DAVI SIMÕES DORO PEREIRA

PIFOP: A WIDE FOR COLLABORATIVE DEVELOPMENT OF MATHEMATICAL MODELS IN AMPL

Belo Horizonte
2021

Universidade Federal de Minas Gerais
Escola de Engenharia
Graduate Program in Production Engineering

PIFOP: A WIDE FOR COLLABORATIVE DEVELOPMENT OF MATHEMATICAL MODELS IN AMPL

Dissertation presented to the Graduate Program in Production Engineering of the Universidade Federal de Minas Gerais in partial fulfillment of the requirements for the degree of Master in Production Engineering.

DAVI SIMÕES DORO PEREIRA

Belo Horizonte
2021

Universidade Federal de Minas Gerais
Escola de Engenharia
Pós-Graduação em Engenharia de Produção

PIFOP: UMA WIDE PARA DESENVOLVIMENTO COLABORATIVO DE MODELOS MATEMÁTICOS EM AMPL

Dissertação apresentada ao Curso de Pós-Graduação em Engenharia de Produção da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Engenharia de Produção.

DAVI SIMÕES DORO PEREIRA

Belo Horizonte
2021

Pereira, Davi Simões Doro.
P436p PIFOP [recurso eletrônico]: a WIDE for collaborative development of mathematical models in AMPL / Davi Simões Doro Pereira. - 2021.
1 recurso online (xii,152 f. : il., color.) : pdf.

Orientador: Ricardo Saraiva de Camargo.

Dissertação (mestrado) - Universidade Federal de Minas Gerais, Escola de Engenharia.

Apêndices: f. 129-143.
Bibliografia: f. 144-152.

1. Engenharia de produção - Teses. 2. Modelos matemáticos - Teses. 3. Otimização matemática – Teses. I. Camargo, Ricardo Saraiva. II. Universidade Federal de Minas Gerais. Escola de Engenharia. III. Título.

CDU: 658.5(043)

Ficha catalográfica elaborada pela Bibliotecária Letícia Alves Vieira - CRB-6/2337
Biblioteca Prof. Mário Werneck - Escola de Engenharia da UFMG

UNIVERSIDADE FEDERAL DE MINAS GERAIS

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE PRODUÇÃO

UFMG

Approval Sheet

This master's thesis, entitled LAMODAL: A WIDE FOR COLLABORATIVE *DEVELOPMENT OF MATHEMATICAL MODELS IN AMPL*, prepared and submitted by **Davi Simões Doro Pereira** *as* a partial requirement for a Master of Science degree in **Production Engineering**, main research area **Operations Research and Interventions in SocioTechnical systems**, and sub research area **Optimization and Simulation of Logistics and Large Scale Systems**, has been examined and recommended for approval and acceptance by the defense committee listed below.

Belo Horizonte, September 16, 2021.

Prof. Robert Fourer, Ph.D.

DocuSigned by:
Gilberto de Miranda Jr.
7CFBB68668AF481...

Prof. Dr. Gilberto de Miranda Junior

Prof. Dr. Raoni Barros Bagno

Prof. DOUGLAS MOURA MIRANDA
UFMG
Siape: 1801111

Prof. Dr. Douglas Moura Miranda

Prof. Dr. Ricardo Saraiva de Camargo (advisor)

Acknowledgments

God is the creator and maintainer of all things. Everything that happens, happens by his determination. As stated by the psalmist, "all the days ordained for me were written in your book before one of them came to be" (Psalms 139:16, New International Version). So if I am here today, it is because of God's favor towards me. Couldn't he have made me blind, or deaf, or mute, or poor, or shorten my life with an early death? Yes, he could. "Who gave human beings their mouths? Who makes them deaf or mute? Who gives them sight or makes them blind? Is it not I, the Lord?" (Exodus 4:11). "The Lord brings death and makes alive; he brings down to the grave and raises up. The Lord sends poverty and wealth; he humbles and he exalts" (1 Samuel 2:6,7). But instead of putting me in unfortunate circumstances like these, he decided to surround me with many undeserved gifts, through which he has enabled me to do the present work. I would like to mention a few of these gifts.

First, my parents, Cornélio and Hilda. Although you are not doctors, when you taught me the bible you were teaching me things much more important than all the things that I have learned from the doctors I have had the privilege to study under.

Second, my siblings, Thiago and Mariana, and siblings-in-law, Daiane and Rodrigo. Our family reunions are always a good distraction from the everyday work.

Third, my brothers and sisters from Comunidade Evangélica do Castelo. You were always ready to pray for me when I asked, and also when I didn't.

Fourth, my advisor, professor Dr. Ricardo Camargo. I'm not exaggerating when I say that this project would never come into being without your expertise, support and patience. Not only you have equipped me with necessary skills to do

this work, but you're also the idealizer of the whole thing. It has been a privilege to work with you for all these years, and I'll hopefully continue to have this privilege for many years to come.

Fifth, professor Dr. Carlos R. V. de Carvalho. You were always very supportive of this project from the beginning, and has promptly accepted the challenge of being the first teacher to use PIFOP in your classes, even though you didn't know how well it would work.

Sixth, the students that have participated in our study. You have very patiently endured the bugs and limitations of PIFOP, and your feedback has been essential for us to improve the application.

Seventh, the millions of Brazilians that unknowingly have financially contributed to this project. I wish you had the option not to. Nevertheless, I recognize the good that you've done for me through CAPES, the institution that has funded my graduate studies.

Thank you all for your support. And thank you God for all these people. Above all, thank you for the gift of faith in Jesus Christ, the most undeserved of all gifts, through which men have eternal life (John 3:36).

Agradecimentos

Deus é o criador e mantenedor de todas as coisas. Tudo o que acontece, acontece por sua determinação. Como diz o salmista, "todos os dias determinados para mim foram escritos no teu livro antes de qualquer deles existir" (Salmos 139:16, Nova Versão Internacional). Portanto, se eu estou aqui hoje, é por causa do favor de Deus para comigo. Não poderia ele ter me feito cego, ou surdo, ou mudo, ou pobre, ou encurtado minha vida com uma morte prematura? Sim, ele poderia. "Quem deu boca ao homem? Quem o fez surdo ou mudo? Quem lhe concede vista ou o torna cego? Não sou eu, o Senhor?" (Êxodo 4:11). "O Senhor mata e preserva a vida; ele faz descer à sepultura e dela resgata. O Senhor é quem dá pobreza e riqueza; ele humilha e exalta" (1 Samuel 2:6,7). Mas ao invés de me colocar em circunstâncias infelizes como essas, ele decidiu me cercar com muitos imerecidos presentes, através dos quais ele me habilitou a realizar o presente trabalho. Eu gostaria de mencionar alguns desses presentes.

Primeiro, meus pais, Cornélio e Hilda. Embora vocês não sejam doutores, quando vocês me ensinavam a bíblia vocês estavam me ensinando coisas muito mais importantes do que todas as coisas que aprendi dos doutores com os quais eu tive o privilégio de estudar.

Segundo, meus irmãos, Thiago e Mariana, e cunhados, Daiane e Rodrigo. Nossas reuniões em família são sempre uma boa distração da rotina de trabalho.

Terceiro, meus irmãos e irmãs da Comunidade Evangélica do Castelo. Vocês sempre estavam prontos para orar por mim quando pedia e também quando não pedia.

Quarto, meu orientador, professor Dr. Ricardo Camargo. Eu não estou exagerando quando digo que este projeto nunca teria vindo à existência sem sua expertise, apoio e paciência. Não apenas você me equipou com as habilidades

necessárias para realizar este trabalho, como também você é o idealizador de toda a coisa. Tem sido um privilégio trabalhar com você todos esses anos, e espero continuar tendo este privilégio por muitos anos por vir.

Quinto, o professor Dr. Carlos R. V. de Carvalho. Desde o início você esteve nos apoiando neste projeto, e prontamente aceitou o desafio de ser o primeiro professor a utilizar o PIFOP em suas aulas, ainda que você não soubesse quão bem a ferramenta iria funcionar.

Sexto, os estudantes que participaram de nosso estudo. Foi com muita paciência que vocês toleraram os *bugs* e limitações do PIFOP, e seus comentários foram essenciais para que pudéssemos melhorar a aplicação.

Sétimo, os milhões de Brasileiros que, sem saber, contribuíram financeiramente com este projeto. Eu gostaria que você tivesse a opção de não fazê-lo. Não obstante, eu reconheço o bem que vocês fizeram a mim através da CAPES, a instituição que financiou meu mestrado.

Obrigado a todos pelo apoio. E obrigado Deus por todas essas pessoas. Acima de tudo, obrigado pelo dom da fé em Jesus Cristo, o mais imerecido de todos os presentes, através do qual homens têm a vida eterna (João 3:36).

Abstract

The present dissertation studies the challenges faced by providers of web IDEs for optimization and their users, and our efforts to address these challenges with a web IDE (WIDE) developed by us. Mathematical programming is one of the most important techniques for solving optimization problems, and the AMPL language is undoubtedly one of the most adopted languages for implementing mathematical models. Even though the operations research community has many options of optimization tools to choose from, few of them facilitate the collaboration between modelers and the remote execution of optimization solvers. One of our goals is to present our tool that can help operations research students, educators and practitioners in their work. PIFOP (**P**rogramming **I**nterface **F**or **O**ptimization **P**roblems) is a WIDE specially crafted to help modelers to develop mathematical programs in AMPL. Additionally to presenting our application, here we also discuss i) the problems involved in the provisioning of WIDE services, ii) the usability of our tool in an operations research class and iii) the challenges faced by mathematical programmers, especially AMPL users.

Keywords: web IDE, AMPL, optimization modeling tools, optimization teaching.

Resumo

A presente dissertação estuda os desafios enfrentados pelos provedores de web IDEs para otimização e seus usuários, e os nossos esforços para superar estes desafios através de uma web IDE (WIDE) denvolvida por nós. Programação matemática é uma das mais importantes técnicas de resolução de problemas de otimização, e a linguagem AMPL indubitavelmente é uma das mais adotadas para implementação de modelos matemáticos. Embora a comunidade de pesquisa operacional possua muitas opções de ferramentas de otimização, pocas delas facilitam a colaboração entre modeladores e a execução remota de resolvedores. Um dos nossos objetivos neste trabalho é apresentar nossa ferramenta, que pode ajudar estudantes de pesquisa operacional, educadores e profissionais da área em seus trabalhos. PIFOP (**P**rogramming **I**nterface **F**or **O**ptimization **P**roblems) é uma WIDE especialmente desenvolvida para ajudar modeladores a desenvolver programas matemáticos em AMPL. Além de apresentar nossa aplicação, aqui nós também discutimos i) os problemas envolvidos no provisionamento de serviços oferecidos por WIDEs, ii) a usabilidade de nossa ferramenta em um curso de pesquisa operacional e iii) os desafios enfrentados por programadores matemáticos, especialmente os usuários do AMPL.

Keywords: web IDE, AMPL, ferramentas de modelagem para otimização, ensino de otimização.

List of Figures

List of Algorithms

Contents

Chapter 1

Introduction

PIFOP (**P**rogramming **I**nterface **F**or **O**ptimization **P**roblems) is a web integrated development environment (WIDE) for AMPL (**A** **M**athematical **P**rogramming **L**anguage). In its early days, it was no more than a web interface for developing and submitting AMPL models to the *NEOS server*[1] (Czyzyk et al. (1998), Dolan (2001), e Gropp and Moré (1997)), a free optimization server hosted at the Wisconsin Institute for Discovery. Today it is possible to use PIFOP to execute AMPL remotely in most Linux machines, a feature that graduate and undergraduate students at UFMG have been using to do AMPL modeling from their own machines but using the laboratory machines to solve their models.

AMPL is one of the most popular algebraic modeling languages available. According to the usage statistics of the *NEOS server*, the language is the most popular among their users. Of the 2,351,976 problem submissions between July 2020 and July 2021, 55% used AMPL as the input language, 42% used GAMS and 3% used other languages (NEOS Statistics, 2021). Additionally, according to Fourer (2020), in 2020, AMPL was being used in 685 courses of 312 universities in 53 countries. These numbers show that AMPL is a widely adopted language for mathematical programming. Hence it would be natural to have a WIDE for AMPL models.

A WIDE is a web application that provides various facilities to help them in the production of an artifact from a source code. Due to the conveniences provided by WIDEs in comparison to desktop IDEs, it is becoming increasingly common for

[1] `neos-server.org`

the latter to make their way into the web. More recently, Microsoft has announced its new product Visual Studio Codespaces (Molnar, 2020), which is essentially a cloud-based version of their popular desktop IDE Visual Studio. Though many other WIDEs for general programming have emerged in the last years (see Chapter 5), WIDEs for mathematical optimization are not as common.

In this work, we present to educators and practitioners of the operations research community this new tool, which, to the best of our knowledge, is one of the very few WIDEs specifically designed for mathematical optimization. The tool, initially called PIFOP (**Lab**oratório de **Mod**elagem **Al**gébrica), started being developed in 2018, as part of our undergraduate thesis. In 2019, PIFOP has appeared in the 30th European Conference on Operational Research, in Dublin. In 2020, the registration process of PIFOP in the INPI (Instituto Nacional da Propriedade Industrial, or National Institute of Industrial Property) has been initiated (process number: 512020001695-2), and an article about our tool has been accepted in the journal Pesquisa Operacional para o Desenvolvimento (Pereira and Camargo, 2021). In this year of 2021, PIFOP began to be publicly offered under the name of PIFOP[2], the same name of the company that we've opened to commercialize the tool. Teachers and instructors that use AMPL now have a more efficient way to work with their students, and the students have a better way to work in groups. Additionally, researchers can easily share their models with reviewers, and AMPL modelers in general have an alternative way to use machine resources remotely. In the next section we will talk about what has motivated us to start this project.

1.1 Motivation: the challenges of mathematical modeling

As engineers, we are always looking for intelligent solutions to the practical problems we encounter. A typical operations research work will usually propose solutions for other people's problems. But our concern here is with problems faced by operation research students and practitioners themselves. Throughout the years as production engineering students, we have observed the challenges associated

[2] https://pifop.com

with the mathematical modeling activity, some of which we have struggled with ourselves as students, and some that we knew that were faced by our teachers and by practitioners in the industry. We'll start off by pointing out issues related to operations research practice in general, as they can appear in different contexts, and then we'll talk about the challenges in an educational context, more specifically, challenges that affect operations research students and teachers.

Generic challenges

There are two essential tools that a modeler needs in order to solve a particular optimization problem: a text editor and a solver. With the text editor the modeler implements the optimization model in the language of their choice and use that implementation as input for the solver. However, there are situations in which these tools will not be sufficient. Often modelers will be working on large problems that require a substantial amount of computational power to be solved, problems that the average consumer-grade machine will not be capable of solving. For these cases, the modelers will need to access more powerful machines, and unless they can access them physically they will do it remotely, which will require the use of some kind of remote connection tool.

A SSH client (a program that uses the Secure Shell network protocol to establish secure connection with a remote machine) can certainly serve this purpose, but it can be very inconvenient for modelers to have to login, transfer files and call the solver every time they make a change to their models. Much like general programming, mathematical programming is an iterative process, in which the optimization program will be changed and tested multiple times during development. And for this reason we think that it would be very useful for modelers to have a simpler way to execute their models remotely, which would decrease the development iteration time.

Aside from the issues related to the usability of SSH tools, the SSH approach also poses some challenges to machine administrators. For instance, it will require extra caution if they want to restrict what modelers can do with the machines they are given access to, limiting the usage of the solver and the consumption of the amount of memory and CPU time of their processes. To give a concrete

example, in the Systems Simulation and Optimization Laboratory (LASOS) at UFMG they want undergraduate students to be able to use the solvers in some machines remotely without giving them access to anything else. Additionally, they also want to have different limits for the memory and CPU time consumption by the students, researchers and teachers. In this scenario, which could very well have a similar occurrence in some companies and organizations, granting SSH remote access is more than what is actually needed, and as a result extra work is needed in preventing an undesired usage of the machines, which could happen either intentionally or unintentionally.

We should mention that the need of remote access to solvers is not only due to the demand for more computational resources by modelers but also due to how some optimization tools are licensed. For instance, AMPL offers a type of license that allows the product to be run by any number of users in any number of simultaneous processes, but on one designated computer. Since copying the product to other machines is expressly disallowed, and since the acquisition of more licenses would incur in a great increase in the costs by institutions that want to have multiple users using the product, a remote usage solution is much needed.

Educational challenges

Now we turn our attention to the challenges faced by operations research teachers and students. In every applied mathematics field of study, the educators in that field will have to face the challenge of achieving a balance between teaching 1) the mathematical theory and 2) the application of that theory. Often the application of the theory will require the use of computer tools. In the field of operations research the situation is not different: at the same time that the students need to acquire a lot of mathematical background before they start solving optimization problems, the application of the theory happens almost exclusively through the use of computer tools.

One of the most used tools in operations research is the optimization problem solver, or just solver, which is a computer program that actually implements the algorithms the students learn about in a classroom (Simplex, branch-and-bound, etc). Interfacing with solvers is not straightforward, especially for those students

who are having their first contact with programming languages. Moreover, teaching these students how to use a solver can be very challenging and demanding. Often the teacher will spend a good deal of time managing issues which are not directly related to the class subject but with the usage of the computer tools, such as the command line terminal, and the IDE (if they are using one), or with configuration related issues, such as the modification of environment variables of the operational system, or even dealing with issues related to the particular operating system the student may be using.

The time spent by the teacher in helping students with these kinds of difficulties is time not spent in the teaching of the actual subject of the class. Likewise, the time spent by the student dealing with these kinds of issues is time not spent learning the actual subject of the class. So if we could minimize the occurrences of students needing help in these regards, then we would have increased the time dedicated to the class subject both by the teacher and the students.

Many of the difficulties encountered by the students in getting started with implementing optimization models that we have mentioned so far could be mitigated by a tool that is platform-agnostic. That is, if there was a tool that is operational system independent, the issues related to download, installation and configuration would have been mitigated.

Another kind of difficulty emerges from the need of the students having to share their model implementations both with other students and with teachers, a situation which is common when there is a group assignment to be done by the students and evaluated by the teacher. Working collaboratively to complete an optimization modeling assignment can be very challenging without proper tools that facilitate the task. If the group decides to meet physically to do the work, there will be the time constraints of each member of the group. If they decide to work together remotely, they will have to rely on sharing copies of documents and make sure that each member knows what the other is doing without actually seeing what they are doing. But since the emergence of web applications like *Overleaf*, *Google Docs* and *Replit*, that allow users to work simultaneously in the same project, we see no reason why there should not be such a tool for mathematical modeling.

1.2 Research Methodology

Although as engineers we tend to be very pragmatic, we are also researchers, and as researchers we know that intelligent solutions for practical problems can be more easily devised after some investigation of the issues involved.

Strictly speaking, we cannot confine our work to a single research methodology. We aim to investigate multiple problems related to what we are trying to do, which is to provide a web application that can help mathematical programmers, both in the academy and in the industry, in their daily works. To do so, we have adopted different research methodologies depending on the problem we are studying.

At a higher level, however, when we look at our work as a whole rather than at each of its parts separately, we can say that our overall procedure merges characteristic from a qualitative research aimed at answering a set of research questions, and from a design science research project.

"Design Science Research (DSR) can be defined as a research strategy aimed at the development of generic solutions for field problems (end) through designing and testing them in the field (means)" (Van Aken and Berends, 2018). A DSR project can be divided into five steps: Awareness, Suggestion, Development, Evaluation and Conclusion (Takeda et al. (1990), Lacerda et al. (2013)). First, the researcher becomes *aware* of the problems. Next, they *suggest* one or more ways in which these problems can be solved and *develops* the suggested solution(s). The solution is then *evaluated*, which can take the researcher back to the first step if the devised solution is problematic or can lead him to *conclude* the project, deciding on a solution to be adopted and planning their next action.

Although the order in which we present the results of our efforts is not exactly the order presented above, all of the steps can be found in our work to some extent: we study the problems we have to face, suggest how they can be solved, develop the solution, evaluate it and plan our future actions. However, we say that we have DSR characteristics rather than that we strictly follow this strategy because, although the problems we discuss are present in other contexts and the solutions we devise are generic enough as to be applied in these contexts, we make little to no effort to generalize our solutions and to indicate how and when they can be applied in other contexts.

In any case, in comparing our overall procedure with that of a DSR project we want to prepare the reader for what they are about to read. Our research is primarily aimed at developing technological solutions that can solve practical problems. With that goal in mind, we set ourselves to answer four research questions, which we expect i) will lead us to better decisions as we continue to improve PIFOP, and ii) will help us to evaluate the decisions we've made so far.

WIDE providers and their challenges

The first two questions are about the technical challenges of WIDE providers, which is what we are. WIDE designers and developers have to face some unique challenges in order to bring to the web the traditional features of desktop IDEs. The simplest functionalities, such as opening and displaying a file on the screen, raise in complexity due to the separation between the WIDE server and the user interface. And running compilers and interpreters is no longer as simple as calling a program on the user's machine. On the other hand, the fact that WIDEs already require the integration with web technology makes some features more feasible than it would be for a desktop application, such as real-time collaborative editing and project sharing.

Different systems have been developed in order to provide IDE features in a web environment. In the literature, we have found many WIDE projects, but most of them seem to have been discontinued: Arvue (Aho et al., 2011), Web2Compile (Santos et al., 2014), IDE 2.0 (Itahriouan et al., 2014), CEclipse (Wu et al., 2011), Harmonik (Yulianto et al., 2017) and WeScheme (Yoo et al., 2011) are some of them. Arvue enabled the development and publishing of Vaadin based web applications (Vaadin is a full stack framework for building web apps in Java). Web2Compile was a tool for developing applications for wireless sensor networks. IDE 2.0 provided an environment for developing PHP, HTML and JavaScript projects. CEclipse was a platform for Java development in the cloud. Harmonik allowed users to create programs in a pseudo-code language, which got translated to programming languages like C++ and Java, compiled and executed from the browser. WeScheme is an environment for Scheme and Racket programming, and

is the only WIDE of the previous list that is still available[3].

Although a reasonable amount of information is provided on the inner workings of these systems, the fact that most of them have been discontinued prevents us from testing if the system successfully supports the features that they advertise.

Outside the literature there is a number of successful WIDEs in the market readily available for testing. A selection of them will be presented in Chapter 5. These have been more extensively used than the ones presented in the literature, which means their systems have been passing the test of practice to some extent. Learning how they provide their features can give important insights to system designers and developers. However, although there is much to be learned from popular WIDEs, information about their systems can be sparse and come from a variety of different sources in the web. If we can consolidate in a single work the information about the different approaches that have been taken to provide the more important features of an IDE in a web environment, we will have produced a reference material that could be useful for software engineers and computer scientists. To put it in form of a research question: **(RQ1) How do WIDE providers overcome the main challenges they have to face in order to effectively meet user demands?**

Additionally, we would also like to study the particularities of WIDEs for optimization. Different audiences have different needs. For instance, while programmers may need the ability to use a number of libraries and a variety of tools, for a modeler it may be sufficient the ability to call an interpreter and a solver engine. On the other hand, while programmers may need no more than 1 or 2 GB of memory to run their programs, 10 GB may not be enough for a modeler to solve their problem. These differences in needs will affect how a WIDE system is designed and what features are provided. To explore how these differences impact the architecture of a WIDE for optimization, we set ourselves to answer the question: **(RQ2) How do providers of WIDEs for optimization overcome the main challenges they have to face in order to effectively meet user demands?**

There are not nearly as many WIDEs for optimization as there are for general programming. After looking for WIDEs designed for mathematical optimization over the literature and over the web, only two have been found (more on that on

[3] wescheme.org

Chapter 5). This, and the fact that popular optimization languages like AMPL and GAMS still haven't made their way into the web, shows the useful novelty of a work that both analyses and compares the currently available WIDEs for optimization while also presenting a new one.

Mathematical modeling education

The other two questions are related to mathematical modeling education, a field to which we are eager to contribute with our web application and website. Eady and Lockyer (2013) discuss some purposes for which technology can be employed for education, among which there are i) *collaboration* and ii) *creation*. Technology for collaboration is one that supports a situation in which two or more students work together to solve a problem, and technology for creation is one that provides the means for creating some deliverable, that is, the resulting artifact of some work. In the context of mathematical optimization, this artifact may be the implementation of an optimization model or the results of a computer experiment. Since PIFOP does enable multiple students to work in the production of the same artifact, our tool does have at least these two kinds of application in a educational context.

However, the effectiveness of integrating our tool in this context still needs to be assessed, which we intend to do by answering the following question: **(RQ3) What are the student's perception of PIFOP as a tool for collaborative work and optimization model creation?** In answering this question we expect to provide educators in general some guidance on what software features to look for when searching for tools to integrate in their classes, and also to provide some insights to software developers on what features are most relevant in order to facilitate collaboration and creation in an educational environment.

Our approach to answer this question will be to use PIFOP as the means through which students in the undergraduate level will make their assignments, both individually and collaboratively. Then we plan to assess the student's perceptions of the tool through a questionnaire covering a variety of different topics related to the usefulness of the tool in assisting them to do different tasks.

In this study we are not going to focus on any teaching methodology, but rather let the teacher employ our application in the way they find it most appro-

priate. There is a variety of teaching/learning approaches that can be taken by teachers/students, of which we mention a few. *Lecture-based learning* is the traditional method of teaching in which the students passively listens to the teacher talk for a period of time (Konopka et al., 2015). *Self-directed learning* is a process in which a student is responsible for organizing and managing their learning activities (Segen's Medical Dictionary, 2011). *Blended learning* is the practice of using both online and in-person learning experiences when teaching students (The Glossary of Education Reform, 2020). *Peer learning* refers to the process through which learners acquire knowledge and skills through active helping and supporting among status equals or matched companions (Mustafa, 2017). *Project-based learning* is a teaching/learning model that involves students in problem-solving tasks, allows students to actively build and manage their own learning, and results in students-built realistic deliverables (Zafirov, 2013). These approaches, rather than being mutually exclusive, can and are employed together, although one of them may be prioritized by a teacher/student.

Despite the differences that there are between teaching/learning approaches, often students will be involved in some kind of practical activity at some point, and, in the context of mathematical optimization, this activity will most likely involve the computer implementation of optimization models and execution of computer experiments. Depending on the teaching strategy adopted, this may be done through many small homework assignments, or one medium-sized project to be done in a semester, or laboratory practice tasks, or computer-aided final exams, etc. Because of this omnipresence feature of practical activities within the teaching/learning strategies, computer tools that can facilitate the implementation of such activities can be adopted with a variety of educational strategies.

As a secondary contribution for education, we would like to help students and practitioners to find solutions to commonly faced issues when programming in AMPL. Following what Cline et al. (1998) and Peavy (2009) have done for C++, Hornik et al. (2002) for R, Kanerva (1997) for Java, and Overleaf for LaTeX[4], as a long-term goal, we would also like to produce a repository of knowledge on the AMPL language and publish this repository in our website. There already is an

[4] overleaf.com/learn

AMPL language FAQ at their website[5], however, it could be enriched to cover a much broader range of issues AMPL users may face when modeling. In order to achieve this long-term goal, a preliminary work is necessary, which is to answer the following question: **(RQ4) What are the most common issues faced by AMPL users?** Our procedure shall be to investigate the issues experienced by AMPL Google Group[6] users, analyze and categorize them and, later, devise guides addressing common issues that can be accessed through our website.

1.3 Dissertation Outline

The present work is divided into two major parts. Part I is a detailed description of PIFOP. The goal here is to present to the reader the tool that we have developed. Chapter 2 shows how PIFOP users interact with the tool. Chapter 3 presents an overview of the architecture of the system. Finally, Chapter 4 provides greater details of some of the more important technical aspects of our system.

Part II aims at answering the research questions presented in Section 1.2. In Chapter 5 we set ourselves to answer **RQ1** and **RQ2**, which relate to the workings of WIDEs. In Chapter 6 we present the case study we have conducted in order to answer **RQ3**. In Chapter 7, our question **RQ4** about common issues faced by AMPL users is answered.

We conclude in Chapter 8 with a summary of the lessons we have learned during this project, and a comment on future work possibilities for ourselves and for others.

[5] ampl.com/resources/faqs
[6] https://groups.google.com/g/ampl

Part I

System Features

Chapter 2

PIFOP Usage

The goal of this chapter is to provide the reader with a general notion of what PIFOP users can do and how can they do it. This will help the reader to better understand the references we make to PIFOP in other chapters. Here we are going to demonstrate how to use PIFOP to 1) create modeling projects, 2) run optimization problem solvers, 3) share projects with other users and collaborate with them and 4) host an optimization server. The reader is welcomed to try the application for themselves at pifop.com.

2.1 Creating Projects

Only a few steps are needed to start working on an optimization model. PIFOP has two screens: 1) the starting screen (Figure 2.1), where the user can view their projects, the optimization servers they have access to and information about their account and 2) the application screen (Figure 2.2), where the user works on their models. When accessing PIFOP for the first time, the user enters the starting screen, where they use the *New Project* button to create a new project.

After clicking this button, the user will be asked to give a name to the project and, optionally, to upload some files. An optimization project in AMPL is typically composed of at least three files: a file containing the model, a file containing the input data and a file containing the script to be executed. To exemplify, here we are going show how to create a project with the files of one of the examples in

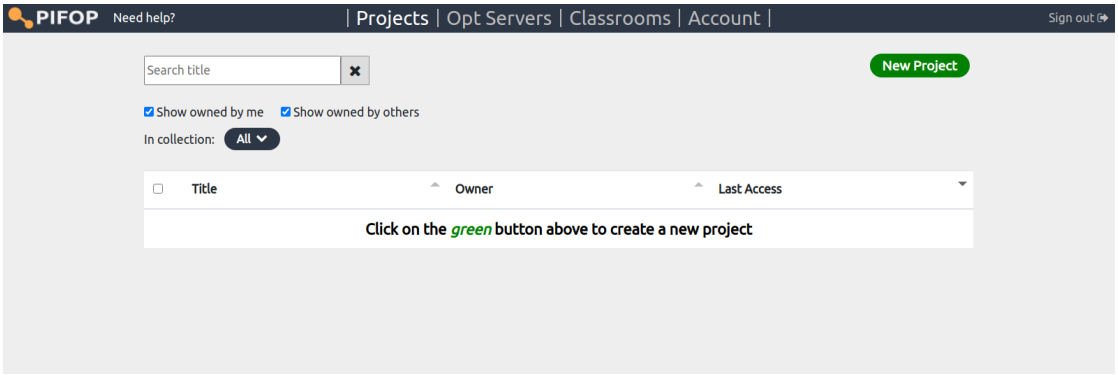

Figure 2.1: Starting screen.

the AMPL reference book (Fourer et al., 2002). It is a small linear optimization model to maximize the profit of a steel producing company. The files of this model can also be found in our AMPL examples page[1], in the example entitled "Steel production model".

After completing this step, the user will be directed to the application screen (Figure 2.2, where the newly created project will be loaded and ready to be edited and executed. In order to edit a file, the user can open it in the editor by double clicking the file name in the file explorer on the left. Once the user is done editing, the next step is to solve the model using an optimization solver. To this end we use the terminal on the right, through which the user can execute solvers using their files as input.

2.2 Running a solver

There are three ways in which users can solve their models. First, users can run the GLPSOL solver locally in the browser. GLPSOL is a free solver for mixed linear programming problems that we enable for in-browser execution through *WebAssembly* technology. To use it, there is a `glpsol` terminal command, which takes as input the name of the model and data files, as shown below.

```
> glpsol -m <model> -d <data>
```

[1] https://pifop.com/help/ampl_examples.html

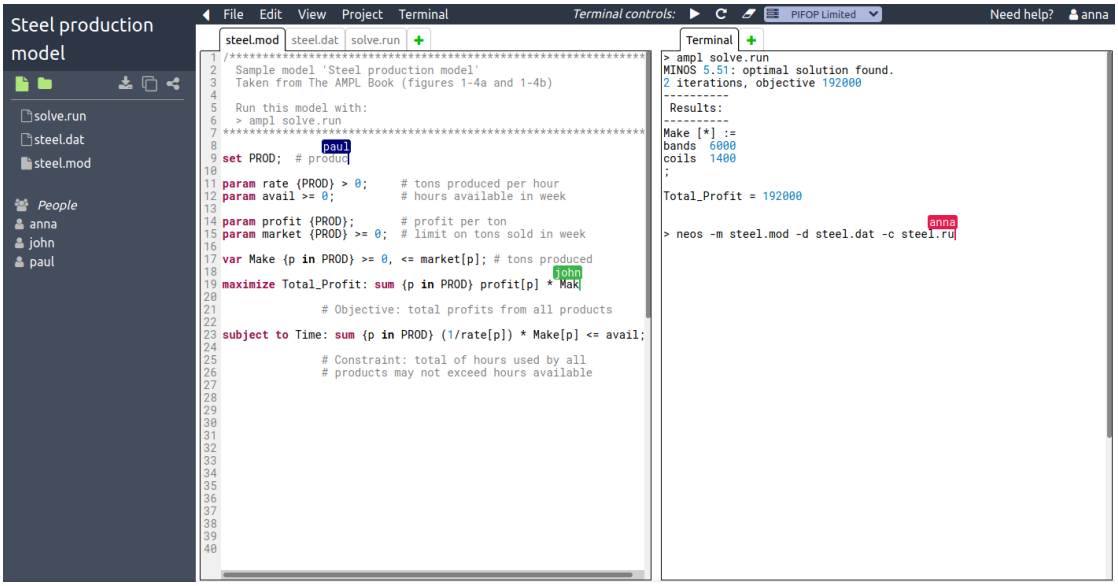

Figure 2.2: Application screen.

Second, users can submit their files to the NEOS server to be solved remotely. There are many available solvers in this free optimization service, including commercial ones, e.g., CPLEX. The terminal command to use the service is `neos`, which takes as input the model, data and commands files and the name of the solver to be used.

```
> neos -m <model> -d <data> -c <commands> -s <solver>
```

Third, users can run solvers remotely in an optimization server hosted by a PIFOP user. More will be said about hosted optimization severs in Section 2.4. A user that has permission to use a given optimization server can do it by selecting the desired server in the selector just above the terminal and entering an `ampl` terminal command just as if they were using the AMPL desktop command line interface.

```
> ampl <main-file>
```

Each of these three commands have other options that can be configured in the command line. To list the available options, the user enters into the terminal the command name followed by `--help`, e.g., `glpsol --help`.

The execution output is printed into the terminal in real-time to all the online project collaborators. If the model execution requires many hours or days to finish, users can close the browser tab without the execution being interrupted, and then return in another moment to see the results.

2.3 Sharing Projects

PIFOP offers two sharing options: the first allows users to view a shared project but not to edit. Users can then make a personal copy of the shared project and edit as they wish. The second one allows users to view and edit the shared project. To exemplify these types of sharing options, let's suppose the following situation: A teacher wants to give a modeling exercise to his students. In this exercise, students must complete an optimization model with some missing information. For instance, the teacher removes the constraint and objective function from the example of the previous section. In this scenario, it is desirable to have each student or group edit their own copy of the project containing the incomplete model and, after completing the task, share their solution with the teacher. The teacher will share their project with the students with permission for viewing only, so that the teacher's own copy of the project is not edited, but only accessed to be copied. To do this, the teacher will click on the *Share Project* button in the menu on the left and copy the URL address that gives users permission for viewing only. The teacher will then share this URL with their students using any communication channel.

By accessing the project through this URL, the students will only be able to view it. To create an editable copy of the project, the student will click on the *Copy* button in the left menu. Once copied, the project can be edited and executed as described in the previous subsection. If the work is in a group, the URL address that gives permission to edit the project can be shared with the students within that group, and all of them will be able to work on the project simultaneously. After the exercise is finished, it is the student's turn to share their project with the teacher. The teacher will then be able to access the project and evaluate the correctness of the proposed solution.

Another scenario for using project sharing in view only mode can be seen in our help page. There the reader will find several examples from the AMPL language book ready to be accessed. Even in this access mode the user can use the terminal and see the results of the model execution. If the user wants to make changes to one of these examples (for instance, add a set of constraints) they can just copy the project and edit the model file. A teacher who wants to demonstrate to their students a more advanced modeling technique, e.g., dynamic cut insertions or column generation, can easily create an example project and share it with their students. And a researcher that is proposing a new mathematical formulation for an optimization problem can easily share their model, for instance, with the reviewers of the journal to which they are submitting their article. In addition to containing their own formulation, the researcher may include in the project the formulations already known in the literature, to facilitate the performance comparison by the reviewers.

2.4 Hosting an Optimization Server

Besides the possibility of collaborating remotely and synchronously, one of the advantages of PIFOP is the remote execution of the models. Though users have access to our demo server and to the *NEOS server*, they can also host a server on a Linux machine of their own. That way they can have full control over who can access their server. Note that servers can be set for either private or share usage.

An example of the former usage is when the user has the AMPL installed on their personal computer and want to use it via PIFOP. The second type of usage is thought for research groups and universities. It allows users to use the resources of a machine and its solvers without giving them direct access to it.[2]

To host an optimization server, the user only needs 1) some version of AMPL (either the full version or one of the free options) and 2) the optimization server program itself. It is important to highlight that our service is offered apart from AMPL, and that users interested in purchasing the AMPL should do so by access-

[2]It is important to mention that this functionality is found in an experimental phase, and that its gratuity will last as long as this phase lasts.

ing the official product website. Below we show command lines for download the *demo* version of AMPL and the optimization server.

```
> wget https://ampl.com/demo/ampl.linux64.tgz
> wget https://pifop.com/download/opt-server.zip
```

The user must then extract each file in the directories of their choice and enter the extracted directory for the `opt-server.zip` file and run the program `opt-server`.

```
> tar xf ampl.linux64.tgz
> unzip opt-server.zip
```

```
> cd opt-server
> ./opt-server
```

The first time the server is executed, the user will be asked to give the server a name and enter the full path to the folder containing the AMPL interpreter, that is, the absolute path of the folder that was extracted from `ampl.linux64.tgz`.

```
1) Give your server a name: My Server
2) Full path to the directory containing ampl: /path/to/ampl-dir
```

Once the optimization server connects to the PIFOP main server, the user will sign in with their account and the optimization server will be ready to be used. This can be confirmed by accessing any project on PIFOP: above the terminal, in the optimization server selection field, the user will see their server listed among the options.

If the server is for shared use, it will be necessary to list the users that can use it and their permissions and computational limits. This is done by modifying the `config.json` file that was generated on the first execution of the opt-server, located in the server folder. First, the user hosting the server needs to create user groups and, second, add users to those groups. For this, a `user-groups` section containing the permissions of each group is added to the configuration file. For example, if we wanted to define a group called "Students" and establish limits of memory and time usage of processes initiated by these users, our configuration file should look like Figure 2.3.

```
{
    "name": "My Server",
    "remember-me": true,
    "ampl-directory": "/path/to/ampl-directory",

    "user-groups": {
        "Students": {
            "max-memory": 20,
            "max-time": 60
        }
    },

    "users": {
        "Students": [
            "mariabazotte",
            "jvitor",
            "davidoro@ufmg.br"
        ]
    }
}
```

Figure 2.3: Example of a `config.json` configuration file.

A complete list of limits and permissions that can be configured for each group of users can be found in our website[3]. Finally, the server is restarted for the changes take effect.

[3] https://pifop.com/help/hosting_opt_server.html

Chapter 3

System Overview

In this chapter our goal is to give the reader an overview of the system, hoping we can contribute with those involved in building systems or applications similar to PIFOP. We do not intend to affirm that our system is the model to be followed. Rather, the description made here should only be taken as an example how to structure a system to provide PIFOP's main functionalities, namely collaborative editing and remote process execution. Although our application is for the specific context of mathematical modeling, the system, in general, is independent of this specialization that we made of it, and so we believe that learning about it can help developers with their design decisions.

3.1 Three-Layers Architecture

Our system's architecture is composed of three layers: presentation, intermediation and execution (Figure 3.1). The *presentation* layer consists of the graphical interface with which the user interacts using their browser. The optimization models are executed on the optimization servers, which are in the *execution* layer. Between these two layers there is the *intermediation* layer, responsible for transmitting the user files to the execution layer and for transmitting the results of the execution to the presentation layer, and also for propagating edits made in the projects to all connected collaborators.

In the two extreme layers (presentation and execution) there can be more than

one computer, while in the middle layer there is only one: the *hub*. In addition to the functions of intermediation between users and optimization servers, and between collaborators of a project, the *hub* also performs other subsidiary functions, such as authentication, file storage and control of user access to optimization servers. There is no direct communication neither between collaborators nor between users and optimization servers. In this way, we guarantee 1) that the only users who will be able to access a project are the users with the access key for that project and 2) that the only users who can use an optimization server are the users that the server administrator has allowed. Between optimization servers there is no communication whatsoever, either direct or indirect through the *hub*.

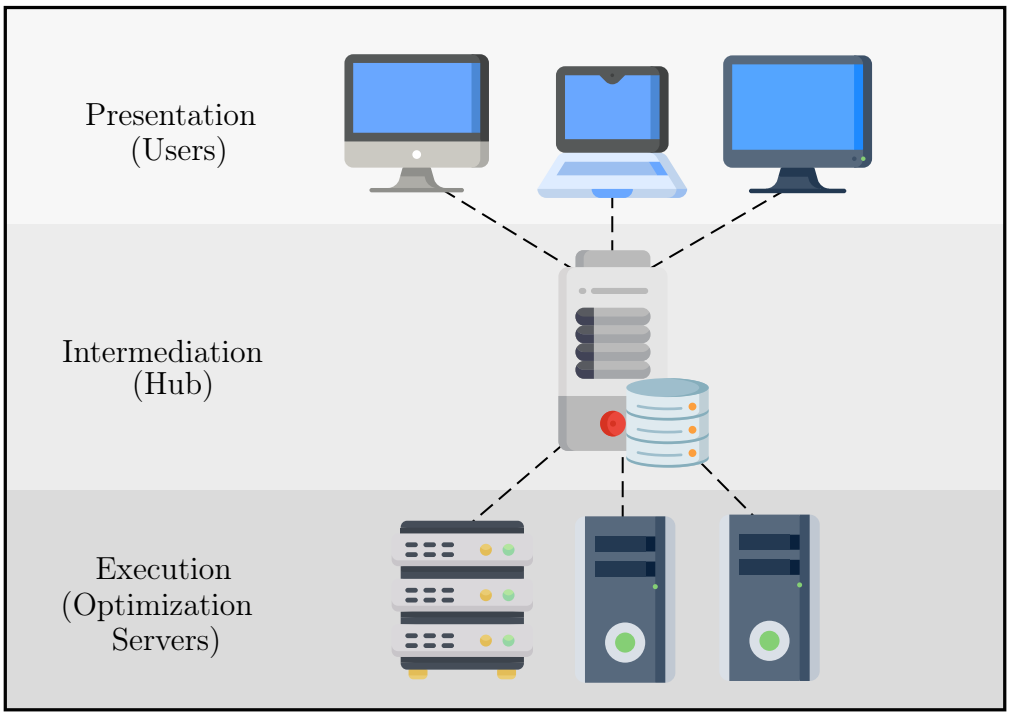

Figure 3.1: The three layers of the system.

Below we will see some of the most important characteristics of each layers.

[0]Icons created by Freeplk from flaticon.com

3.2 Presentation Layer

Implemented in *javascript*, the presentation layer provides the interface through which the user interacts with the system. Communication between the interface and the *hub* takes place through a persistent and bidirectional connection using the communication protocol *WebSocket* (Fette and Melnikov, 2011), which allows either party to transmit messages to the other at any time. Of the three layers, this is the simplest, consisting largely of controls through which the user sends requests to the *hub*, such as creating and accessing projects, editing files, moving folders and running optimization programs. Its most complex component is, undoubtedly, the text editor, mainly because of three functionalities: 1) highlighting of AMPL syntax 2) collaborative editing and 3) support for long files, with hundreds of thousands of lines.

These features of our editor would not would be possible using the standard HTML text input control *textarea*. Although we could have chosen to use some third-party text editor, such as *Ace*[1], *Monaco*[2] and *CodeMirror*[3], either way we would have to make the necessary adaptations to integrate them into our system, and none of them has support for collaboration or syntax highlighting in native AMPL, we've preferred to develop our own editor with these features.

In fact, support for collaborative editing is not just the responsibility of the editor itself, but of the subsystem that includes both the editor and the *hub*. This subsystem is based on the collaboration mechanism known in the literature as *Operations Transformation* (Ellis and Gibbs, 1989a), on which other collaborative editors are also based, such as the editors of the *Overleaf* (Overleaf How-to Guides, 2020), and *Google Docs* (Xu et al., 2014). This mechanism allows edits made by users to be immediately applied locally without definitive divergence between the views of the different project collaborators.

To interact with the optimization server and execute their models, the user interacts with a terminal. In the same way that files are shared among the collaborators of a project, the terminals are also shared. This allows optimization

[1] ace.c9.io
[2] microsoft.github.io/monaco-editor
[3] codemirror.net

programs to be executed only once and its output to be shared among the collaborators, which is especially useful when the program will need hours of execution. In addition, users can open multiple terminals and run different programs on each one. That is, it is not necessary to wait for an execution to finish before starting another one, and different collaborators can work independently, each of them using a different terminal.

3.3 Intermediation Layer

Both the *hub* in the intermediation layer and the optimization server in the execution layer were implemented in C++. In both cases the motivation for this decision was to avoid, at least in part, performance issues. The *hub* has several responsibilities, the most important being that of intermediating communication between users and their collaborators, and between users and optimization servers. This makes it a strong candidate to be the bottleneck of the entire system: the *hub* needs to communicate with all parties, in addition to authenticating users and making database queries, whereas the other parts of the system only need to communicate with the hub.

To establish and maintain communication in *WebSocket* we use the open source library *libwebsockets*[4], by Andy Green. We also decided to keep the database on the same machine as the *hub* and use the SQLite database manager to manage it, which allows us to access it directly from our program using the SQLite C API.

When a user requests access to a project, files stored in the database are loaded into the *hub's* memory and a project editing session is created. If other users access this project, they will be linked to the same editing session. This editing session contains the most recent versions of the project's files, which are automatically saved every 5 seconds. When a user makes a modification to a file, this modification is propagated to all users linked to the editing session. Like mentioned in the previous section, we use the *Operational Transformation* mechanism to resolve conflicts between modifications carried out by different users at the same time, so that there is no definitive divergence between the views that users have of the files

[4]libwebsockets.org

they are editing, although there may be temporary divergence while the changes are not propagated. The editing session remains open as long as there are users connected to the project or as long as there are optimization programs running on their terminals.

In addition to functioning as a gateway to projects, the *hub* is also a gateway to optimization servers. When a user submits a program to be executed on an optimization server, the *hub* checks if they have permission to use that server. If so, a link is created between the terminal through which the user submitted the program and the process executed on the optimization server, so that the output of the process is printed out on the terminal it is linked to.

3.4 Execution Layer

As we've mentioned in the previous section, the optimization server was implemented in C++ and we've used the *libwebsockets* library for communication with the *hub*. It is important that the optimization server has a good performance because, although the communication is one-to-one, the *hub* actually transmits the message of many. The optimization server, therefore, can receive many requests from different users and can run several programs in parallel, sending the outputs of each program in real time back to *hub*, being why it needs to have a good performance.

When the optimization server receives a request to run an optimization program, it first saves the files submitted locally, i.e., the AMPL model, data, and script files. Then the AMPL interpreter is called with these files as input in such a way that the output, instead of being written to the standard output device, is redirected to the server program itself.

We provide a security layer for the machine that is hosting the optimization server by running the process within a *sandbox* created via *nsjail*[5], an unofficial open source tool from *Google* developed primarily by Robert Swiecki. The *sandbox* is an environment that restricts the programs executed within it, limiting its access to files, its memory usage and its run time. The *sandbox* configuration will depend

[5]github.com/google/nsjail

on the limits of the user using the server. In section 2.4 we've showed how these limits are established, which is through the definition of user groups and the limits imposed on memory usage, run time, and other parameters.

It is noteworthy that, conceptually, the *NEOS server* is in the execution layer of the system, but the relationship between PIFOP and NEOS is that we only use this free service. We participate neither in its development nor its maintenance, and therefore the features of the optimization server developed by us are not available to NEOS users. For instance, it is not possible to see the memory usage rate of the machines in the NEOS server through our application.

3.5 System Workings Example

The figure 3.2 represents the flow of communication between the layers of the system when a user accesses a project, modify it and runs a model on an optimization server.

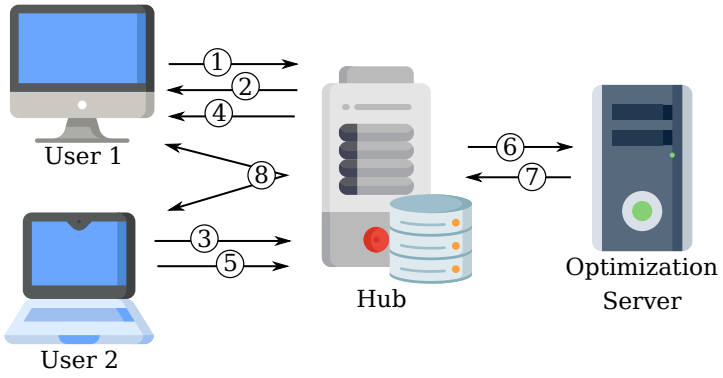

Figure 3.2: Example of how the system works.

First, user 1 requests access to a project to edit it (1). After adding user 1 to the edit session of the project, the *hub* responds with the files and other project data (2). Then user 2 modifies one of the files, and the modification is sent to the server (3) and propagated to the other collaborators of the project (4). Then user 2 decides to run the modified model, by submitting the files for execution in the selected optimization server (5). The *hub* creates a link between the terminal used by the user 2 and the process that will be executed on the optimization server

and sends the files to this server (6). Finally, the optimization server returns the execution output to *hub* (7), which in turn sends the output to the users connected to the project (8).

Chapter 4

Technical Details

The previous chapter is enough to provide the reader with an overall notion of the system's architecture. Here we are going to discuss in greater detail the working of two of the more important parts of the system, namely the group editor and the optimization servers.

4.1 Group Editor

The development of a real-time collaborative text editor can be very challenging depending on what you want to achieve in terms of user experience. In our case, we wanted the user experience to be exactly the same as if they were using an offline desktop editor, but with the exception that multiple users would be able to simultaneously edit the same file and see what other users were typing (or deleting, of course) in real-time. Despite the fact that this goal can be simply stated and understood, achieving it is rather complicated. You would think that because text editors are among the first types of software ever developed and so many knowledge about them have been accumulated ever since, adding a single feature to it, which is basically adding the ability to have one extra user, would not involve that much complexity. But that is not true.

The problem is that it is not enough to implement an algorithm and integrate it with an existing software: a whole system is needed, and not just one component. The developers of the CKEditor, a rich text editor featuring collaborative

editing, say that when they started implementing this feature they realized that "full support for collaborative editing for rich-text data cannot be added on top of existing projects. A proper architecture has to be designed and implemented from scratch, with real-time collaboration treated as a first-class citizen in the entire project" (Cofalik and Tomanek, 2018). Our experience has been the same. When we started the implementation of the real-time collaboration feature, so much of the old code has been modified or thrown away that little of it remained in the final result.

Collaborative Editing Challenges

There is extensive literature pointing out the challenges of making real-time collaborative editors. One of the simplest problems to be stated (but not so much to be solved) is this: suppose Andy and Bob are editing the same file, which is initially empty. Andy types the letter 'A', and sees 'A' on their screen. This operation of adding the letter 'A' at the beginning of the file is then sent to Bob. While this operation is not received at Bob's side, Bob types 'B', and sees 'B' on their screen. This operation is then sent to Andy. What will happen when Bob receives Andy's operation? An 'A' will be added to the beginning of the file, and Bob will see 'AB' on their screen. What will happen when Andy receives Bob's operation? Letter 'B' will be added to the beginning of the file, and Andy will see 'BA' on his screen. As a result, the file will have different contents at both sides. Notice how simple the scenario is: two peers, one empty file, one letter added by each peer. Imagine having multiple users performing all sorts of operations over a non-empty file.

The above illustration supposed a naive peer-to-peer system in which each peer 1) sends their operations to the peers immediately after performing them and 2) applies the received operations immediately after receiving them. The following examples will also suppose this naive system. At the moment, we don't care about solutions, only about the problems. It is easier to explain what a system does by what problems it is trying to solve. And the first problem any group editor will have to solve is how to guarantee that all the file copies, i.e., the copies of the file at each peer, will have the same content. A group editor needs to be capable of

Convergence.

Suppose another situation: Andy, Bob and Carl are editing the same file, which is also initially empty. Andy types ‘A’ and Bob types ‘B’ simultaneously. Carl receives ‘B’, then ‘A’ and then types ‘C’ at the end of the file. Latency issues make so that Andy receives ‘C’ before it receives ‘B’. What problem will arise from this situation? Well, when Carl typed ‘C’, he typed it after position 1 (counting from zero), because he typed it at the end of the file, which already contained ‘AB’. But when ‘C’ arrives at Andy’s side, their file contained only an ‘A’. How can anything be inserted after position 1 if there is only one character (at position zero) in the file?

This illustration shows that some operations only make sense considering the operations that were performed before them. It is important to observe in what order the operations are performed. If the insertion of ‘C’ happens after the insertion of ‘B’ and ‘A’, this order should be preserved. A group editor needs to be capable of **Precedence preservation**.

One last situation, and then we will have covered the major challenges we have in a group editor. Andy and Bob are viewing the same file, which initially contains ‘135’. They both want to complete the sequence to obtain ‘12345’, so Andy typed ‘2’ at position 1, or, between ‘1’ and ‘3’, and Bob typed ‘4’ at position 2, or, between ‘3’ and ’5’. After Bob receives Andy’s operation, his file will contain ‘12345’, which is what Bob and Andy intended. But when Andy receives Bob’s operation, his file will contain ‘12435’. Why? Because Bob’s operation consists in inserting ‘4’ at position 2, and the file at Andy’s side contained ‘1235’ when it received the ‘4’.

Aside from illustrating the divergence problem, because the file contents didn’t converge, the above situation shows another problem. From Bob’s perspective, he was not inserting a ‘4’ at position 2, meaning that that is not what he was thinking when he did what he did. He had an intention, which was to insert a ‘4’ between ‘3’ and ‘5’, and this intention was not captured at Andy’s side. A group editor needs to be capable of **Intention Preservation**.

The above mentioned situations give us an idea of what problems a group editor will have to deal with. For developers, these three challenges, and the three respective capabilities needed to solve them constitutes a set of goals that will

guide the development. This set of goals or model that the developer will try to implement constitutes the **Consistency model** proposed by Sun and Ellis (1998).

A cooperative editing system is said to be *consistent* if it always maintains the properties of convergence, causality-preservation and intention-preservation. Sun says: "In essence, the convergence property ensures the consistency of the final results at the end of a cooperative editing session; the causality-preservation property ensures the consistency of the execution orders of dependent operations during a cooperative editing session; and the intention-preservation property ensures that the effect of executing an operation at remote sites achieve the same effect as executing this operation at the local site at the time of its generation, and the execution effects of independent operations do not interfere with each other."

Now, what approach should a group editor take in order to achieve consistency? The first thing to be noted is that different systems with different architectures can achieve the same set of goals. For instance, both peer-to-peer as well as server-based group editors can be consistent. Multiple and varying solutions have been devised to prevent the above problems from happening, some which are actually correct, but many of them proved to be wrong after being published. It turns out that collaborative editing has too many edge cases, which makes it quite hard to prove the devised algorithms. Instead of talking briefly about each proposed solution, we will talk in detail about the solution which we have chosen and adapted for PIFOP.

Overcoming the Challenges with Operational Transformation

Our approach is not different from the Google Docs approach (Xu et al., 2014), which in turn is not *much* different from the solution proposed by Nichols et al. (1995). As the reader can see, the ideas we've implemented are not new, having its origin 25 years ago and being used in applications of great popularity. The central mechanism of the system is known as **Operational Transformation** and it is well described in the literature (see Ellis and Gibbs (1989b), Nichols et al. (1995), Sun and Ellis (1998) and Sun et al. (1998)). The few differences that arise between our approach and Google Docs are due to the fact that we deal with different kind

of documents. Google Docs is an application for editing rich text files, whereas our editor is for simple text only.

It must be said that the Operational Transformation literature is much richer than what is going to be presented in this work. The reader interested in the theoretical background of the technique can refer to the papers we've just cited. The present section will be mainly interesting for those seeking to implement the technique themselves and for those who wish to know how the theory comes into practice.

In PIFOP, a *file* consists of an array of lines, and a line consists of an array of characters. This is different from other group text editors because many of them represent a file as a single array of characters. Although we haven't made performance tests, there is an argument to be made that this representation is better for our context because it allows lines to be updated independently from each other. The computational cost of inserting and removing characters will be lower because rather than moving the memory of a section of the file, these operations will only have to move the memory of a section of a line.

When we exemplified some of the problems involved in group editing, we've presented one kind of edit operation that can be performed over files: the string insertion operation. Actually there are four kinds of operations that can be performed over a file. They can be formally defined as follows:

1. String Insertion SI(l, c, s): insert string s at line l before column c.
2. String Deletion SD(l, c, k): delete k characters at line l starting at column c.
3. Lines Insertion LI(l, L): insert list of lines L before line l.
4. Lines Deletion LD(l, k): delete k lines starting from line l.

It does not matter what edits a user performs over a file, their actions can always be broken down into one or more operations. For instance, suppose a file with 3 lines, as shown bellow. A user deletes the range (0,1) to (2,1), that is, they delete the characters in the closed interval starting at line 0 column 1 and ending at line 2 column 1. From the user perspective, they perform a single action: the

deletion of a range of characters. But internally, this single user action consists of a sequence of operations, as show in Figure 4.1.

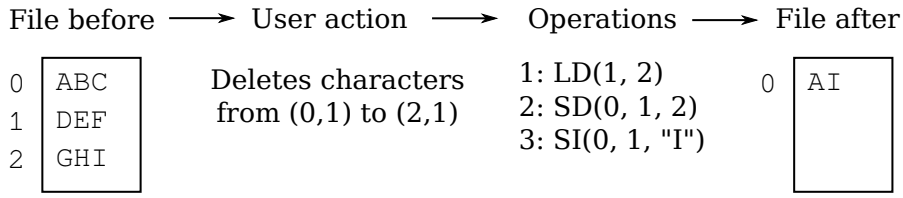

Figure 4.1: Example of a single user action being translated into multiple edit operations.

Other user actions may also be composed of multiple operations. Another instance where this will happen is on replacing a range of characters with some content being pasted. This action will be composed of deletion operations followed by insertion operations.

In our system, when a client executes an operation it is immediately applied in their local file and sent to the server. This operation will also enter the queue of the *non-acknowledged operations* (NAO queue, for short) maintained by the client. Before sending another operation to the server, the client needs to receive a server acknowledgment of the last sent operation. If the user performs other operations while the acknowledgment is not received, these will be applied locally but not sent to the server. They will also enter the NAO queue.

The server maintains a history of the operations it has applied, so each operation in the server can be identified by their position in the history. Each peer keeps the number of the last operation they have received from the server. A peer is in sync with the server if 1) the number of the last operation they have received is the same as the length of the server history and 2) the peer does not have queued operations.

When a peer sends an operation to the server, it also sends the number of the last operation they have received. With this information, the server knows at which point in the file history the user performed the operation. If the operation has been performed after all the server operations, it is ready to be appended to the history and broadcast to the peers. If not, the operation needs to be transformed before being appended to the history.

The purpose of transforming operations before applying them is to achieve intention preservation. Let us return to the intention violation example. Initially, the file contains '135' at both Andy's and Bob's side and also in the server. The file history is empty. Andy executes SI(0, 1, '2') and Bob executes SI(0, 2, '4'). Andy's operation arrives first at the server, which immediately appends it to the history and sends it to Bob. When Bob's operation arrives at the server, the server will check the number of the last operation Bob had received from the server when they performed the operation and will see that he had not received Andy's operation at the time. This means that both operations had been performed when the file was in its initial state. So Bob's operation needs to be transformed before being applied to compensate for the fact that 1 character has been added at position (0,1). This means that Bob's insertion needs to be "shifted to the right" by 1 in order to preserve the intention of Bob's operation. That is, instead of applying SI(0, 2, '4'), the server will apply SI(0, 3, '4') and send this transformed operation to Andy. Assuming Andy and Bob do nothing else while all of this is happening (we will get to what happens if that is not the case later), the file copies on the peers and on the server will converge, and all of the operation's intentions will be preserved.

Formally speaking, a transformation function $T(o_a,\ o_b)$ is a function that takes as input two operations and returns a list of operations O_a^t such that applying o_b then O_a^t achieves the same result as applying O_b^t then o_a, where O_b^t is the result of $T(o_b,\ o_a)$.

In order for the system to be able to handle every kind of operation concurrency, there needs to be one transformation function for every pair $(o_a,\ o_b)$ that may require o_a to be transformed. Some pairs of operations do not interfere with one another, so they can always be applied without transformation. For instance, line insertions will never need to be transformed against string insertions because no matter in which line or column a string is inserted, the parameters of the line insertion operation will remain the same. Appendix A contains every transformation functions that are needed. The transformation functions that take two string operations as input are the same presented by Sun et al. (1998). Since transforming an array of characters is not that different from transforming an array of lines, the transformation functions that take two line operations as input are slight

adaptations of Sun et al. (1998). The other transformation functions, that take one string operation and one line operation, simply set the string operation line parameter appropriately.

The reason why the transformation function returns a list of operations rather than a single one is to cover the case where deletions and insertions inside the deletion range occur simultaneously. Suppose a file containing 'ACD'. Andy performs SI(0, 1, 'B') and Bob performs SD(0, 0, 2) simultaneously. Andy's operation arrives first at the server, which appends the operation to the history and sends it to Bob. Upon receiving Bob's operation, the server will brake SD(0, 0, 2) into 2 operations in order to maintain Andy's insertion and Bob's deletion, i.e, SD(0, 0, 1) and SD(0, 2, 1). These operations will preserve Bob's operation original intention, which was to delete characters 'A' and 'C', and Andy's intention of inserting character 'B', resulting in a file containing 'BD'.

Now it is the time to take a closer look at what the client does with the operations it receives. The reader may have noticed that in the above situation, Andy's operation will have to be transformed at Bob's side somehow, otherwise a 'B' will be inserted after 'D', which is incorrect.

The operations are broadcasted from the server to the clients along with the *id* of the author of the operation, which is a unique identifier of the client. Even if the server transforms an operation before broadcasting it, the author of this transformed operation will be the same as of the original operation. When a client receives an operation, they will first check to see who is the author. If they are not the author, the operation has to be applied; otherwise, the message is an acknowledgment that the server has received and propagated the last operation the client has sent, so the client can send another one. In the first case, in which the operation has to be applied, the client checks if the client NAO queue is empty. If it is, this means the client is in sync with the server, so the operation is ready to be applied. Things get more complicated if the NAO queue is not empty, because that means that the client has applied operations locally that the server does not know about yet.

To understand what needs to be done in that case, it is useful to represent the file changes happening in the client and server as a movement through a state space (proposed by Ellis and Gibbs (1989b)). To simplify the examples and diagrams we

are going to provide, we have ignored the cases in which *T(o_a, o_b)* returns more than one operation. Instead, for didactic purposes, we are assuming it returns a single operation.

In the following diagrams, each vertex represents a file state, i.e., its contents in a given moment. Each operation performed by the client moves the file state to the bottom left, and each operation performed by the server moves the file state to the bottom right. The operations are represented as solid lines when they are performed in the same place they are originated from. If the operation originated in the client but is performed in the server, it is represented by a dashed line. Likewise, if the operation originated in the server but it is performed in the client, it is also represented by a dashed line. Figure 4.2 shows a scenario in which the client and the server both altered the initial state of the file. The client and the server applied the operations a and b, respectively, being therefore out of sync.

Figure 4.2: Operational transformation guarantees the file copies will *eventually* converge.

When operation b is received by the client, it needs to be transformed against operation a before being applied. So the client, instead of applying b, will apply b', where b' is *T(b, a)*. Likewise, the server will apply a' instead of applying a, where a' is *T(a, b)*. By definition, applying a then b' will get the file to the same state as if it were applied b then a'. Hence, after performing the transformed operations on both ends, client and server will be synchronized again.

A gray line represents an operation in the NAO queue. Operation a in figure 4.2A was initially in the NAO queue. The fact that the client has received the acknowledgment of the operation, removing a from the NAO queue, is represented by the line becoming black in figure 4.2B. The next diagrams may also contain bold dotted lines, which represent operations used only temporarily in intermediate

steps in a chain-transformation process, and thin dotted lines, which are only there to represent operations that are no longer relevant because they have been transformed and moved in the diagram.

Now let us consider an example in which client and server diverge by more than one operation, as shown in figure 4.3A. When the client receives operation b, it has two operations that the server does not know about: a and c. So in order to preserve the effects of both a and c, b needs to be transformed against a and this transformed operation needs to be transformed against c. The intermediate result *T(b,a)* is, of course not applied, because what we really want is to perform an operation that is going to preserve the intentions of all three operations: a, b and c. That is, the only applied operation is b', which is *T(T(b,a), c)*. In sum, the first thing the client needs to do when receiving a foreign operation is to *chain-transform* it against every operation in the NAO queue, then apply the transformed operation, as indicated in figure 4.3B.

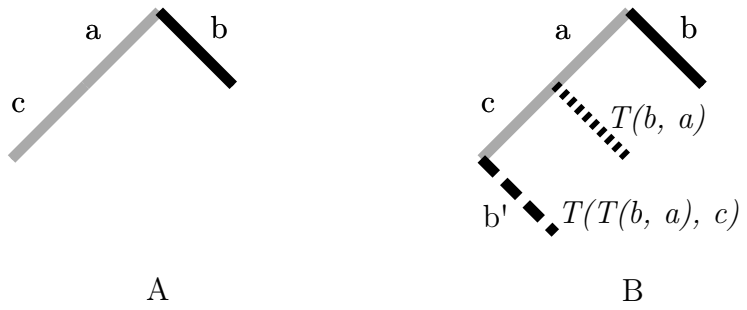

Figure 4.3: *Chain-transformation* of b against the NAO queue $[a, c]$

There is a second thing the client needs to do which is to update the NAO queue. All the operations in the queue will eventually be sent to the server, one by one. But they cannot be sent in their original form if the user has received and applied an operation from the server, because their original form doesn't consider the effects of the operations received by the server. Instead, every time the client receives a foreign operation, the operations in the NAO queue need to be updated to consider the effects of the received operation. In the diagram, the queue is "shifted" to the bottom right (see figure 4.4). This guarantees that the next received operation will be transformed against a queue that starts at the same

state of the operation that has been applied in the server.

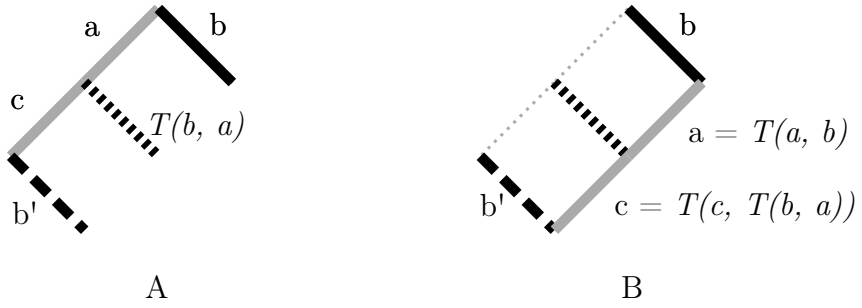

Figure 4.4: Transformation of the NAO queue [a, c] against b

One more example will now be given to show that the system can maintain consistency no matter how complex the scenario is. Figures 4.5 through 4.8 show a scenario in which the client keeps editing the file while it waits for the acknowledgment of the first operation and keeps receiving operations broadcast by the server. The transformations involved are indicated in the captions.

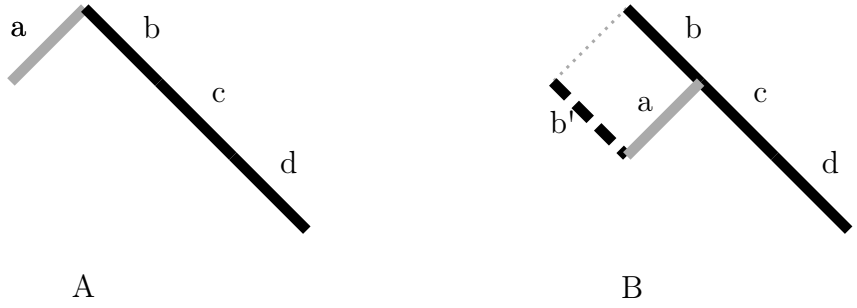

Figure 4.5: A) Client has one NAO operation and has not yet received three of the server operations. B) Upon receiving b, two things happen: 1) b'= *T(b,a)* is applied and the NAO queue is transformed against b, leading to a new a = *T(a,b)*

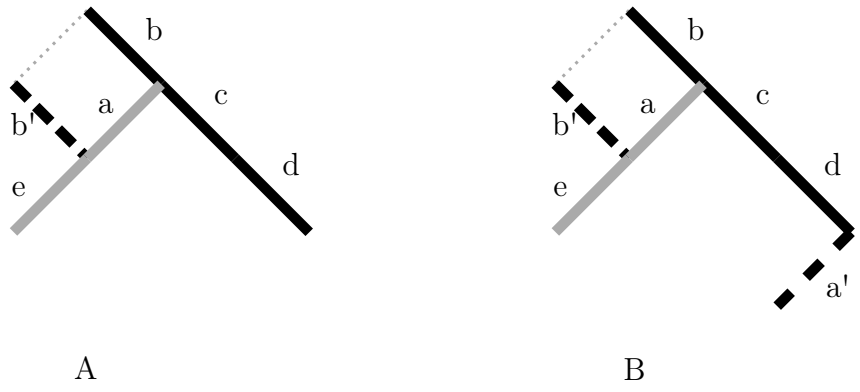

Figure 4.6: A) Client performs operation e locally and adds it to the NAO queue. B) Server receives a and applies a', which is *T(T(T(a,b),c),d)*

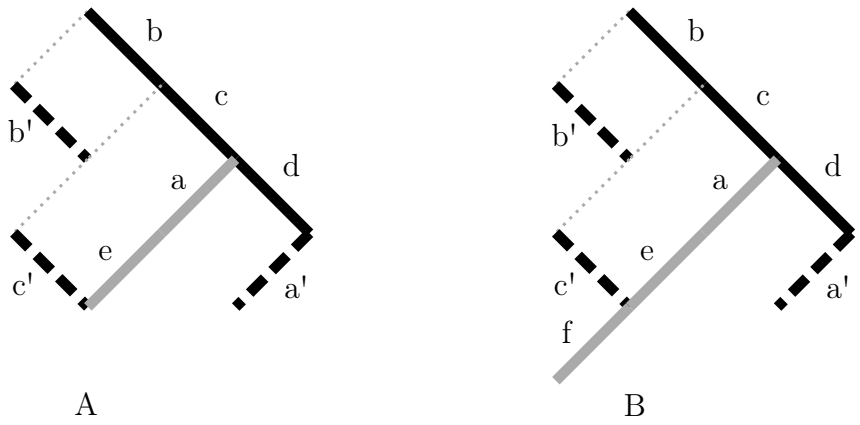

Figure 4.7: A) Client receives c and performs c'. Then transforms the NAO queue [*a, e*] against c, resulting in a new queue [*T(a,c), T(e, T(c,a))*]. B) Client performs f and adds it to the NAO queue

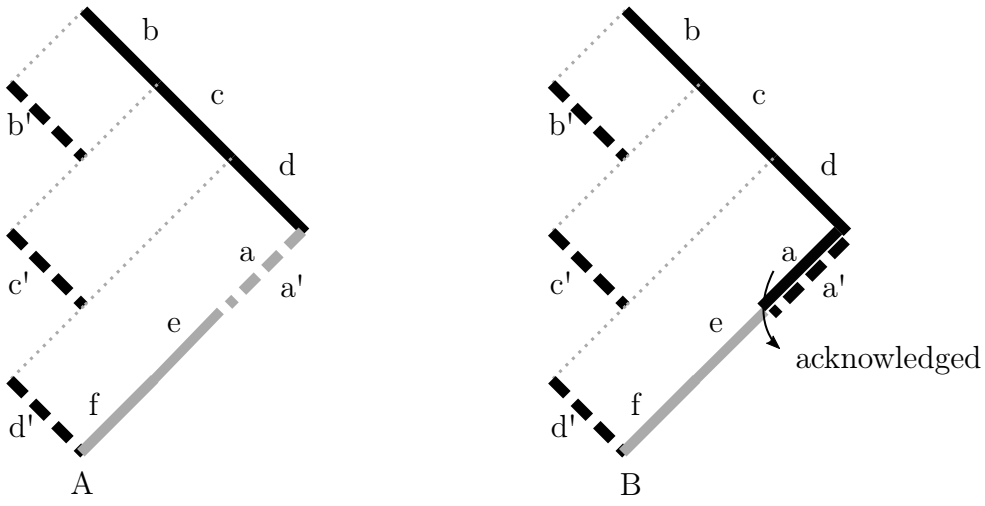

Figure 4.8: A) Client receives d and performs d'. Then transforms the NAO queue [*a, e, f*] against *d*, resulting in a new queue [*T(a,d), T(e, T(d,a)), T(f, T(T(d,a),e))*]. B) Client receives the acknowledgement of operation *a*. At this moment, *a* is removed from the queue and the next operation in the queue (e) can be sent to the server.

To close this section, we note that there is a potential optimization that can improve the performance of the whole system. At the instant shown in figure 4.8B, the client is about to send operation *e*. But if *e* could be combined with *f*, both operations could be sent in one go, meaning the number of messages going between client and server would be diminished, as would the delay between the execution of an operation and the propagation of it to the other users. This and others improvements will eventually be made in our system.

Graphical Interface Performance Related Challenges

Separated from the challenges related to the support of collaborative editing, there are difficulties related to the support of large files. Some model and data files can be very large, containing hundreds of thousands of lines. The fact that the front-end of our application is browser-based demands that we pay special attention to the browser rendering process. We will briefly comment on the issues here involved before we move on from the group editor to our next topic.

In the previous subsection, we've mentioned that a file in PIFOP consists of an array of lines, and that this representation is arguably better than a single array of characters because the insertion and removal of individual characters will only require the manipulation of a section of the file's memory. There is a secondary reason why this representation is better than the alternative just mentioned: it allows us to easily exclude from the browser rendering process all the lines that are not currently visible in the editor.

The browser maintains a tree structure that contains all the visible (and some invisible) elements in the web page: the DOM (Document Object Model) tree. This structure is used for positioning and sizing the elements, a process that is called Layout (or Reflow). Not every modification in the DOM elements will require the browser to recalculate the position and size of the elements, but some of them, such as updating the content of an input box, will require not only the recalculation of its own position and dimensions but also those of all the elements affected by that update, which will include at least the ascending and descending elements, if not the siblings.

The painting process is critical to browser renddering. If positions and dimensions are handled in the layout phase, colors are handled in the paint phase. Whenever a color or text is updated, a repaint is needed. This is why using a single *textarea* HTML document would be slow at dealing with large files. Adding characters to a large file would require the repaint of the whole *textarea* content, even of the part outside the scrolling region.

Our approach to avoid the overloading of the layout and paint processes has been to maintain a fixed number of elements to show the visible lines in the group editor. Whenever the scrolling region changes, the editor content is updated, but no new elements are added to the DOM and no old elements are removed from it. That way, even if an update requires to recalculate the layout parameters of the whole page or to repainte the whole page, the page itself will only contain elements that are actually visible. So the costs of rendering a file with a hundred of lines and of rendering a file with one hundred of thousand lines will be virtually the same. The same approach can be seen in other browser-based code editors, e.g., Ace, Monaco Editor and CodeMirror.

4.2 Optimization Servers

We now move our attention from the group editor to the optimization servers. What has been broadly described in the previous chapter about the optimization servers will now be explained in greater detail. This explanation will be focused on how can we offer a self-hosted optimization server option that can 1) be hosted by machines behind routers, 2) be accessible only by the allowed users and 3) execute and manage multiple processes in parallel.

Technically speaking, optimization servers are in fact clients of the *hub*, because they do not listen to a port waiting for client connections. But since from the end user perspective they are servers, we thought it made more sense to call them servers rather than clients.

In practice, what that means is that the optimization server is the one that initiates the connection with the *hub* and not the other way around. This design decision allows optimization servers to be run even from behind routers and still be accessed from outside. This would not be possible if the *hub* were the one to initiate the connection with the optimization servers because they would need to be reachable from the outside, either by virtue of it having a public ip address or by port forwarding in the router. This approach has it's advantages and disadvantages.

The main advantage is that any Linux machine with access to the internet can host an optimization server without the need of any additional setup. The main disadvantage is that the only way in which users can access the optimization server is indirectly, through the *hub*, even to perform administrative or monitoring tasks which the *hub* does not care about. But since most of what the optimization servers do would require it to communicate with the *hub* anyway, for instance, propagating the terminal output, the impossibility of direct communication between user and optimization server is not a significant downside.

The same design decision of having no direct communication between users and optimization servers allow us to better control the access to them, providing an extra security layer. As explained in Chapter 3, the optimization server configuration file contains a list of the users that are allowed to use that server. Upon the launch of the server, this list is sent to the *hub*, which stores it for future reference.

When a user accesses PIFOP, the *hub* checks which optimization servers the user is allowed to use and sends their names and identifiers to the user so that they can select the server they want to use for the execution of any given AMPL program.

Despite of what the user may think, the terminal they see in the IDE is not a regular Linux terminal through which they can send `bash` commands. It is not even an AMPL interpreter command line session. It is only a way the user interacts with the selected optimization server using a limited set of possible commands. This means that there is no real link between the users or terminals and the optimization server until a command is entered, and as soon as the command is finished, the link is undone, since it is no longer needed.

To be more specific, this link between a terminal and the optimization server consists in a pair of identifiers: the project terminal identifier and the request identifier. A request is any command a user can enter in the terminal, so a query of the machine usage through the `show-usage` command and the submission of a AMPL program for execution through the `submit` command are both types of requests the optimization server can receive. When the user enters a command in the terminal, a request is created in the *hub* and sent to the optimization server. Each request has a unique identifier, so when the optimization server sends back to the *hub* the results of a request, the *hub* knows which terminal that request has come from and thus can edit that terminal and propagate the changes to the connected users.

For initiating and managing processes requested by the users, the optimization server program leverages different Linux mechanisms. Let us see what the server does when it receives an optimization program submission. The server stores the request id that it has received from the *hub* and saves the submitted files locally, so that they can be used as input of the AMPL interpreter. Then a pipe is created via `pipe` and a new process is spawned via `fork` with its output being redirected to the pipe.

As mentioned in section 3.4, we use the *nsjail* tool for creating a sandbox inside of which the AMPL interpreter is called. The sandbox configuration prevents the interpreter from accessing files in the machine it should not be able to and also limits the time and memory used by it, which depends on the permissions of the user that is using the server.

The interpreter is called in the new process using the command line submited by the user, e.g., `ampl model.mod data.dat script.run`. The parent process, i.e. the optimization server program, adds that process to the list of processes it has spawned.

The optimization server runs in a loop. In every iteration the server goes through all its processes to see if there is new data to send to the *hub*. This is done in fixed intervals to prevent the connection with the *hub* to be clogged up with too many packets in the write queue. If there is new data, it enters the write queue to be sent later. If there is no data and the process is finished, the child process is disposed and a message is sent to the *hub* saying the process is finished.

As the reader can see, the optimization server is mostly a spawner and manager of processes, and is not closely tied to AMPL. This gives us a lot of flexibility for future feature additions in the optimization server, for instance supporting other interpreters and tools. We mention this so that readers interested in developing a similar system notice that the optimization server can, in fact, be used for remote execution of any program.

Part II

Research Questions

Chapter 5

A comparative study on WIDEs

Our goal in this chapter is to investigate the workings of other WIDEs in order to provide answers to two of our research questions. First, **(RQ1) How do WIDE providers overcome the main challenges they have to face in order to effectively meet user demands?** We are not the first ones to think about the challenges involved in providing IDE features on the web, and we believe that there is much to be learned from those who preceded us in doing so. Not only this information should be useful for us as we continue to improve our tool but also to others that are involved in similar endeavors. Second, **(RQ2) How do providers of WIDEs for optimization overcome the main challenges they have to face in order to effectively meet user demands?** Mathematical programmers have their own particular demands that need to be addressed by WIDE providers, being of our interest to know how this can be done.

The way in which we are going to address these research questions is by comparing WIDEs in different aspects, looking at the features provided by them and the mechanisms and technologies involved in providing them. The answers to our questions will emerge from this comparative analysis.

Our procedure will be the following. We'll begin by looking at other studies on WIDEs in Section 5.1. Then, in Section 5.2, we'll introduce our definition of the term WIDE and also the WIDEs that we've selected for our comparative analysis, which will follow in Section 5.3. In Sections 5.4 and 5.5, we'll formulate the answers to questions RQ1 and RQ2, respectively. We'll conclude in Section

5.6 with a brief discussion of the state of WIDEs in the past, in the present and what we believe will be the trends for the future.

5.1 Related works

About nine years ago, Iqbal et al. (2012) presented a technical report comparing nine WIDEs available at the time: *CodeRun*, *Cloud9*, *eXo Cloud*, *Bespin*, *Kodingen*, *Bungee Connect*, *Codeanywhere*, *ECCO* and *WonderFL*. The WIDEs were compared in terms of programming language support, integration with source control systems, project collaboration features and deployment options. Much has changed in the WIDEs' ecosystem over the years. Seven of the nine WIDEs in that study are no longer available, either due to the project termination or due to merges with other projects (such as *Bespin*, which has merged with the *Ace* code editor project[1]). *Cloud9* and *Codeanywhere* are the only WIDEs from that study that are still available today, but nowadays they do much more than what they did nine years ago. While the authors comment that "these web-based programming environments can (*nearly*) replace our desktop IDEs and code editors" (emphasis added), today we can say that, depending on the use case, they are fully capable of replacing desktop IDEs. There are, of course, trade-offs to be made when opting for a WIDE rather than a desktop IDE, but yet, feature-wise, WIDEs are not much behind IDEs.

In the same year, Kats et al. (2012) proposed a research agenda for software development environments on the web. The motivation was the recognition that, although software development environments on the web was an appealing vision, it was far from reality. Moving IDEs to the web at the time required the overcoming of many challenges, both technical and social, which were outlined by the authors as topics for further research. Kats et al. (2012) posed many questions that should be answered by WIDE developers if the migration from the desktop is to be successful, such as: How does the edit-compile-run cycle work on the web? What happens when the developer is offline? How does the WIDE integrate with existing tools? How can we specialize the IDE for teaching? When all develop-

[1] https://blog.mozilla.org/labs/2011/01/mozilla-skywriter-has-been-merged-into-ace

ers are online, how does team collaboration change? Who controls the installation and maintenance of the web IDE? Since then, different WIDE providers have given their own answers to these questions, with varying levels of success. Here one of our goals is to present to the reader the answers that have been given and also to propose some answers of our own.

Four years later, Fylaktopoulos et al. (2016) presented an overview of the state of the art technologies for software development in cloud environment. It included a comparison between seven programming environments in the cloud: *Compilr*, *jsFiddle*, *Cloud9*, *Codenvy*, *Eclipse Orion*, *Koding* and *Codeanywhere*. Among other criteria, they were compared in terms of their features, integration with version control services and the possibility of running the IDE on-premises. Some of those programming environments are no longer available. To the best of our knowledge, *Compilr* and *Koding* have been terminated[2], and *Codenvy* has been integrated into OpenShift[3], a Red Hat's online platform that provides a variety of tools for building, deploying and monitoring software in the cloud.

In a paper in which a new WIDE called *CodeCircle* is introduced, Fiala et al. (2016) presented a brief comparison between some of the existing online coding interfaces at the time, such as *Cloud9*, *jsFiddle* and *CodePen*. Among other criteria, the tools were compared in terms of purpose, features and sharing options. As the comparison was not the main part of the work, not much was said about their differences and similarities.

More recently, during the COVID-19 pandemic, Kusumaningtyas et al. (2020) made a comparative analysis between eleven WIDEs that support the Python programming language. The authors were concerned with the fact that, although WIDEs can be very effectively integrated in virtual classes, many students throughout the world are limited on the internet quotas they can afford, so they compared the WIDEs based on their internet data usage and other criteria, such as the presence of advertisements and the support of popular Python libraries. They concluded by recommending *Replit*[4], as it offers the least internet data usage without

[2]We've found no official note confirming that Koding has been terminated, but it appears to be so, as the project's repository is now archived and their blog and social media pages have been inactive for years. See https://github.com/koding/koding/issues/11518

[3]https://openshift.com

[4]https://replit.com

losing in user friendliness and library support.

The aforementioned works have influenced us in our choice of WIDEs to compare and in the adopted comparison criteria used, as we'll see in the following sections.

5.2 Overview of the selected WIDEs

There are many WIDEs that we could have selected for this comparative study. What was our criteria to choose one WIDE rather than another? Before answering that, it is important to define what we consider to be a WIDE in this study.

A WIDE is a web application that provides various facilities to help them in the production of an artifact from a source code. The type of artifact a WIDE is designed for will determine the kinds of facilities they provide and the source code they support. If the WIDE is designed for building Python programs, it may include a debugger, a compiler and an editor with syntax highlighting for that programming language as tools to facilitate the developing process. However, our definition is broad enough as to not limit ourselves to general programming languages. This allows us to talk about WIDEs for optimization programs, WIDEs for PDF document creation, WIDEs for music/sound creation, and so on.

Although WIDEs may differ in the artifacts they enable the users to produce, we consider that there is a common set of basic features that should be present in every WIDE. First, since the artifact is produced from a source code, it is expected that the WIDE will provide a source code editor, ideally with syntax highlighting. Second, the user should be able to interact with the artifact being produced. In the case of a programming WIDE, the user should be able to run the program and see its output. In the case of a PDF creation WIDE, the user should be able to visualize the document. Third, the user's files, as well as the IDE itself, should be stored in the cloud and provided on request. That is to say a WIDE is not just an IDE running on a browser (which would make them no different from a desktop IDE, except for the fact that they would not need to be installed in the system), it is a web application that enables the user to continuously develop an artifact from different devices without the need to re-upload the files every time they want to continue their work.

On imposing the need of these basic characteristics we wanted to be loose enough to include in our comparison web applications designed to facilitate the building of different kinds of artifacts (especially mathematical optimization programs) but strict enough to exclude the ever increasing number of web applications designed for quick testing of simple ideas. In this last type of web applications we include *jsFiddle*[5], *Compiler Explorer*[6], *Ideone*[7], *C++ Shell*[8], *Coding Ground*[9], *The Go Playground*[10], *JDoodle*[11], *Linear Optimization Solver*[12], *Try AMPL*[13] and several others. These tools may be effectively used for prototyping an idea or sharing snippets of code with other people. However they are not designed for continuous development of an artifact (and most of them don't identify themselves as WIDEs anyway). The continuous development is made impractical either due to the lack of persistent cloud storage of the project or due to the inability to easily work with multiple files, which quickly makes a medium size project unmanageable.

Even with this WIDE definition in view, there are still many candidates to choose from. We've prioritized WIDEs for mathematical optimization, our the main interest in this work. Of this kind, only two were found: *Watson Studio Decision Optimization Model Builder* and *RASON IDE*. Then we've looked for well established WIDEs for general programming, from which we've chosen *Cloud9*, *Codeanywhere*, *Replit* and *Google Colab*[14]. Finally, we wanted to include WIDEs designed for purposes other than general and mathematical programming in order to enrich our discussion. So we've added *Overleaf* and *CSound IDE* to our list. Below, we briefly describe each of the selected WIDEs.

Watson Studio[15] (**WS**) is a web platform that offers a great variety of tools and technologies for data scientists and developers, among which there is the *Decision*

[5] https://jsfiddle.net
[6] https://godbolt.org
[7] https://ideone.com
[8] https://cpp.sh
[9] https://tutorialspoint.com/codingground.htm
[10] https://play.golang.org
[11] https://jdoodle.com
[12] https://online-optimizer.appspot.com
[13] https://ampl.com/try-ampl/try-ampl-online
[14] We wanted to include *Visual Studio Codespaces* in our comparative study, but the WIDE is currently in closed beta testing, and we haven't had our request to participate approved.
[15] https://dataplatform.cloud.ibm.com/docs/content/DO/DODS_Introduction/buildingmodels.html

Optimization technology from IBM. There are different ways in which users can work with Decision Optimization at Watson Studio, but we are going to focus on the *Model Builder* tool. This tool allows users to develop and solve optimization models in OPL (Optimization Programming Language), an algebraic modeling language tailored for the CPLEX solving engines[16].

RASON IDE[17] (**RS**) allows users to develop optimization models in the RASON language (Restful Analytic Solver Object Notation). The language has a JSON-like specification, although it is not strictly a JSON[18] subset. It was created by Frontline Systems to work well with the RASON REST Server and other REST API optimization servers that users of the language may want to develop. *RASON IDE* is basically a web front-end for using the RASON REST Server.

Cloud9[19] (**C9**), *Codeanywhere*[20] (**CA**) and *Replit*[21] (**RP**) are similar and competing WIDEs for general purpose programming. They all support most popular programming languages, such as Python, C/C++ and Java, with a few important differences in their provided features and in their underlying systems.

Google Colab[22] (**GC**) is a *Jupyter Notebook* environment hosting service. A *Jupyter Notebook* is a type of document that integrates source code, rich text and computation output, allowing users to produce a much richer source code documentation and explanation than it is possible with plain source code comments. With Google Colab, users can work in *Jupyter Notebooks* from the browser and share their documents with colleagues.

Overleaf[23] (**OL**) is a WIDE for LaTeX projects. LaTeX is a declarative programming language used to create stylized PDF documents. Overleaf provides the interface for writing the source code and the computer resources for compiling the project.

Csound IDE[24] (**CS**) is a WIDE for Csound projects (Yi et al., 2019). Csound

[16] https://www.ibm.com/analytics/cplex-optimizer
[17] https://rason.com
[18] https://www.json.org
[19] https://aws.amazon.com/cloud9
[20] https://codeanywhere.com
[21] https://replit.com
[22] https://colab.research.google.com
[23] https://overleaf.com
[24] https://ide.csound.com

is a declarative programming language for audio synthesis, sound processing, composition and sound design. With the *Csound IDE*, users can develop, compile and hear the output using the browser.

By examining the characteristics of this diverse selection of WIDEs we are presented with different solutions to similar problems, which will make our discussion richer, as we will not be just looking at different implementations of the same system, but different solutions altogether. Throughout the text, we will be referring to each WIDE with their two letter abbreviation, as presented above.

5.3 Comparative Analysis

We've compared the WIDEs using nine different dimensions. The dimensions of comparison have emerged mostly from our reading of the discussions presented at the related works, with the addition of a few that were missing and which we wanted to cover, such as "provisioned computer resources" and "long running processes". These are the nine dimensions of comparison:

1. **GUI characteristics**: Which code editor does it use? Which terminal front-end library does it use? How are user files organized and presented? What is the layout of the application? Is the WIDE mobile friendly?

2. **Coding and debugging facilities**: What kind of debugging features are available? Is there real-time static code analysis? Is there integration with third party debuggers?

3. **Compilation/execution environment**: Where is the code compiled and executed? What kind of process isolation mechanisms are in play? Which infrastructure service is used (Google Cloud, Microsoft Azure, etc)?

4. **Provisioned computer resources**: What are the specifications of the provisioned virtual machines? What are the file or project size limits? Are there domain specific limitations, such as maximum number of variables in the WIDEs for optimization?

5. **Persistent storage**: How is data stored? Where does it lives? Is there integration with cloud storage services (Dropbox, Google Drive, etc)?

6. **Long running processes**: How does the WIDE handle long running processes? What happens when the user leaves the WIDE while a process is running? Does it stays running? For how long?

7. **Group work**: Does it have features to facilitate collaborative work? Can multiple users work on the same project at the same time? What happens when multiple users are editing the same file at the same time? Is there a built-in chat? Can users make comments on files?

8. **Version control**: Is there some kind of built-in version control system? Can users see the history of a file and go back to a previous version? Is there integration with cloud version control services (Github, Gitlab, etc)?

9. **Projects discoverability**: Are user projects public? Are there built-in mechanisms to find public projects? Can users keep their projects private if they want?

The information needed to answer these questions has been collected basically from three types of sources. First, there is the official documentation and manuals of each WIDE. Second, there are blog and social media posts of the WIDE provider. Third, there is what can be gathered simply by using the application, with occasional inspection of the front-end client using the developer tools of Google Chrome to get some information about the application, such as libraries and infrastructure services used. For **CS** there is an extra source, the paper of Yi et al. (2019), which describes the design and implementation of the WIDE.

1. GUI characteristics

We begin our analysis from the graphical user interface of the applications. There are several questions that can be asked regarding the user interface of each WIDE.

First, is the code editor component custom or does the WIDE integrate a third party code editor? Most of the WIDEs integrate with well known open source code editors. **WS**, **RS**, **C9** and **OL** all integrate the *Ace* code editor, which

is maintained by **C9** and Mozilla[25]. **OL**, however, is considering migrating to *CodeMirror*[26], which development they are sponsoring[27] and which is the editor component used at **CS**. **RP** uses *Monaco*[28], a code editor maintained by Microsoft and used in *Visual Studio Code*[29] (or *VS Code*, a free Microsoft IDE). At the time of this writing, **CA** is transitioning to a new interface built upon *Eclipse Theia*[30]. *Theia* is an open source extensible framework for building cloud and desktop IDEs. The framework also uses *Monaco* as the code editor component. **GC** is the only WIDE that has its own custom code editor.

Second, which terminal front-end does the WIDE integrates, if any? **C9** uses *Ace* not only for code editing but also as a terminal front-end. **CA** and **RP** both use *xterm.js*[31], a terminal front-end that is also used at *VS Code*. These are the only three WIDEs that have terminals which allow both input and output. **WS**, **RS**, **OL** and **CS** only have read-only areas where the output of the execution/-compilation is displayed. As for **GC**, the output is displayed just below the source code cells of the document, which is the way that output is displayed in *Jupyter Notebooks*.

Third, what is the layout of the application two main components (code editor and output displayer)? A typical WIDE layout puts the code editor on top of the output displayer. That's the default layout of **RS**, **CA**, **C9** and **CS**. Alternatively, some WIDEs go for a left-right layout, where the editor is on the left and the output on the right, which is the default layout of **WS**, **RP** and **OL**. **GC**, as already mentioned, displays the source code and the output on the same file, the output being just below a source code cell. With exception of **WS**, **CA** and **C9**, the layout of the applications cannot be changed by the user.

Fourth, how are user files organized and presented to the them? The most common way of organizing the user files is by using a typical file system structure of directories, which is presented to the user in a file explorer consisting of a tree of directories and documents. This is the way files are organized and presented at

[25] https://ace.c9.io
[26] https://codemirror.net
[27] https://www.overleaf.com/blog/our-tips-for-running-a-remote-hackathon
[28] https://microsoft.github.io/monaco-editor
[29] https://code.visualstudio.com
[30] https://theia-ide.org
[31] https://xtermjs.org

C9, **CA**, **RP**, **OL** and **CS**. **GC** also works with a file system structure. Jupyter Notebook is designed to be completely contained in a single file, but the main file at **GC** can still read from other files in the system. **WS** and **RS** have no file system structure. Users can have multiple files in a project, but they are presented to the user as a flat list.

Fifth, is the WIDE mobile friendly? We've tested each WIDE in a 720 by 1520 pixels smartphone to see how they behave. Overlapping or hidden interface components and excessively small buttons indicates that the WIDE was not designed for mobile usage. **RS** is virtually unusable on a phone. Most of the other WIDEs have partial mobile support, meaning they *can* be used on phones but it is clear that they were not designed for that. For instance, some WIDEs basically use the same interface used in desktop browsers scaled down to fit on mobile screens, which results in a crowded interface with small widgets and buttons. That's the case of **WS**, **C9**, **CA** and **OL**. As for **RP**, **GC** and **CS** they all have interfaces designed specifically for smartphones, and we've found no difficulty accessing the main WIDE functionalities through them.

2. Coding and debugging facilities

How do the WIDEs facilitate the coding and debugging activities? Here we describe four types of programming aids provided by one or more of the WIDEs.

First, there is the most basic coding assistance a programmer can ask for: syntax highlighting. All of the WIDEs provide syntax highlighting for the languages they support.

Second, there is the live code check, which is a code analysis that occurs as the user is typing, before the code is compiled or executed. This allows users to identify syntactic and semantic errors faster. With varying levels of error coverage, this feature can be found in **RS**, **CA**, **C9**, **RP** and **OL**.

Third, there is built-in integration with the language documentation, which allows users to quickly check what is the correct usage of a language construct, e.g., what parameters a standard library function takes and what does it returns. This integration can be found in **C9** and **CA** in the form of pop-ups that show up when the user enters the name of a known language construct. It is also present in

CS, which allows users to search for a language construct from within the WIDE and returns information about that construct, including playable code examples.

Fourth, there is built-in integration with third party debuggers, e.g. integration with *gdb* for C++ and with *pdb* for Python, which is more than just allowing users to use these debuggers. Built-in integration means the WIDE provides an interface for interactively using these tools (setting breakpoints, stepping over statements, watching variables, and so on). This feature can be found in **C9** and **CA**. The WIDEs **RP** and **GC** do allow users to run third party debuggers but provide no interactive interface for using them.

It should be noted that for the three WIDEs aimed at multiple general programming languages (**C9**, **CA** and **RP**) the level of support for each language may not be the same, which means that some of the features mentioned in this section may not be available for all languages, but are available for *at least one* of the supported languages.

3. Compilation/execution environment

Where is the user's code compiled and/or executed? With exception of **CS**, the user's code is compiled/executed via a remote code execution (RCE) solution within the WIDEs we're analyzing. **C9**, **CA** and **RP** provide the user with a private virtual machine (VM) to which they have *sudo* access. Users can then download and install packages, execute build scripts, run programs and do virtually anything within the environment they have. **GC** works in a similar way, but terminal access to the VM is only given to paying users. **WS**, **RS** and **OL** run the user's code on their internal servers, without exposing the execution environment to the user. All the users get from these three WIDEs is the output of the compilation/execution.

The infrastructural details of each RCE solution vary. In **C9**, the infrastructure service behind the VMs is the Amazon Web Services (AWS)[32]. The WIDE allows users to choose between various operating systems, including Linux, MacOS and Windows Server. The VMs provided by **CA** are *OpenVZ*[33] containers running

[32] https://aws.amazon.com

[33] *OpenVZ* is a virtualization system that allows multiple Linux instances to run in the same physical server. https://openvz.org

either Ubuntu or Cent OS. **RP** utilizes Google Cloud[34] preemptible VMs to provide the RCE environments for the users. "Preemptible" means that the VMs can be stopped at any moment. IBM Cloud[35] is the infrastructure provider for the **WS**, whereas Microsoft Azure[36] is the provider for **RS**. Unfortunately, we don't know much about the infrastructure of **OL**[37].

Do these WIDEs offer a way to run user code in machines other than those provided by the WIDE? Some of them do. In **C9**, **CA** and **GC** users can bind the WIDE to any machine they want through SSH/SFTP[38]. For these cases, the connected machine becomes the environment in which user code is executed. At **CA**, the WIDE server still acts as a middle-man between the user front-end and the execution environment[39], which is probably the case for **C9** and **GC** as well, but we have no confirmation on that.

RS and **OL** offer a different solution for users that want to store/run their code in their own private servers. Both WIDEs provide on-premises solutions[40,41] (unavailable for free users), allowing companies to run the WIDE back-end (**RS**/**OL** servers) internally in their own private clouds. This internal server, then, is the environment in which user code is compiled/executed.

In **GC** and **CS**, user's code can be executed locally (in the case of **CS**, that's the only way user's code can be executed). **GC** can be connected to a Python Notebook server running locally, and **CS** provides the user with a *WebAssembly*[42] port of the Csound engine (Yi et al., 2019), which allows their code to be executed in the browser.

[34] https://cloud.google.com

[35] https://www.ibm.com/cloud

[36] https://azure.microsoft.com

[37] According to a 2017 Overleaf blog post, each project is compiled within its own virtual machine. This information may or may not be outdated. https://www.overleaf.com/blog/507-overleaf-for-book-production

[38] In Google Colab, the process of connecting to user machines is less straight forward, but it is still possible. https://research.google.com/colaboratory/local-runtimes.html

[39] https://docs.codeanywhere.com/#sftp-ssh

[40] https://www.solver.com/Introducing-rason-dedicated-server

[41] https://www.overleaf.com/for/enterprises/when-to-use-overleaf-on-premises-vs-in-the-cloud

[42] https://webassembly.org

4. Provisioned computer resources

How much computing power and storage is made available for users? With exception of **CS**, which doesn't have a paid option, the amount of computer resources that can be used depends on the user subscription plan or demand. Most of the WIDEs have a free tier option (at least for a limited period of time). The following numbers are what we could gather from what is publicly disclosed by each WIDE provider, so some information will be unavoidably missing.

At **WS**, free users are granted execution environments with up to 4 vCPUs[43] and 16 GB of RAM for a limited period of time. The maximum computing power a user can pay for is 16 vCPUs, 96 GB of RAM and 4 GPUs.

At **RS**, free users are limited in problem size and in execution time. There are various problem size limits that apply depending on the type of problem the user is trying to solve, e.g., the maximum number of decision variables is limited to 200. The solving time cannot surpass 1 minute per problem and 4 hours per month. Other problem size limits can be found in the WIDE documentation[44]. As for the limits of paying users, they are not publicly disclosed.

At **C9**, users select the VM configuration of their WIDE environment from a subset of the VMs available at AWS Elastic Computing Cloud. New AWS accounts can setup a **C9** environment and use it for free for one year. The free tier environment comes with 1 vCPU or 2 depending on the server location, 1 GB of RAM and 30 GB of storage. The most powerful VM a user can pay for comes with 384 GB of RAM, 96 vCPUs and 16 TiB of storage.

At **CA**, there is no free tier option, only a seven day trial for on of the paid plan. The cheapest plan allows the creation of one VM with 2 GB of RAM and 10 GB of storage, while users at the most expensive plan can create up to 6 VMs with 48 GB of RAM and 120 GB of storage each.

At **RP**, there is a free tier option in which the provisioned VMs come with 0.2 vCPU, 500 MB of RAM and 500 MB of storage. Free users can have up to

[43] A vCPU is a virtual central processing unit, which is basically a portion of the processing power of a physical CPU. The processing power details of a single vCPU vary across VM hosting services. As a point of reference, at Amazon Web Services a vCPU is usually equivalent to a thread of a physical CPU core. In general, this is not an information disclosed by the WIDE providers, so we will limit ourselves to use the term vCPU without detailing its processing power.

[44] https://www.rason.com/Help/Index/RASON-Subscriptions-Intro

20 concurrent projects. The most expensive publicly disclosed plan provides users with VMs with 2 vCPUs, 4 GB of RAM and 10 GB of storage.

GC does not provide computer resource limits because the availability of resources for a given user varies over time, according to a set of priority rules that take into account, among other things, the resources usage pattern of the user and their subscription plan. The machine provided to us in our test as free tier users came with 12.69 GB of RAM and 107.77 GB of storage, but this may not always be the case[45].

At **OL**, the computation limits are all the same for free and paying users, except for the maximum compilation time, which is of 1 minute for free tier users and 4 minutes for paying users. Other limits include 7 MB of editable material per project and 1 MB maximum size for an editable file.

At **C9**, although all the computation is done in the client side, the WIDE back-end is still used for storing user's files, but the user's storage limits are not disclosed.

5. Persistent storage

Where do user's files live? The usual persistent storage solution is to keep the user's files in a database or file system in a server somewhere. But there are two cases which are worthy of a closer look.

First, at **C9** and **CA** user's files live inside the VM to which the WIDE is connected to, meaning there is no separation between the storage environment and the execution environment. Once the VM is deleted, the files are permanently lost. This is different from the persistent storage solutions of **WS**, **RP** and **GC**, in which user files are kept separated from the execution environment.

Second, each project at **RP** has its own private database. The main purpose of this database is to circumvent two limitations of the persistent file storage solution. First, dynamically generated files are stored in the VM executing the program, which means that they are lost once the VM is stopped[46]. Data stored in the project database, however, is persistent, so dynamically generated data can

[45] Quote from the **GC** FAQ: "Colab does not publish these limits, in part because they can (and sometimes do) vary quickly". https://research.google.com/colaboratory/faq.html

[46] https://blog.replit.com/alwayson

be stored in the database instead. Second, the database of a project is private[47]. This separation between code and data allows users to share their project code without sharing their private data.

Integration with cloud storage services is provided by some of the WIDEs. Files in a **OL** project can be kept in sync with Dropbox. At **CA**, users can establish a connection to Dropbox and manage their files from the WIDE file explorer. Google Drive integration can be found at **CA** and **GC**. Just as with Dropbox, **CA** users can connect to Google Drive through the WIDE interface. **GC** uses Google Drive by default to store the main file of a project. Moreover, users can mount their Google Drive to the execution environment and dynamically access their files therein.

6. Long running processes

How does the WIDE handle long running processes? First, we want to know if there is a hard limit on how much time a process can take to finish. At **WS** there are no time limitations. **RS** does not allow free user processes to run for longer than 1 minute. For paying users, this limit is not publicly disclosed. **C9** and **CA** VMs have no maximum lifetime, so there is no limitation on how long a process can run for. As already mentioned, **RP** VMs are Google Cloud preemptible instances, which are stopped after 24 hours[48], so a process cannot run for longer than that. **GC** VMs have a maximum lifetime of up to 12 hours[49]. Finally, at **OL** the compilation cannot take more than 4 minutes, whereas **CS** does not impose any time limitation, since processes run on the browser. Theoretically, they can theoretically run forever as long as the browser tab is kept open.

Second, for the WIDEs that can run user processes for multiple hours (**WS**, **C9**, **CA**, **RP**, **GC** and **CS**), we want to know what happens when a user closes the WIDE while a process is running. At **WS** and **C9**, the process will stay running indefinitely, even if it takes multiple days for completion, as long as the VM in which the process in running is not interrupted. This does not not normally happen without an user action in these WIDEs.

[47] https://docs.replit.com/misc/database
[48] https://cloud.google.com/compute/docs/instances/preemptible
[49] https://research.google.com/colaboratory/faq.html

Processes running at **CA**, **RP** and **GC**, on the other hand, are interrupted in some way or another after the user closes the browser tab. It may not happen immediately, but by default processes will be interrupted or paused after some amount of inactive time. To overcome this limitation, **CA** and **RP** both offer a paid "always-on" option, in which the VMs running the processes are kept awake even if the user is not connected to the WIDE. At **RP**, however, even with this option enabled, a VM may still be occasionally restarted as it is migrated between physical servers[50].

At **CS** processes run locally in threads that are bound to the tab in which the WIDE is open. Once the tab is closed, all threads running in that tab are killed.

7. Group work

What does the WIDE do to facilitate group work? Basically, WIDE providers offer four main features to aid collaborative work.

First and most important, the ability to share a project with multiple users. In order for collaboration to be possible, the least users need to be able to do is to work in the same project as other users. Without this feature, none of the other three are possible. This shared access feature is available at **C9**, **CA**, **RP**, **GC** and **OL**.

Second, there is the real-time collaborative editing, which allows multiple users to edit the same file at the same time with real-time update of the file content. This feature can be found at **C9**, **CA**, **RP** and **OL**. The last three implement operational transformation to guarantee convergence[51,52,53], but the design of the collaborative editing system used at **C9** is unknown. **GC** used to have real-time collaborative editing, but this is no longer the case[54]. Now, when multiple users edit the same file at the same file, changes made by one user are overwritten by changes made by the other.

[50] https://blog.replit.com/alwayson

[51] https://blog.codeanywhere.com/real-time-collaboration-issues-and-solutions

[52] https://blog.replit.com/collab

[53] https://overleaf.com/learn/how-to/Can_multiple_authors_edit_the_same_file_at_the_same_time%3F

[54] https://github.com/googlecolab/colabtools/issues/355#issuecomment-446647296

Third, there is the built-in chat, allowing the collaborating users to converse without leaving the WIDE. This is possible at **C9**, **RP** and **OL**. One may wonder if this feature is really necessary nowadays with all the business communication tools available, such as *slack*[55] and *flock*[56].

Fourth and finally, there is the ability to make comments on the source code. Differently from plain source code comments, these are designed to facilitate discussion about the code. When a user comment over a line, a conversation is initiated. The discussion is kept separated from the chat, which allows for a better organization of conversation topics. The feature is present at **RP**, **GC** and **OL** (only for paying users in this last one).

8. Version control

Are there built-in version control features? The answer is positive for **RS**, **RP**, **GC** and **OL**.

At **RS**, every file change pushed to the server creates a new version of that file. At the file navigation menu, users can browse through the different versions of each file. Moreover, the resulting output of a model is also stored and dated, so users can always see the results of past executions and compare with more recent ones. A model file can be marked as "champion", which helps users to keep track of the current best model formulation.

A similar history tracking system is present at **RP**, **GC** and **OL**, but rather than having to manually push newer versions of a file every time a change is made, the project's history is automatically updated as the user works on it. The user is then able to browse through the project history, compare versions, see file changes side by side in a built-in *diff* tool and restore an older version of the project. Moreover, at **GC** and **OL** users can give custom names or labels to different versions.

Additionally to the history tracker, paying users of **OL** also have access to a change tracker tool. This tool is designed to facilitate code review. It provides a list of changes made by each user, and each change can be accepted, rejected or commented by the reviewer. The feature is comparable to pull requests on Git

[55] https://slack.com
[56] https://flock.com

hosting services, in which changes suggested by users can be accepted or rejected by a reviewer.

How about integration with version control cloud services? Projects at **C9**, **CA** and **OL** can be integrated with any Git repository hosting service (GitHub, GitLab, BitBucket, etc). Over at **RP** and **GC**, users are limited to GitHub integration. By "integration" we mean that the users can interact with the Git hosting service through the WIDE interface to perform common operations such as cloning, pulling and committing.

9. Project discoverability

Do the WIDE provides mechanisms to discover other user's projects? Sometimes users are working on projects that they want to be publicly available (an open source library, for instance). At **RP**, projects are public by default, and sharing/discovering projects is facilitated by public communication channels. Through *Replit Talk*[57] users can share projects, see what other people are working on, ask for feedback on an idea, and so on. Additionally, all the public repositories of an user can be seen in their profile page.

Public projects at **CS** can also be discovered by anyone. Although the WIDE does not have a built-in communication channel like **RP**, there is a search bar that allows users to search through all public projects in the **CS** database. Besides, popular projects are displayed in the initial page of the application, and people can also see the public projects of an user by accessing their profile.

Although we've been looking at project discoverability from a positive perspective, we should also ask: do the WIDEs provide mechanisms to protect the privacy of users' projects? This is only a concern for the two mentioned WIDEs (**RP** and **CS**), since the others provide no way for users to discover and access other people's projects. Both **RP** and **CS** allow users to make their projects private, but at **RP** this is a feature only available for paying users.

* * *

[57] https://replit.com/talk/all

The WIDEs characteristics described in this section have been summarized in Figure 5.1. For compactness, some of the details have been omitted. Additionally to showing how the WIDEs compare among themselves, the figure also shows how PIFOP compares to them.

	GUI characteristics	Coding and dbg facilities	Comp./exec. environment	Provisioned comp. resources	Persistent storage	Long running processes	Group work	Version control	Project discoverability
Watson Studio (**WS**)	Ace No LR No +/-	Yes No No No	IBM No No No	min 4 16 ● max 16 96 ●	No	∞ Yes	No No No	No No	No
RASON IDE (**RS**)	Ace No TB No No	Yes Yes No No	Azure No Yes No	- Problem size limits - 1 min per process - 4 hours per month	No	1min ●	No No No	Yes No	No
Cloud9 (**C9**)	Ace Ace TB Yes +/-	Yes Yes Yes Yes	AWS Yes No No	min 1 1 30 max 96 384 16000	No	∞ Yes	Yes Yes No	No Any Git	No
CodeAnywhere (**CA**)	Mon xt.js TB Yes +/-	Yes Yes Yes Yes	Yes Yes No No	min ● 2 10 max ● 48 120	Dropbox Google drive	∞ Yes*	Yes No No	No Any Git	No
Replit (**RP**)	Mon xt.js LR Yes Yes	Yes Yes No No	GC No No No	min 0.2 0.5 0.5 max 2 4 10	No	24h No	Yes Yes Yes	Yes GitHub	Yes
Google Colab (**GC**)	Cust No SP Yes Yes	Yes No No No	GC Yes No Yes	Resources availability fluctuate and are unpredictable	Google Drive	12h No	No No Yes	Yes GitHub	No
Overleaf (**OL**)	Ace No LR Yes +/-	Yes Yes No No	Yes No Yes No	1 to 4 min compilation ~7 MB per project	Dropbox	4min ●	Yes Yes Yes	Yes Any Git	No
Csound (**CS**)	CM No TB Yes Yes	Yes No Yes No	No No No Yes	Unknown	No	∞ No	No No No	No No	Yes
PIFOP	Cust Cust LR Yes No	Yes No No No	No Yes No Yes	Project size and time limits are still being defined	No	Plan dependent Yes*	Yes No No	No No	No

GUI characteristics

- Which code editor? Mon = Monaco, CM = CodeMirror, Cust = Custom
- Which terminal front-end? xt.js = xterminal.js
- What's the layout? LR = Left-right, TB = Top-bottom
- File system organization?
- Mobile friendly? +/- = partial

Coding and dbg facilities

- Syntax highlight?
- Live code check?
- Integrated documentation?
- Third-party debuggers integration?

Comp./exec. environment

- Does it provides the user with an RCE? From which cloud service? IBM = IBM Cloud, Azure = Microsoft Azure, AWS = Amazon Web Services, GC = Google Cloud
- Can users use their own RCEs?
- Is there an on-premises option?
- Can the local machine be used as execution environment?

Provisioned comp. resources

For the WIDEs that provide RCEs, how much resources can users possibly have? (vCPU, RAM and storage)

For the others, this column shows some relevant resource limits.

Persistent storage

- Is there integration with cloud storage services?

Long running processes

- For how long can processes run at most?
- For WIDEs in which processes can run for multiple hours, does it stays running after the tab is closed? Yes* = paid option

Group work

- Real-time collaborative editing?
- Chat?
- Comments?

Version control

- Built-in VC features?
- Integration with Git cloud services?

Project discoverability

- Are projects publicly visible/discoverable?

Figure 5.1: Summary of our comparative analysis. Icons created by the following flaticon.com users: Becris, Freeplk, Iconixar, Mynamepong, Kiranshastry, Kirill Kazachek, Pixel perfect, Smashicons, Srip and xnimrodx.

In the next three sections, our goal is to digest all this information and provide answers to our research questions. What can we learn from all this information which we were able to gather?

5.4 Challenges faced and overcome by WIDE providers

Our first research question is about the challenges involved in the service provided by WIDEs in general. **(RQ1) How do WIDE providers overcome the main challenges they have to face in order to effectively meet user demands?** In the previous section we have briefly presented the different ways in which different WIDEs work to provide users with an IDE experience in the browser. Now we want to discuss in greater detail what we think are the two most important challenges WIDE providers have to face and how do they overcome these challenges.

Challenge #1: Execution environment

The first big challenge is that of enabling users to compile/execute their code. All the WIDEs would be pretty much useless without this capability. Not only code execution is important from the user's perspective, but from WIDE providers perspective it is often the core of their business (and possibly the most costly too). That is why at **RP** they have stated that remote code execution is their main product[58].

Traditional desktop IDEs can simply run compilers and interpreters on the user's computer. But that is not an option for WIDEs because, since they run in the browser, they are unable to spawn processes on the user machine. So the crucial question WIDE providers have to answer is this: "*where* are we going to compile/execute user code?" Many other features of a WIDE depends on the answer to this question. For instance, the ability to call command-line tools requires that the user have access to a VM; if a VM approach is adopted, user files can be stored in that VM rather than in a database; and the ability to run time consum-

[58] https://blog.replit.com/pwnd

ing programs require the execution environment to be active for a long time. Since the execution environment solution leads to design decisions that will impact the service in many other areas, we proceed to discuss in grater detail the solutions that have been adopted to overcome the challenge of enabling users to run their code.

Solution #1: Black-box execution environment

One way WIDE providers responded to the challenge is by saying that they are going to run user code on their end inside a black-box, meaning that the environment in which the code is going to be executed is unknown to the user. All that the user gets from this black-box environment is the output of their program. That is the main option for users at **WS**, **RS**, **GC** and **OL**. **WS** and **GC** do enable paying users to at least select some configurations of the black-box, such as the machine specifications (vCPU and memory), but still, all they can do with the environment they get is to submit their code and get the results back.

From the user's perspective, the advantage of this solution is that it requires little to no effort from their part in setting up an environment in which their code can run. This is a problem for the WIDE to solve. On the other hand, this solution limits the user on what they can do with the execution environment they have. This is not a problem if all they want from the environment is for it to run the code and return the results, but can become a problem when the user requires the environment to be configured in a particular way or if they need to run other programs that are not the compiler/interpreter nor their own program they are writing.

From the WIDE provider's perspective, there are a few disadvantages, which may or may not be compensated with the benefits provided to the user. One disadvantage is the need of a robust solution for running user code securely. Since they will be running untrusted code in their own servers, it is important that the WIDE prevents users from accessing data they are not allowed to and from interfering with other users' processes. Another disadvantage is that the burden of setting up an execution environment is on them. They are the ones responsible for providing the machine resources needed for user code and making sure the

environment fits the needs of their users (or at least *most* of their users).

Now, it seems to us that this solution is very fitting in situations with the following characteristics. First, when the typical user does not require to run programs other than the compiler/interpreter of the language. Second, when the language being supported does not allow users to do much with the machine anyway, so untrusted code execution is less of a concern for the WIDE provider. Third, when a one-size-fits-all environment covers most of the needs users may have, which makes the lack of environment configurability less of a problem for users.

The first point is characteristic of the situation of the WIDE providers **WS**, **RS** and **OL**. In the case of mathematical programmers and authors working with LaTeX, the users are mostly interested in the solution returned by the solver and the document returned by the LaTeX engine, so if they are unable to run other programs, that is not necessarily a big user concern. Moreover, the declarative characteristic of OPL, RASON and LaTeX makes so that users of these languages cannot do much with the execution environment with the source code they write, so the second point also is descriptive of the situation of **WS**, **RS** and **OL**. Finally, the needs of users of these languages is somewhat predictable, meaning it is not difficult to devise an environment configuration that will meet most use cases. For instance, as long as the **OL** execution environment enables the user to include the majority of LaTeX most used packages, a single environment configuration can meet the needs of most of its users.

GC partially fits the situation described above, but for different reasons. Typically, Jupyter Notebook users are not using the application to write large programs that need to run for a long time. The attractiveness of Jupyter is that it allows a mix of rich text and source code in the same file, which makes it useful for explaining, documenting, prototyping and sharing simple programs with other users. So the average Jupyter user is not in need of running programs other than the Python interpreter. Moreover, since all a Jupyter user will be using is Python modules and packages, an environment with the standard packages with the ability to install other packages is enough for most use cases. The only real concern for **GC** as a WIDE provider adopting this solution is of untrusted code execution, a problem they solve by isolating the user code in a virtual machine.

Solution #2: Provisioned virtual machines

Another way in which WIDE providers have been enabling users to compile/execute their code is by providing them full (*sudo*) access to a remote virtual machine. This is similar to the previous solution in that the execution environment is provisioned by the WIDE, but is different in that the execution is not a black-box. Rather, the user is given full control over the environment. That way they can configure it however they want, download and install packages, run applications and servers, execute system commands and do pretty much whatever they want with the computing resources they get. This is the main option for users at **C9**, **CA** and **RP**.

There are two main variations of this mode of execution environment provisioning. At **C9** and **CA**, the VMs provisioned to users are persistent, meaning every time they access a given project they will be working on the same VM. At **RP**, on the other hand, that is not guaranteed, meaning that users should not expect to be working in the same VM every time they access a project. We call these two approaches *persistent* VM provisioning and *disposable* VM provisioning, respectively.

From the user's perspective, the main advantage is the high level of control they get over the execution environment, which enables them to work with a great variety of packages, tools, programs and programming languages. But this high degree of freedom comes with some disadvantages. Setting up a new project at **C9** has an additional step of selecting the VM specifications (for instance, operating system and computing resources). Then, if the VM does not come with tools and packages they need, users are required to download and configure them themselves. Moreover, when starting a new project at **C9** and **CA**, the user has to wait an additional time for the start-up of the newly created VM. At **RP** the wait time is not as long at the start of a new project, but there is still some latency associated with the VM initialization when accessing the project.

From the WIDE provider's perspective, there is the advantage of not having the responsibility to configure the execution environment for the user. Yes, they still have to start-up the VM and establish user connection with it, but the configuration of the VM (the tools and packages installed, for instance) that's for the

user to decide. On the other hand, since they are providing user with *sudo* access to remote machines, there are more security concerns for the WIDE provider to worry about. Without proper isolation, users can damage the back-end infrastructure or read and write files they should not be able to, either intentionally or unintentionally.

This type of execution environment provisioning seems to be particularly useful when it is hard to predict what users will need to do their work. These are situations in which a one-size-fits-all environment is unfitting (pun intended), because different users will need different tools and packages, so they need the flexibility to configure the environment in a way that works the best for them. The nature of general programming languages is such that they can be used to build a variety of different programs, so the needs of an user are unpredictable. We believe that's the reason why **C9**, **CA** and **RP**, which aim to support multiple general programming languages, opt for this kind of execution environment provisioning.

Solution #3: Execution on user owned machines

The third solution is similar to the previous one, but here the execution environment, rather than being provisioned by the WIDE provider, is provisioned by users themselves. This option is given at **RS**, **C9**, **CA**, **GC** and **OL**, but is not the main mode of operation advertised by any of them.

There are two ways in which execution is made possible on user owned machines. The first is by establishing a connection between the WIDE front-end and the machine in which the user wants to use as execution environment, which is how the solution is implemented at **C9**, **CA** and **GC**. In this first variation, communication with servers of the WIDE provider is still needed for the solution to work. At **C9** and **CA**, the server is in-between the WIDE front-end (user's browser) and the execution environment provisioned by the user. At **GC**, the user's browser is in-between the execution environment provisioned by the user and the servers of the WIDE provider (where the data is stored). In both cases, users still depend on the WIDE provider servers.

The second way in which execution is made possible on user owned machines is through an on-premises solution, which is an option at **RS** and **OL**. Here the

WIDE provider offers users the option to run the WIDE server on their own private clouds. Differently from the solution described above, this completely eliminates the dependency of users to the public WIDE provider cloud.

From the user's perspective, the execution on user owned machines has two main advantages. First, it enables them to use computing resources they already own, or resources they would not be able to have with the VMs provisioned by the WIDE. Second, their data is stored on their own private cloud rather than on the public WIDE provider servers, which is better for companies concerned with the privacy of their data. The main disadvantage is that it requires the user to run and maintain their own WIDE server, which requires a person or team with system administration skills. Moreover, many of the conveniences associated with the public WIDE cloud are potentially lost, like the cloud storage and the ability to access a project from anywhere.

From the WIDE provider's perspective, there is the advantage of meeting the needs of medium to large sized companies that have their own infrastructure and/or care about their data privacy, which are willing to pay more for this kind of solution (especially the on-premises alternative). Additionally, the burden of running user's programs is transferred to the user, which saves the provider the computing resource cost. On the other hand, the on-premises alternative has the disadvantage that this is a product on its own, and thus requires development time and support time, which the WIDE provider may not have available.

This kind of solution is more geared towards companies and teams that want to use their own machines for execution and to which data privacy is a high priority concern. So if the WIDE provider aims to meet the needs of this public, the solution here described is something they should consider.

Solution #4: The browser is the execution environment

Finally, the fourth way in which WIDE providers overcome the challenge of user code compilation/execution is by using the browser as the execution environment. **CS** is the only WIDE in this study that adopts this approach. The way they've managed to do it is by porting the Csound engine, originally written in C, to WebAssembly (Lazzarini et al., 2014), which is also how we at PIFOP execute

GLPSOL in the browser.

From the user's perspective, there is the advantage that the system that they are interacting with is simpler, meaning they don't have to bother with choosing a remote code execution environment provisioned by the WIDE or by themselves, and the latency of the application is potentially lower in many cases. But there are three disadvantages to this approach. First, users are limited to the computing resources they have available in their own machines. Second, this approach makes the persistent execution after the browser is closed impossible. And third, the performance of an application ported to a browser language (WebAssembly or Javascript) is not necessarily the same of the application in its original language.

From the WIDE provider's perspective, the system simplicity that benefits users is also an advantage for them, because the communication network of the whole system has one less component (using Figure 3.1 as a reference, the third layer is eliminated), and the lower the complexity of a system, the easier to maintain it. The drawback is that it requires porting the compiler/interpreter to a browser language. Depending on the characteristics of the program(s) that need to be ported, this may be trivially done with tools like *emscripten*[59], or too difficult of a task due to the complexity of the program or even impossible due to the unavailability of the source code or to licensing restrictions.

This solution is a viable alternative when the porting of the compiler/interpreter is something feasible and when the programs being executed are not too resource or time consuming. That way, the fact that users cannot use other machines with the WIDE is not a problem, and the fact that they can't leave the program running after closing the browser is also not a problem.

Challenge #2: Competition with desktop IDEs

The competition between WIDEs and desktop IDEs is an uneven one. Some desktop IDEs have decades of development and bug-fixing on their back, which makes them very stable and reliable. They can also count with a large ecosystem of tools that have been developed for offline execution: interpreters, compilers, debuggers, and so on. Over the previous decades, desktop IDEs have grown a

[59] https://emscripten.org

large user base which got used to working with them and have efficiently integrated them in their workflow. Now, with the emergence of WIDEs, this large user base rightly asks: "why bother?"

Because of this accommodation of users to their IDE of choice, it is not enough that WIDEs offer the same features provided by desktop IDEs if they want to attract users that already have their preferred desktop IDE. There would be no incentive for these users to migrate to another IDE that does exactly the same thing, but with internet latency. So WIDEs have to compensate their disadvantages in some way or another. But if they can't beat desktop IDEs in their reputation, reliability and integration with other desktop tools, how do they do it? The previous challenge was related to how can WIDEs enable users to compile/execute their code, like desktop IDEs do. Now we want to know what do WIDEs do to go *beyond* what a typical desktop IDE can do. It appears to us that there are two main incentives that WIDEs provide to attract desktop IDE users.

Incentive #1: Group work support

Very rarely we see a desktop IDE with support for collaborative work. *Atom*[60] and *Visual Studio Code*[61] (VSC) are two of the few that have support for real-time collaborative editing, but even then their collaboration solutions are not quite as powerful as those provided by WIDEs. The way in which *Atom* and VSC provide real-time collaboration is by enabling users to "invite" their peers to participate in an editing session, where all collaborators can edit the files of the same project. The limitation lies in the fact that once the session host closes his IDE, the editing session is over, so asynchronous collaboration is made impossible. This means that if the peers of the host wanted to keep working on the project they would have to wait until the host invites them for another session, because the files live in the host's machine. This solution is fine for one-time collaborative tasks, such as when a developer wants help on finding a bug, but it is unsuitable for long-term collaboration, where multiple users need access to the same project at any moment.

[60] https://teletype.atom.io
[61] https://code.visualstudio.com/learn/collaboration/live-share

The advantage of WIDEs that support group work is that the code lives in the cloud and is accessible at any time by anyone with access to that code. This makes it easier for WIDE providers to better support collaborative work. As we've seen, real-time editing is not a feature exclusive to WIDEs. But this feature, paired with the fact that collaborators can access the same project at any moment, makes the collaboration much more productive for long-term projects.

We would say that this feature combination alone can be more than enough compensation for the disadvantages WIDEs have over desktop IDEs. Take *Google Docs*[62], for instance. It is not a WIDE, but it is a good example on how this feature can make a web application more attractive than an equivalent desktop application. There is no doubt that, in terms of rich text editing and processing, *Microsoft Word* can do much more than what *Google Docs* can. But there are many cases in which users are willing to trade advanced editing features for the ability to edit the same file with other people. So even though *Microsoft Word* is as powerful as it is, a single *Google Docs* feature makes up for a large amount of features it lacks.

Other ways in which collaboration is facilitated include built-in chat and the ability to discuss parts of the source code through comments, which also puts some WIDEs ahead of desktop IDEs in the collaboration support.

Incentive #2: Remote code compilation/execution

There are cases in which compiling/executing code remotely adds no benefit to the user. For instance, users developing programs with graphical interfaces would find no use in having their code being executed remotely without the ability to interact with its interface, or if the user's machine is powerful enough for compiling and running the code. But RCE can be very beneficial in many other cases, of which we mention three.

First, when the application being developed is to be remotely deployed. With RCE, the developer can compile and test the application in the same environment in which it will be put into production. Web servers are good examples of applications that are usually not targeted to run in the developer's machine, but

[62] *Google Docs* is a web application for editing rich text documents.https://docs.google.com

on virtual machines in the cloud. WIDEs that allow users to use the production environment as the execution environment can increase the productivity of web developers, as they will be developing and testing in the same environment the system is to be deployed.

Second, when the execution of the code is computer resource intensive, hence requiring more resources than it's available on the user's machine. Some WIDEs enable users to choose the hardware specifications of the execution environment, so they pick an specification that meets their computing resources demands. This is particularly useful for applications that consume lots of memory and CPU cicles to perform expensive calculations, which is the case of mathematical optimization programs.

Third, when there are multiple people working together in the same project from machines with different configurations. RCE eliminates the "it works on my machine" problem because all developers will be compiling/executing the program in the same environment. This facility, of course, is only possible in WIDEs that have group work support. The problem of divergent development environments is not limited to general programming scenarios, but can also appear in multiple other contexts as well, such as LaTeX code compilation, which requires that all the environments in which the code is going to be compiled has all the package dependencies installed therein.

5.5 Challenges faced and overcome by providers of WIDEs for optimization

Now we want to zoom into WIDEs for optimization and see what are the particular challenges they have to overcome, which do not necessarily occur in other contexts. The question we want to answer is this: **(RQ2) How do providers of WIDEs for optimization overcome the main challenges they have to face in order to effectively meet user demands?** Similarly to what we have done in the previous section, here we are going to delineate what we think are the main challenges providers of WIDEs for optimization have to overcome and how do they do it.

Since our sample of WIDEs for optimization is rather small (consisting of only **WS** and **RS**), we are also going to include PIFOP in the discussion of this section and use it as reference of WIDE for optimization as well. The reader will notice that the solutions here presented are closely related to the solutions presented for the first challenge of the previous section, and that we are not doing more than revisiting them with a new challenge in mind. Yes, that's true. After all, a single solution can address multiple challenges.

Challenge #1: Computing resources

Mathematical optimization programs can grow in size very quickly. It is not rare that in order to solve an optimization problem, the program will consume multiple gigabytes of RAM and require multiple CPU cycles for matrix multiplications and other expensive operations. This is not a particularity of optimization programs, of course, but a WIDE for general programming can get away with not addressing this case and still be useful for advanced level programming projects. Mathematical programmers in the advanced level, on the other hand, are often enough working on large optimization models that require powerful machines to solve them.

Solution #1: Resources provided by the WIDE

The first solution of the previous section can be offered in a way that allows users for *some* customization of the black-box environment. The code execution at **WS** occurs in an environment over which the user has no control, with exception of the virtual machine specifications of vCPU count and memory. This approach effectively overcomes the challenge in view. The drawback is that users are then limited to what the WIDE provider has to offer, and have to pay for that resource, not being able to use machines they may have at their disposal.

At **RS**, although the main execution environment available to users is the black-box provided by the server, this black-box is not dedicated nor customizable. Users are limited in many ways on the size of problems they can solve in that environment.

Solution #2: Resources provided by the user

The way in which the challenge can be overcome at **RS** is through the RASON dedicated servers, but we have not enough information to say how effectively does this option addresses the issue. We know that the dedicated servers can run on the premises of companies, allowing them to use the computer resources they have at their disposal, and that the resource usage of these servers can be configured. But we don't know if these dedicated servers can be used via a WIDE interface, or if the only way to interact with them is via the RASON REST API. If the second is the case, the user is left with an execution environment without a WIDE, which means we are not talking about a solution for the WIDE user, but for the general programmer that will be using the REST API.

At PIFOP, users can use any Linux machine at their disposal by running the PIFOP optimization server on that machine, hence transferring to the user the responsibility of providing an execution environment for themselves while still enabling them to use the same browser interface. Since users are free to choose machines with whatever hardware specifications they need, the solution effectively addresses the issue in question. The drawback of this approach is that it requires users to provide the environment themselves, which can be difficult if they don't have one already.

Solution #3: Use of local machine resources via browser

PIFOP users can run the GLPSOL solver in their own browser, which of course will consume the resources available in their own machine. This is only a real "solution" for the challenge in view for users that already have all the resources they need in their own machine, and if they are content with GLPSOL as their solver. But realistically speaking this combination of factors should not be expected for the majority of advanced level mathematical programmers. We only mention this as a "solution" because if it ever so happens that commercial solver enable users to run the solver in the browser, this approach may be attractive for some users.

Challenge #2: Long-running processes

It is not enough that users have access to large amounts of memory and vCPUs. If their processes cannot run for multiple days, mathematical programmers will not be able to solve some large scale optimization problems. We note that, although some programs in general programming languages such as web servers are designed to run "indefinitely", there is an important difference with mathematical programs. Web servers can be interrupted and restarted and nothing is lost, except maybe the time of the users trying to at the moment that happens. An optimization program, however, when interrupted, can potentially lose hours of calculations accumulated during the execution, which will then require the user to restart the calculations from zero.

Solution #1: Persistent execution environment provided by the WIDE

The reason why this is not an issue at **WS** is because the black-box execution environment provisioned by the WIDE is persistent, like the VMs at **C9** and **CA**, which contrasts with the execution environments of **RP** and **GC** that have a lifetime and thus will interrupt the user's program when that lifetime is reached.

The advantage of this solution is that the WIDE provider does not assume that the user has the computer resources they need to solve their problems, either on their on local machine or on remote machines they may have at their disposal. This removes from the user the burden of setting up an execution environment themselves and transfers it to the WIDE provider.

Solution #2: Persistent execution environment provided by the user

At **RS**, the on-premises solution can be used to address this challenge, but then again we don't know if users are still able to interact with the server via a WIDE interface. PIFOP users can host their own optimization servers, and as long as the optimization server is not interrupted, the optimization program should run to completion. There are time limits associated with the subscription plan of the user, but these can be raised to meet the user needs with a better plan.

The in-browser execution is another variant of this solution. The drawbacks are that the user has to keep the browser tab open during the execution, and that they are limited to running programs that can be run in the browser. In the case of PIFOP, GLPSOL is the only option for in-browser execution.

Challenge #3: Proprietary software

Differently from what happens with compilers and interpreters of general programming languages (and some domain specific languages, like LaTeX), there is a significant performance distinction between proprietary and free optimization problem solvers. For that reason, modelers working on large scale problems are likely to require the use of commercial solvers. Enabling access to proprietary solvers is a challenge for WIDE providers for basically two reasons. First, they can be very expensive. And second, the access must be given in such a way that the license being used is protected from being cloned by the user.

The first reason makes it difficult for WIDEs to provide each user with their own licensed copy of a commercial solver, similarly to what happens when WIDEs such as **C9**, **CA** and **RP** provide each user their own copy of free compilers and interpreters. And the second reason discourages the providers of WIDEs for optimization from adopting the execution environment solution of giving users *sudo* access to VMs with the tools they need at their disposal. For instance, if a WIDEs like **RP** included a CPLEX license in the VM, users would be able to clone that license and use it elsewhere. So, how to address these issues?

Solution #1: Black-box execution on the server side

Again, the solution lies in the way code execution is made possible by the WIDE provider. Both **WS** and **RS** run the user code in a black-box environment, which prevents users from being able to download the proprietary software they are using.

But this solution only addresses the license protection issue. What about the expenses with solver licenses? It just so happens that the same companies that provide **WS** and **RS** are also the developers of the solvers which users can access through these WIDEs, so it is just a business decision matter for them on how to commercialize their own products. We can only imagine how it would be if that

was not the case. Maybe a deal between companies would be required, or maybe the WIDE provider would make the investment on acquiring multiple licenses and pricing for the usage accordingly. But these are only speculations.

Solution #2: Black-box execution on the user side

PIFOP addresses the challenge by transferring to the user the burden of having to acquire solver licenses themselves. Once they have the license in hands, they can run the optimization server on the machine with the solver and use it through PIFOP. This solves the expenses issue on the side of the WIDE provider. Now, we've seen that the owner of an optimization server can enable access to multiple users. Are the solver licenses protected from being copied by other users? Yes. Since the user code is executed inside a sandbox on the host machine, there are many things users cannot do, including downloading or uploading files.

The disadvantage of this solution is that, rather than their purchase being a one-stop experience, users have to first purchase the solvers at one place and the WIDE subscription in another. We think that it would be better if, upon subscription, users had the option to acquire the solvers they need, or if they were able to use the solvers and pay-as-they-go. At any rate, the current solution does serve to overcome the challenge in question.

Solution #3: Integration with cloud optimization services

There is an alternative way through which PIFOP users can use commercial solvers, which is by using the *NEOS server*. It is a great service that lets users run commercial solvers for free. The drawback is that the computer resource limitations can be quite restrictive, reason why we haven't mentioned this as a solution for the previous two challenges. Jobs submitted to the *NEOS server* can consume no more than 3 GB of memory and run for, at most, 8 hours. Nevertheless, it is still a viable option for users in need of running commercial solvers.

Generally speaking, the solution consists in the WIDE provider making use of RCE services, which is what *NEOS server* essentially is. For the end user, the execution environment is still a black-box, and the licenses of the software they are using are secured from being cloned. The WIDE provider acts as a bridge

between the user and the RCE server. The advantage for both WIDE providers and their users is the separation of concerns: the WIDE provider can focus on the user interface while the RCE service is responsible for running user's code. On the other hand, the WIDE provider will be limited to offer they users what the RCE service used has to offer. This may or may not be an issue for users, depending on their needs.

5.6 Conclusion: WIDEs past, present and future

In concluding this chapter, we would like to call the reader's attention to how WIDEs have evolved over the years. We opened this chapter with a brief presentation of a few works in which WIDEs were studied and compared. In those works, especially the earlier ones, it is common to see comments on how WIDEs, in general, lack many important features present on desktop IDEs.

For instance, Iqbal et al. (2012) recommends: "Browser-based IDEs must provide interfaces to enable integration with code forges [i.e. cloud repository services like GitHub] to support open source software development in the cloud". Most of the WIDEs today have integration with GitHub or other similar services. Fylaktopoulos et al. (2016) comments that "[d]ebugging and runtime auditing needs to be further supported while a full set of languages and components (e.g. databases) need to be made available to the developer". Kats et al. (2012) makes comments in the same direction, pointing out the limitations some WIDEs have on their language parsing capabilities. But we see that, today, the debugging support provided by **C9** and **CA** is comparable to what can be found in traditional desktop IDEs, and they support dozens of different languages, as does **RP**. Moreover, there are different ways in which users of these WIDEs can work with databases. **RP** even provides its own built-in database solution.

Another comment made by Fylaktopoulos et al. (2016) is this: "The majority of users and enterprises are reluctant to trust sensitive data to cloud environments, and this is the main reason for the development of private clouds". As we have seen, some WIDE providers, like **OL** and **RS**, already offer solutions for users with this concern by enabling on-premises execution of the WIDE.

This sample of quotations, when contrasted with today's WIDEs characteristics, show that the service they provide have evolved considerably and continues to evolve. If it was true in the past that WIDEs were too far off from replacing desktop IDEs, that statement no longer holds. Undoubtedly, there are still trade-offs to be made when choosing a WIDE over a desktop IDE, and there are situations in which there isn't enough incentive to do the migration. But depending on the user's needs, a WIDE can be more than equally as good as a desktop IDE, but even better, especially when the need for collaboration and for remote code deployment is great.

In this study, we've shown what's the present state of the WIDEs market. In our comparative analysis, we've seen what they have to offer. And in answering our research questions, we've highlighted the main challenges they have to overcome and how do they overcome them. In short, there are two main challenges any WIDE provider has to face, which are the challenge of enabling users to compile/execute their code and the challenge of staying competitive against desktop IDEs. WIDEs for optimization, in particular, have three additional challenges that may not appear in other contexts. These emerge from the needs of mathematical programmers working on large scale models, which require large amounts of computer resources, can run for long periods of time and can only be efficiently solved with commercial software. None of these five challenges are insurmountable, and we've seen that each one can be overcome in multiple ways, some more effective than others.

If we were to make recommendations for readers just starting to build a WIDE, this is what we would say. First, know the demands of your users. Giving your users multiple options on where to run their code is great, but they are not all equally effective for meeting your users' demands. Knowing if your users need access to powerful machines or access to proprietary software will help you to focus your efforts on providing solutions that will actually be relevant for them. Second, enable users to do what they can't with desktop IDEs. The fact that WIDEs are built with web technology from the ground up makes it easier for them to provide features that depend on that technology. We've mentioned two ways in which WIDE providers take advantage of this: by providing support for collaborative work and by enabling users to compile/execute their code remotely.

And third, if the compiler/interpreter of your language can be ported to a browser language, do it. Supporting in-browser execution is a great way of providing value for users while also saving the costs of having to run the user code on your end.

Finally, our expectation is that we can apply on PIFOP what we've learned from the practices of other WIDE providers. And we also hope that this study can help the reader to see the advantages and disadvantages of each solution WIDE providers have given to the challenges they have to face, so they can make intelligent decisions on their own endeavors. Moreover, this chapter can serve as a guide for technology researchers in devising new solutions that can better solve the challenges we've discussed.

Chapter 6

Usability of PIFOP in the classroom

Our third research question is the following: **(RQ3) What are the student's perception of PIFOP as a tool for collaborative work and optimization model creation?** In order to provide an answer to this question we have conducted a case study with test group of students. In this chapter, we briefly present related case studies in Section 6.1, followed by a discussion on our methodology in Section 6.2 and an analysis of our results in Section 6.3. We conclude in Section 6.4 with a summary our findings regarding our research question.

6.1 Related Work

Here we mention a few other case studies related to the integration of web applications that support collaborative work in the classroom.

Berssanette et al. (2018) investigated the potential that digital technologies (DT) have to broaden the learning space by extending the classroom to the virtual world and to aid the learning of computer programming fundamentals. In their work, they evaluated the user experience of a test group of 39 students after using multiple DTs. The test group was composed of second year high school students who were taking a technical course on Informatics for the Internet. The DTs that they have used during the course include a web IDE (*Replit*[1]), a social

[1] https://replit.com

media (*Facebook*[2]), and several learning resources available online (*Hour of Code*[3], *Codecademy*[4], *URIOnlineJudge*[5] and *SoloLearn*[6]). 74% of the students thought that the use of DTs contributed to their development in the course subject. *Repl.it*, in particular, was considered to be important or very important by 92% of the students.

Valez et al. (2020) conducted a survey to understand students' perceptions towards *Kodethon*[7] and usage of its features. *Kodethon* is a WIDE developed by the authors of that work, which, at the time the paper was written, already had been used by more than 3000 students in 15 courses at the University of California, Davis. 140 students chose to participate in the survey. The usage of the tool varied: 48% said to use the tool "very often" or "often", and 52% said to use it "sometimes" or "rarely". The three features of *Kodethon* that have been considered to be the most useful by the participants were: the web-based characteristic (selected by 65% of the participants), the fact that it does not require installation (selected by 61% of the participants), and the automatic assignment grading feature (selected by 52% of the participants). As for the overall usefulness of the tool, the results were mixed: 32% agreed on the usefulness of *Kodethon*, 30% were neutral and 38% disagreed.

Zhou et al. (2012) evaluated the effectiveness of using Google Docs in an out-of-class collaborative writing activity for an introductory psychology course at the University of Georgia. 35 undergraduate students that have participated in two group assignments were, afterwards, asked to answer a questionnaire concerning their experience. Most students were unfamiliar with Google Docs prior to the study. The results show that 79% of the participants thought that the Google Docs influenced their group's collaborative experience positively or very positively, and 93% of the participants considered the application a useful tool for group work.

Škorić et al. (2016) applied the method of Analytic Hierarchy Process (AHP) to select the most appropriate WIDE for learning programming. In short, AHP is a

[2] https://facebook.com
[3] https://hourofcode.com
[4] https://codecademy.com
[5] https://urionlinejudge.com.br
[6] https://sololearn.com
[7] https://kodethon.com

method to aid establishing weights for the different criteria a decision maker is trying to meet. In their analysis, when it came to the choosing of a WIDE for learning programming, collaboration support was the second item in the priority list, just behind of embedded editor. Additionally, the study explored the perceived quality of *CodeBoard*[8] when used for educational purposes. *CodeBoard* was examined in an introductory computer science programming course, during a class session in which the tool, after being introduced to the students, was used for presentation of code examples by the instructor and for completing simple assignments by the students. At the end of the class, students were asked to complete a post-use questionnaire related to their perception of the tool. Students rated *CodeBoard* as simple and useful tool that has met their expectations in the context of learning programming. They have commended its simplicity and the code sharing feature, especially in interaction with the teacher.

Yildirim et al. (2017) carries out a study on the learner's perception of *Visual Studio Online* (VSO)[9], a WIDE by *Microsoft*. After 8 weeks of using VSO for group assignments, the perception of 100 students on the web application usability was collected by means of a questionnaire. Among the results highlighted by the authors, they've found that the average participant of the study did not find VSO easy to be used, and that it did "not provide any convenience compared to traditional learning platforms".

6.2 Methodology

Here we applied the action research methodology, which is a type of action investigation that makes use of traditional research methods to inform the action to be chosen to improve the practice (Tripp, 2005). In our case, the practice that we aim to improve is that of the work of students in doing their various optimization modeling related tasks, e.g., implementing models and executing computer experiments.

[8] https://codeboard.io

[9] Today the application goes by the name of *Visual Studio Codespaces*. https://marketplace.visualstudio.com/items?itemName=ms-vsonline.vsonline

As pointed out by Engel (2000), action research is situational in character. It does not aim to make universal scientific statements, but rather it looks for achieving relevant results for a specific situation. Thus, whatever results are obtained from an action research, the application of them in other situations is not a primary concern. Our focus in this chapter, therefore, is in our tool and how it can help mathematical optimization students in their day-to-day activities. That said, our findings should provide insights on how to effectively design WIDEs for it to be integrated in the classroom.

In this study we have assessed the user perception of a group of 18 students that were enrolled in the *Mathematical-Computational Implementation of Operations Research Models* course (OR-Lab for short) of the Production Engineering undergraduate program at UFMG in the first semester of 2020[10]. The course is usually ministered in the second year of the program, concomitantly with *Operations Research I* course. The goal of this pair of courses is to simultaneously teach students both the theory and the practice of operations research. Together, they amount to 90 hours of class in a semester, which are ministered in sessions of 100 minutes thrice a week — two sessions of *Operations Research I* and one session of OR-Lab — by professor Dr. Carlos R. V. de Carvalho.

Due to the pandemic crisis, the classes are being ministered virtually, via Microsoft Teams, with the aid of PIFOP. The professor uses PIFOP to implement mathematical models live, as the students are watching, and the students then follow along at home implementing the same model themselves. PIFOP is also used as a tool by the students to complete the practical assignments, which are to be done in groups of two or three students. There are usually five practical assignments in total, which essentially consist in the implementation of a given formulation for an optimization problem and the execution of computer experiments. Appendix B contains one of such assignments as an example, in which students are asked to implement the single commodity based formulation for the traveling salesman problem.

For the year of 2020, the perception assessment of PIFOP was done using the traditional method of a questionnaire, which was applied at the end of the academic

[10]There were actually 33 students enrolled in the course, which were all using PIFOP, but only 18 students were willing to answer our questionnaire.

semester. The questionnaire consisted of six closed questions: five of them being traditional multiple choice questions, and the sixth being a Likert-based question (Likert, 1932) with eleven scale items. We also had two open questions which the respondents were not required to answer. These two open questions gave the respondents the opportunity to i) report bugs and issues they had with our tool, and ii) suggest new features and improvements to it. See Appendix C for the full questionnaire.

We've devised each question and scale item with one of three goals in mind: i) the characterization of the test group, ii) the usage pattern identification and iii) the evaluation of PIFOP features from the user's perspective.

6.3 Results and discussion

As we talk about the questionnaire results we will be referring to its questions by their number, which can be seen in Figures 6.1 and 6.2 along with the responses. Throughout the text, when we refer to the responses of question six, the reader will note that we have grouped together agreement levels 1 and 2, as being those who disagree with the assertion, levels 4 and 5, as those who agree with the assertion, and level 3, those who are neutral to the assertion. This makes our report and analysis easier to read, but if the interested reader would like to know the detailed level of agreement of the respondents, they should refer to Figure 6.2 for it.

Test group characteristics and usage pattern

We have already mentioned one of the characteristics of the test group, namely that the students were all enrolled in the OR-Lab course. This gives us some information about their experience with computer programming in general and mathematical programming in particular. At the time, the UFMG Production Engineering department placed this course in the fourth academic semester of the undergraduate program. Regular students taking the OR-Lab course is assumed to have completed two courses on general computer programming, namely *Algorithms and Data Structures I and II*, and it is at the point of being first introduced to operations research theory and practice.

Because of the program structure it is usually the case that a student enrolled in the OR-Lab course will be working with AMPL for the first time. But, for various reasons, this ought not be the case, e.g, if students happen to be repeating the course. The first question removes any doubt in this regard: all the respondents were having their first contact with AMPL.

The second question tells us about what has been their preferred browser for accessing PIFOP. This was important for us to know because at that point of PIFOP development we only officially supported Mozilla Firefox and Google Chrome, so if it happened that the user of another browser revealed a significantly worse experience we would have to take that into account in our interpretation of their responses. Google Chrome was the most popular, used by 13 (72%) of the respondents, followed by Mozilla Firefox and Microsoft Edge, each with 2 (11%) of the participants, while only 1 (5%) used some other browser. Fortunately for us we have not identified any significant difference in user experience across the different browsers, whether they were officially supported or not.

Question three shows us the frequency with which the students were accessing our tool. Most commonly (8 or 44%) the participants have accessed PIFOP approximately once a week, which is in accordance with the fact that the OR-Lab classes happened once a week as well. One possibility for divergent access frequency from this is the fact that different students have different levels of interest in the various subjects of a program, which can be translated into their engagement with a given course. In our case, 6 (33%) of the students presented a higher course engagement, accessing PIFOP two or more times a week, and 4 (22%) were less engaged, accessing PIFOP once every two weeks or less.

The first item of question six help us to see how familiarized were the students with the type of user interface we provide in PIFOP. The responses showed that there was no homogeneity in this regard. The group was composed of students with varying levels of experience with integrated development environments, 6 (28%) being familiar with them, 8 (44%) not familiar and 6 (28%) more or less familiar.

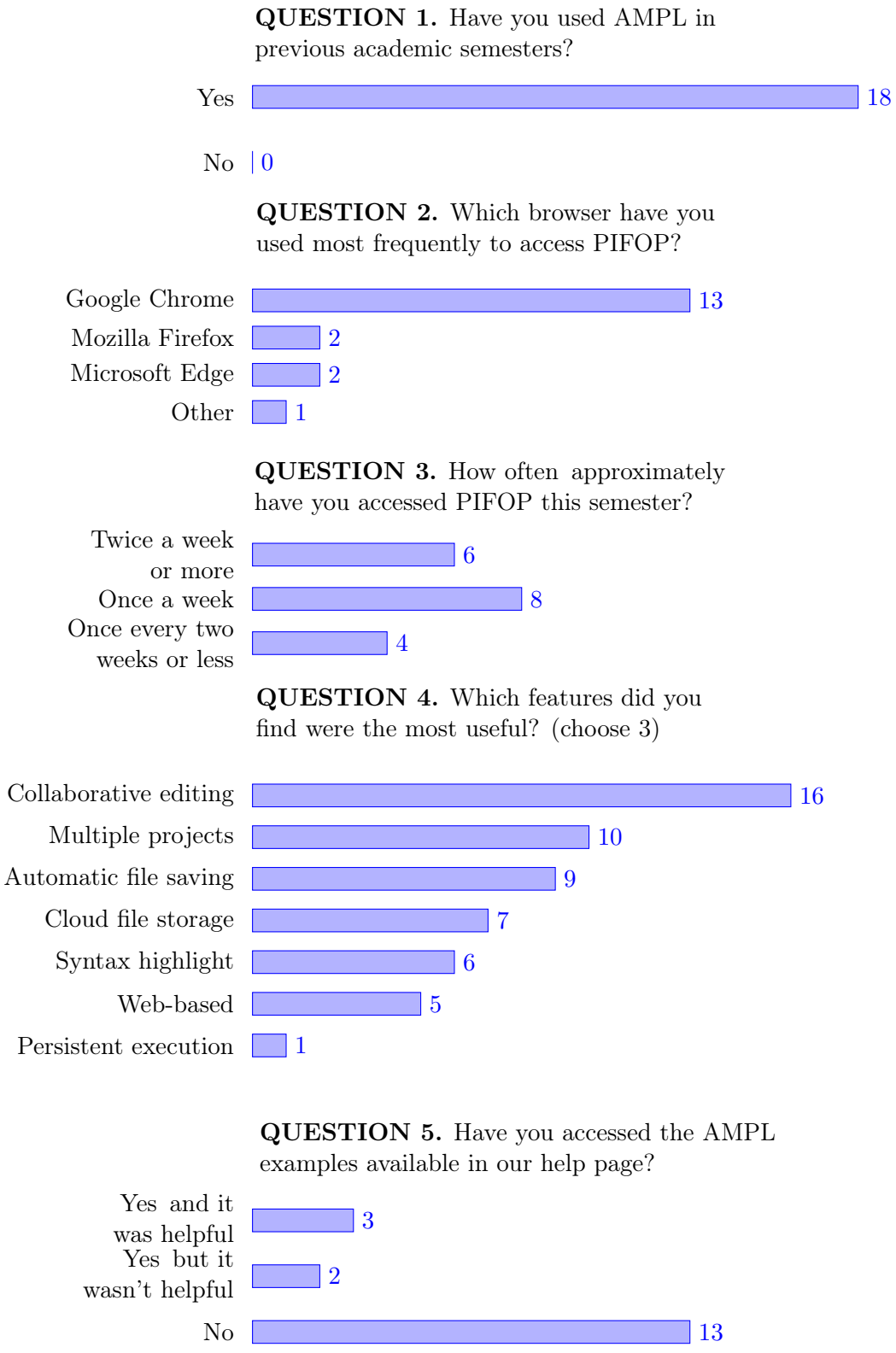

Figure 6.1: Summary of responses for question 1 to 5.

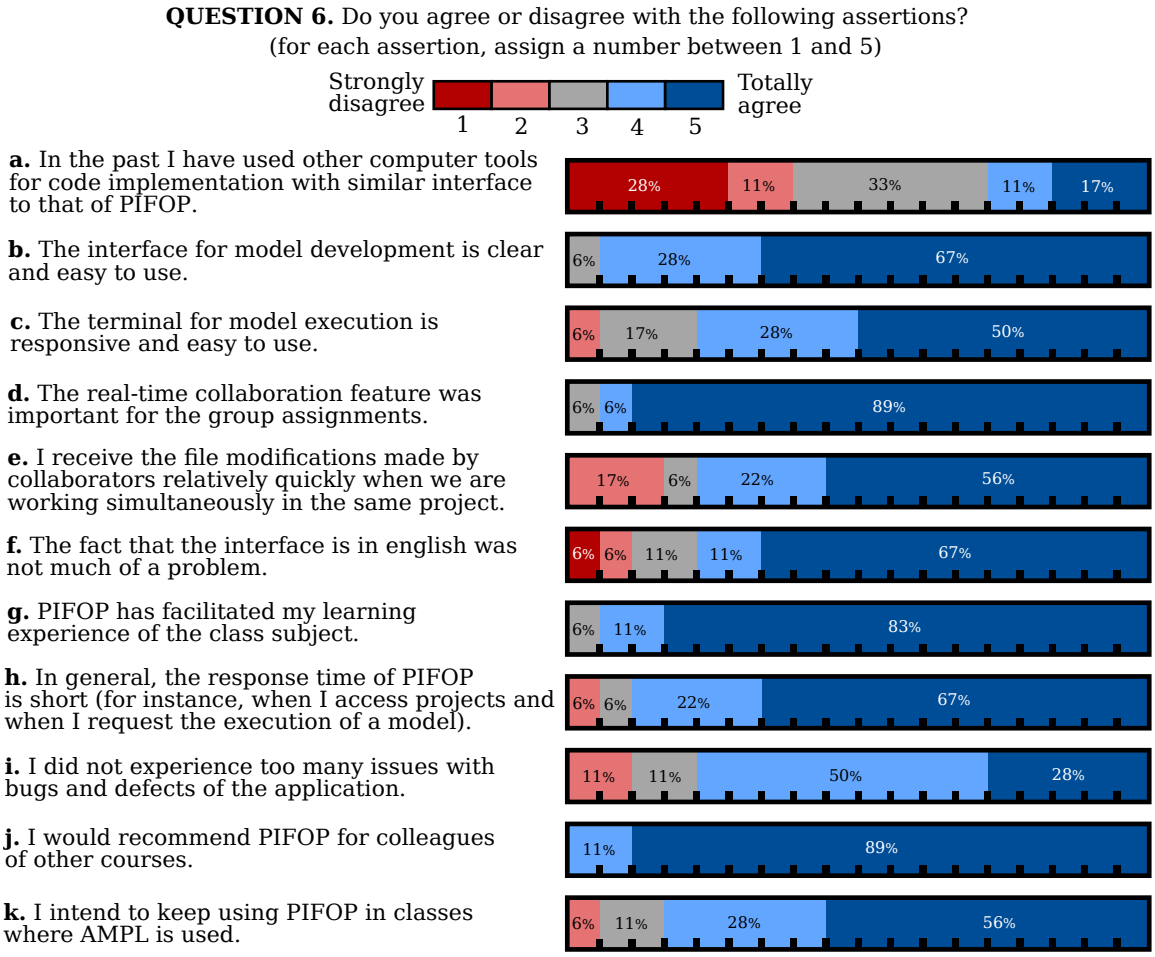

Figure 6.2: Visual summary of the responses for question 6. The width of each smaller rectangle inside a bar corresponds to how many of the 18 respondents have chosen the level of agreement indicated by the color.

User perception assessment

All the remaining questions and scale items were devised for gathering user experience information, which is the whole purpose of this study. In question four we have provided the participants with a list of seven features, and asked them to select which three features they have found to be the most useful. Collaborative editing leads the ranking, appearing in 16 (88%) of the responses. This is what we would expect, since the students were required to work together in multiple as-

signments. This number also shows that our implementation of the collaborative editing feature as described in Chapter 4 has been successful, as it can satisfactorily meet the demands of a small group of modelers working together.

In second place we have the ability to create multiple projects, which appears in 10 (56%) of the responses, followed by automatic file saving in 9 (50%) of the responses, cloud file storage in 7 (39%) of the responses and syntax highlight in 6 (33%) of the responses.

It is somewhat surprising for us that the fact that PIFOP is web-based would rank so low in the results, appearing only in 5 (28%) of the responses. This contrasts with Valez et al. (2020), in which survey the web-based feature of *Kodethon* was ranked first by the participants. It is possible that it was not clear enough for the participants what does this characteristic of the application entails, such as 1) the fact that it does not require download and installation of a software, and 2) the fact that it is readily accessible from any of the user's computers. But it is also possible that they simply thought that this is not so much of a benefit: if a desktop application were to provide the other features, the need for download and installation would not hinder them from using it. In any case, it is understandable that five of the other features would rank above this one, since if it were not for them the application would not be very useful anyway.

What is not surprising at all is that persistent execution has been the least useful feature according to the students. This feature gives the user the ability to leave PIFOP while there are still jobs running in their projects. This is only useful for users that are running time-consuming jobs, which takes hours to finish. But this is not the case for undergraduate students in an introductory course on operations research. Most jobs executed for the assignments would take seconds or, at most, a few minutes to complete, which explains why persistent execution has appeared in only 1 (6%) of the responses.

Question five concerns the help provided by the AMPL examples available in our web page. It must be said that we have not at any moment told the students about these examples, which would only be found by those who happen to be browsing thorough our website. We believe that this is why only 5 (28%) of the students have accessed the examples, but this is too small of a sample to reach any conclusions about the usefulness of this resource. At any rate, as a future work, it

would be interesting to know from students that access the examples but don't find them helpful what they were looking for that they did not find in the examples. This would help us to produce better guides and introductory resources.

Now we turn our attention to the scale items of question six. We've already discussed item **a** in the previous section. Items **b**, **c** and **f** refer to different aspects of the graphical user interface. In relation to the model development interface (item **b**), i.e., the editor and the file explorer on the left side of the application, 17 (94%) participants have considered it clear and easy to use, whereas 1 (6%) participant was neutral to the assertion. In relation to the terminal used for model execution at the right side of the application (item **c**), 14 (78%) participants have considered it to be responsive and easy to use, 1 (6%) have thought the contrary and 3 (17%) were neutral to the assertion. In relation to the interface language being English (item **f**), 14 (78%) participants did not find it to be a problem, 2 (11%) did find it to be a problem and 2 (11%) were neutral to the assertion. The responses for these three items shows that in general the evaluation of the interface has been positive, although it can be improved by providing multiple interface language options and a more responsive and easier to use terminal.

Items **d** and **g** refer to the usefulness of PIFOP as a tool to be used in modeling courses. In item **d** we evaluate specifically if the collaborative editing feature was important for the group assignments. 17 (94%) of the participants did find it important, whereas 1 (6%) participant was neutral to the assertion. In item **g** we evaluate if the application in general has facilitated the learning experience of the class subject. 17 (94%) of the participants did think it facilitated their learning experience, whereas 1 (6%) participant was neutral to the assertion. These results show that PIFOP is very much capable of meeting the needs of undergraduate students in introductory mathematical modeling courses, allowing the students to better assimilate the class subject.

Items **e** and **h** refer to PIFOP response times of the collaborative editing feature in particular and of the application in general. 14 (78%) of the participants agreed that edits made by collaborators were received relatively quickly (item **e**), while 3 (16%) disagreed with that assertion and 1 (6%) participant was neutral. As for PIFOP's features in general (item **h**), 16 (89%) participants considered PIFOP to have a short response time, 1 (6%) participant did not consider the response time

to be short and and 1 (6%) participant was neutral to the assertion. The responses for item **e**, although mostly positive, indicate that the delay between the edits of a user and the update on a collaborator's screen has been bothersome for some students, which is something for us to investigate. It may or may not be related to our server, because it can also be related to the user internet speed. But if we take into consideration the overwhelmingly positive evaluation of the collaborative editing feature (questions 4 and 6.d), the delay may not be that much of a problem. In general, as indicated by item **h**, the application performs well.

Item **i** is related to the presence of bugs and defects in PIFOP. 14 (78%) of the participants did not experience too many issues with bugs and defects in the application, while 2 (11%) disagree with that assertion and 2 (11%) participants were neutral. Related to this item is the optional question seven, which was an open question asking if the respondent would like to report an issue. Unfortunately, those who have disagreed with item **i** have not reported any issues, which makes it difficult for us to solve them. Five issues have been reported: two related to not receiving collaborators edits (being necessary a page refresh to fix it), two related to edits not being saved and one related to the text editor not properly displaying the text cursor. The respondents that have reported issues in question seven were neutral to the assertion of item **i**.

Items **j** and **h** evaluate how useful the respondents think PIFOP would be for other courses in which AMPL is used. The responses for item **j** show that all participants would recommend PIFOP for colleagues in other courses. As a matter of fact, we estimate that during the academic semester around 44 new users have come to PIFOP after recommendation of the students. We've arrived at this estimate by counting the number of our users that have registered during the academic semester that were not part of the test group nor have come by recommendation of the authors of the tool. Since the URL for accessing PIFOP was only known by PIFOP users, most likely these 44 new users come by recommendation of one of the 45 PIFOP users (that is, the 33 students of professor Carlos plus 12 other testers we had at the time). This means that, on average, every user has brought around one more user with them, which is consistent with item **j** responses.

Item **h** informs us about the intention of the users to keep using PIFOP in future courses in which AMPL is used. 15 (83%) participants intend to keep using

PIFOP, 1 (6%) do not have that intention and 2 (11%) are neutral in this regard. We note that the UFMG Production Engineering program at the time, although it offered many courses based on mathematical modeling, most were not required for completing the program, and the use of AMPL was not always required by the instructor. These factors may or may not have influenced the the answers for this item, since the respondent may not envision themselves taking other modeling courses as they are not aiming to be operations research practitioners. At any rate, we are pleased to know that most students intend to keep using PIFOP, not because of it being required by the instructor, but of their own volition.

Finally, in question eight, which was another optional open question, we've asked the respondents if they missed features on PIFOP. Two answers were given. The first missed a button in the application screen to return to the initial screen. The button already existed, on the top right corner of the screen, but the user did not find it, which indicates to us that, for some users, it is not clear enough what that button does just by looking at it (the button only has the name of the user written on it). The second answer suggested that the application could "detect errors before compilation". This is something many IDE's do for general programming languages, and we agree that it would be a very useful addition to PIFOP. Although we don't have plans for it to be added in the near future, since it is not a trivial functionality to implement, it is something we would like to add at some point.

6.4 Conclusion

The objective of this study was to provide an answer to the question: **(RQ3) What are the student's perception of PIFOP as a tool for collaborative work and optimization model creation?** Overall, we can say that PIFOP is positively perceived by the students, more specifically, of the undergraduate level. To give a more complete answer, let us summarize our findings about students' perception regarding 1) the application interface, 2) its features, 3) its performance and 4) its ability to meet undergraduate students' needs.

First, the part of the application interface for model creation and edition has been considered to be clear and easy to use by most students (94%). Other aspects

of the interface, such as the language and the terminal, although have not received the same level of approval, have still been well evaluated.

Second, according to 88% of our test group, collaborative editing is one of the most useful features of the application. The persistent execution feature has not been very much appreciated, since the group have not had the need to use it. As a future work, we would like to assess the usefulness of this feature with a test group that would actually benefit from it (graduate students, professionals, etc).

Third, in general PIFOP has been considered to be performant by 88% of the students. Some issues have been experienced with the collaborative editing feature, but still the performance of this feature have been approved by 78% of the group.

Fourth, PIFOP has been able to meet the students' needs, as have become evident from the fact that 94% of them thought the application has facilitated the learning experience and that 100% of them would recommend for other courses in which AMPL is used.

The results of this study are very much encouraging and shows the potential of the tool we have in hands. With the responses of this survey, we can better direct our efforts towards the improvement of the features currently provided as well as the design of new features.

We are thankful to professor Dr. Carlos R. V. Carvalho for accepting the challenge of pioneering the use of a newly created web application in his classroom, and are also thankful to the students for patiently enduring the issues of PIFOP during their course.

Chapter 7

Common issues experienced by AMPL modelers

In this chapter we turn our attention to our last research question: **(RQ4) What are the most common issues faced by AMPL users?** We begin by talking about our motivations in Section 7.1. Then we proceed to present related works in section 7.2. In Section 7.3 we talk about our data source and our methodological choice for classifying the issues described in this source. The categorization scheme is presented in Section 7.4 along with the number of issues in each category. We follow that with a discussion of the results in Section 7.5. Finally, we end in Section 7.6 with some final remarks on what we have achieved in this study and what are our plans for future works.

7.1 Motivation

On first sight, our research question may look rather unambitious because all there is to the answer is the enumeration of the types of issues APML users experience and the counting of them. Indeed, these are the necessary steps to answer our research question, but when there are no frameworks to use as reference on how to classify issues experienced by AMPL modelers or by users of any algebraic modeling language in general, nor a data source that can be easily mined via computer tools, the task, which appeared to be simple at first, turns to be a great

endeavor. At any rate, the value of what we are doing is better perceived by taking into consideration 1) the larger project from which this research is only a part of, and 2) the novelty of this kind of work in the literature on mathematical optimization practice.

In the introduction we have mentioned our intention to produce instructive content that can guide AMPL users in solving the issues they come to face with the language. One idea is to create a bank of answers to frequently asked questions, following what what Cline et al. (1998) and Peavy (2009) have done for C++, Hornik et al. (2002) for R, and Kanerva (1997) for Java. Alternatively, the instruction can come in the form of longer guides addressing different aspects of the language, such as those that Overleaf has created for LaTeX[1]. But the decision of the form that the instructive content will take can be postponed for later stages of this project. If we are going to create a FAQ or a series of guides and tutorials, this is not our present concern. First, we need to know what topics should we cover when we produce this material. The answer to our research question will help us to produce a content that is actually relevant for AMPL users, especially for undergraduates.

As for the coverage of this topic in the literature, the issues faced by modelers when implementing mathematical optimization programs in algebraic modeling languages (AMLs) has not received much attention. In fact, the practice of implementing mathematical models using AMLs in general is not an activity that has been the focus of any study that we know of. Surely people are using AMPL, GAMS and other AMLs to solve optimization problems, as it is attested by the papers in any operations research journal. But what are the struggles modelers go through to make their implementation work? What problems do they face and how are they overcome? And how can teachers better equip the students with the skills necessary to both prevent and solve implementation issues? We believe that research on these topics would greatly contribute for the adoption of AMLs in operations research courses and in industry practice.

We do not have a definitive answer for why it is the case that the activity of model implementation in AMLs has not received much attention in more than 30 years of AMLs existence. One possibility is that there are already plenty of

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

works addressing the challenges in the activity of general programming, which is a similar activity to that of mathematical programming using AMLs. But we argue that there is enough difference between the two that they should be considered separately. Programs in general programming languages and in programs in AMLs have different types of constructs, different purposes, different capabilities and different computational demands, so the issues that emerge in one activity may never arise in the other. For instance, a modeler may never experience a buffer overflow write issue, while a programmer may never find themselves debugging the cause of a theoretically infeasible solution of a linear problem. As pointed out by Berry (2013), the activities of mathematical modeling and programming are distinct enough that programmers may never do any mathematical modeling, and modelers may never do much programming. So the issues that emerge in the mathematical modeling activity will only hardly ever be addressed in the literature on general programming.

Another explanation for this seldom presence in the literature may be because AMLs have not been the preferred technology among teachers to be used for teaching students how to use computers to solve optimization problems. Kallrath (2012) points out that many optimization courses opt for the use of spreadsheet software (such as Excel) to solve operations research problems. A quick look in articles from the past decade in the *INFORMS Transactions on Education* journal shows that this still seems to be the case. Williams et al. (2018), Carroll and White (2017), Willoughby and Teare (2017), Nurre and Weir (2017), Isken (2014), Liu et al. (2013) all involve the use of spreadsheet optimization approaches in university courses for solving different kinds of operations research problems. To be sure, AMLs are used in many places, both in the academy and in the industry, but still it seems that the predominant use of spreadsheets has made AMLs a comparatively uninteresting topic of discussion among educators. We, however, are convinced that we can greatly contribute to the adoption of AMLs in university courses by providing better guidance to both beginners and experts on how to prevent and overcome common issues in AMPL modeling.

7.2 Related work

Although mathematical programming and general programming are two distinct activities, we do recognize that there are many similarities between the two, and for that reason we think that the findings of the literature concerning the issues faced by programmers are most relevant to the discussion of this chapter. It should not be surprising that most works in this section are from the field of computer science rather than from the operations research one. As mentioned previously, the activity of implementing mathematical optimization programs in AMLs has received only cursory attention in the literature.

One noteworthy exception is the work of Pannell et al. (1996), which discusses principles and strategies for debugging mathematical programming models. Based on their experience using and maintaining a large optimization model, the authors identified types of formulation bugs and their symptoms, and developed guidelines and strategies to both prevent and solve these bugs. The authors presented the discussion in a language-agnostic way, focusing not on any one AML in particular but rather on model formulation issues in general.

Since then, in the field of mathematical optimization, the only other work that somewhat relates to what we are doing is that of Brown and Dell (2007), which compiled a number of small guides on how to model different situations that are commonly seen in optimization problems, such as common types of constraints and common types of indexing. The purpose of these small guides is to prevent common mistakes modelers struggle with during the formulation phase of an optimization project.

Undoubtedly, both works contributed for our understanding of the issues experienced by modelers in the formulation phase of an optimization project. And it is an essential contribution, because in order to have a working AML implementation of a model, it is certainly required that the model formulation is correct. However, an optimization program in an AML language is not limited to the description of a model formulation. The program may also read and write files, dynamically generate data, perform all sorts of calculations, interact with a solver, include and remove problem constraints, and do a variety of other operations that the modeler may struggle with. It is not enough to have a correct model, it is also necessary

to have a working optimization program. The present work differs from the above ones in that we include implementation issues as well as formulation issues in our discussion.

The types of issues mathematical programmers and general programmers experience are not exactly the same, but there certainly is an overlap between the two. The most obvious kind of difficulty present in both optimization and general programs is difficulties with the syntax: the rules of a language must be strictly followed. As for the non-syntax-related difficulties of general programmers, different authors have categorized them in different ways (McCauley et al., 2008). We will see that some categories of issues can be found in optimization programs, while others are specific to the mathematical programming activity.

Authors looking to find out what are the common challenges faced by programmers have applied different methodologies to complete the task. Bajaj et al. (2014) presented a study on common challenges and misconceptions among web developers. Their approach has been to classify *Stack Overflow*[2] questions concerning the three main web languages: JavaScript, HTML5 and CSS. For each language, they've found that each question would fall into one of 15 (or 17 in the case of HTML5) categories, ending up with 47 categories in total. The most common topics of discussion were "cross browser compatibility" for JavaScript, "Canvas API" for HTML5 and "DOM layout" for CSS. A similar approach can be seen in Pinto et al. (2015), which concerns the most popular questions about concurrent programming and also uses *Stack Overflow* as their data source.

Bryce et al. (2010) sought to understand the problems for which the students in an introductory computer science course ask tutors for assistance. They have divided the issues into 20 categories. Most issues would fall into the "problem solving" category, which is the inability to understand a problem and/or how to use a programming language to implement a solution, followed by the "loops and switch statements" category in second, and by "arrays" in third place.

Lahtinen et al. (2005) studied the difficulties of novice programmers by analyzing an international survey with more than 500 students/teachers. When it comes to the programming concepts that the participants found to be the most

[2] https://stackoverflow.com

difficult to learn, "pointers and references" occupy the first place, followed by "error handling" and "abstract data types" in second and third places, respectively.

The previous works focused on challenges faced by novice programmers. Donald Knuth, the creator of TEX, differs from those in the fact that his focus was on mistakes made by an expert (himself). Knuth (1989) classified the errors he has committed during the 10 years of TEX development. Based on a log of changes he has made to the program, the author divided the errors into 15 categories. Most commonly, the bugs would fall into one of three categories: algorithm awry (meaning the original solution was insufficient or incorrect), blunder or botch (where he knew what to write, but wrote something else) or data structure debacle (meaning the usage of variables or structures was incorrect or insufficient).

Other works focused on the identification of common issues experienced by programmers can be found in the literature review on debugging of McCauley et al. (2008), which discussed not only types of bugs but also why they occur, what are the strategies to identify and solve them and how can we improve the learning and teaching of debugging.

7.3 Methodology

Our goal is to identify common issues experienced by AMPL modelers. In order to achieve this goal, first we need a collection of issues that AMPL users have reported experiencing. The characteristics of the data source we have selected have influenced our methodological choice, for reasons we shall briefly discuss. After that, we will present the method we have chosen to identify and classify recurring themes in our data source.

Data source

There are three well known online platforms in which AMPL questions can be publicly asked and answered by anyone within the community. First, there is the *AMPL Google Group* [3] (AGG), which, according to the AMPL website[4], is the

[3]Website: https://groups.google.com/g/ampl
[4]https://ampl.com/about-us/contact-ampl

official communication channel for asking for help with AMPL and its supported solvers. Second, there is the *Operations Research Stack Exchange*[5] (ORSE), which aggregates questions in the field of operations research in general. Third, there is the already mentioned *Stack Overflow* (SO), which aggregates questions related to programming in general.

The AMPL Google Group is by far the most used communication channel within these three to ask AMPL related queries. From January 2020 to January 2021, around 540 discussion threads were opened at AGG, while only 13 questions with the "AMPL" tag have been asked at ORSE in the same period and 48 at SO. The fact that AGG offers us a larger data set of AMPL related questions and issues is the main reason why we have selected this as our data source.

Unlike ORSE and SO, AGG has no tag system, filters, the concept of an 'answer' or up votes and down votes. These features would be helpful for making a preliminary selection of relevant discussion threads, but without being able to even filter questions by its view count, we are very limited on what we can achieve with automatizing. For this reason, we thought that manual classification methods would be more adequate for this study.

The decision on the number of threads that we would analyze was purely based on what we thought would be manageable with the time frame we had. 500 hundred threads is the number we have arrived at, based on the pace at which we could read and classify the threads. Since this number was close to the total number of active threads between January 2020 and January 2021, we decided to read all threads in that interval, which were 543 in total[6].

Related works about difficulties experienced by programmers usually can be classified by the level of expertise of the programmers: novice or expert. In our case, an important characteristic of our data source is that the level of expertise of the authors of the threads varied. Some discuss topics of great complexity, others ask how to fix simple syntax errors. Rather than concentrating on issues experienced by novice or by experts, this study is concerned with issues experienced by AMPL users in general.

[5] https://or.stackexchange.com

[6] Threads that we quickly identified as not being relevant to our purposes, such as duplicate questions and AMPL announcements, were immediately filtered away. This number represents the remaining threads after *after* this filter being applied.

Classification method

Card sorting is a well known and widely used technique to derive categories and taxonomies from input data. Simply put, the technique consists in sorting a collection of cards, each containing some data, into categories that make sense to the sorter. This method can be applied with a variety of different purposes. In the field of Information Architecture, card sorting is often applied to gain insight into how the sorter thinks. It allows information architects to better structure websites to match the user thinking pattern and to enable users consistent access to the correct data (Resmini and Rosati, 2012).

Although the technique has indeed been predominantly used in that field and for that purpose (Righi et al. (2013), Nawaz (2012), Camargo (2010), Bussolon (2009), Hannah (2008), Barker (2014), Spencer and Warfel (2004), etc), its usefulness has been proven in a variety of different fields of study, e.g., Information Science (de Oliveira Solano and Rocha (2015), de Faria (2010)), Cartography (Roth et al., 2011) and Computer Science (Bacchelli and Bird (2013), Nurmuliani et al. (2004)). In our case, the card sorting technique will be applied to derive categories of issues experienced by AMPL modelers.

Just as there are no prescribed goals that should be pursued when applying the technique, the card sorting exercise can also vary in type (closed or open), number of sorters (individual or groups), number of cards, sorting analysis and many other aspects, as pointed out by Hannah (2008). Since the literature shows a great number of variations on the technique, we will limit ourselves to explain our own implementation of it and the reasons for our decisions.

We've applied an *open* card sorting technique, which differs from the *closed* type in that the card sorting in the first one starts without any predefined categories, while in the second one the cards are sorted into categories defined before-hand. Since we are pioneering studies in this area, no one as of yet have suggested how AML modeler issues should be categorized, so an open sorting approach both meets our need of deriving issue categories, and our goal of reporting how often issues in each category are experienced.

The task of deriving categories of AMPL issues has been split into two card sortings performed subsequently. The "cards" of the first sorting were the AGG

threads themselves. The most basic assumption we had while doing the sorting is that any kind of issue can be reduced to the lack of some kind of knowledge. For each thread, we would ask ourselves several questions, trying to identify what kind of information or knowledge is the thread author in need of. For instance: "In what kind of AMPL statement is the error occurring?" If it happens in an assignment statement, the root of the issue is that the user lacks the understanding on how to properly use or write that kind of statement. "What goal is the user trying to achieve, but is having difficulty in achieving it?" If they are having trouble reading data from a spreadsheet file, the root of the issue is their lack of knowledge on how to properly use the `amplxl` DLL extension. "What solver is being used and what is it returning?" If the solver reports that the problem is infeasible, the root of the issue is most likely in the mathematical formulation, and not directly related to any aspect of the AMPL language, which indicates a lack of understanding by the part of the modeler on what are the implications of the constraints and objective function used to model the problem.

This procedure of asking questions about the roots of the issue being experienced allowed us to connect most threads with one or more "sources of confusion" (Brown and Dell, 2007), which are the missing pieces of information or knowledge that the user needs in order to solve a problem or to accomplish whatever they are trying to accomplish. The source may or may not be the lack of understanding of aspects of the AMPL language itself. It can be a lack of knowledge on mathematical optimization theory, or on the workings of an operating system, and so on. In order to identify the missing pieces of information, we have read both the questions being asked and the answers being given, knowing that a correct answer would give us precisely what we were looking for, that is, the knowledge that the AMPL user is in need of, which is the whole reason why the issue is being reported in the first place.

In practice, by the end of the first sorting we had listed on a spreadsheet all the links to the AGG threads selected for this study in a column and in the column to the right we had one or more descriptive tags regarding the sources of the issue being reported. For instance, if one source of the issue was a lack of understanding on the features of the KNitro solver, that thread would receive the tags "solver" and "knitro". If another source of the issue was a lack of understanding on how to

omit variables with value zero when using the display command, the thread would receive the tags "display" and "omit zeros". We had no rules on how specific or how broad these tags should be. The only rule was that the tag should inform us about the main pieces of information that the author of the thread was in need of. Some of them were applied to many issues, such "solver" and "constraint declaration", but most of them were very specific to the issue being raised, such as the "benders decomposition implementation" tag, which appeared only once.

The next step would be to sort these tags to derive broader issue categories. In preparation to this second card sorting, we have made a second pass over the tags, specially the very specific ones, to see if the threads they applied to could be tagged with broader tags. For instance, the thread tagged with "benders decomposition implementation" would also receive the "algorithm implementation" tag, which was already being applied to threads in which the author was looking for help on how to implement a given algorithm. In this second pass we also looked for tags that described the same thing but with different words. In these cases we decided to keep one and discard the other, replacing the occurrences of the discarded one with the preserved one. For instance, we had the tags "warm start" and "initial solution". Since they both refer to the act of setting a starting solution to the solver, we have preserved the "initial solution" tag and replaced the occurrences of "warm start" with it.

After this second pass, we had a final list of 226 tags were that used as the "cards" for the second card sorting. The goal of this second sorting was to finally derive meaningful categories of issues experienced by AMPL modelers. To keep up with our long term goal of producing guides for AMPL users and to help us with the task of naming the categories being formed, during the sorting process we tried to answer the question: "How would we categorize the guides with the information needed to solve the issues being described by these tags?".

Our procedure in this second sorting was essentially the same of the first sorting. We had listed on a spreadsheet all the tags obtained from the first sorting. Next to each tag we would put the name of the category we thought that tag belonged to. We took the number of occurrences of the tags as a hint on what categories we should create. For instance, "solver" was the most common tag, appearing 66 times. Thus, that tag alone suggested a "solver" category, which we ended up

calling "solver usage and interaction" to give it a more descriptive name. Other categories, such as "translation" and "formulation and theory", have also been derived in the same way. During the process, as more categories were being created, we have given preference to including a tag into an existing category rather than creating a new one. By the end, we had 17 categories. After a second pass over them, merging and renaming categories, the number was reduced to 13, which will be described in the next section. The whole procedure described in this section is summarized in Figure 7.1.

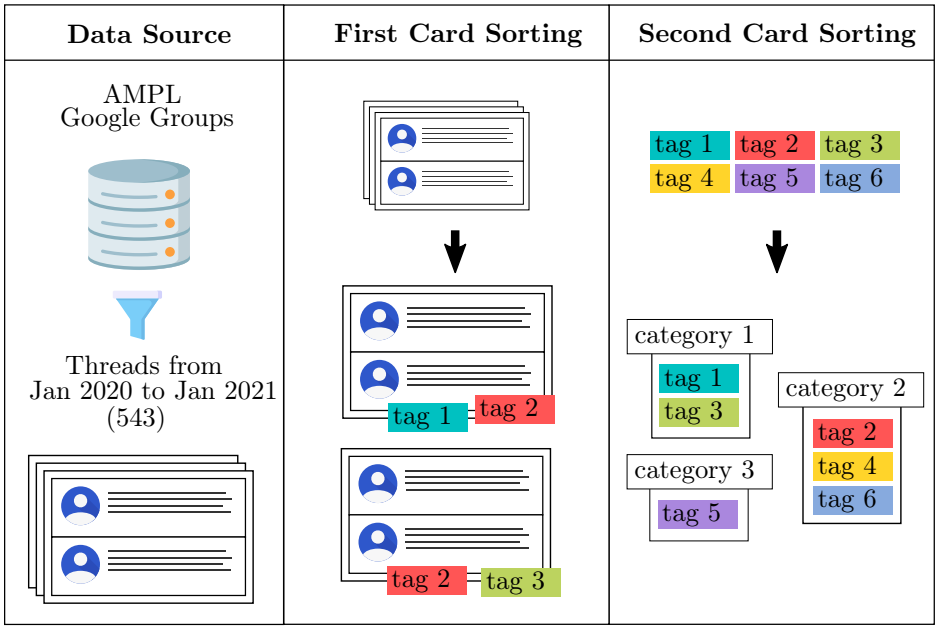

Figure 7.1: Steps taken to derive issue categories from our data source. The input for the first card sorting was the 543 AGG threads and the output was 226 descriptive tags. These tags became the input for the second card sorting, from which 13 issue categories were derived.

7.4 Issue Categories

These are the categories we have arrived at by the end of our sorting exercise:

[6]Icons created by `Freeplk` and `bqlqn` from `flaticon.com`.

1. Model entities (27.19%)
2. Solver usage and interaction (17.88%)
3. Formulation and theory (13.97%)
4. AMPL APIs, extensions and external tools (11.73%)
5. Data assignment (10.61%)
6. Indexing expressions and subscripts (9.87%)
7. Functions and operators (7.26%)
8. Translation from algebraic notation (6.89%)
9. Table reading and writing (6.70%)
10. Scripting and algorithm implementation (5.40%)
11. Performance (3.91%)
12. Printing and output formatting (2.98%)
13. File reading and writing (1.86%)

The percentage in parenthesis is the percentage of threads that would fall into that category due to it being tagged with one or more tags of that category. Since a thread could be tagged with multiple tags, it could also be included in multiple issue categories. For instance, a thread tagged with "constraint declaration" and "invalid subscript" falls into both "Model entities" and "Indexing expressions and subscripts" categories.

As a general rule of interpretation of the names of these issue categories, they indicates a type of knowledge the authors of the threads within that category are lacking, which is preventing them from accomplishing whatever they are trying to accomplish. Since this lack of information is the root issue of the problem being experienced, the categories can be properly called "issue categories". Below we explain each one in more detail and describe what types of issues fall into them.

1. Model entities (27.19%)

Every AMPL program involves the declaration and usage of different types of model entities, such as *sets, parameters, variables, objective functions, constraints* and, sometimes, *problems*. Most of the issues in our data source arise from the improper declaration/usage of these entities. Issues in this category are often related to the syntax of a malformed entity declaration statement, but they can also occur due to the lack of conceptual understanding, such as the difference of *sets* and *parameters*, when and how to use an *indicator constraint*, how to work with multiple models in the same program — in which case a *problem* entity may be required — and what kinds of introspection mechanisms are available for getting information about the implemented model, such as the number of variables and constraints, which are available as built-in AMPL parameters.

The number of occurrences of the tags within this category shows that *constraints* are the most "problematic" type of model entity. 41% of the threads within this category were tagged with "constraint declaration", which was applied when there was an error in the syntax in a declaration statement of a constraint or when the modeler needed help in expressing a given constraint in AMPL.

2. Solver usage and interaction (17.88%)

A great number of issues arises from the lack of understanding on how to use and interact with the solvers. There is a variety of things that a modeler may want to instruct a solver to do and a variety of information that they may want the solver to return. Issues in this category include the lack of understanding on how to select a solver, how to suppress, redirect or format its output, how to set a time limit, what types of variables and constraints the solver is able to handle, how to interpret a solver message, how to relax or restore integrality constraints, and so on. We also included here cases in which the user wanted to retrieve extra information from a `solve` call, such as the dual values and reduced costs of variables, which are accessed using model entity *suffixes*.

If the thread discussion was about a specific solver, it received tag with the name of that solver. CPLEX was the most used solver tag, appearing in 17% of

the threads in this category, followed by KNitro (14%), Baron (9%) and Gurobi (7%).

3. Formulation and theory (13.97%)

This category comprises issues that are not directly related to AMPL, but rather relate to mathematical optimization theory in general and/or with a model formulation in particular. For instance, the modeler may be lacking theoretical background on specific techniques, e.g., Lagrangian relaxation, or there may be a misunderstanding on the characteristics and requirements of different problem classes, such as MIQP and general NLP. But more often, issues in this category are from users that are trying to figure out why their formulation, which is syntactically correct, is returning unexpected results, such as "infeasible" or "unbounded". In other words, the issue is not in the implementation level, but in the mathematical formulation level.

Although the root of the issue is not in the AMPL implementation, AMPL can often be used for debugging a mathematical formulation, for instance, by helping the modeler in the identification of problematic constraints. So the formulation issues in this category may also be thought to arise from a lack of understanding on how to use AMPL to track down a problem in a mathematical formulation.

4. AMPL APIs, extensions and external tools (11.73%)

These issues are related to software that are in the AMPL "ecosystem", that is, software that interacts with AMPL or depends on AMPL in some way or vice-versa. In this category we find issues related to the usage of AMPL's APIs for different programming languages, e.g., Python and C++, issues on how to create or use AMPL DLL extensions, such as `amplgsl` and `amplfunc`, and issues related to different types of tools that interact with AMPL, such as the AMPL IDE, Jupyter Notebooks and the NEOS server.

Most commonly, the threads in this category are tagged with "AMPL IDE", which have been applied to 32% of them. Following closely is the "amplpy" tag, which applies to issues with AMPL's Python API, and appears in 29% of the threads in this category.

5. Data assignment (10.61%)

This category comprises a variety of issues that arise when the users are trying to assign values to model entities. It includes issues on assignment statements found in data files, misunderstandings on how to set default values for *parameters*, how to assign initial values for *variables*, misuse of uninitialized *sets*, and issues with the `let` statement in general. Moreover, often users find themselves in difficulty trying to assign random values to their *parameters*, a type of issue also included in this category.

The threads discussing data assignment issues are mostly related to regular assignment statements on data files. 32% of them have been tagged with "param assignment" and 12% with "set assignment". But issues with random data generation are also common, present in 19% of the threads in this category.

6. Indexing expressions and subscripts (9.87%)

Indexing expressions are used to define an indexed collection of values, being found in a variety of contexts in an AMPL program. They can occur in declaration statements, `for` loops, `display` statements and others. This pervasive presence makes this kind of expression a somewhat common source of confusion. The issues here are mostly about the syntax of this kind of expression: what can and cannot appear in this kind of expression, how to properly write the `such-that` part of the expression, how to access a specific value of an indexed collection and what makes a subscript invalid are examples of misunderstandings in this category.

72% of the threads within this category are also in "Model entities". This significant overlap is explained by the fact that the declaration of model entities, a common source of difficulties, will often involve the use of indexing expressions, and certainly the access of a value in an entity collection will require the use of subscripts.

7. Functions and operators (7.26%)

The AMPL language has a variety of built-in functions and operators that can be used for different purposes, such as arithmetic calculations (`abs, sqrt, min`, etc) and set operations (`union, inter, cross`, etc). Issues in this category involve

the misuse of a function/operator or misunderstanding on how it works: what kind of input a function requires, what is the resulting set of a set operator, what is a valid context for a given function to appear in, and so on.

Since the declaration of model entities will often require the use of functions and operators, it is not surprising that 69% of the threads within this category are also in "Model entities". The `sum` operator is the most commonly misused one, being an issue in 20% of the threads in this category. Other functions with which users have had issues with include the `if-then-else` logical function (sometimes confused with the `if-then-else` statement) and the `abs` function (sometimes misused to express constraints with absolute variable values).

8. Translation from algebraic notation (6.89%)

Every time the thread author was looking for help on how to translate a given formulation or expression from algebraic notation (provided as an image or other file) to AMPL it was tagged with "translation". This is a common thing users are trying to accomplish, and the translation is not always straightforward. Sometimes there is no one-to-one correspondence between a mathematical symbol and an AMPL operator, sometimes in order to translate the indexing expression of a sum auxiliary sets are needed in AMPL, and so on.

The issue of translating a mathematical expression, such as a constraint, is also an issue of how to declare that constraint using the rules of the AMPL language. As a result, 70% of the translation issues are also included in "Model entities".

9. Table reading and writing (6.70%)

AMPL provides features to facilitate reading data from table file formats, such as ".xlsx", as well as writing data in these formats. Issues in this category include syntax errors on `table` statements, misunderstandings related to the usage of the `amplxl` DLL extension, how to properly configure spreadsheet files to be read, and how to write data to spreadsheets.

10. Scripting and algorithm implementation (5.40%)

More often than not, users will not need to do much with the scripting features of AMPL, aside from calling the solver and printing results. But sometimes a more advanced scripting is required for specific calculations, analysis and algorithmic implementation. Issues in this category include difficulties on how to use flow-control statements, such as `repeat while` loops and `if-then-else`, and, in a broader level, difficulties on how to implement a given algorithm, such as a Benders decomposition or a specific calculation the user is trying to do.

11. Performance (3.91%)

Two kinds of performance issues are comprised in this category: time consumption and memory consumption. Threads in this category are from users looking for advice on how to improve the time spent by the solver on a problem, or on how to reduce the memory it takes to solve it, something users might look for after an "out-of-memory" message from the solver.

12. Printing and output formatting (2.98%)

Issues in this category include misunderstandings on the proper usage of the `display` and `print` statements, how to format strings, how to change the precision with which numbers are printed out, and so on.

13. File reading and writing (1.86%)

Aside from using spreadsheets to read and write data, users can also read and write straight from text files. Issues in this category include misunderstandings on how to use the `read` and `write` statements with files, what is causing a "file not found" error, and how to format a file so the data in it can be assigned to a collection of parameters.

7.5 Discussion

We have selected a few topics of discussion to express what we think are the most relevant takeaways of this study. First, we will talk about the categorization scheme we have arrived at. Second, we will talk about the frequency in which the AGG threads fell into each category. Third, we'll highlight some similarities and differences between general programming issues and mathematical programming issues. Fourth, we have some suggestions for mathematical programming educators.

About the categorization scheme

Just as there is no point in discussing what is the *right* way to slice a pie[7], it does not make sense to discuss if a categorization scheme is *correct* or *incorrect*. It is better to judge them by how *useful* they are for achieving the purpose to which they were designed for. In our case, we said from the beginning that this study is part of a larger project, which final goal is to provide helpful guides for AMPL modelers. How can this categorization scheme help us to achieve this goal?

Firstly, it can help us to cover a broad enough range of subjects in our guides so that they can be useful to nearly all AMPL users that may be in need of information on how to solve an issue. We were able to classify the vast majority (97%) of the threads in our data source within one or more of our 13 categories. Those which did not fit any category discuss topics unrelated to the scope of this study. More specifically, either there were no issues being reported in these threads or the issue were too rare. So if our guides cover these 13 topics, there is a good chance that AMPL users facing issues with the language will find some information on our guides that is relevant to the issue they are experiencing.

Realistically speaking, of course, the usefulness of the guides will depend on the actual content we are going to provide, the topics we will cover and the complexity of the issue the user is trying to solve. But as we devise the guides, the categories

[7]Some slices will be larger than others, some will have cherries on top, others will not. A categorization scheme is only one way to group items together.

can help us to keep track of the coverage of subjects we are achieving and decide what topic should we cover next.

Secondarily, an important feature of any information system is the searchability and findability of the information therein (Martin et al., 2010). This categorization scheme can make it easier for AMPL users to find the information they are in need of. Admittedly, the proposed categorization scheme may need some adjustments to meet the way AMPL users other than us think if we decide to turn this categorization scheme into navigation menus. Nevertheless, even if these categories are never exposed to the site visitor, they still may help us in other ways, such as in the development of a search system based on keywords typed by the user, in which case not only the categories would be useful but also the tags we have created in the process.

Most common issues faced by AMPL users

What can we learn from the number of threads in each category? We can say that it gives us an answer to our research question: **(RQ4) What are the most common issues faced by AMPL users?** We say that it gives us *an* answer rather than *the* answer to be realistic about the limitations of this study. We have only considered the issues reported by AMPL users at AGG in the last year, so the reader should take that into consideration when interpreting our findings. But as long as we don't reach out to conclusions that are too far beyond the scope of this study, we should be able to provide some useful information for those interested in an answer to this question.

The three most recurring types of issues reported at AGG in the last year are 1) model entity issues, 2) solver issues and 3) formulation/theory issues.

By model entity issues we mean difficulties users have in properly declaring and using AMPL entities, such as *sets*, *parameters*, *objective functions*, *variables* and *constraints*. These difficulties are often related to the syntax of the language, but can also occur due to misconceptions about what can and cannot be done with an entity, where it can appear, how it differs from other types of entities, how it can be used to meet a given purpose, and so on.

By solver issues we mean difficulties users have in using and interacting with

solvers. These difficulties include issues on how to instruct a solver to do this or that, how to get the solver to return some information, how a specific feature of a solver can be enabled or disabled, which solver is fit for a given problem type, and so on.

By formulation and theory issues we mean difficulties users have which are due to a lack of understanding of the implications of their problem formulation in particular, or a lack of knowledge on mathematical optimization theory in general. These include issues on how to fix a problem formulation that the solver is returning to be infeasible, misconceptions about classes of optimization problems, difficulties understanding theoretical aspects of optimization techniques, and so on.

Aside from answering the proposed question, these numbers also tell us something about the information needed to solve these issues that is currently available online. Assuming that before going through the trouble of writing a post explaining their difficulty and waiting a day or two for a response most users will look for an answer elsewhere in the internet, we can see what kind of information sources are being most often sought after by the AGG users.

Undoubtedly, the AMPL book (Fourer et al., 2002) is the most complete, reliable and well known source of information about AMPL freely available on the internet. Most likely, it contains relevant information to solve any kind of model entity related issue. Not that the information therein should be sufficient for any issue of this kind, but it will be relevant nevertheless. If our assumptions are correct, that makes the information contained in the AMPL book the most sought after by the AGG users.

But one thing is for an information to be relevant and sought after, and another thing is for it to be *findable* (Martin et al., 2010). Searching the AMPL book for a specific piece of information about an specific model entity can be difficult. The way in which the AMPL book is made available, in PDF format, does not help search engines in the indexation of that content. Moreover, since the book is intended to be an introduction both to mathematical programming and to AMPL, the arrangement of the book's content is by complexity, advancing from basic concepts to advanced concepts of both mathematical programming and AMPL. This type of arrangement is well suited for the purpose to which the book was designed for, but since the information about the usage of an AMPL construct, such as a

constraint, is scattered throughout the book as the discussion on mathematical programming advances in complexity, finding a specific piece of information will require the user to read through many paragraphs unrelated to what they are looking for.

This comment about the AMPL book, however, is a digression on the point we were just making. Our point is that the numbers suggest that the most sought after type of content by AMPL users, before they decide to ask a question at AGG, is reference material on how to properly declare and use model entities, followed by reference on how to interact with solvers and then reference on debugging mathematical programs and optimization theory in general.

Comparison with issues faced by programmers

The categories above should help us to highlight important distinctions between the general programming and mathematical programming activities. Ko and Myers (2005) suggests that the programming activity can be divided into three types of sub-activities: specification (design and requirements), implementation (code manipulation) and run-time (testing and debugging). The mathematical programming activity can be divided in a similar way, except that "mathematical modeling" would be a more suiting name for the first sub-activity. Let us see how they differ in each of these sub-activities.

First, mathematical programmers start by creating a mathematical model of an optimization problem, while general programmers start by defining functional and non-functional requirements of the program they want to implement, and by devising a plan on how to achieve this goal. The knowledge needed to perform these two sub-activities and their resulting products are not the same. A programmer may never devise an optimization model, and a modeler may never need to define the behavior of a computer program. That means that the struggles of a modeler in this phase are not the struggles of a computer programmer: they have different concerns, different purposes and need a different type of theoretical background.

Second, in the implementation phase, mathematical programmers will use AMLs, while general programmers will use general programming languages (GPLs). There is a sharp distinction between the paradigms of these two language

types. AMLs are more *declarative*, while GPLs are usually *imperative* (Kallrath, 2012). Simply put, *declarative* languages are more concerned with *what* should be done, but not *how* it should be done. For instance, in AMPL you may declare an *ordered set*, but you do not say how it should be ordered. Of course, the declarative-imperative distinction is a spectrum. AMLs do provide ways to control the flow of the program with scripts. But, by design, they were made to *express*, not to *do*. Indeed, modelers are very much limited on what they can *do* with them, which puts AMLs more towards the *declarative* end of the spectrum.

In practice, what that means is that the modeler and the programmer are dealing with different types of language constructs. There is no direct correspondence between model entities, such as variables, constraints and objective functions, and the typical constructs of GPLs. There is nothing similar to indexing expressions or set operators either. As a result, the implementation issues of the modeler and the programmer will have little in common. In fact, we have seen that issues related to "scripting and algorithm implementation", which concern the usage of the features of AMPL that are most similar to GPLs, are not nearly as common as the issues concerning the "declarative" aspects of the language.

Third, when testing and debugging, modelers and programmers are dealing with very different kinds of bugs, and the strategy to find and fix them varies greatly. The contrast can be seen when comparing the works of Pannell et al. (1996), which presents strategies to debug mathematical models, and McCauley et al. (2008), which, among other things, presents strategies to debug programs. Most of the bugs a programmer deals with on a daily basis would fall into "scripting and algorithm implementation" issues if we translate them into "AMPL equivalent" problems, which, again, is only one type of issue and is not among the most common ones.

For these reasons, we think that the mathematical programming community would greatly benefit from more studies focusing on issues faced by modelers and strategies on how to solve these issues. The conclusions of the literature on the general programming activity are applicable to the mathematical programming activity only to a limited extent. Debugging, in particular, is an important skill for both mathematical and general programmers, but little attention has been given to the bugs and strategies to fix them in the mathematical programming

context.

Implications for education

The categories above provide educators some guidance on what to focus on when teaching AMPL and other AMLs to students. In order to students to become proficient mathematical programmers, it is important that they have the knowledge necessary to prevent and overcome the types of issues we have listed, especially those which are the most common ones.

First, it is important that students have a solid understanding of the concepts of a mathematical programming language. Model entities are the basic building blocks with which an optimization program is constructed, and most of the mathematical programming activity in AMPL will involve the declaration and usage of those entities. Other features of the language, such as its scripting capabilities, are of lesser importance.

We wanted to stress this point because, based on our own experience as undergraduates, production/industrial engineering departments have not been giving enough attention to the important distinctions between general and mathematical programming. It is a mistake, we believe, to assume that because students have been introduced to general programming in earlier courses of the program they will easily grasp the concepts of algebraic modeling languages. Fernández and Fernández (2015) have also made some comments in this direction. In their work, which is about the usage of web 2.0 tools for teaching linear programming in an introductory level, they say that the students that would be participating on the study had already taken courses on general programming, in which they have learned Java in an advanced level. "However", they say, "a programming language (as it is known in Computer Science) is not the same as a modeling language for mathematical programming" (Fernández and Fernández, 2015), so they took measures in their study to properly deal with the novelty of the concepts required to do mathematical programming in Mosel (an AML developed by FICO Xpress). Other educators in the operations research community should be equally aware of these differences when devising a curriculum and when teaching students that have already been introduced to general programming.

Second, it is important that students have a proper understanding on what a *solver* is and what it does. There is much confusion regarding the relationship between AMLs and solvers among students. Without the knowledge on what an AML does and what a solver does, students may think that a difficulty they are having is related to the language, when in fact it is related to the solver. Part of the problem resolution process on the mathematical programming context is the identification of the origin of the issue. For instance, it is important that students know if a message printed out in the console is coming from the solver or from the AML interpreter. This helps the modeler to narrow down the reasons why they are having an issue and what to do to overcome it. The knowledge on what a solver does also helps to prepare the students for mathematical programming at a more advanced level. At a beginner's level, the interaction with solvers is minimal, but as the problems they are trying to solve become more complex, interacting with the solver becomes an important step when implementing an optimization program.

Third, many of the issues experienced by modelers are formulation issues rather than implementation issues. With this in mind, teachers should show the students how to leverage AML capabilities in practice when trying to debug a formulation issue. We have already mentioned the work of Pannell et al. (1996), which presents strategies to fix and prevent mathematical programming bugs. Teachers would do well to teach students how to implement those strategies and others in practice, showing students how to check shadow costs and prices, how to assign values to variables in order to check the feasibility of a solution, how to relax integrality constraints, how to identify if the formulation is suffering from numerical instability (problems related, for instance, with Big M values and with integrality tolerance), and so on.

7.6 Conclusion and future directions

The main goal of this chapter was to provide an answer to the question: **(RQ4) What are the most common issues faced by AMPL users?** The answer we have arrived at from checking the discussions at the AMPL Google Group is that the most common issues are those related to the declaration and usage of

model entities, those related to solver usage and interaction, and those related to formulation and/or mathematical optimization theory.

As we discussed the categorization scheme and the answer for the question, other topics naturally emerged. We have tried to show how different the mathematical and general programming activities are by comparing the types of issues modelers and programs have to deal with. We have also made some suggestions for educators interested in teaching mathematical programming based on our findings. Together with Kallrath (2012), we also enthusiastically recommend the use of AMLs in universities as they are considerably more flexible and powerful than spreadsheet approaches, but also much simpler than general programming languages.

The next step for us is to implement the plan that has been laid down in the first part of Section 7.5, which is to create helpful guides for AMPL users and make them available on our website. We also hope that other researchers on the operations research community will take notice of the lack of studies on the mathematical programming activity and will offer their own contribution for the discussions we have initiated here.

Chapter 8

Lessons Learned

This dissertation has a strong problem-solving orientation. In Chapter 1 we have pointed out some of the struggles of operations research students and practitioners that we would like to mitigate with the tool we have developed. In trying to do so, we have found ourselves in the need of understanding the issues that WIDE providers, which is what we are, have to solve. So our attention had to be divided between the two parties of this "service-consumer" relationship: WIDE providers and operations research students and practitioners.

This divided attention resulted in a work with two notable features. First, the chapters of this dissertation are very much independent from one another, and it doesn't really matter in which order they are read. In one chapter we direct greater attention to one of these parties, and in the next to the other. Second, it is a work which depth is shorter than its width. We cannot say that that what we have in hands is an exhaustive research on a single problem, because it isn't. We thought that focusing on issues faced by the two parties of the relationship would lead us to better solutions for both, even if that approach compromised the depth of our discussions.

Our efforts to understand the problems of both contexts have lead us to interact with multiple areas of research: software architecture, web technology, collaborative editing algorithms, WIDEs' industry, software usability, computer-aided education, information architecture, algebraic modeling languages, and debugging strategies, to name a few. The combination of our divided attention between

WIDE providers and users and this broad interdisciplinary interaction can make it easy for the reader to lose sight of what is the unifying theme that brings everything together.

In this short last chapter, our main goal is to help the reader to digest all that we have seen, revisiting the topics we have discussed and highlighting our contributions. This will be done in Sections 8.1 and 8.2, the first showing our *main* contributions and the second showing our *collateral* contributions. Following that, in Section 8.3 we will briefly share with the reader some of our own struggles during this project. Then we will discuss in Section 8.4 some of our thoughts about future work possibilities, both for us and for others. Finally, we conclude in Section 8.5 by thanking God for the ability He has given us to do everything that we have done.

8.1 Challenges, problems, issues, difficulties...

As already noted, this dissertation has a strong problem-solving orientation. The reader may have noticed how often we use the words "challenges", "problems", "issues" and "difficulties" throughout the text, especially on Part II. Part I is basically the presentation of how PIFOP addresses some of the issues discussed in Part II, and for the issues which are not currently addressed by the tool, it is the presentation of the platform through which they can (and hopefully will) be addressed in the future. If we were to summarize what this dissertation is about, we would say that it is about the challenges of providers of WIDEs for optimization and their users. Let us once again go through the issues we have discussed in the previous chapter and see what are the contributions we are leaving in this work.

Starting with the challenges faced by providers of WIDEs for optimization, in Chapter 5 we've highlighted five of them: the challenge of enabling users to run their code, the challenge of competing with desktop IDEs, the challenge of supporting large scale optimization problems, the challenge of running user programs for multiple days and the challenge of providing access to commercial solvers. Based on what we could gather from our research on how WIDEs provide their services, for each challenge we were able to identify multiple solutions that have been proposed and used by different providers. Some of our own solutions have

been presented in that chapter, and can also be seen in more detail in Part I. More than helping the reader to understand the issues faced by providers of WIDEs for optimization, we believe that our main contribution in this regard is in the compilation of possible solutions for the challenges, each with their advantages and disadvantages. As WIDE providers ourselves, we look forward to incorporate in PIFOP some of the solutions we have discussed.

Then we have the challenges faced by operations research students and practitioners. These issues were what motivated us to develop a WIDE for optimization in the first place, and we've talked about them in the Motivations section in Chapter 1. In short, we have five challenges as well: the challenge of running solvers remotely, the challenge of administrating a machine to which multiple modelers need to be granted access, the challenge of students and teachers in focusing on mathematical modeling rather than on software-related issues, and the challenge of collaborating with other mathematical modelers. These issues have been addressed with the application we have developed, and our solutions have been put to test in the study of Chapter 6.

Moreover, we have also studied the challenges faced by AMPL users in general in Chapter 7, where we tried to identify those issues which were the most common so that we could, in the future, provide on our website relevant guides for students and practitioners. Our research resulted in a categorization scheme for AMPL related issues, and three types of issues stand out as the most common: "model entities", "solver usage and interaction" and "formulation and theory". More than a promise that we will produce a repository of AMPL knowledge to our website, we would say that our main contribution with this study is that it provides teachers some insights on what they should focus on when teaching AMPL to their students.

8.2 Collateral contributions

Even though our main focus has been on how to solve the problems of these two actors, namely, WIDEs for optimization providers and operations research students and practitioners, there are two "collateral" contributions that we believe are worth mentioning as well, which may be of the interest of computer science researchers and practitioners.

First, our implementation of the operational transformation algorithms has the novelty of distinguishing between string-wise operations and line-wise operations. To the best of our knowledge, this is not done by other authors. We said that our motivation in doing so was the suspicion that, for large files, a string insertion would be computationally expensive if the whole remaining content after the string had to be shifted. In our system, a string insertion only manipulates the memory of the line in which the operation is being made. However, we haven't made any performance test. The interested reader is welcomed to investigate our claim.

Second, the way in which we provide RCE with PIFOP optimization servers doesn't seem to be common among WIDEs. The most distinctive feature of the optimization server is that the host machine is the one that initiates connection with the central server. This contrasts with solutions provided by the WIDEs we have studied because, when they enable users to connect to machines of their own, the connection is usually established via SSH/SFTP and initiated by the WIDE provider. But our solution has the advantage that the host machine doesn't need to have a public IP address through which a SSH connection can be made. During our research, the only WIDE which we came across that seems to provide a similar solution is the *GitLab WIDE*[1], were users can execute jobs on their machines via the *GitLab Runner*[2], an application that runs in the host machine and establishes a connection between the execution environment and the *GitLab WIDE*. So even though the usefulness of this type of RCE solution has been proven in different contexts, it has seldom presence among WIDEs.

8.3 Challenges faced by the author

Speaking of challenges, some of the readers may be interested in knowing what were the challenges we ourselves had to face in this project. To be brief, we will only talk about two of our struggles, one more technical and one more project management related.

When we first started implementing our operational transformation system, we didn't really know what we were getting ourselves into. It took us about a month

[1] https://docs.gitlab.com/ee/user/project/web_ide/
[2] https://docs.gitlab.com/runner/

and a half to get it into a *mostly* working state, which is not so bad. After we've deployed the feature and students started using it, a new bug would be discovered about every few days, which is not unexpected given the amount of "moving parts" of the mechanism. As more and more bugs were identified and fixed, the failures became less frequent. However, some of the bugs that would only cause a failure on very specific conditions have managed to remain undetected for more than nine months. It was very frustrating to see the system fail over and over again, spend many hours searching for the cause, and find nothing. Fortunately, today we got to a point in which we haven't had OT-related issues for months now.

What have we learned from this experience? That debugging a *system* is significantly harder than debugging a *standalone software*. A minimal OT interaction has not one, not two, but three actors (two clients and the server), which have to interact with each other in a very specific manner. If something bad happens, like a bad transformation, the system may not fail immediately, but only after other certain conditions are met. So when a failure finally occurs, even though we know what was being done at the moment it happens, the root of the issue can be anywhere in each of the components of the system. Next time we find ourselves working on the implementation of a system, we will remember to not underestimate the time needed to debug it.

Then there is the time management challenge. We didn't have a clear idea of what we were going to do in this work from the beginning (March 2019). The decisions on what contributions we wanted to make came over time. But even when they came, we didn't know how much time each of our efforts would take, and our estimations were often wrong. Our time had to be fragmented between the technical implementation and the research we were doing, not to mention the graduate program classes and extra-academic commitments. How much time should we allocate to each of these tasks? That's the question to which we didn't have a clear answer.

Looking back, given the innovative character of what we are doing, we don't think there is something we could have done to devise a clear plan to the whole project from the beginning either. To some extent, that is true to any research, since they always involve some degree of innovation. One thing that we think has helped us in being more productive with the use of our time is to do always *do*

something. Sometimes we would get stuck on how to make progress on our research, so we would do something else, like revise what we have already written, or read the bibliography about another research subject. So our advice to researchers just starting their journey is that they don't spend too much time trying to decide what's the best way to spend their time. Instead, just *do something*. If you get stuck on one thing, move on to do something else. That said, our advice must be taken with a grain of salt, because we still have failed to deliver the project in the time frame we initially had, so what do we know.

8.4 Future works

Given everything we have discussed in this dissertation, what do we expect to see in the future? First, PIFOP will hopefully continue to evolve and attract more AMPL users, and possibly users of other languages as well if we decide to support other AMLs. Additionally, the plan discussed in Chapter 7 will be put into practice and we'll create a knowledge repository on our website.

Second, we expect to see more and more WIDEs designed for other domain specific languages. Most WIDE providers have been focusing on general programming language users, and we would say that that market is nearly full. What is not even close to full is the market of WIDEs for other domain specific languages, and we think that *Overleaf* shows the potential that they have of attracting lots of users. As more WIDE designers take notice of this, more WIDEs for domain specific languages may appear.

Third, as we have seen in Chapter 5, the WIDEs market has changed significantly in the past few years. Most WIDEs featured in previous studies on the area have already disappeared, and those that continued to provide their services have greatly evolved. It is possible that big changes in the WIDEs market will continue to happen, in which case a new research opportunity similar to Chapter 5 could appear. Otherwise, if the market stays virtually the same for many years to come, another research question could be made: what were the most important features of WIDEs that have granted them the stability that they've got? In any case, The WIDEs market is something for researchers to keep an eye on.

Fourth and finally, we expect that the present work may stimulate other researchers in the operations research community to discuss problems of the community itself. Most of the works in the area aim at solving optimization problems encountered in different contexts, which is great. But we think that the activities of operations research students, educators and practitioners would greatly benefit from more technical and theoretical contributions *from* people within the community *for* people within the community.

8.5 All glory be to God

I could not end this dissertation without reminding myself and the reader about the God that has made this work possible. Sinful hearts like mine are quick to brag about any small achievement. But "who makes you different from anyone else?", asks the apostle Paul. "What do you have that you did not receive? And if you did receive it, why do you boast as though you did not?" (1 Corinthians 4:7, New International Version). Indeed, all the things required for this dissertation to come into being have been received from God.

Moses, in a speech directed to the Israelites just before they took the promised land away from the Canaanites, also adverts them against being proud of having produced whatever wealth they would produce in that land: "You may say to yourself, 'My power and the strength of my hands have produced this wealth for me.' But remember the LORD your God, for it is he who gives you the ability to produce wealth" (Deuteronomy 8:17,18). Likewise, the ability to produce this dissertation is a gift from God.

In the same direction goes Isaiah, who attributes to God's teaching the knowledge a farmer needs to do their work. First, the prophet makes a series of rhetorical questions proving that farmers act intelligently: "When a farmer plows for planting, does he plow continually?". No. "Does he keep on breaking up and working the soil?". Of course not. "When he has leveled the surface, does he not sow caraway and scatter cumin?". He does. "Does he not plant wheat in its place, barley in its plot, and spelt in its field?". Yes, indeed. Now, where does all this knowledge of the farmer comes from? "His God instructs him and teaches him the right way"

(Isaiah 28:24-26). Every knowledge required to do any work is a gift from God, and the present work is not an exception.

For all things through which God has enabled me to do this work, from the everyday meals to the comfort of my bed, from the birth in a loving family to the education received throughout my life, from the keyboard I'm using to type these words to the mind which is thinking them, I'm grateful. Above everything else, I'm thankful that, although the sentence for the sins I have committed is death (Romans 1:32), Jesus Christ decided to die in my place. Much more important than all the words of the dozens of preceding pages is that which we read in the Gospel of John: "Whoever believes in the Son has eternal life, but whoever rejects the Son will not see life, for God's wrath remains on them" (John 3:36). My prayer is that the reader may recognize the truth in these words.

Appendix A

Operational Transformation Functions

All of the algorithms here take two edit operations o_a and o_b. They can be one of the four operations described in section 4.1, which are:

1. String Insertion SI(l, c, s): insert string s at line l before column c.
2. String Deletion SD(l, c, k): delete k characters at line l starting at column c.
3. Lines Insertion LI(l, L): insert list of lines L before line l.
4. Lines Deletion LD(l, k): delete k lines starting from line l.

A transformation function $T(o_a, o_b)$ is a function that takes as input two operations and returns a list of operations O'_a such that applying o_b then O'_a achieves the same result as applying O'_b then o_a, where O'_b is the result of $T(o_b, o_a)$. The goal of the transformation functions is to make the files in the client and server sides converge despite the fact that they may apply operations in a different order and with different parameters.

The algorithms in this appendix check the parameters of the operations and then, if needed, transform o_a, modifying one or more of its parameters. Every transformation returns a list of operations O'_a. Most transformations return a single operation, some return two, but never more than that. If no transformation is required, O'_a will contain the operation o_a unaltered. This list may also be empty

if o_a is to be discarded. In all algorithms, assume O'_a starts empty and that it is returned at the end. References to parameters of an operation will be made within brackets. For instance, if o_a is a string insertion operation, $o_a[l]$ refers to the line in which the insertion is being made.

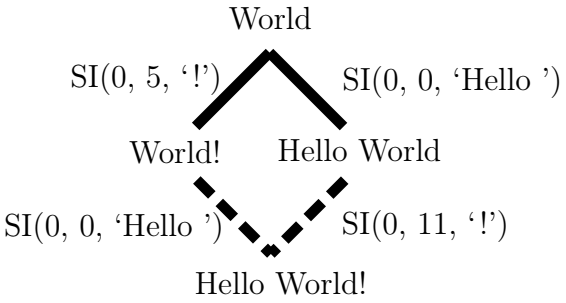

Figure A.1: Simple example demonstrating the role of operational transformation functions in achieving file convergence.

Consider the example given in figure A.1. The file starts with a single line containing 'World'. Let o_a be a string insertion performed at the client's side, with parameters $l = 0$, $c = 5$ and $s =$ '!'. Let o_b be a string insertion performed at the server's side with parameters $l = 0$, $c = 0$ and $s =$ 'Hello '. By the time the server receives o_a, it will have already applied o_b, which means o_a needs to be "shifted to the right" so that the exclamation mark is inserted just after 'World'. Algorithm 1 handles the case of transforming string insertion against string insertion. If the lines in which the insertion occurs are the same ($o_a[l] = o_b[l]$) and the insertion point of o_a is greater than or equal to the insertion point of o_b ($o_a[c] \geq o_b[c]$), o_a will be shifted to the right ($o'_a[c] = o_a[c] + length(o_b[s])$). The resulting O'_a will contain a single operation SI(0, 11, '!').

To transform o_b against o_a, the same function is used but with swapped parameters. The order of the parameters matter because it indicates which operation has already been applied. The already applied operation in the server side is o_b, while the already applied operation in the client side is o_a. When the client receives o_b, it will call the exact same function but with swapped parameters, i.e., $T(o_b, o_a)$. Since the insertion o_b happens in the beginning of the file, prior to the exclamation mark the client has inserted, o_b need not to be transformed. Thus, a single operation SI(0, 0, 'Hello ') is returned.

Algorithm 1: Transform string insertion o_a against string insertion o_b

$o'_a := o_a$;
if $o_a[l] = o_b[l]$ **then**
  if $o_a[c] < o_b[c]$ **then**
    AppendToList(O'_a, o'_a);
  else
    $o'_a[c] = o_a[c] + length(o_b[s])$;
    AppendToList(O'_a, o'_a);
  end
else
  AppendToList(O'_a, o'_a);
end

Algorithm 2: Transform string insertion o_a against string deletion o_b

$o'_a := o_a$;
if $o_a[l] = o_b[l]$ **then**
  if $o_a[c] \leq o_b[c]$ **then**
    AppendToList(O'_a, o'_a);
  else if $o_a[c] \geq o_b[c] + o_b[k]$ **then**
    $o'_a[c] = o_a[c] - o_b[k]$;
    AppendToList(O'_a, o'_a);
  else
    $o'_a[c] = o_b[c]$;
    AppendToList(O'_a, o'_a);
  end
else
  AppendToList(O'_a, o'_a);
end

Algorithm 3: Transform string deletion o_a against string insertion o_b

$o'_{a1} := o_a$;
$o'_{a2} := o_a$;
if $o_a[l] = o_b[l]$ **then**
  if $o_b[c] \leq o_a[c]$ **then**
    $o'_{a1}[c] = o_a[c] - length(o_b[s])$;
    AppendToList(O'_a, o'_{a1});
  else if $o_b[c] \geq o_a[c] + o_a[k]$ **then**
    AppendToList(O'_a, o'_{a1});
  else
    $o'_{a1}[k] = o_a[c] + o_a[k] - o_b[c]$;
    $o'_{a1}[c] = o_a[c] + o_a[k] - o'_{a1}[k]$;
    AppendToList(O'_a, o'_{a1});
    $o'_{a2}[k] = o_b[c] - o_a[c]$;
    AppendToList(O'_a, o'_{a2});
  end
else
  AppendToList(O'_a, o'_a);
end

Algorithm 4: Transform string deletion o_a against string deletion o_b

$o'_a := o_a$;
if $o_a[l] = o_b[l]$ **then**
  if $o_a[c] + o_a[k] \leq o_b[c]$ **then**
    $O'_a := \{\varnothing\}$;
  else if $o_a[c] \geq o_b[c] + o_b[k]$ **then**
    $o'_a[c] := o_a[c] - o_b[k]$;
    AppendToList(O'_a, o'_a);
  else if $o_a[c] \geq o_b[c]$ ***and*** $o_a[c] + o_a[k] \leq o_b[c] + o_b[k]$ **then**
    $O'_a := \{\varnothing\}$;
  else if $o_b[c] \geq o_a[c]$ ***and*** $o_b[c] + o_b[k] \leq o_a[c] + o_a[k]$ **then**
    $o'_a[k] := o_a[k] - o_b[k]$;
    AppendToList(O'_a, o'_a);
  else if $o_a[c] < o_b[c]$ ***and*** $o_a[c] + o_a[k] > o_b[c]$ **then**
    $OverlapCount := o_a[c] + o_a[k] - o_b[c]$;
    $o'_a[k] := o_a[k] - OverlapCount$;
    AppendToList(O'_a, o'_a);
  else if $o_b[c] < o_a[c]$ ***and*** $o_b[c] + o_b[k] > o_a[c]$ **then**
    $OverlapCount := o_b[c] + o_b[k] - o_a[c]$;
    $o'_a[k] := o_a[k] - OverlapCount$;
    $o'_a[c] := o_b[c]$;
    AppendToList(O'_a, o'_a);
  end
else
  AppendToList(O'_a, o'_a);
end

Algorithm 5: Transform line insertion o_a against line insertion o_b

$o'_a := o_a$;
if $o_a[l] < o_b[l]$ **then**
  AppendToList(O'_a, o'_a);
else
  $o'_a[l] = o_a[l] + length(o_b[L])$;
  AppendToList(O'_a, o'_a);
end

Algorithm 6: Transform line insertion o_a against line deletion o_b

$o'_a := o_a$;
if $o_a[l] \leq o_b[l]$ **then**
  AppendToList(O'_a, o'_a);
else if $o_a[l] \geq o_b[c] + o_b[k]$ **then**
  $o'_a[l] = o_a[l] - o_b[k]$;
  AppendToList(O'_a, o'_a);
else
  $o'_a[l] = o_b[l]$;
  AppendToList(O'_a, o'_a);
end

Algorithm 7: Transform line deletion o_a against line insertion o_b

$o'_{a1} := o_a$;
$o'_{a2} := o_a$;
if $o_b[l] \leq o_a[l]$ **then**
  $o'_{a1}[l] = o_a[l] - length(o_b[L])$;
  AppendToList(O'_a, o'_{a1});
else if $o_b[l] \geq o_a[l] + o_a[k]$ **then**
  AppendToList(O'_a, o'_{a1});
else
  $o'_{a1}[k] = o_a[l] + o_a[k] - o_b[l]$;
  $o'_{a1}[l] = o_a[l] + o_a[k] - o'_{a1}[k]$;
  AppendToList(O'_a, o'_{a1});
  $o'_{a2}[k] = o_b[l] - o_a[l]$;
  AppendToList(O'_a, o'_{a2});
end

Algorithm 8: Transform line deletion o_a against line deletion o_b

$o'_a := o_a$;
if $o_a[l] + o_a[k] \leq o_b[l]$ **then**
  $O'_a := \{\varnothing\}$;
else if $o_a[l] \geq o_b[l] + o_b[k]$ **then**
  $o'_a[l] := o_a[l] - o_b[k]$;
  AppendToList(O'_a, o'_a);
else if $o_a[l] \geq o_b[l]$ ***and*** $o_a[l] + o_a[k] \leq o_b[l] + o_b[k]$ **then**
  $O'_a := \{\varnothing\}$;
else if $o_b[l] \geq o_a[l]$ ***and*** $o_b[l] + o_b[k] \leq o_a[l] + o_a[k]$ **then**
  $o'_a[k] := o_a[k] - o_b[k]$;
  AppendToList(O'_a, o'_a);
else if $o_a[l] < o_b[l]$ ***and*** $o_a[l] + o_a[k] > o_b[l]$ **then**
  $OverlapCount := o_a[l] + o_a[k] - o_b[l]$;
  $o'_a[k] := o_a[k] - OverlapCount$;
  AppendToList(O'_a, o'_a);
else if $o_b[l] < o_a[l]$ ***and*** $o_b[l] + o_b[k] > o_a[l]$ **then**
  $OverlapCount := o_b[l] + o_b[k] - o_a[l]$;
  $o'_a[k] := o_a[k] - OverlapCount$;
  $o'_a[l] := o_b[l]$;
  AppendToList(O'_a, o'_a);
end

Algorithm 9: Transform string insertion o_a against line insertion o_b

$o'_a := o_a$;
if $o_a[l] \geq o_b[l]$ **then**
  $o'_a[l] := length(o_b[L])$;
  AppendToList(O'_a, o'_a);
else
  AppendToList(O'_a, o'_a);
end

Algorithm 10: Transform string deletion o_a against line insertion o_b

$o'_a := o_a$;
if $o_a[l] \geq o_b[l]$ **then**
  $o'_a[l] := length(o_b[L])$;
  AppendToList(O'_a, o'_a);
else
  AppendToList(O'_a, o'_a);
end

Algorithm 11: Transform string insertion o_a against line deletion o_b

$o'_a := o_a$;
if $o_a[l] \geq o_b[l]$ ***and*** $o_a[l] < o_b[l] + o_b[k]$ **then**
  $O'_a := \{\varnothing\}$;
else if $o_a[l] \geq o_b[l] + o_b[k]$ **then**
  $o'_a[l] := o_a[l] - o_b[k]$;
  AppendToList(O'_a, o'_a);
end
AppendToList(O'_a, o'_a);

Algorithm 12: Transform string deletion o_a against line deletion o_b

$o'_a := o_a$;
if $o_a[l] \geq o_b[l]$ ***and*** $o_a[l] < o_b[l] + o_b[k]$ **then**
  $O'_a := \{\varnothing\}$;
else if $o_a[l] \geq o_b[l] + o_b[k]$ **then**
  $o'_a[l] := o_a[l] - o_b[k]$;
  AppendToList(O'_a, o'_a);
end
AppendToList(O'_a, o'_a);

Appendix B

Practical Assignment Example

During the academic semester, professor Carlos R. V. de Carvalho gave five different assignments to the students. To give the reader an idea of what was required from the students in these assignments, we include here the instructions given for the fourth assignment, which was to be done in groups of two or three people.

Assignment: Traveling Salesman Problem
Author: Professor Carlos Roberto Venâncio de Carvalho

The traveling salesman problem (TSP) is defined as follows: a traveling salesman lives in a city within a region where there are roads directly connecting every city to all the others. The salesman wants to know which path to take in order to visit all the cities of his region to sell his product in such a way that he travels the least total distance possible.

This problem is also modeled in Graph Theory and is known as the Hamiltonian Cycle Problem (HCP), defined as follows: given a graph $G(V, E)$ where N is the set of vertices of $G(V, E)$ or nodes of $G(N, U)$, and $U = \{(i, j) : i \in N, j \in N, \forall i \neq j\}$ is the set of arcs — directed graph — or edges — undirected graph — (i, j) of $G(V, E)$. A value c_{ij} is associated with each arc or edge $(i, j) \in E$ of the graph, $G(V, E)$ weighted in the arcs or edges, to identify the distance between two cities. A hamiltonian cycle (undirected graph) or circuit (directed graph) of graph $G(V, E)$ is a sub graph $H(V, C)$ of $G(V, E)$ with all its vertices and a subset of the edges in such a way that, starting from any of its vertices, it is possible to visit all the

vertices a single time without re-visiting any of them.

The traveling salesman problem has been extensively studied in the literature and there is a variety of versions of the problem and mathematical programming models for each variation, for instance: all the cities are directly connected to each other (complete graph, i.e, $\exists(i,j)\forall i \in V, j \in V, i \neq j$); the distance when going from city a to city b is different from the distance when going from city b to city a (asymmetric directed graph: $c_{ij} \neq c_{ji}$); the distance when going from city a to city b is equal to the distance when going from city b to city a (symmetric directed graph: $c_{ij} = c_{ji}$); etc.

In the more general case, the traveling salesman problem has all the cities directly connected to each other (complete graph) and the distance between two cities depends on the traveling direction (asymmetric directed graph). Here we present only one of the formulations existent for this general problem, with simplifications. The formulation of any variation of the TSP can be obtained from the formulation and AMPL implementation shown here.

To model this problem, Makhorin makes use of Graph Theory concepts. The model presented here uses a formulation based on Flow Networks, where the salesman starts in the city he lives with a quantity of his product equal to the number of cities to be visited. In each city he visits he leaves one unity of his product. That way, at the end of his journey, when he goes back to his city, the salesman has no more products. This technique is used to guarantee that the solution found contains no sub circuits.

Data

- V: set of vertices $i \in V$;
- $n = |V|$: total number of cities in the salesman region;
- $i = 1$: city where the salesman lives;
- E: set of arcs (i,j), such that $i \in V$ and $j \in V$;
- $c_{ij} : (i,j) \in E$: distance between cities $i \in V$ and $j \in V$; c_{ij} may or may not be equal to c_{ji}.

Variables

- $x_{ij} \in \{0, 1\}$, for all $(i, j) \in E$, such that:

$$x_{ij} = \begin{cases} 1 & \text{if the salesman travels directly from city } i \text{ to city } j \\ 0 & \text{otherwise} \end{cases}$$

- $y_{ij} \geq 0$, for all $(i, j) \in E$: number of products the salesman has when going from city i to city j.

Model

$$\begin{aligned}
\text{minimize} \quad & z = \sum_{(i,j)\in E} c_{ij}x_{ij} && && (1)\\
\text{subject to} \quad & \sum_{(i,j)\in E} x_{ij} = 1 && \forall i \in V && (2)\\
& \sum_{(i,j)\in E} x_{ij} = 1 && \forall j \in V && (3)\\
& (n-1)x_{ij} \geq y_{ij} && \forall (i,j) \in E && (4)\\
& \sum_{(j,1)\in E} y_{j1} + n = \sum_{(1,j)\in E} y_{1j} + 1 && && (5)\\
& \sum_{(j,i)\in E} y_{ji} = \sum_{(i,j)\in E} y_{ij} + 1 && \forall (i,j) \in E \vee i \neq 1 && (6)\\
& x_{ij} \in \{0, 1\} && \forall (i,j) \in E && (7)\\
& y_{ij} \geq 0 && \forall (i,j) \in E && (8)
\end{aligned}$$

The objective function (1) minimizes the total sum of the distances traveled by the salesman. Constraints (2) and (3) are allocation constraint: (2) ensures that the salesman leaves only once from each city $i \in V$; (3) ensures that the salesman arrives only once at each city $i \in V$. These two constraints are not sufficient to eliminate solutions with sub cycles. The constraints (4), (5) and (6) are introduced precisely to eliminate the possibility of a solution containing sub cycles. Constraints (4) guarantee that the maximum number of products that the salesman carries is equal to the number of cities yet to be visited and also that he only carries products from city i to city j is the salesman actually travels from i to j. Considering constraint (2), the traveling salesman leaves his own city (vertex

$i = 1$) only once to a single city, so constraint (5) ensures that he leaves his city with exactly $n - 1$ units of his product. Constraints (6) guarantee that, for all cities which are not the origin ($i \neq 1$), the number of products that the salesman carries **from** that city is equal to the number of products that he has carried **to** it minus one unit. Remember that constraints (2) and (3) ensure that the salesman leaves and arrives a city only once.

Task: Create an AMPL model file (`.mod`) and a data file (`.dat`) to describe and solve this problem. Share the files with the professor.

Appendix C

Case Study Questionnaire

These are the questions that composed the questionnaire that was applied to assess the students perception of PIFOP.

1. Have you used AMPL in previous academic semesters?

☐ Yes.

☐ No.

2. Which browser have you used most frequently to access PIFOP?

☐ Google Chrome.

☐ Mozilla Firefox.

☐ Internet Explorer/Microsoft Edge.

☐ Other (which one?):

3. How often, approximately, have you accessed PIFOP this semester?

☐ Two or more times a week.

☐ Once a week.

☐ Once every two weeks or less.

4. Which features did you find were the most useful? (choose 3)

☐ Web-based.

☐ Cloud file storage.

☐ Syntax highlight of the editor.

☐ Collaborative editing.

☐ Ability to create multiple projects.

☐ Persistent execution (ability to leave processes running while I'm offline).

☐ Automatic file saving.

☐ Other (which one?):

5. Have you accessed the AMPL examples available in our help page?

☐ Yes, and they were very helpful.

☐ Yes, but they weren't very helpful.

☐ No.

6. Do you agree or disagree with the following assertions? (for each assertion, assign a number between 1 and 5, where 1 means you strongly disagree and 5 means you totally agrees.)

a. In the past I have used other computer tools for code implementation with similar interface to that of PIFOP.

b. The interface for model development is clear and easy to use.

c. The terminal for model execution is responsive and easy to use.

d. The real-time collaboration feature was important for the group assignments.

e. I receive the file modifications made by collaborators relatively quickly when we are working simultaneously in the same project.

f. The fact that the interface is in english was not a big problem.

g. PIFOP has facilitated my learning experience of the class subject.

h. In general, the response time of PIFOP is short (for instance, when I access projects and when I request the execution of a model)

i. I did not experience many issues with bugs and defects of PIFOP.

j. I would recommend PIFOP for colleagues in other classes.

k. I intend to keep using PIFOP in classes where AMPL is used.

7. [Optional] Is there any bug in PIFOP you want to report?

8. [Optional] Did you miss any feature in PIFOP?

Bibliography

Aho, T., Ashraf, A., Englund, M., Katajamäki, J., Koskinen, J., Lautamäki, J., Nieminen, A., Porres, I., and Turunen, I. (2011). Designing ide as a service. *Communications of Cloud Software*, 1(1).

Bacchelli, A. and Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE.

Bajaj, K., Pattabiraman, K., and Mesbah, A. (2014). Mining questions asked by web developers. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 112–121.

Barker, I. (2014). What is information architecture?. 2005. *Dostopno na: http://www. steptwo. com. au/papers/kmc_ whatisinfoarch/index. html.*

Berry, D. M. (2013). The essential similarity and differences between mathematical modeling and programming. *Science of Computer Programming*, 78(9):1208–1211.

Berssanette, J. H., de Francisco, A. C., and Baran, L. R. (2018). Espaços ampliados apoiados por tecnologias digitais para o ensino de programação de computadores. *Revista Pleiade*, 12(25):39–51.

Brown, G. G. and Dell, R. F. (2007). Formulating integer linear programs: A rogues' gallery. *INFORMS Transactions on Education*, 7(2):153–159.

Bryce, R. C., Cooley, A., Hansen, A., and Hayrapetyan, N. (2010). A one year empirical study of student programming bugs. In *2010 IEEE Frontiers in Education Conference (FIE)*, pages F1G–1. IEEE.

Bussolon, S. (2009). Card sorting, category validity, and contextual navigation. *Journal of Information Architecture*, 1(2).

Camargo, L. S. d. A. d. (2010). Metodologia de desenvolvimento de ambientes informacionais digitais a partir dos princípios da arquitetura da informação.

Carroll, P. and White, A. (2017). Identifying patterns of learner behaviour: what business statistics students do with learning resources. *INFORMS Transactions on Education*, 18(1):1–13.

Cline, M. P., Lomow, G., and Girou, M. (1998). *C++ FAQs*. Pearson Education.

Cofalik, S. and Tomanek, A. (2018). Lessons learned from creating a rich-text editor with real-time collaboration. https://ckeditor.com/blog/Lessons-learned-from-creating-a-rich-text-editor-with-real-time-collaboration/. Retrieved August 3 2020.

Czyzyk, J., Mesnier, M. P., and Moré, J. J. (1998). The neos server. *IEEE Journal on Computational Science and Engineering*, 5(3):68–75.

de Faria, M. M. (2010). Card sorting: noções sobre a técnica para teste e desenvolvimento de categorizações e vocabulários. *RDBCI: Revista Digital de Biblioteconomia e Ciência da Informação*, 8(1):1–9.

de Oliveira Solano, V. and Rocha, J. A. P. (2015). O card sorting no ensino de biblioteconomia: uma experiência em sala de aula. *RDBCI: Revista Digital de Biblioteconomia e Ciência da Informação*, 13(2):404–417.

Dolan, E. D. (2001). The neos server 4.0 administrative guide. Technical Memorandum ANL/MCS-TM-250, Mathematics and Computer Science Division, Argonne National Laboratory.

Eady, M. and Lockyer, L. (2013). Tools for learning: Technology and teaching. *Learning to teach in the primary school*, 71.

Ellis, C. A. and Gibbs, S. J. (1989a). Concurrency control in groupware systems. *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 399–407.

Ellis, C. A. and Gibbs, S. J. (1989b). Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 399–407.

Engel, G. I. (2000). Pesquisa-ação. *Educar em Revista*, (16):181–191.

Fernández, J. and Fernández, P. (2015). Introducing web 2.0 tools for teaching linear programming. *Procedia-Social and Behavioral Sciences*, 191:1392–1396.

Fette, I. and Melnikov, A. (2011). The WebSocket Protocol. RFC 6455.

Fiala, J., Yee-King, M., and Grierson, M. (2016). Collaborative coding interfaces on the web. In *Proceedings of the International Conference on Live Interfaces*, pages 49–57. REFRAME Books, University of Sussex.

Fourer, R. (2020). Youtube video: New ampl features in 2020. https://www.youtube.com/watch?v=a4pUzPupCjE. Retrieved August 3 2020.

Fourer, R., Gay, D. M., and Kernighan, B. W. (2002). The ampl book. ampl: A modeling language for mathematical programming. *Duxbury Press/Brooks/Cole Publishing Company,*.

Fylaktopoulos, G., Goumas, G., Skolarikis, M., Sotiropoulos, A., and Maglogiannis, I. (2016). An overview of platforms for cloud based development. *SpringerPlus*, 5(1):1–13.

Gropp, W. and Moré, J. J. (1997). Optimization environments and the neos server. In Buhman, M. D. and Iserles, A., editors, *Approximation Theory and Optimization*, pages 167–182. Cambridge University Press.

Hannah, S. (2008). Sorting out card sorting: Comparing methods for information architects, usability specialists, and other practitioners.

Hornik, K. et al. (2002). The r faq.

Iqbal, A., Hausenblas, M., and Decker, S. (2012). Developing in the cloud. Technical report, Citeseer.

Isken, M. W. (2014). Translating a lab based spreadsheet modeling course to an online format: experience from a natural experiment. *INFORMS Transactions on Education*, 14(3):120–128.

Itahriouan, Z., Aknin, N., Abtoy, A., and El Kadiri, K. E. (2014). Building a web-based ide from web 2. 0 perspective. *International Journal of Computer Applications*, 96(22).

Kallrath, J. (2012). Algebraic modeling languages: Introduction and overview. In *Algebraic Modeling Systems*, pages 3–10. Springer.

Kanerva, J. M. (1997). *The Java FAQ*. Addison-Wesley Longman Publishing Co., Inc.

Kats, L. C., Vogelij, R. G., Kalleberg, K. T., and Visser, E. (2012). Software development environments on the web: a research agenda. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pages 99–116.

Knuth, D. E. (1989). The errors of tex. *Software: Practice and Experience*, 19(7):607–685.

Ko, A. J. and Myers, B. A. (2005). A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1-2):41–84.

Konopka, C. L., Adaime, M. B., Mosele, P. H., et al. (2015). Active teaching and learning methodologies: some considerations. *Creative Education*, 6(14):1536.

Kusumaningtyas, K., Nugroho, E. D., and Priadana, A. (2020). Online integrated development environment (ide) in supporting computer programming learning process during covid-19 pandemic: A comparative analysis. *IJID (International Journal on Informatics for Development)*, 9(2):66–71.

Lacerda, D. P., Dresch, A., Proença, A., and Antunes Júnior, J. A. V. (2013). Design science research: método de pesquisa para a engenharia de produção. *Gestão & produção*, 20:741–761.

Lahtinen, E., Ala-Mutka, K., and Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. *Acm sigcse bulletin*, 37(3):14–18.

Lazzarini, V., Costello, E., Yi, S., and Fitch, J. (2014). Csound on the web. In *Proceedings of the Linux Audio Conference*, pages 77–84.

Likert, R. (1932). A technique for the measurement of attitudes. *Archives of psychology*.

Liu, Q., Zhang, X., Liu, Y., and Lin, L. (2013). Spreadsheet inventory simulation and optimization models and their application in a national pharmacy chain. *INFORMS Transactions on Education*, 14(1):13–25.

Martin, A., Dmitriev, D., and Akeroyd, J. (2010). A resurgence of interest in information architecture. *International journal of information management*, 30(1):6–12.

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., and Zander, C. (2008). Debugging: a review of the literature from an educational perspective. *Computer Science Education*, 18(2):67–92.

Molnar, N. (2020). Introducing visual studio codespaces: cloud-hosted development for wherever you're working. https://devblogs.microsoft.com/visualstudio/introducing-visual-studio-codespaces/?WT.mc_id=reddit-social-thmaure. Retrieved August 3 2020.

Mustafa, G. M. (2017). Learning with each other: Peer learning as an academic culture among graduate students in education. *American Journal of Educational Research*, 5(9):944–951.

Nawaz, A. (2012). A comparison of card-sorting analysis methods. In *10th Asia Pacific Conference on Computer Human Interaction (Apchi 2012). Matsue-city, Shimane, Japan*, pages 28–31.

NEOS Statistics (2021). NEOS Solver Statistics. https://neos-server.org/neos/report.html. Retrieved July 21 2021.

Nichols, D. A., Curtis, P., Dixon, M., and Lamping, J. (1995). High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 111–120.

Nurmuliani, N., Zowghi, D., and Williams, S. P. (2004). Using card sorting technique to classify requirements change. In *Proceedings. 12th IEEE International Requirements Engineering Conference, 2004.*, pages 240–248. IEEE.

Nurre, S. G. and Weir, J. D. (2017). Interactive excel-based gantt chart schedule builder. *INFORMS Transactions on Education*, 17(2):49–57.

Overleaf How-to Guides (2020). Can multiple authors edit the same file at the same time? https://www.overleaf.com/learn/how-to/Can_multiple_authors_edit_the_same_file_at_the_same_time? Retrieved June 23 2020.

Pannell, D. J., Kingwell, R. S., and Schilizzi, S. (1996). Debugging mathematical programming models: principles and practical strategies. *Review of Marketing and Agricultural Economics*, 64(430-2016-31525):86–100.

Peavy, M. D. (2009). C++ faq.

Pereira, D. and Camargo, R. (2021). Pifop: Uma aplicação web para desenvolvimento colaborativo de modelos matemáticos em ampl. *Pesquisa Operacional para o Desenvolvimento*, 14:1–15.

Pinto, G., Torres, W., and Castor, F. (2015). A study on the most popular questions about concurrent programming. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 39–46.

Resmini, A. and Rosati, L. (2012). A brief history of information architecture. *Journal of information architecture*, 3(2).

Righi, C., James, J., Beasley, M., Day, D. L., Fox, J. E., Gieber, J., Howe, C., and Ruby, L. (2013). Card sort analysis best practices. *Journal of Usability Studies*, 8(3):69–89.

Roth, R. E., Finch, B. G., Blanford, J. I., Klippel, A., Robinson, A. C., and MacEachren, A. M. (2011). Card sorting for cartographic research and practice. *Cartography and Geographic Information Science*, 38(2):89–99.

Santos, A., Farias, M., Rocha, G., Pereira, M. V., de Farias, C. M., França, T. C., Costa, R. O., and de Janeiro-RJ-Brazil, R. (2014). Web2compile: uma web ide para redes de sensores sem fio. *Simpósio Brasileiro de Redes de Computadores e Sistema Distribuídos (SBRC)*, pages 1037–1044.

Segen's Medical Dictionary (2011). Self-directed learning. https://medical-dictionary.thefreedictionary.com/self-directed+learning. Retrieved August 2 2020.

Škorić, I., Pein, B., and Orehovački, T. (2016). Selecting the most appropriate web ide for learning programming using ahp. In *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 877–882. IEEE.

Spencer, D. and Warfel, T. (2004). Card sorting: a definitive guide. *Boxes and arrows*, 2(2004):1–23.

Sun, C. and Ellis, C. (1998). Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 59–68.

Sun, C., Jia, X., Zhang, Y., Yang, Y., and Chen, D. (1998). Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1):63–108.

Takeda, H., Veerkamp, P., and Yoshikawa, H. (1990). Modeling design process. *AI magazine*, 11(4):37–37.

The Glossary of Education Reform (2020). Blended learning. https://www.edglossary.org/blended-learning/. Retrieved August 2 2020.

Tripp, D. (2005). Pesquisa-ação: uma introdução metodológica. *Educação e pesquisa*, 31(3):443–466.

Valez, M., Yen, M., Le, M., Su, Z., and Alipour, M. A. (2020). Student adoption and perceptions of a web integrated development environment: An experience report. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 1172–1178.

Van Aken, J. E. and Berends, H. (2018). *Problem solving in organizations*. Cambridge university press.

Williams, J. A. S., Reid, R., Gallamore, K., and Rankin, M. (2018). Introducing troubleshooting for model formulation, spreadsheet development, and memo communication with feedforward. *INFORMS Transactions on Education*, 18(2):102–115.

Willoughby, K. A. and Teare, G. F. (2017). How big should my dot be? using spreadsheet simulation to evaluate process improvement data collection strategies. *INFORMS Transactions on Education*, 17(3):93–98.

Wu, L., Liang, G., Kui, S., and Wang, Q. (2011). Ceclipse: An online ide for programing in the cloud. In *2011 IEEE World Congress on Services*, pages 45–52. IEEE.

Xu, Y., Sun, C., and Li, M. (2014). Achieving convergence in operational transformation: conditions, mechanisms and systems. *Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing*, pages 505–518.

Yi, S., Sigurðsson, H., and Costello, E. (2019). Csound web-ide. In *Web Audio Conference*.

Yildirim, S., Bölen, M. C., and Yildirim, G. (2017). Learners' views about cloud computing-based group activities. In *SHS Web of Conferences*, volume 37, page 01033. EDP Sciences.

Yoo, D., Schanzer, E., Krishnamurthi, S., and Fisler, K. (2011). Wescheme: the browser is your programming environment. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, pages 163–167.

Yulianto, B., Prabowo, H., Kosala, R., and Hapsara, M. (2017). Harmonik=++(web ide). *Procedia computer science*, 116:222–231.

Zafirov, C. (2013). New challenges for the project based learning in the digital age. *Trakia Journal of Sciences*, 11(3):298–302.

Zhou, W., Simpson, E., and Domizi, D. P. (2012). Google docs in an out-of-class collaborative writing activity. *International Journal of Teaching and Learning in Higher Education*, 24(3):359–375.