Mathematical Programming Computation
https://doi.org/10.1007/s12532-022-00225-1

FULL LENGTH PAPER

On technical debt in mathematical programming: An exploratory study

Melina Vidoni[1] · **Maria Laura Cunico**[2]

Received: 7 December 2020 / Accepted: 5 July 2022

Abstract
The Technical Debt (TD) metaphor describes development shortcuts taken for expediency that cause the degradation of internal software quality. It has served the discourse between engineers and management regarding how to invest resources in maintenance and extend into scientific software (both the tools, the algorithms and the analysis conducted with it). Mathematical programming has been considered 'special purpose programming', meant to program and simulate particular problem types (e.g., symbolic mathematics through Matlab). Likewise, more traditional mathematical programming has been considered 'modelling programming' to program models by providing programming structures required for mathematical formulations (e.g., GAMS, AMPL, AIMMS). Because of this, other authors have argued the need to consider mathematical programming as closely related to software development. As a result, this paper presents a novel exploration of TD in mathematical programming by assessing self-reported practices through a survey, which gathered 168 complete responses. This study discovered potential debts manifested through *smells* and attitudinal causes towards them. Results uncovered a trend to refactor and polish the final mathematical model and use version control and detailed comments. Nonetheless, we uncovered traces of negative practices regarding *Code Debt* and *Documentation Debt*, alongside hints indicating that most TD is deliberately introduced (i.e., modellers are aware that their practices are not the best). We aim to discuss the idea that TD is also present in mathematical programming and that it may hamper the reproducibility and maintainability of the models created. The overall goal is to outline future areas of work that can lead to changing current modellers' habits and assist in extending existing mathematical programming (both practice and research) to eventually manage TD in mathematical programming.

✉ Melina Vidoni
melina.vidoni@anu.edu.au

Maria Laura Cunico
laura-cunico@santafe-conicet.gov.ar

[1] Australian National University, CECS School of Computing, Canberra, Australia

[2] Institute of Design and Development (INGAR CONICET-UTN), Santa Fe, Argentina

Published online: 05 August 2022

Keywords Mathematical modelling · Soft-or · Software engineering · Technical debt

Mathematics Subject Classification 68N15 · Programming languages 68N01 · General

1 Introduction

Mathematical Programming (MP) is an essential part of Operational Research (OR) that goes beyond optimisation [45]. Its applications abound in many disciplines of science and engineering [23, 45, 51, 64]. In the later years, many trends have aligned OR and MP with other areas such as machine and deep learning, and data science, among others [59, 73, 81, 88, 93].

Because of this, other authors argued MP is akin to software development, grounded on the origin of both disciplines [85], and in the dichotomy of general and special-purpose programming [6]. MP has been considered 'special purpose programming', meant to program and simulate particular problem types (e.g., symbolic mathematics through Matlab) [62]. Likewise, more traditional mathematical programming has been considered a special-purpose known as 'modelling programming' [47], which allows programming models by providing programming structures required for mathematical formulations (e.g., GAMS, AMPL, AIMMS). Because of this, many newer languages for MP are based or inspired by traditional software languages; specific examples are Pyomo and Julia [40, 55], whose popularity for scientific software development (namely, made to understand a problem) increased considerably in later years [90].

However, the practices required by software developers and modellers are somewhat akin [59, 88], even though the 'users' (namely, the modellers, developers, and researchers using these languages) seldom identify themselves as developers [13, 26]. This brings an attached consequence–these 'users' disregard code quality given they do not consider themselves as developers [63], leading to another question: *What is software quality in special-purpose programming, and more particularly, MP?* This question cannot be solved straightforwardly, as several authors have pointed that there is a gap between Software Engineering (SE) and scientific programming, which poses a severe risk to the production of reliable scientific results [79].

In software engineering, *Technical Debt* (TD) is a metaphor used to encapsulate, broadly, a "shortcut for expediency" [28]; it indicates a trade-off between short-term goals and long-term goals in the development[1] [11] and is also related to the implied cost of additional rework caused by choosing an easy solution instead of a better approach that would take longer to implement [22]. More importantly, TD can be introduced unintentionally (namely, unknowingly for the developer) [5, 32].

The usefulness of the TD concept prompted the SE community (both academics and practitioners alike) to study it further [3, 20, 33, 54, 70, 72]. Nowadays, TD is regarded as an essential consideration when developing software [33, 74], which can even sway developers' morale [11]. Moreover, it has also been expanded to cover scientific software [19, 52]. Nonetheless, though there is a plethora of work related

[1] This is not related to the problem situation being modelled or addressed.

to improving processes related to project management in OR interventions (i.e., Soft OR) [1, 27, 50, 86], to the authors' knowledge, approaching TD in MP remains a gap in the literature that has been previously highlighted [85]. We consider this to be complementary to the well-regarded area of Soft OR.

This study aims to address this gap by providing a first exploratory study of TD in MP. We focused on three specific TD types: Code (the most commonly admitted by developers [7, 22, 75, 89], and one of the most researched [33, 54]), Documentation (the most impactful and interesting for scientific software reviewers [19]), and Versioning (given the relevance of versioning for open-science and handling TD [15, 56]). To do this, we conducted an online, anonymous survey with well-established OR academics and practitioners, and gathered 168 full responses regarding self-reported practices that may favour (or control) TD. Given that results are self-reported by participants and subjected to participant bias, we use this to detect traces of TD as a first step to nudge OR/SE research in this direction. To our knowledge, there is no formal specification of TD for this paradigm; therefore, we drew from traditional SE definitions.

Overall, our study uncovered modellers' tendency to refactor and polish the final model and use version control and very detailed comments. Nonetheless, we discovered traces of negative practices regarding Code and Documentation Debt (e.g., dead and duplicated code, and outdated or incomplete documentation). We also observed hints that TD appears to be deliberately introduced, with modellers being aware that their practices are not the best–this seems to align with prior findings related to scientific software practices [4, 63]. We also highlight four future areas of work to continue unveiling what TD means for OR. Finally, although the goal may be ambitious, this paper aspires to stimulate reflective thinking and promote a novel and different line of action and research among OR practitioners in search of two goals. First, achieving better programming habits during model development, and second, approaching SE research in OR programming.

Paper structure. Section 2 introduces the SE concepts that are later analysed in the context of OR, while Sect. 3 presents the methodology for this study. After that, Sect. 4 discusses the results of the survey. Section 5 summarises findings to answer our research questions and presents threats to the validity of our study. Section 6 concludes the paper.

2 Software engineering concepts

This Section introduces the SE-specific background concepts underlying this work and the reason for selecting some of them.

2.1 Technical debt

Technical Debt (TD) was mentioned for the first time in 1992 as a metaphor derived from finances and referred to the need to rework a piece of code in the future, emerging from technical choices of low quality, in order to obtain short-term advantages [72]. Since then, the concept was revisited, and nowadays, TD has been redefined by Avgeriou et al. [5] as follows:

> *TD is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. TD presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability.*

However, although expanding the TD concept was relevant, a subsequent study by Rios et al. [70] (a tertiary systematic literature review) noted that:

> *(...) it has become increasingly common to associate any impediment related to the software product and its development process with the definition of TD. This can bring confusion and ambiguity in the use of the term. Thus, it is important to know the different types of debt that can affect a project so that one can establish the limits of the concept and, therefore, to work on the definition of strategies that allow its management.*

Several authors investigated the differences across different types of TD [11, 33, 54, 70, 74, 75], even demonstrating that scientific software may have either different frequencies and also exclusive types [19, 52]. However, to narrow the scope of the project (and to limit the survey length to a 'participant-friendly' timespan), we selected three types only–Code, Documentation and Versioning Debt.

Code Debt (poorly written code that violates best coding practices or rules), has been demonstrated to be the most commonly admitted by developers [7, 22, 75, 89], and one of the most researched [33, 54, 70]. *Documentation Debt* (insufficient, incomplete or outdated documentation) has reportedly been considered the most impactful and interesting for scientific software reviewers [19] given its influence on software reuse and replicability [71]. Finally, *Versioning Debt* (related to problems in producing and tracing previous versions of the system) is fundamental for collaborative work, given the relevance of versioning for open-science and handling TD [4, 15, 56]).

Note that *Testing Debt* (issues found in testing activities that can affect the quality of the product) is also a commonly addressed TD type [33, 54, 70]. However, work related to 'software testing' in MP remains scarce [83] and thus was not possible to study further.

TD gained relevance among the SE community [54], as scrutiny of systems' quality has become more pronounced [3]. They have approached specific procedures to handle it in software factories [57, 71], how it affects (and is affected by) agile practices [33, 43], how developers approach it [19, 28, 75] what are the "common beliefs" about it [11], and how developers' attitudes increase or decrease its existence [3, 20].

TD is classified into types (each with different techniques for measurement, prioritisation, identification and repayment [74]), and the issues (or symptoms of issues) caused by TD are known as *smells* [5, 17], which have been adopted by the many forms of TD. Different works in SE have identified a list of common smells in traditional Object-Oriented Programming (namely, OOP) [20, 54, 70, 71, 92].

Addressing TD has become essential as *"the accumulation of technical debt may severely hinder the maintainability of the software"* unless it is continuously monitored and managed [94]. Repayment has been considered "of critical importance" for software maintenance [74], although in order to repay TD it is essential to *identify* it first [70].

Table 1 Fowler's quadrants [31], defining causes of TD introduction

	Reckless	Prudent
Deliberate	**(RD)** Knowingly decides to do something wrong, under a given excuse. Does not consider potential consequences. Brushes off adequate practices as unnecessary or not having time for it	**(PD)** Is aware of good practices and smells. Is aware of constraints (i.e., time). Is aware of the consequences of smells. Deliberately decides to go ahead with smells due to constraints, but forces to fix that in the future
Inadvertent	**(RI)** Is not aware of good practices that avoid smells	**(PI)** Recognises that did something wrong in a past project. Learns good practices to avoid that smell in the future. Decides to apply that in future projects

2.2 TD quadrants

Several authors discussed that TD *"is not necessarily identified by who has made such choices"* [72], and that introducing TD is not always done willingly or knowingly–a developer may do so due to their lack of knowledge [70].

An accepted classification establishes that developers (or in this case, modellers) incur in TD in two ways [10, 20, 58]: *intentionally*, which happens deliberately (e.g., writing code that does not matches the agreed coding standards) or *unintentionally*, incurred inadvertently due to low quality work or lack of knowledge (e.g., junior programmer unaware of how to improve the code). Works in this area have disclosed a relationship between inexperienced developers and high-level TD [10].

A second classification extends the above to include a second dimension [31]. These are summarised in Table 1. In this model the first dimension (intentional/unintentional) is renamed as **deliberate** and **inadvertent** (indicated in the vertical axis). A second dimension, **reckless** or **prudent**, indicates if the TD had been introduced due to thoughtless or strategic actions.

The number of SE studies based on Fowler's quadrants continues to increase [8, 12, 14, 20, 75, 91], due to being a simple model that allows an in-depth classification. As a result, the quadrants were used to model our survey, as introduced in Sect. 3.

3 Methodology

This Section presents the methodology used for this study. First, we describe the goal and research questions; second, we give an overview of the survey, the questionnaire, and how responses were collected. The methodology used in this manuscript, described in the sections below, was approved by a Human Ethics Research Committee (HREC).

3.1 Research questions (RQs)

The goal of this study was to understand if common cases of TD (as studied in SE) can be identified into MP, under the premise that MP is scientific programming under the umbrella of special-purpose programming. This led to the following research questions.

RQ1. Which self-reported practices may hint at the existence of TD in MP?
RQ2. What attitudes do modellers have regarding TD?

Because TD is a decision taken (knowingly or not) by developers [5], we decided to approach this topic from a human-centric point of view through a survey. As such, our RQs depend on self-reported practices and perceptions, subjected to 'participant bias'; this is further discussed in Sect. 3.4, where we highlight the steps taken to mitigate or contain the related risks. Nonetheless, we believe that, considering the gap of research regarding TD in MP, we first needed to conduct an initial study to understand "where to look"–i.e., which types of debts appear regularly, and where to focus future research efforts. This is not the first study of its kind [63, 71, 92] and will be appropriately considered in terms of threats (see Sect. 3.4).

Henceforth, we will refer to mathematical modellers, OR researchers, scientific software developers and similar roles as 'modellers' since it is the commonly accepted name in OR [85].

Finally, given that the primary goal of this paper is to ignite a discussion regarding the existence of TD in mathematical modelling, Sect. 5 will present a discussion and implications analysis, without the need for an additional research question.

3.2 Survey construction

Online surveys are commonly affected by a low response rate, generally caused by long or complex surveys [35], and it has been demonstrated that Likert-style, close-ended surveys generally have a higher response rate [18]. Moreover, participants are often biased in their responses, which is both a desirable treat (since it allows assessing human perception), but also a threat to validity (since it can affect the responses) [35]. Although open questions allow more natural responses, even a strict coding analysis will be subjected to researcher bias [38].

Quantitative, closed-option surveys have been used in both disciplines related to this study (SE and OR). On the one hand, SE has extensively applied this methodology to address different aspects of TD, especially when aiming to uncover the developers' perspectives or knowledge about this topic [10, 11, 28, 33, 43, 57, 63]. On the other hand, OR has also used surveys (both quantitative and qualitative) as methodologies on a myriad of topics [1, 2, 60, 67].

As a result, we structured this study as an *exploratory survey* meant to "look at a particular topic from a different perspective" as a pre-study to "help to identify unknown patterns" [35].

The survey was divided into three main sections. First, a participant information sheet explained the study's goal, requesting participants to confirm they were above 18 years old. Second, demographic questions with fixed responses: age group, job

position, area of work, OR approaches, and residential country. Then, a section of programming practices to be assessed with 5-point Likert scales, divided into three pages [18]. The following subsections will detail the development and testing stages, while Sect. 6 discusses the replication package.

3.2.1 Phased development

The survey was developed through a lengthy construction process, in which we included different original approaches. We conducted the following phases:

PHASE 1. We considered existing surveys related to common challenges in the development of scientific software [63] and works related to specific TD types. As explained in Sect. 2, we considered only Code, Documentation and Versioning debt–besides the aforementioned technical considerations, another reason was to limit the survey scope to keep the length manageable and appealing to participants. In terms of Documentation Debt, we selected practices the three smells highlighted in an extensive study by other authors [71]. For Code Debt, we originally considered the complete updated list of smells by Fowler [32]. For Versioning Debt, we considered previously highlighted common issues in scientific software development [15].

Note that some Code Smells were exclusively dependent on the OOP paradigm or intrinsic to a given programming functionality. Given that we aimed to obtain a generalised view rather than assessing the languages' capabilities, some Code Smells were discarded. In particular, these are: 'primitive obsession', 'switch statements', 'parallel inheritance hierarchies', 'lazy class', 'middle man', 'inappropriate intimacy', 'alternative classes', and 'refused bequest'.

PHASE 2. Using the relationships between smells, refactoring, and quality attributes [49], we drafted statements (namely, s-statements) written from the modeller's point of view to reflect practices often translated into each smell; e.g., *"I often leave unused code in my models just in case I need it in the future"*. Using prior work as references [22, 33, 49, 70], these statements were worded in a way that agreeing with them resulted in a trace of a smell; this will be further discussed in Sect. 3.2.2.

Considering prior findings regarding survey wording [68] both authors carefully revised these statements to (a) avoid double-barrelling (namely, asking two questions per statement), (b) frame all questions in the context of MP, and (c) framing the statements as neutrally as possible, so as not to bias the respondents. Additionally, we consider (d) the questions' order [68] to organise this block in three parts (one per TD Type). These statements only aimed to answer RQ1, and the assessment was done in the next phase.

PHASE 3. Once these statements were ready, we assessed them with four OR practitioners (two native English speakers)–two middle-career researchers, a postdoc, and an early-career researcher. This was done in two groups (pairs), with both authors present in each group. The discussion was unstructured, and each practitioner was asked to evaluate if the statement was neutrality worded (point c) and if they read two or more questions (a).

During this process, the practitioners in the first group commented that some of the statements were for 'software developers' and not 'modellers', implying they would not answer the survey if they received such questions (namely, self-selection bias);

Table 2 Questionnaire's organisation in Sect. 2, for RQ1–2

TD type	Smell	Definition [33, 49, 54, 70]
Code	Duplicated code	Two or more code fragments that look almost identical
	Shotgun surgery	Completing any modifications requires making many small changes to many parts of the code
	Dead code	A piece of code that is no longer used but remains in the final version of the code
	Speculative generality	Unused pieces of code often created to support anticipated features that are not currently in development
	Incorrect naming	Using names that are not semantic or meaningful, or an inconsistent mixture of naming conventions
	Excess comments	A large number of comments inside the code that often mask unreadable, non-intuitive code
Documentation	Insufficient comments	There are none, or few comments inside the code, making it difficult to read
	Non-existent docs	No documentation (comments or specifications) are available during or after the project is finished
	Outdated docs	There is some level of documentation but refers to older versions of the model, and most information is not applicable anymore
	Incomplete docs	The documentation is incomplete, leaving gaps at different points
Versioning	Code repository	An incorrect control and trace of previous versions of the system limits the ability to compare them or move between them

both authors requested the practitioners to mark these statements. When consulting with the second pair, a similar discussion arose without being prompted; given that the statements selected (and thus, the Code smells) were the same, both authors agreed to remove them from the survey.

The discussions were unrecorded and informal, and no inter-rater agreement was calculated. The consulted practitioners had no objections to the statements outlined for Documentation and Versioning Debt. Note that the participants did not have access to our research question, nor the link between TD type, smell and statement, to prevent additional bias. After finalising PHASE 3, only the statements related to the Smells outlined in Table 2 remained part of the survey.

PHASE 4. After completing PHASE 3, we had 13 statements in total (see Sect. 6). For each of them (and taking the same considerations as previously outlined in PHASE 2),

we drafted four additional statements (q-statements). These new sentences were aimed to convey an attitude for the introduction, reflecting the TD Quadrants introduced in Sect. 2.2. For example, "I copy-paste code inside the same project because it is faster and easier" is associated with 'Duplicated Code' and an RD attitude because it indicates awareness, presents an excuse and does not consider consequences (top-left cell in Table 1). These q-statements were aimed to answer RQ2.

PHASE 5. As previously done for PHASE 3, we reunited with the same pairs of OR practitioners to analyse the statements drafted in PHASE 4. Once again, the process was an informal discussion conducted virtually with all four practitioners while analysing an online, shared document. Additionally, the q-statements were clearly linked to each corresponding s-statement and the intended attitude, and the practitioners had Table 1 available for consultation.

During this informal discussion, the following suggestions were made. First, some s-statements led to similar q-statements (e.g., 'dead code' and 'speculative generality') and were thus merged (e.g., a single set of q-statements for two s-statements); this had the added benefit of reducing the time required to complete the survey. Second, for two groups of s-statements, PD was removed after the practitioners could not agree on an example. The consensus was that MP models are short-lived; this argument has been previously raised regarding scientific software development [39, 63, 79].

Finally, the questions order [68] was also considered, accounting for (PHASE 1 d). Therefore, each page of the survey third part was divided into blocks of related s-statements followed by the q-statements (always that order). Please, refer to Sect. 6.

PHASE 6. Following best practices, we conducted a pilot study with ten OR practitioners/researchers recommended by the four OR practitioners that assisted with the sanity checks of previous phases; these included a range of PhD candidates to mid-career researchers and a single late-career researcher. Given that the original four OR practitioners reached out via email first, all pilot-participants agreed to provide feedback. The survey was distributed in PDF form via email, and the pilot participants were advised that no responses would be recorded.

However, in their feedback email, half of the pilot-participants did not attempt the survey arguing it was for 'software developers' and not 'OR practitioners'. This is a known problem regarding scientific software development, as the 'users' (e.g., the researchers, students or anybody writing scientific software in any shape) tend not to consider themselves as software developers [6, 62, 63]. Given that a response rate of about 10–18% is considered acceptable in SE surveys [61], we decided to continue with this survey structure.

The five emails that provided feedback had minor aesthetic comments, wording questions or notes used to improve the survey wording and presentation. The only recommendation regarding the demographics section was to change "country of origin" for "country of residence", given academic mobility. The survey instrument resulting from this phase was approved by the HREC in an Ethical Protocol and is available in the replication package of Sect. 6. The threats regarding the survey construction are discussed in Sect. 3.4.

Table 3 Likert-scale interpretation for each type of statement (Questionnaire, Sect. 2)

RQ	Statement	Likert interpretation
RQ1	Statements represent a single TD smell practice, written from the developers' point of view. For example *"I often copy a piece of code and paste it in the same project, slightly modified."*	**1–2:** Good practice, does not indicate the smell
		3: Neutral, indicating a mixed or inconsistent practice
		4–5: Bad practice, reinforcing the presence of this smell
RQ2	Statements used to identify the reasons for each type of smell. Each statement belongs to a smell and a quadrant. For example, *"I copy-paste code inside the same project because it is faster and easier"*	**1–2:** Does not belong to this quadrant/this quadrant is not the reason for the smell
		3: Mixed practices with a potentially inconsistent case (weak belonging)
		4–5: This quadrant can be a cause for this smell (strong belonging)

3.2.2 Likert evaluation

In Sect. 3.2.1 PHASE 2 we developed the Likert scales used in the survey. All statements were rated using a common Likert scale of 1–5 (from "Strongly Disagree" to "Strongly Agree"). This approach was selected as Likert is a known, proven approach that most people can intuitively understand [18]. However, in our survey, the Likert value was interpreted differently depending on the type of statement. This is is summarised in Table 3. It was decided not to use a 6-point Likert to favour the traditional approach and have a middle point.

Note that statements about practices have time qualifiers (e.g., 'usually' or 'often'); this was purposefully added (and discussed in the phased development) because there is always a chance developers depart from their habits even if temporarily due to multiple reasons [36]. Moreover, the survey results are used to determine *traces* of behaviours that may *hint* at the presence of TD or to behaviours that can incur in TD.

Therefore, using prior TD taxonomies that included smells and causes leading to TD introduction (in one of the TD types assessed) [33, 49, 54, 70], we worded the statements in a manner that 'strongly agree' would always lead to *hintinng* the *trace* of a smell or the cause (quadrant) for incurring in such smell. For example, *"My models often have fragments of code that are no longer used or are outdated (they may be commented out)"* represents the smell 'unused code'. Therefore, if the respondent

agreed (indicating that 'often' is true), it hinted at a common smell; likewise, the opposite would be true–not doing this often possible hints at a healthy programming practice.

This is an exploratory study, and related threats are considered in 3.4, including those related to the survey construction.

3.3 Survey distribution

We used convenience sampling to invite participants to our study [35]. We manually generated a list of OR researchers and graduate students by browsing the websites of Universities and Research Institutes around the world and gathering the publicly available emails of those academics listed on the faculty or staff pages. This approach has been commonly used in the area of SE to investigate developers' concerns or points of view [33, 72, 92], understanding them as *field studies* [76]. Moreover, this type of contact is often positively regarded, as it also allows a better definition of the sample of candidates [16].

The target size of the invitation list was decided after comparing it to similar survey studies. Similar approaches have demonstrated an expected response rate of 10–20% of the original sample [33, 72, 92]. Since we aimed to have at least a hundred responses, we set to collect almost 2000 emails. After removing duplicates (i.e., academics that have moved institutes and had different emails), **our list included 1849 emails**.

The survey was implemented in Qualtrics, an advanced survey system with powerful result analysis capabilities[2]. In terms of response time, Qualtrics estimated a response time of 15 minutes, and the average response time after the distribution was 17.6 minutes.

We used Qualtrics embedded to send an automated invitation email to the list of selected participants[3]. We used the extracted name and affiliation to automatically customise the email, which also included the invitation and highlighted main aspects of the research data distribution. Using these tools, we configured the 'response address' as the author's email address to facilitate responses and reduce the risk of our email being filtered as spam. The first email was sent at the beginning of September 2020. After two weeks, we sent a reminder email to those who had not responded yet (or not finalised) and whose emails had not bounced or opted-out. The survey closed by the end of September 2020.

From 1849 emails, 32 emails bounced back. After that, 208 surveys were started but not finished. These incomplete responses were ignored in this study because they only had the demographics completed and none of the responses regarding the TD constructs. We assumed this was aligned with the feedback obtained during the Pilot Study (see Sect. 3.2.1, on PHASE 6).

We obtained 168 full responses, totalising a 9.3% response rate (calculated excluding the bounces). Because this rate was slightly below what it is expected in Software

[2] See: https://www.qualtrics.com/.

[3] Qualtric's guide on how to compose these emails is available at: https://www.qualtrics.com/support/survey-platform/distributions-module/email-distribution/emails-overview/.

Table 4 Precedence of invited participants and respondent per region, including response rate

Region	Invited	Responded	Response rate
North America	504	28	5.55 %
Central America	26	2	7.69 %
South America	583	86	14.75 %
Europe	516	32	6.2 %
Oceania	215	20	9.3 %

Engineering [33, 72, 92], and the 'newness' of the topic for OR (see Pilot Study on Sect. 3.2.1 PHASE 6), the number was considered suitable for an exploratory study.

Table 4 shows how many participants of each region were contacted, how many responded, and the response rate. As researchers, we cannot control who decides to participate in the survey. Though the final number of responses is similar to that of similar works [76], participation is skewed towards South America–it had a considerably higher response rate. However, the survey did not question topics relevant to specific ethnically or culturally-relevant practices; therefore, the demographic does not impact the outcomes of this study.

3.4 Threats to validity

To a certain extent, the results of our study are subject to limitations related to its experimental design. In particular, the following *construct*, *external*, and *internal* threats may affect the validity of our findings and conclusions:

Construct threats stem from the degree to which scales, constructs, and instruments measure the properties they intend to [66]. The most critical threat is the survey artefact, given that it has not been previously validated. We considered using known TD-centric surveys (e.g., InsighTD [33, 71] or 'Naming the Pain' [63]), but discarded it upon a quick consultation with the researchers that assisted in the phased development. To mitigate the threats due to the survey, we: a) followed the guidance provided by known studies [68], b) we derived them from practices previously assessed in SE [22, 33, 49, 70], c) assessed the questionnaire with OR practitioners and through a pilot study (see Sect. 3.2.1), and d) consider our results as *traces* of TD, and not as a certainty that TD itself exists. Given that this study is the first of its kind, it was considered a reasonable threat to push forward a new line of work.

Internal threats refer to influences that may affect the study's independent variables in terms of causality [21]. *Participant selection* is the primary threat to internal validity. To minimise this, we created the invitation list by manually browsing the websites of Universities and Research Institutes in the sections of Faculty or Staff members. We mainly aimed for those belonging to departments such as Mathematics o Business Management but included anyone that listed OR as a research interest or background, as well as any other keywords related to the area (e.g., approaches, area of work, topics). To further improve this, the collection was done in two steps: (1) each author searched websites from a different region, and (2) we switched places and reviewed the selection according to published manuscripts.

We also aimed to obtain similarly-sized samples per region (as seen in Table 4). Central America was small because few countries were scouted, and Oceania included Southeast Asian countries. Note that we were limited by our languages, as some websites were not translated to English and/or the Google-translated version hindered our browsing (e.g., when we searched academics with the keyword "operational research", the website provided no results).

However, our response rate is slightly below 10% (approximately 9.3%, excluding bounces). Therefore, our results could suffer from *non-response bias*: a case in which the opinions of those who chose to participate may differ from those who did not. Nonetheless, the analysed responses provided a rich data source in a novel area for OR. It is also possible that our results are affected by the practices of the regions that answered our survey. However, as researchers, we cannot control who decides to participate in the survey [35]. This was considered during PHASE 6 of the survey development (Sect. 3.2.1) and may have affected the response rate. The authors also assume that the 208 demographic-only responses (later removed from the analysis) were caused by *participants' self-selection bias* (considering they received the email by mistake or that the survey was too programming-centric for them) [37]. This is enhanced in this case as the premise of this study lies in the proposed similarities between SE traditional programming and OR's mathematical programming, previously discussed.

Additionally, we did not perform a cross-check to determine potentially "contradictory" answers (e.g., the same participant indicates agreement with two possible opposed attitudes). This is because it is possible that a participant had one type of attitude when using approach X, and another with approach Y (likewise, with the programming languages). However, to achieve such fine-grained detail, the survey would have to enquire about the approach and programming language *on each question* rather than once. Given that this was an exploratory study, it was decided not to exhaust participants and instead obtain a general view. A fine-grained analysis remains future work.

External threats relate to conditions that may affect the generalisability of the study results [21]. Our survey respondents may not adequately represent all modellers, as practices may differ between disciplines, expertise and even country. A limitation is that we mainly targeted academics, with few of them self-reporting a mixed industry-academic affiliation; our conclusions may be limited to this set of respondents. As such, this study may not fully reflect the practices of those practitioners working almost exclusively in the industry.

4 Smell & attitude traces

The remaining subsections analyse each code smell. Each table has a tag with an idea (e.g., [Question Q9]); that ID was given by Qualtrics[4], and are used to refer to related plots available in the Replication Package.

[4] Note that Qualtrics assign the IDs in order of creation of the question (not display order), and if a question is deleted, the IDs are not regenerated. In any case, they are not shown to the participants.

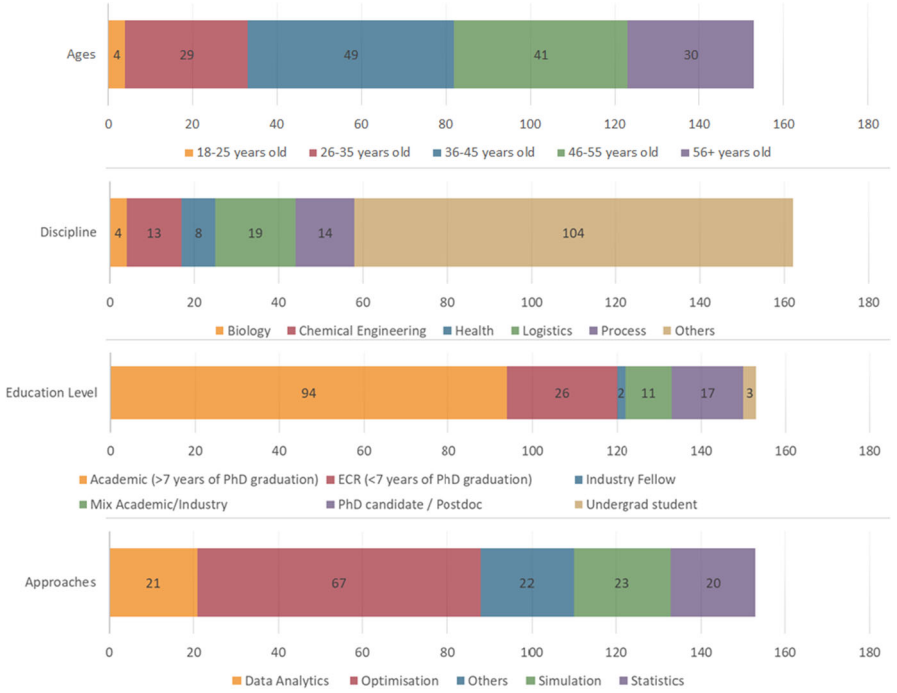

Fig. 1 Aggregated self-reported demographics, except region

A summary of findings and a formal answer to our RQs are presented in Sect. 5.

Figure 1 summarises the responses to all demographic questions, except region (that data is available in Table 4). The discipline selection was a multiple-choice (thus, allowing multiple responses), and 15 participants did not select any option (leaving it blank); from the reminder, 'Other' was the most combined (with existing choices) and selected in itself. In terms of age groups, a large number are mid-career or late-career researchers over 36 years of age; it is possible that this demographic affected the *self-selection bias* of the survey, but analysing such a hypothesis was out of scope. In terms of approaches, 'Optimisation' was the most popular. In all answers, participants had the choice of leaving the answer blank (per our Ethics Protocol).

4.1 Code debt

Regarding *Code Debt*, we analysed **duplicated code**. In SE, it has been proven that duplicated code is often caused by copy-pasting the code instead of refactoring to extract it [25, 34]. Table 5 summarises the values obtained in this question.

About 43.92% of responses ('Strongly' plus 'Somewhat Agree') indicate that copy-pasting pieces of code are common practice; nonetheless, about 36.45% deny it (strongly and somewhat disagree). Interestingly, researchers between 18–25 years old only answered 'Strongly or Somewhat Agree', and most researchers between 26–35 years old selected 'Somewhat Agree'. Given that this practice is considered harmful

Table 5 Results regarding duplicated code, and possible attitudes towards it [Question Q9]

Statement	Type	S.agree	Sw.agree	Neither	Sw.disagree	S.disagree
I often copy a piece of code and paste it in the same project, slightly modified	Smell	10.28%	33.64 %	1.63%	19.63%	16.82%
I don't know what I could do to stop copy-pasting code	RI	3.92%	8.82%	21.57%	24.51%	41.18%
I copy-paste code inside the same project, because it is faster and easier	RD	19.23%	27.88%	21.15%	17.31%	14.42%
I copy-paste when testing something new, but I improve the "final" version of the code (e.g., extract it in a segment)	PD	29.13%	35.92%	14.56%	10.68%	9.71%
I used to do this before, but now I prefer to avoid duplicates	PI	13.73%	18.63%	34.31%	21.57%	11.76%

in traditional SE programming, it is a trace of TD in MP. As a result, this is considered an inconsistent practice that may be **prone** to happening.

Since no code analysis was done to corroborate the number of function clones or code clones in MP, we cannot infer *how* common this behaviour is. A plot of ages per response is available in the replication package as `Q9_Age`.

In terms of possible causes, it is *hinted* that the cause is not a lack of knowledge regarding acceptable practices (RI) since almost 65.7% of responses disagreed with the statement. The remaining three possible reasons are somewhat related: the modellers prefer quick practices while developing the model and trying approaches, but they are primarily aware that having duplicated code is not a good practice (PD, PI); as a result, they come back and remove duplicates (notice that about 65% of responses agreed to some extent with the PD statement). This is consistent with the distribution of responses in the statement related to the smell.

This block highlights the presence of *refactoring*–"a process of improving software systems by applying transformations that should preserve their observable behaviour" [49]. It may be possibly related to the academic background of most participants, and the need to refine a model before publishing (also related to PD); however, further investigating this remains future work.

Duplicated code is often related to the *shotgun surgery*–a smell that happens when, given an excessive redundancy, a change impacts multiple parts of the code. We explored this in conjunction with copy-pasted code, as it is somewhat related; if code is copy-pasted throughout the project (instead of extracted and reused) if said piece of code needs to be changed, it is possible it must be altered in all of its occurrences. The responses are summarised in Table 6.

This case is mixed, as there is about a 10% difference in responses leaning to non-existence (i.e., disagreeing with the practice) compared to the existence of the smell (i.e., agreeing with the practice). In both cases, about 20% of respondents neither agree nor disagree, potentially indicating mixed or inconsistent practices. There is

Table 6 Results regarding shotgun surgery smells, and possible attitudes towards it [Question Q10]

Statement	Type	S.agree	Sw.agree	Neither	Sw.disagree	S.disagree
My models often have similar parts repeated all over it	Smell	4.85%	30.10%	20.39%	26.21%	18.45%
I often find myself making multiple minor modifications in several places of the model to account for a new change in the problem situation or requirements	Smell	9.80%	26.47%	21.57%	31.37%	10.78%
I believe that having similar repeated parts is good for my model	RI	2.91%	9.71%	28.16%	29.13%	30.10%
I don't have time or I don't want to remove repeated parts (e.g., by extracting them in a more general, reusable segment).	RD	3.96%	13.86%	27.72%	30.69%	23.76%
I know it is inconvenient to have repeated parts, but I prefer to deal with that later	PI	4.90%	20.59%	27.45%	23.53%	23.53%

no correlation to age or career stage either (see Replication package, `Q10_Age` and `Q10_CareerStage`). Figure 2 summarises how this affects the approaches used (only for the s-statement), Optimisation is stable across 'Somewhat Agree' to 'Strongly Disagree', but Simulation and Statistics lean towards agreeing with the statement, which may be related to the programming languages used.

As a result, the *shotgun surgery* was considered as **plausible**.

Regarding attitudes (quadrants) related to these smells, lack of knowledge regarding best practices may not be a cause (RI) (as almost 60% of responses disagreed with the statement); this aligns with the demographics that indicate a large number of senior participants. There is also a trend to disagree with PI, somewhat correlated to the first block of questions (namely, Q10), where participants reported the practice of polishing the model before producing the final version.

Furthermore, unlike SE, lack of time for delivery does not appear to be a common cause of duplicating code and finishing earlier (RD). This is possibly related to the fact that most participants are academics, without the pressure to deliver caused by an industry partner. Such a hypothesis aligns with other authors' findings regarding data scientists' programming behaviours [63].

As a result, none of the three evaluated attitudes had enough agreement to hint at a possible reason. Therefore, further studies are needed in this regard.

Two other smells were explored in combination with each other: **dead code** and **speculative generality**. These are summarised in Table 2 since they are related to unused, outdated code in the model's final version. The responses obtained are summarised in Table 7.

An important number of responses (about 30% on each smell) indicate that *dead code* and *speculative generality* may be a concern, each response received about 48.5% agreement, with disagreements of 31.7% and 35.7% (respectively). Given there is more

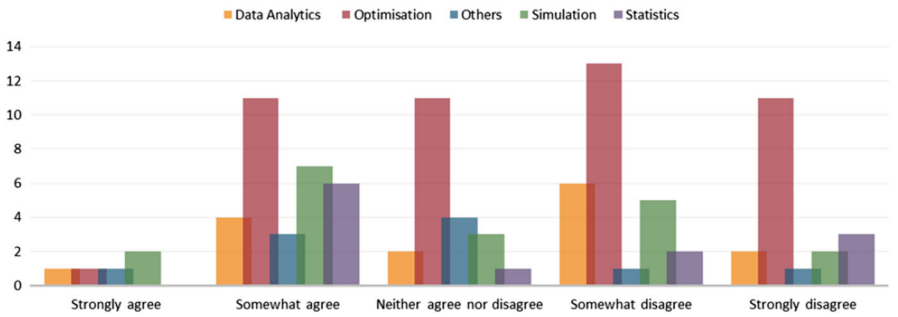

Fig. 2 Shotgun surgery responses in terms of approaches [Question Q10]

Table 7 Survey results regarding smells concerning unused code [Question Q11]

Statement	Type	S.agree	Sw.agree	Neither	S.disagree	S.disagree
My models often have code fragments no longer used or outdated (they may be commented out)	Smell	7.77%	40.78%	20.39%	18.45%	12.62%
My models often have unused code in case I need it in the future	Smell	12.87%	35.64%	15.84%	21.78%	13.86%
I am not sure if my models have unused pieces of code	RI	0.98%	5.88%	8.82%	36.27%	48.04%
I don't want or care to go back to the model to remove or find those unused pieces of code	RD	2.94%	17.65%	26.47%	33.33%	19.61%
I leave in unused code while working, and then remove it for the final version	PI	9.90%	42.57%	21.78%	18.81%	6.93%
I often go back to the model after finishing it, and remove unused/duplicated parts	PD	6.86%	46.08%	18.63%	21.57%	6.86%

than 12% of difference between the agreements, we assumed a a *strong trace* of both assessed smells. Therefore, they are labelled as **recurrent** and **prone**, respectively.

When analysing the reasons, PI and PD hints to be the cause of the smells, with over half participants supporting each of these statements.

This may be related to the academic background of participants, the previously established revision before the final version, and the polishing of models before publication. It may be possibly related to a scarce use of version control [15] (which removes the need to keep dead code) and possibly to the trend not to follow SE practices highlighted by previous studies [63]. This is also aligned with the other findings of our survey, indicating a trend to refactor models.

However, even though code may be cleaned up before publication, the existence of *dead code* during development may cause other smells not assessed in this survey. Therefore, future works are needed to explore these areas.

Table 8 Survey results regarding *incorrect naming* in mathematical programming [Question Q12]

Statement	Type	S.agree	Sw.agree	Neither	S.disagree	S.disagree
My variables usually follow a mathematical notation only, like `x(i,j)`, or `x,y,z` and so on	Smell	18.63%	18.63%	14.71%	27.45%	20.59%
My variables often have names related to what they represent in the problem situation (e.g., `isChosen(i,j)` or `isChosen(product, supplier)`)	Smell	38.83%	43.69%	9.71%	4.85%	2.91%
I name my variables thinking from a mathematical point of view, not a programming one	RI	16.67%	24.47%	29.41%	17.65%	9.80%
I don't care about "semantic" names. I won't come back to the code, share it, or need it	RD	2.94%	9.80%	14.71%	35.29%	37.25%
I prefer a mathematical notation, and do not like larger names, as they are annoying, uncomfortable or other reason	RD	7.77%	25.24%	17.48%	29.13%	20.39%
I often write in mathematical notation because it is faster or more natural, even if I know that semantic names are better, but I have to finish earlier	PD	7.77%	14.56%	25.24%	33.01%	19.42%
My naming style changed through the years. Initially more mathematical, now it is more semantic	PI	19.42%	26.21%	24.27%	17.48	12.62%

The last smell investigated concerning *code debt* is **incorrect naming**; SE research has proven that a readable, intuitive code can reduce human mistakes when coding [9, 42]. Thus, Table 8 summarises responses in this regard.

In SE, naming conventions are considered more semantic if they are closer to a natural language description [42]. Therefore, a direct 'translation' of that idea in OR would imply that the second row is *not* a smell, but a correct practice (i.e., an "anti-smell"); in this case, any disagreement ('Somewhat' or 'Strong') implies the smell. Reversing Likert scales in some cases is a common practice [18], and we did it here so that both s-statements are worded in a *neutral way* rather than using wording such as 'instead' which could bias the response [68].

Therefore, we explored both options–mathematical and semantic naming, first and second rows of Table 8, respectively. As can be seen, a dramatic 82.5% of responses support semantic naming. However, it is possible that the statement was ambiguously worded; for example, 'to what they represent in the problem situation' can vary per discipline and may be inherently 'meaningful' or 'semantic' for a participant but vague for external readers'. As a result, in the Replication Package, we included two plots `Q12_Math_Approach` and `Q12_Semantic_Approach`, which corresponds to both s-statements in Table 8.

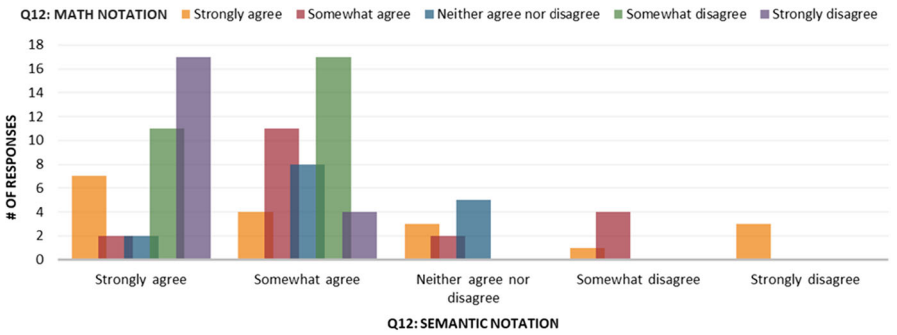

Fig. 3 Crossing responses to mathematical and semantic notation (s-statements in Table 8, respectively)

Additionally, Fig. 3 presents the correspondence between both s-statements. Namely, how many participants (flat count) responded to a given Likert in mathematical notation (first statement, colours) for each Likert of Semantic notation (horizontal axis).

This comparison is curious, as about 23% of responses agreed (to some extent) with both statements, but 47.6% agreed with the semantic notation (the second statement) but disagreed with the mathematical notation. Although this discloses respondents' bias, it also indicates that smells related to the notation are **plausible**. Namely, there are *traces of inconsistent practices* that should be further investigated.

When exploring the quadrants as potential reasons for this, it can be seen that RI weakly hints at a reason with a mixed agreement: most answers are located between Likert 2–4 (instead than in the 'Strong' agreements or disagreements). This is probably due to the participant's selected approaches, which in turn affects the programming languages they use (see Replication Package, figure `Q12_RI`); while those favouring 'optimisation' approaches tended to agree, those favouring 'data analytics' are predominantly neutral or had a strong disagreement.

Participants seem aware of the benefits of semantic names (i.e., PD responses) but have a diverse approach when regarding *why* they prefer mathematical notations. The RD-2 statement provides some reasons (pure preference or disliking longer names). The disagreement could be related to the fact that respondents do not fully align with the statement or with external choices such as the mathematical language.

Regarding PD, those using either 'optimisation' or 'simulation' approaches disagreed with this cause (see Replication Package, figure `Q12_PD`). This may *hint* that they do not consider 'semantic names' as meaningful or usable; a hypothesis is that this is related to the programming languages used or even the style accepted in relevant academic venues. However, further studies are needed to investigate this hypothesis.

Finally, PI appears to be a secondary reason, indicating that their practices have changed and improved over time; this is visible throughout all assessed approaches, as seen in Fig. 4.

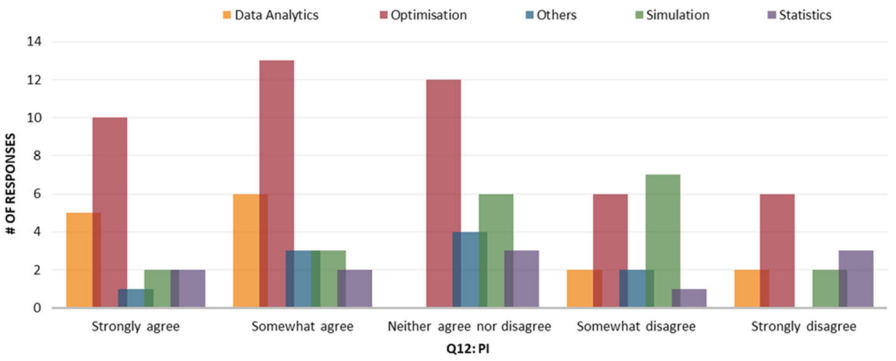

Fig. 4 PI responses to *naming conventions* per approach (flat count)

4.2 Documentation debt

Insufficient comments in the code is a common documentation smell. In SE, it has been proven that lack of comments leads to a lowered readability of the code, thus complicating its maintainability [29]. Moreover, recent studies have demonstrated scientific software has a similar usage of source code comments in data science [84], which is also positively perceived by students in computational sciences [87]. Therefore, Table 9 presents the results related to this.

As can be seen, the responses regarding this smell are skewed towards the negatives; this was another response with a reversed Likert scale [68]. Moreover, the trend is consistent across approaches used by participants (see Replication Package, figure `Q14_Approaches`). These results are *traces* of correct practices (thus including comments). Therefore, commenting is deemed a potentially **safe** practice.

When analysing the reasons, the negativity in RD hints that respondents may be aware of the benefits of commenting models. This seems related to the demographics, which indicate a higher proportion of experienced academics–those over 46 years old had an apparent disagreement, while those younger than 36 years old had a minor proportion of agreements (see Replication Package on `R14_RD`). This is consistent with SE research, which demonstrated that experienced developers write more comments in their code [78].

Regarding the *prudent* quadrant, the balance of agree/disagree answers in PD aligns with previous findings that indicate that participants often return to the model to improve the final version. However, more insights are available when analysed per favoured approaches (see Fig. 5). Those using 'optimisation' seem ambiguous, as there is a similar amount of responses between 'Somewhat Agree' to a 'Strong Disagree'. However, the other three main approaches (sans 'Others' are inclined toward an agreement. This may also be related to the programming languages used for each approach, although further studies are needed.

Thus, PI results indicate mixed practices, as results are somewhat balanced (namely, a roughly similar selection rate on each response, sans on 'Strongly Agree'). An interesting point is that those using 'optimisation' approaches tended to disagree with

Table 9 Results regarding lack of comments in mathematical programming [Question Q14]

Statement	Type	S.agree	Sw.agree	Neither	S.disagree	S.disagree
I don't usually write comments in my code, or write very few of them	Smell	7.22%	13.40%	13.40%	29.90%	36.08%
I don't write comments in my code, because a) I will remember what I wrote, or b) I don't have time to do it or c) I won't need this code again	RD	2.11%	13.68%	8.42%	29.47%	46.32%
I don't know how written comments may help me in reading or remembering code	RI	2.08%	2.08%	6.25%	26.04%	63.54%
I often go back to the code after the model is finished, and write a few comments to remember the general idea, or to have an overview at hand	PD	6.35%	36.46%	20.83%	20.83%	15.63%
I didn't write comments until I had to go back to my code and didn't remember what I wrote. Now, I write comments more often	PI	11.58%	18.95%	21.05%	22.11%	26.32%

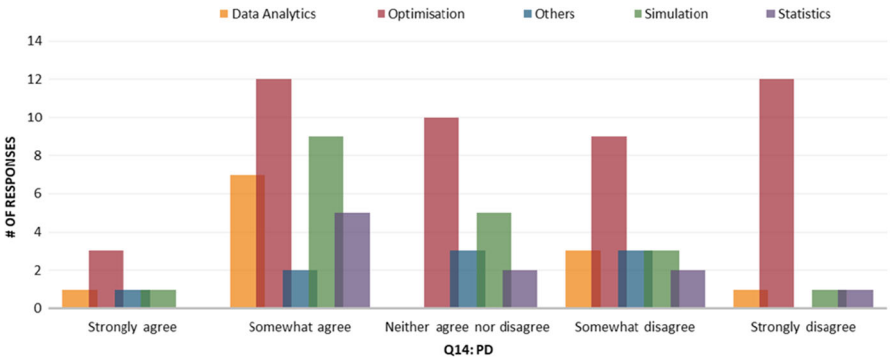

Fig. 5 PD responses to *insufficient comments* per approach [Question Q14]

this statement (see Replication Package `Q14_PI`); this represented almost a 17% of 'Strong Disagree' for this group.

Several areas for future works arise from these results, presented in Sect. 5.

We also enquired about **non-existent documentation** (as a **documentation debt** smell), and *excess comments* (related to *code debt*). These two are somewhat related to the above case and highlight that there is a very delicate balance in the proportion of adequate comments compared to lines of code. Given how similar they are to each other, we organised them in the same block. Table 10 presents the responses.

In both cases, *excess comments* (first row) obtained about 50% of agreement with only 25% of disagreements; however, *non-existent documentation* (second row) had about 58% of agreement with almost 25% of disagreements. Furthermore, Fig. 6 show-

Table 10 Results regarding excess comments and non-existent documentation [Question Q15]

Statement	Type	S.agree	Sw.agree	Neither	Sw.disagree	S.disagree
I often need to have a lot of comments in the code to clarify or remember what I did. Especially in complex cases or problems	Smell	15.63%	34.74%	16.84%	14.74%	10.53%
I prefer to write many comments, instead of keeping separated notes or documentation for the model	Smell	23.16%	34.74%	16.84%	14.74%	10.53%
I prefer to write many large comments, instead of simplifying the code or keeping separate notes	RD	9.47%	13.68%	23.16%	36.84%	16.84%
I don't know what I could do to stop needing large comments and to simplify my code, without changing assumptions for the representation of the problem	RI	2.11%	11.58%	21.05%	35.79%	29.47%
I don't know what kind of notes or documentation I could keep	RI	6.32%	7.37%	17.89%	32.63%	35.79%
I used to have complex code with large comments, but now I prefer to rework my code, keep notes or documentation	PI	2.11%	10.53%	34.74%	32.63%	20.00%

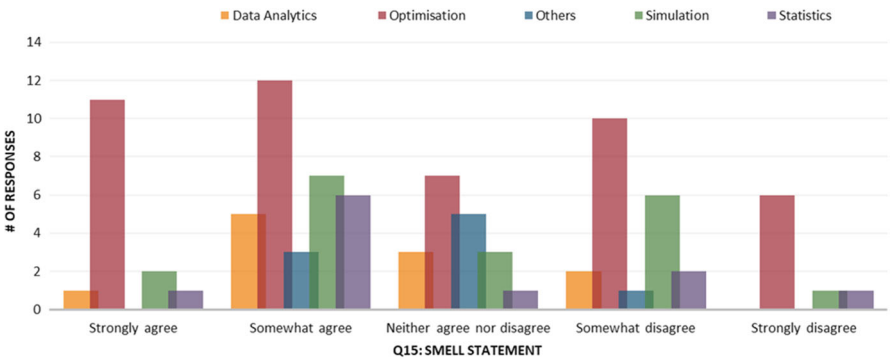

Fig. 6 Responses to excess comments per approach [Question Q15]

cases the responses according to the approach used by the practitioners. 'Statistics' seem somewhat neutral, while 'simulation' leans towards agreement (if combining both agreement answers). Also, while 'optimisation' leans towards an agreement, the responses are somewhat balanced. This is also consistent with the prior responses, indicating traces of favouring comments: if participants prefer comments, it is possible they just document the initial analysis of the model and nothing more. However, assessing this hypothesis remains future work. As a result, we considered these as *traces of the smell*, and considered it **recurrent**.

Table 11 Results concerning outdated and incomplete documentation [Question Q16]

Statement	Type	S.agree	Sw.agree	Neither	Sw.disagree	S.disagree
I keep notes at the start of the project, but don't often update them after changes in the model or problem	Smell	8.33%	28.13%	20.83%	33.33%	9.38%
I keep notes, but I often need to go back to the model, because my notes are insufficient to report the results or to write a paper about it	Smell	5.32%	31.91%	27.66%	29.79%	5.32%
Updating the documentation is too much effort, and I'm not going to go back to that model	RD	5.26%	21.05%	27.37%	33.68%	12.63%
I never realised that I should keep notes alongside a model, to remember and document what was done (besides writing an academic paper)	RI	3.16%	17.89%	20.00%	26.32%	32.6%
Before, I didn't care about my side notes or documentation, but now I update them because I find them useful to be able to go back to the previous models	PI	3.19%	18.09%	34.04%	30.85%	13.83%
I prefer to focus on the model first, and then write, update or improve my side notes or documentation	PD	13.68%	37.89%	21.05%	18.95%	8.42%

When assessing the causes, neither appear to have a substantial agreement, as they are heavily inclined to disagree. Therefore, we did not uncover traces of attitudes leading to the introduction of these smells. Exploring these by age groups or favoured approaches provided no additional insights. It can be hypothesised that excess comments are introduced as MP is scientific software of higher complexity, which could (to some extent) relate to findings on previous works [63]. However, a different approach and investigation will be needed to evaluate such a hypothesis.

Another *documentation debt* smell is the state of the accompanying notes or documents: they can often be **outdated** or **incomplete**, loosing helpfulness and being potentially damaging. This is a common problem in software development [54, 70], and was thus selected to be explored in OR. Table 11 summarises the responses in this area.

Both smells presented here have very close and even agree/disagree responses, indicating mixed practices. When analysed by the favoured approach (see Fig. 7), those in 'optimisation' mostly answered Likert 2–4, although they lean more towards agreeing with incomplete documentation. Those in 'simulation' are mostly neutral about incompleteness, but tend more to disagree with; the opposite happens with 'statistics'. Meanwhile, those in 'data analytics' are somewhat neutral to both smells. Therefore, we considered this as *traces of the smells*, and labelled them **plausible**. Practices seem to differ from each approach, possibly caused by programming languages or

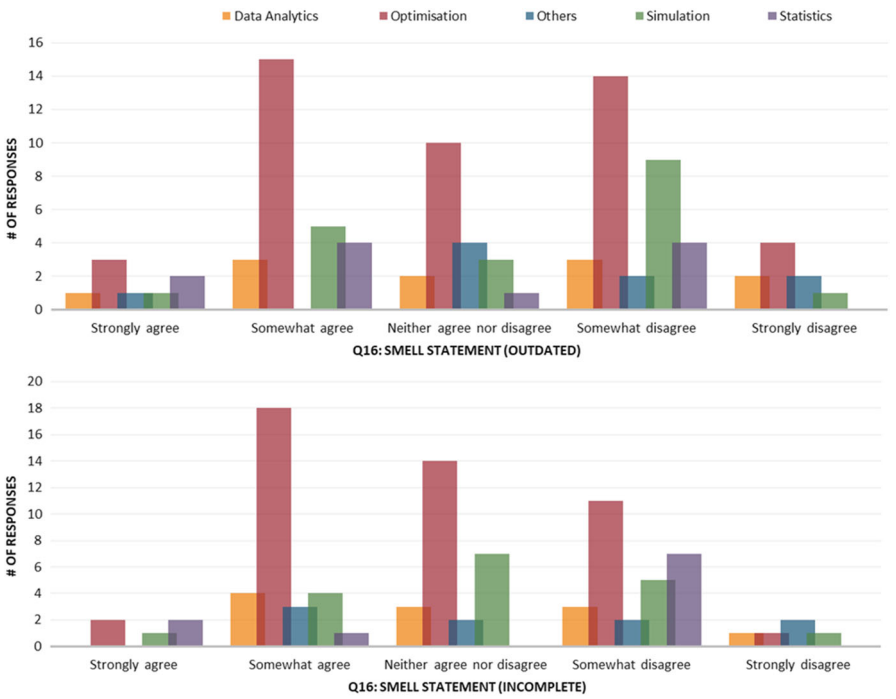

Fig. 7 Responses to incomplete/outdated documentation, per approach [Question 16]

styles needed for publication in academic venues. As a result, future works should be conducted in this regard, especially analysing source code.

In terms of attitudes (quadrants), RD and PI seem to have similar trends, oscillating between 'somewhat agree and disagree', but leaning towards the disagreement. Something more extreme happens with RI, as most responses are skewed towards the disagreement, regardless of age or MP approach; see Replication Package at `Q16_RI`.

However, the leading cause is *hinted* to be a PD attitude, which is presented in Fig. 8. This is consistent with previous responses indicating that respondents return to the model to polish it for the final version. When analysed per approach, it can be seen that those using 'simulation' do 'somewhat agree' but not strongly, which could hint at a variability of reasons or practices. 'Statistics' clearly leaned towards an agreement, while 'data analytics' does so to a lesser extent. Interestingly, those participants using mostly 'optimisation' approaches lean towards agreeing with the reason but have a considerable number of participants (about 26.3%) ranging from neutral to disagree. This may indicate that PD is not always the reason for those using 'optimisation'. Further analyses are needed to understand whether the programming language or academic publishing venues influence these reasons.

4.3 Versioning debt

The last smell explored is **code repository**. Version control is a class of systems responsible for managing changes to computer programs or collections of information.

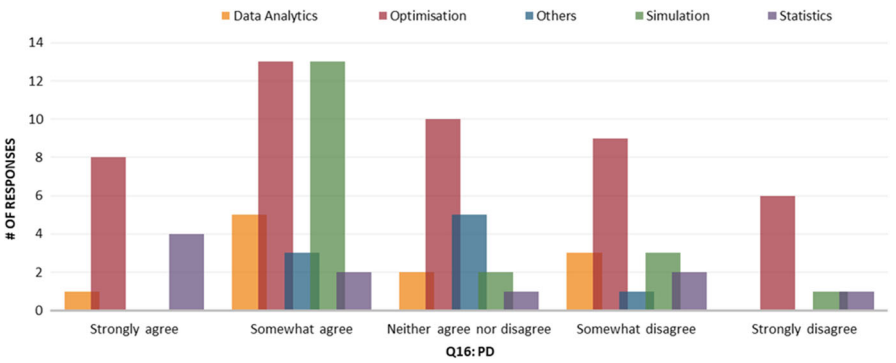

Fig. 8 PD responses to *incomplete/outdated documentation* per approach [Question Q16]

Its popularity has grown exponentially over the last decade [56], and it is also being taught in statistical or mathematical courses [15, 30, 46, 77]. Table 12 summarises the findings in this area.

The format of this question was suggested by the practitioners (Sect. 3.2.1), as what they experienced as 'usually seen'–namely, compressing project folders. However, the responses to the survey indicate strong traces of disagreement with this behaviour. When dividing this by approach, those using 'simulation' slightly lean toward agreement, but the difference remains minimal. The additional plot is available in the Replication Package as `Q17_Approach`. Thus, it is not possible to see traces of the 'counter example' practice of proper version control, and thus we labelled this as **safe**.

Furthermore, when exploring the quadrants' statements, most respondents seem aware of what version control is (RI1 has almost 61% of responses disagreeing, with 40.6% being 'Strong Disagree'). Likewise, almost 64% of participants disagreed with RI2 (not knowing what version control is; note that although we provided a list of systems (git, GitHub/Lab, BitBucket), the survey did not include a textual definition of version control, which could have biased the result).

RD1 (renaming files and commenting out) is interesting because there about 30.5% of responses agreed with the RD1 statement in Q11 (version control) but disagreed with Q11 (having commented-out dead code in their models). This is visible in Fig. 9. Currently, the survey did not provide enough evidence to understand why this contradiction happens, although respondents' bias is possible. Further studies are needed to understand the reason.

Lacking a team (RD2) had a balanced response slightly leaned towards the disagreement; therefore, there were few hints indicating this as a cause. Finally, PD had over 55% responses disagreeing with the statement; therefore, training and knowledge were not a cause not to use version control. The latter could be related to the current efforts of teaching version control in statistical or mathematical courses [15, 30, 46, 77]; even if our respondents were mostly mid- or late-career researchers, we could hypothesise they learned version control in order to teach it. Nevertheless, the information regarding version control was limited, and further studies are needed.

Table 12 Responses regarding the use of version control in OR [Question 17]

Statement	Type	S.agree	Sw.agree	Neither	Sw.disagree	S.disagree
I often compress or zip my model project, so I can "save" what I did in a previous version	Smell	7.29%	20.83%	14.58%	23.96%	33.33%
I thought that compressing or zipping projects was the best way of keeping previous versions	RI	3.13%	9.38%	27.08%	19.79%	40.63%
I don't know what version control is (e.g., [...])	RI	12.77%	14.89%	8.51%	13.89%	50.00%
I used to compress or zip my code, but then I discovered version control (e.g., [...])	PI	10.64%	15.96%	22.34%	14.89%	36.17%
I don't keep versions of my code. I just rename files or comment out fragments of outdated code	RD	9.47%	15.79%	14.74%	20.00%	40.00%
I don't work in teams (i.e., multiple people writing the code), so I don't need version control	RD	11.70%	19.15%	17.02%	17.02%	35.11%
I am aware of the benefits of version control (such as git), but my team is not. Training them would require time or budget we don't have	PD	8.42%	10.53%	22.11%	22.11%	36.84%

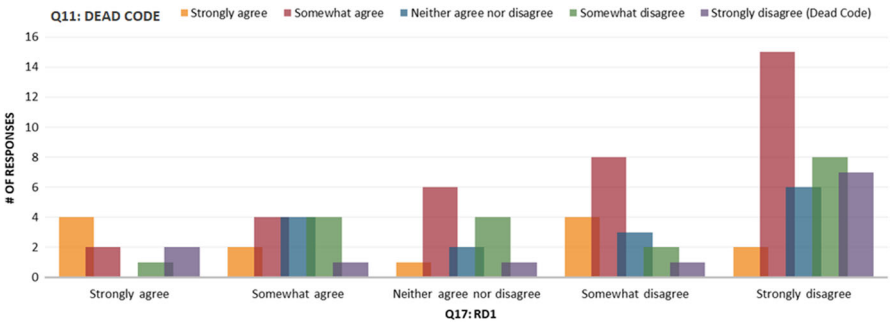

Fig. 9 Version Control's RD1, compared to Q11 (dead code) smell

5 Results & discussion

This section presents a detailed answer to each RQ based on the survey findings. Alongside the results, we also discuss the implications of the study.

5.1 RQ1: possible TD smells

As mentioned in Sect. 3, the survey only allowed us to infer traces of a behaviour/practice as a first step to direct the discussions further. During the analysis of survey

responses to each smell (see Sect. 4), each smell was categorised into one of four categories listed below. This categorisation was done based on visible trends between responses. Note that the analysis of the responses in Sect. 4 was presented as flat percentages of the total answers for that block of responses (i.e., empty responses were not counted). As a result, the differences were also established as flat percentages. The four categories are:

1. **Safe**. This represents a percentage difference favouring Likert values of 1–2 (Strongly/Somewhat Disagree). This difference ought to be over 25%, with a neutral value lower than 15%, indicating that most participants hinted at a trend towards best practices known for counteracting this smell.
2. **Plausible**. This was considered such in two situations. First, when all values (agreement, neutral and disagreement) approached 30%, indicating a balanced response with no clear trend to any side; e.g., the case of *incomplete documentation*. Second, with a difference between 7–10% towards best practices (Strongly/Somewhat Disagree), but a neutral value of about 20%; this usually meant that, although there was a hint of best practices, the neutrality could have represented mixed practices. We considered that, though there is a group of practitioners not favouring these smells, there is still a considerably large number exhibiting them through inadequate practices.
3. **Prone**. Opposed to *plausible*, it happens when the percentage difference is skewed towards the agreement (Strongly/Somewhat Agree), with a neutral of about 20%. As before, the large number of neutrals could indicate mixed practices, although when combined with more negative practices (i.e., the agreement), the traces of such behaviour were stronger.
4. **Recurrent**.These results provide strong traces that a TD smell may occur frequently. To be here, the percentage difference between agreement/disagreement had to be closer to 20% or larger (favouring agreement), with a neutral of about 16–20%.

The belonging of each smell to a category is summarised in Fig. 10, while individual descriptions smell-by-smell were addressed in Sect. 4.

Regarding specific smells, it can be summarised that:

– *Duplicated code* was the only smell evaluated twice in the survey. One of the assessments was prone and the other plausible; given the risk such practice represents for the maintainability of code, the authors decided to favour the 'riskier' category and thus label this smell as *prone*.
– Prior research demonstrated that problems with the documentation reduce the reproducibility of findings [80], effectively dampening the continuation of research through future works [69]. Our survey indicated several traces of problems regarding the documentation of MP. The chosen programming languages may affect documentation debt. However, considering the current trends and efforts toward open-science [82], such a lack should be further researched and assessed.
– Dead and duplicated code disclosed enough hints in the survey to be considered a strong trace. Further studies should be carried out to understand why this happens and how this can be solved through better programming habits and better support from languages/tools.

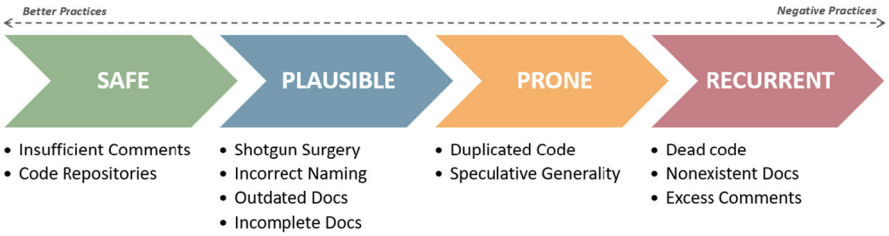

Fig. 10 TD smells trends uncovered through the survey's responses

Note that this analysis only indicates trends based on self-reported practices. Therefore, it is not possible to assert whether such behaviour happens at a given frequency; however, it does represent enough evidence to confirm that TD smells are happening in MP and should be further studied. Likewise, it does confirm that the similarity between MP and traditional software development, at least in regards to TD and programming practices [6, 62], is somewhat similar.

This exploration was done using a minimal adaptation of the well-established SE concepts into OR programming. As a result, it is possible that many nuances of TD in MP have not been detected; this is not considered a threat to the validity of this study, given that there is no knowledge of taxonomy and this survey was simply exploratory in nature. However, our findings enable a wide range of studies focusing on *how mathematical models are written* instead of what techniques and methods are applied (i.e., a purely Hard-OR approach [85]).

Additionally, this study was laid out with the primary goal of understanding what path to take in future research and in which areas to focus future works. Although there are many solid practices (i.e., as defined in SE) likely accepted in the MP community (i.e., commenting code and using version control), there are still many areas that need more exploration (i.e., excess of comments, dead code, and others). This avenues are further discussed in Sect. 5.3.

Question. *Which self-reported practices may hint the existence of TD in MP?*
Answer. Several practices have *strong traces* of behaviours often labelled as negative practices in SE. Duplicated and Dead Code had clear traces of negative behaviours, although further studies should be conducted, especially by analysing available code.
These were related to Speculative Generality (i.e., one minimal change requires multiple changes), the hypothesis linking this smell to code duplication should be further assessed, given that it may: a) introduce new errors in the code, b) complicating the maintenance, and c) increase the cost of sustaining the models over time.
Finally, Documentation (beyond code comments) had *strong traces* of negative practices. Given its effects on reproducibility [69, 80] and the current rise of open science [82], this should also be assessed.

5.2 RQ2: modellers' attitudes

The participants' responses to the statements related to the modellers' attitudes (i.e., the q-statements) when introducing TD were classified as a **weak trend** or a **strong trend** to a particular smell. Like before, these trends highlight a problem, hinting at a problematic attitude; in all cases, specialised investigations are required.

As indicated in Table 3, this was done according to the percentage of responses for a given Likert value associated with a particular quadrant. This categorisation was not as strict as the classification of the smells, mostly to account for subjectivity and other possible threats. Figure 11 summarises this information.a

We considered a *weak trend* when responses were balanced (namely, equally distributed) among agreement-neutral-disagreement (e.g., PI for *incorrect naming*), or when there difference between extremes was about 7–12% (e.g., RD in *duplicated code*). Note that PI in *shotgun surgery* was considered very weak (as discussed in Sect. 4.1), but included in the results. Likewise, a *strong trend* was considerably skewed towards agreement (strongly plus somewhat) with optionally a high value in the neutral cause. An example of this is PI and PD for the *dead code* smell.

Most causes of TD (code and documentation) tended towards *prudent* answers, with *inadvertent* being more common. As defined by Fowler [31], a *prudent* person is aware of best practices but introduces TD due to different, well-grounded reasons. In these cases, introducing TD is either a deliberate decision (PD) or a good-but-not-perfect solution (PI), improved by reflecting on previous and current practices. Given that this is the first study of its kind and we also had to develop the survey, it is possible that the PI answers were caused by respondent bias due to 'guessing ideal answers' in the survey. Nonetheless, as discussed in Sect. 3.4, we completed several steps to mitigate this risk.

Regarding specific practices, participants indicated a tendency to return to the model to polish the final version, including the documentation. These can be understood as *refactoring*. SE research has uncovered multiple advantages and disadvantages of refactoring and documentation at different stages of the life-cycle [48, 71]. However, these practices can also be affected by the fact that MP, as scientific programming is exploratory [6, 62] and perhaps more inclined to it. Like before, further studies are needed to assess refactoring and documentation in scientific software/MP.

These results are also consistent with the demographics of the survey, which point towards experienced academics. It has been repeatedly proven in SE that developers improve and polish their approaches and solutions as the years go by [10, 78]; thus, it is reasonable to assume that the same "evolution" happens in OR. However, surveys and studies regarding scientific software have demonstrated this is primarily coded by junior researchers [39, 63]; thus, it may be possible that a discrepancy between code and this survey exists. However, performing a 'mining software repositories' study of MP code was out of the scope of this exploratory analysis.

The final question of the survey also posed an open text space for participants to leave any insight considered valuable. Although no specific open-coding protocol was followed, upon close inspection (by both authors), it was possible to associate responses into two groups:

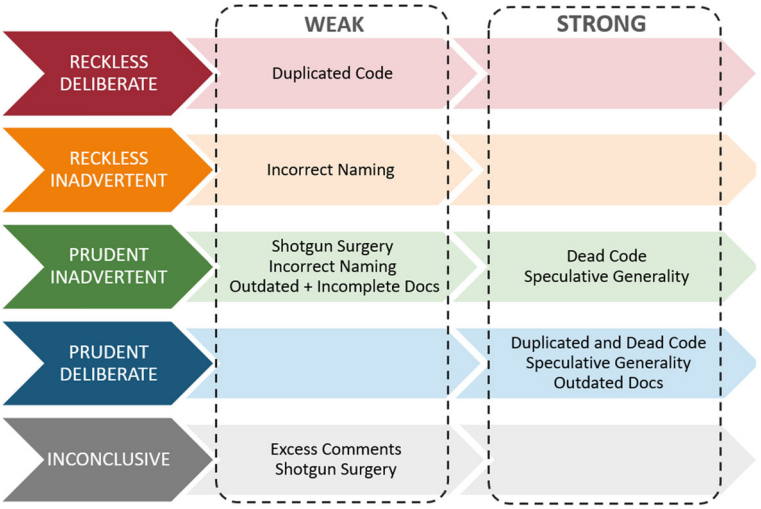

Fig. 11 Self-reported modellers' attitudes in terms of Fowler's TD Quadrants

– Several participants agreed on the similarities between "conventional programming" (i.e., software) and MP; one of them even stated that *"conditions hold in both situations"*. Furthermore, one of them addressed the positive effects of understanding computer architecture and programming to harness the full potential of MP. They said that *"there is also a need for modellers to understand how the underlying computer architecture affects a model (e.g., random numbers, floating-point arithmetic) as these can lead to significant errors over the life of a model in code"*.
– Many respondents disclosed that their practices change according to the experience of the workgroup members. For example, *"I follow different practices depending on the type of project (collaborative? consultancy? long-term?)"*. Teaching specific practices to new colleagues appears to be somewhat common, *"I use Git and Github for individual projects, group projects, and I also take the time to train new team members on using these technologies"*. Related to this, teamwork has also improved many participants' practices, leading them towards those limiting TD. Specifically, one participant commented that *"working with others has improved my code style greatly, commenting/documentation, variable names, use of version control"*.

Question. *What attitudes do modellers have regarding TD?*
Answer. Most causes are reportedly *prudent*, with a larger trend towards *inadvertent*. This could mean that, at any stage, the survey participants' practices evolved by 'tuning the coding style' (due to non-assessed reasons). Some PD responses confirm MP's exploratory nature (as it is a type of scientific software [39, 62]), but the process of maintenance and changes should be further studied. This is related to the *reckless deliberate* attitude on duplicated code matching the strong traces of that smell.

However, some smells were inconclusive in terms of attitudes (excess comments and shotgun surgery), and more analyses are needed.

5.3 Future research efforts

The overarching goal of this exploratory study was to identify where to focus future research efforts. As a result, we propose the following areas and research questions:

Duplicated code and shotgun surgery. Our survey indicates that modellers are aware of code duplication in their models, leading to strong traces of this smell. This smell is traditionally defined for conventional OOP software development [54, 70], were common programming paradigms (i.e., OOP) allow for an extensive reduction of duplicated code. Though newer MP languages provide some facilities in this area [40, 55], it remains an under-developed area of study. Moreover, our survey did not cover programming languages but approaches.

To study *duplicated code in MP*, we propose the following future questions. What are the causes for code duplication in MP? Are some mathematical approaches more prone to duplication than others? Is this enhanced or limited by the mathematical language? What are the possible mathematical solutions for this? Furthermore, questions tailored explicitly to *shotgun surgery* can also be posed. How can we measure the impact of a change when there is duplicated code in MP? How can we limit this? Are shotgun surgeries introducing programming errors in MP?

Incorrect naming. Results in this area align more with semantic naming but with fair use of mathematical notation. Thus, this was deemed as ambiguous, as the answers could have been affected by respondent bias, favoured programming languages, and participants' backgrounds. Therefore, it may be possible that the meaningfulness of names changes for some demographics (i.e., according to the programming background, the area of work, interdisciplinary work, and others). Furthermore, it is interesting to study if there are any 'implicit' naming conventions (e.g., using a specific naming convention for a type of variable).

Hence, some questions can be explored in this area. Is the meaningfulness of a naming convention subject to specific demographics? Are naming conventions influenced by programming background, size of teams, area of work and interdisciplinary (among others)? Is the naming affected by the type of programming language selected? Are there any specific "silently agreed" conventions commonly used in a particular area of work? How does this affect the reusability of models and their linkage with other discipline's products?

Excess of comments. This survey hints at modellers being prone to comment their code. Prior SE works investigated source code comments as a way of communication inside a team of developers [78], which opens two specific areas of future research for TD in MP: 1) Why do modellers need so many comments? 2) Do they disclose *self-admitted technical debt*?

The first one could be traced to the complexity of code, the versatility of work teams [39], how communication is handled, and the existence of different development life-

cycles [63]. It is possible to study several future questions. Is the number of comments somehow related to the complexity of the code? What is the 'tone' (i.e., sentiment analysis) of those comments? Which demographics (i.e., areas of work, interdisciplinary teams) are more prone to write more comments? How does the presence of comments affect the model documentation? What kind of information is being passed through the comments? Does this affect the model's solution to a problem situation?

The second one addresses a trending topic in SE. *Self-admitted technical debt* (SATD) happens when a developer (or modeller, in this case) willingly uses the code comments to indicate the presence of TD in the code [7]; this may occur either knowingly (e.g., "I solved it like this because it is faster") or unknowingly (e.g., "I don't know how to simplify this piece of code"). Since several SE studies have focused on SATD [7, 22, 44, 65, 89], it would be interesting to conduct differentiated replicas of those studies to discover more similarities and differences between traditional programming and MP.

Documentation (Non-existent, Outdated, Incomplete). Though previous research has posed some documentation standards, they are mostly oriented to documenting the problem situation and are used to draw agreement between stakeholders [1, 27, 50, 86]. Nonetheless, to the best of the authors' knowledge, there is little work regarding code documentation and the maintainability of the MPs as pieces of software.

Some future research questions are as follows. How can documentation improve reusability and maintainability in MP? What type of documentation would be suitable (i.e., little effort for high gains)? How can this documentation be aligned with a project life-cycle? Can this documentation reuse approaches from other disciplines (i.e., SE, deep learning, data science) to cater to interdisciplinary OR?

Besides pursuing the suggested "future questions", a recommended next step will be reinforcing our results through code and groupwork explorations. The former can be achieved by conducting studies of the type *mining repositories*, which implies systematically collecting code and searching for specific data inside it [24]. Moreover, the latter can be explored through grounded-theory-led workshops and unstructured interviews [41] to understand typical dynamics. Likewise, both could be combined in mixed-methods studies. Finally, multiple programming languages are used in MP and, more broadly, in OR; investigating how the provided functionalities affect development practices should also be pursued.

6 Conclusions

Mathematical Programming (MP) is an intrinsic part of Operational Research (OR). However, although it is known that MP is a different type of software development, scarce investigations have addressed programming practices in MP. Since this topic has often been addressed in Software Engineering (SE), this paper conducted a novel exploratory study into MP practices and attitudes based on the definitions of *technical debt* (TD) outlined by SE. Specifically, TD is a metaphor reflecting the implied cost of additional rework caused by choosing an easy solution now instead of using a better

approach that would take longer. This was based on the concept of MP being somewhat akin to software development, sharing many technical and process-related similarities, thus drawing concepts from the latter into OR.

We analysed results from a worldwide, online anonymous survey with 168 valid responses. It was developed iteratively and meant to assess specific TD smell for Code, Documentation and Versioning Debt. Results hinted that *code and documentation debt* have strong traces, thus being possibly common in MP. Other practices such as detailed commenting and *versioning debt* did not provide enough evidence of negative practices. Regarding attitudes, we determined that most debts are hinted to be deliberately introduced during development but then removed, as our results indicate that modellers are prone to rework their models.

Although similar studies are commonly conducted in SE, their application in OR and MP is unique. Therefore, the findings of this study are of interest to many groups, including modellers aiming to improve their development practices and those developing (or extending) languages used for MP. We also identified four areas for future work in terms of TD for MP: addressing duplicated code and shotgun surgery smell, incorrect naming, excess comments and the possibility of addressing self-admitted technical debt in OR, and code documentation. Finally, further studies (e.g., mining repositories and workplace exploration) are required to complement the data obtained through this study. Nonetheless, this first study successfully provides a direction to continue exploring this topic.

Funding Open Access funding enabled and organized by CAUL and its Member Institutions.

Data Availability Statement We provide a partial replication package in: https://doi.org/10.5281/zenodo.6757598. It includes the complete survey structure and the email invitation (with the Qualtrics' embedded fields). The participant collection sheet used for the convenience sample is shared empty to demonstrate the type of data collected; note that we cannot provide the completed sheet (which included the name, email and affiliation of invited participants) because our Ethical Protocol requires us to preserve participants' identity. This is a problem known as the 'privacy vs utility paradox' [53], and its study was out of scope for this inves-tigation. As mentioned before, the survey was implemented and distributed in Qualtrics. Qualtrics provides an advanced WYSIWYG ('what you see is what you get') editor for results reporting that also provides plots and summarises data (Qualtrics' official documentation is available at https://www.qualtrics.com/support/survey-platform/reports-module/results-section/reports-overview/?parent=p002). As a result, we used this system to produce the aggregated data. The tables in Sect. 4 were generated through Qualtrics (including Table 4); thus, there is no code/script available for this. Finally, some additional plots not included in the manuscript are part of the replication package, showing aggregated, unidentifiable data. The repository is in Zenodo, available at: https://doi.org/10.5281/zenodo.6757598.

Declarations

Conflict of Interest The authors declare that they have no conflict of interest.

Ethical Protocol The methodology used in this was approved by the ANU Human Ethics Research Committee (HREC), with project code 2020-23416-11101.

References

1. Abuabara, L., Paucar-Caceres, A., Belderrain, M.C.N., Burrowes-Cromwell, T.: A systemic framework based on soft or approaches to support teamwork strategy: An aviation manufacturer brazilian company case. J. of the Oper. Res. Soc. **69**(2), 220–234 (2018). https://doi.org/10.1057/s41274-017-0204-9
2. Ackermann, F., Alexander, J., Stephens, A., Pincombe, B.: In defence of soft or: Reflections on teaching soft or. J. of the Oper. Res. Soc. **71**(1), 1–15 (2020). https://doi.org/10.1080/01605682.2018.1542960
3. Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P.: The financial aspect of managing technical debt: A systematic literature review. Inf. and Software Technol. **64**, 52–73 (2015). https://doi.org/10.1016/j.infsof.2015.04.001
4. Arvanitou, E.M., Ampatzoglou, A., Chatzigeorgiou, A., Carver, J.C.: Software engineering practices for scientific software development: A systematic mapping study. J. of Syst. and Software **172**, 110848 (2021). https://doi.org/10.1016/j.jss.2020.110848
5. Avgeriou, P., Kruchten, P., Ozkaya, I., Seaman, C.: Managing Technical Debt in Software Engineering. Dagstuhl Reports **6**(4), 110–138 (2016). https://doi.org/10.4230/DagRep.6.4.110
6. Baek, N., Kim, K.J.: Prototype implementation of the opengl es 2.0 shading language offline compiler. Cluster Comput. **22**(1), 943–948 (2019). https://doi.org/10.1007/s10586-017-1113-z
7. Bavota, G., Russo, B.: A Large-Scale Empirical Study on Self-Admitted Technical Debt. In: Proceedings of the 13th International Conference on Mining Software Repositories, Association for Computing Machinery, MSR '16, pp. 315–326. USA (2016). https://doi.org/10.1145/2901739.2901742
8. Bedi, J., Kaur, K.: Understanding factors affecting technical debt. Int. J. of Inf. Technol. (2020). https://doi.org/10.1007/s41870-020-00487-9
9. Beniamini, G., Gingichashvili, S., Orbach, A.K., Feitelson, D.G.: Meaningful Identifier Names: The Case of Single-Letter Variables. In: IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp. 45–54 (2017)
10. Besker, T., Martini, A., Edirisooriya, L.R., Blincoe, K., Bosch, J.: Embracing Technical Debt, from a Startup Company Perspective. In: IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 415–425 (2018)
11. Besker, T., Ghanbari, H., Martini, A., Bosch, J.: The influence of technical debt on software developer morale. J. of Syst. and Software **167**, 110586 (2020). https://doi.org/10.1016/j.jss.2020.110586
12. Borup, N.B., Christiansen, A.L.J., Tovgaard, S.H., Persson, J.S.: Deliberative technical debt management: An action research study. In: Wang, X., Martini, A., Nguyen-Duc, A., Stray, V. (eds.) Software Business, pp. 50–65. Springer International Publishing, Cham (2021)
13. Bostelmann, H.: Automated assessment in a programming course for mathematicians. MSOR Connect. **18**(2), 36–44 (2020)
14. Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., Zazworka, N.: Managing Technical Debt in Software-Reliant Systems. In: FSE/SDP Workshop on Future of Software Engineering Research, Association for Computing Machinery, FoSER '10, pp. 47–52. USA (2010). https://doi.org/10.1145/1882362.1882373
15. Bryan, J.: Excuse me, do you have a moment to talk about version control? The Am. Statistician **72**(1), 20–27 (2018). https://doi.org/10.1080/00031305.2017.1399928
16. Cadavid, H., Andrikopoulos, V., Avgeriou, P., Klein, J.: A Survey on the Interplay between Software Engineering and Systems Engineering during SoS Architecting. Association for Computing Machinery, chap 2, pp. 1–11. New York, NY, USA (2020). https://doi.org/10.1145/3382494.3410671

17. Capilla, R., Mikkonen, T., Carrillo, C., Fontana, F.A., Pigazzini, I., Lenarduzzi, V.: Impact of opportunistic reuse practices to technical debt. In: 2021 IEEE/ACM International Conference on Technical Debt (TechDebt), pp. 16–25 (2021). https://doi.org/10.1109/TechDebt52882.2021.00011
18. Chyung, S.Y.Y., Roberts, K., Swanson, I., Hankinson, A.: Evidence-Based Survey Design: The Use of a Midpoint on the Likert Scale. Perform. Improv. **56**(10), 15–23 (2017). https://doi.org/10.1002/pfi.21727
19. Codabux, Z., Vidoni, M., Fard, F.H.: Technical Debt in the Peer-Review Documentation of R Packages: A rOpenSci Case Study. In: IEEE/ACM 18th International Conference on Mining Software Repositories, IEEE, pp. 195–206. USA (2021). https://doi.org/10.1109/MSR52588.2021.00032
20. Codabux, Z., Williams, B.: Managing technical debt: An industrial case study. In: 4th International Workshop on Managing Technical Debt (MTD), pp. 8–15 (2013)
21. Cruzes, D.S., ben Othmane, L.: Threats to validity in empirical software security research. Empirical Res. for Software Secur.: Foundations and Exp. **1**(1), 277–302 (2017)
22. da Silva Maldonado, E., Shihab, E., Tsantalis, N.: Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt. IEEE Trans. on Software Eng. **43**(11), 1044–1062 (2017)
23. D'Ambrosio, C., Lodi, A., Wiese, S., Bragalli, C.: Mathematical programming techniques in water network optimization. European J. of Oper. Res. **243**(3), 774–788 (2015). https://doi.org/10.1016/j.ejor.2014.12.039
24. de F. Farias, M.A., Novais, R., Júnior, M.C., da Silva Carvalho, L.P., Mendonça, M., Spínola, R.O.: A Systematic Mapping Study on Mining Software Repositories. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, Association for Computing Machinery, SAC '16, pp. 1472–1479. USA (2016) https://doi.org/10.1145/2851613.2851786
25. dos Reis, J.P., e Abreu, F.B., Carneiro, G.D.F.: Code Smells Incidence: Does It Depend on the Application Domain? In: 10th International Conference on the Quality of Information and Communications Technology (QUATIC), pp. 172–177 (2016)
26. Durán, A.J., Pérez, M., Varona, J.L.: The misfortunes of a trio of mathematicians using computer algebra systems. can we trust in them. Notices of the AMS **61**(10), 1249–1252 (2014)
27. Eden, C., Ackermann, F.: Theory into practice, practice to theory: Action research in method development. European J. of Oper. Res. **271**(3), 1145–1155 (2018). https://doi.org/10.1016/j.ejor.2018.05.061
28. Ernst, N.A., Bellomo, S., Ozkaya, I., Nord, R.L., Gorton, I.: Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt. In: 10th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, ESEC/FSE 2015, pp. 50–60. USA (2015). https://doi.org/10.1145/2786805.2786848
29. Fakhoury, S., Ma, Y., Arnaoudova, V., Adesope, O.: The Effect of Poor Source Code Lexicon and Readability on Developers' Cognitive Load. In: IEEE/ACM 26th International Conference on Program Comprehension (ICPC), pp. 286–28610 (2018)
30. Fiksel, J., Jager, L.R., Hardin, J.S., Taub, M.A.: Using GitHub Classroom To Teach Statistics. J. of Statistics Education Publisher, Taylor & Francis, UK **27**(2), 110–119 (2019). https://doi.org/10.1080/10691898.2019.1617089
31. Fowler, M.: Technical debt quadrant. Martin Fowler pp. 14–0 (2009). https://martinfowler.com/bliki/TechnicalDebtQuadrant.html
32. Fowler, M.: Refactoring: Improving the design of existing code. Addison-Wesley Professional (2018)
33. Freire, S., Rios, N., Mendonça, M., Falessi, D., Seaman, C., Izurieta, C., Spínola, R.O.: Actions and Impediments for Technical Debt Prevention: Results from a Global Family of Industrial Surveys. In: Proceedings of the 35th Annual ACM Symposium on Applied Computing, Association for Computing Machinery, SAC '20, pp. 1548–1555. USA (2020). https://doi.org/10.1145/3341105.3373912
34. Fu, S., Shen, B.: Code Bad Smell Detection through Evolutionary Data Mining. In: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–9 (2015)
35. Ghazi, A.N., Petersen, K., Reddy, S.S.V.R., Nekkanti, H.: Survey research in software engineering: Problems and mitigation strategies. IEEE Access **7**, 24703–24718 (2019). https://doi.org/10.1109/ACCESS.2018.2881041
36. Graziotin, D., Fagerholm, F., Wang, X., Abrahamsson, P.: On the unhappiness of software developers. In: 21st International Conference on Evaluation and Assessment in Software Engineering, ACM, EASE'17, pp. 324–333. USA (2017). https://doi.org/10.1145/3084226.3084242
37. Greenacre, Z.A.: The importance of selection bias in internet surveys. Open J. of Statistics **6**(3), 8 (2016). https://doi.org/10.4236/ojs.2016.63035

38. Guevara-Vega, C., Bernárdez, B., Durán, A., Quiña-Mera, A., Cruz, M., Ruiz-Cortés, A.: Empirical strategies in software engineering research: A literature survey. In: 2021 Second International Conference on Information Systems and Software Technologies (ICI2ST), pp. 120–127 (2021). https://doi.org/10.1109/ICI2ST51859.2021.00025
39. Hannay, J.E., MacLeod, C., Singer, J., Langtangen, H.P., Pfahl, D., Wilson, G.: How Do Scientists Develop and Use Scientific Software? In: ICSE Workshop on Software Engineering for Computational Science and Engineering, IEEE, pp. 1–8. Publisher IEEE, Vancouver, Canada (2009). https://doi.org/10.1109/SECSE.2009.5069155
40. Hart, W.E., Laird, C.D., Watson, J.P., Woodruff, D.L., Hackebeil, G.A., Nicholson, B.L., Siirola, J.D.: Pyomo-Optimization Modeling in Python, vol. 67. Springer (2017)
41. Hoda, R., Noble, J., Marshall, S.: The impact of inadequate customer collaboration on self-organizing agile teams. Inf. and Software Technol. **53**(5), 521–534 (2011). https://doi.org/10.1016/j.infsof.2010.10.009
42. Hofmeister, J., Siegmund, J., Holt, D.V.: Shorter identifier names take longer to comprehend. In: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 217–227 (2017)
43. Holvitie, J., Leppänen, V., Hyrynsalmi, S.: Technical Debt and the Effect of Agile Software Development Practices on It - An Industry Practitioner Survey. In: 2014 Sixth International Workshop on Managing Technical Debt, pp. 35–42 (2014)
44. Huang, Q., Shihab, E., Xia, X., Lo, D., Li, S.: Identifying self-admitted technical debt in open source projects using text mining. Empirical Software Eng. **23**(1), 418–451 (2018). https://doi.org/10.1007/s10664-017-9522-4
45. IE, Grossmann, Apap, R.M., Calfa, B.A., García-Herreros, P., Zhang, Q.: Recent advances in mathematical programming techniques for the optimization of process systems under uncertainty. Comput. & Chem. Eng. **91**, 3–14 (2015). https://doi.org/10.1016/j.compchemeng.2016.03.002. (**12th International Symposium on Process Systems Engineering & 25th European Symposium of Computer Aided Process Engineering (PSE-2015/ESCAPE-25), 31 May - 4 June 2015, Copenhagen, Denmark**)
46. Jones, M., Megeney, A.: Programming in groups: Developing industry-facing software development skills in the undergraduate mathematics curriculum. MSOR Connections (2020)
47. Kallrath, J.: Mathematical Optimization and the Role of Modeling Languages, pp. 3–24. Springer US, Boston, MA (2004). https://doi.org/10.1007/978-1-4613-0215-5_1
48. Kaya, M., Conley, S., Othman, Z.S., Varol, A.: Effective software refactoring process. In: 6th International Symposium on Digital Forensic and Security (ISDFS), pp. 1–6 (2018). https://doi.org/10.1109/ISDFS.2018.8355350
49. Lacerda, G., Petrillo, F., Pimenta, M., Guéhéneuc, Y.G.: Code smells and refactoring: A tertiary systematic review of challenges and observations. J. of Syst. and Software **167**, 110610 (2020). https://doi.org/10.1016/j.jss.2020.110610
50. Lami, I.M., Tavella, E.: On the usefulness of soft or models in decision making: A comparison of problem structuring methods supported and self-organized workshops. European J. of Oper. Res. **275**(3), 1020–1036 (2019). https://doi.org/10.1016/j.ejor.2018.12.016
51. Lee, E.K.: Innovation in big data analytics: Applications of mathematical programming in medicine and healthcare. In: IEEE International Conference on Big Data (Big Data), pp. 3586–3595 (2017)
52. Li, J., Huang, Q., Xia, X., Shihab, E., Lo, D., Li, S.: Is Using Deep Learning Frameworks Free? Characterizing Technical Debt in Deep Learning Frameworks. In: 42nd International Conference on Software Engineering: Software Engineering in Society, ACM, ICSE-SEIS '20, pp. 1–10. USA (2020). https://doi.org/10.1145/3377815.3381377
53. Li, T., Li, N.: On the Tradeoff between Privacy and Utility in Data Publishing. In: 15th International Conference on Knowledge Discovery and Data Mining, pp. 517–526. ACM, USA (2009)
54. Li, Z., Avgeriou, P., Liang, P.: A systematic mapping study on technical debt and its management. J. of Syst. and Software **101**, 193–220 (2015). https://doi.org/10.1016/j.jss.2014.12.027
55. Lubin, M., Dunning, I.: Computing in Operations Research Using Julia. INFORMS J. on Comput. **27**(2), 238–248 (2015). https://doi.org/10.1287/ijoc.2014.0623
56. Majumdar, R., Jain, R., Barthwal, S., Choudhary, C.: Source code management using version control system. In: 6th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), pp. 278–281 (2017)

57. Martini, A., Vajda, S., Vasa, R., Jones, A., Abdelrazek, M., Grundy, J., Bosch, J.: Technical Debt Interest Assessment: From Issues to Project. In: Proceedings of the XP2017 Scientific Workshops, Association for Computing Machinery, XP '17, p. 6. USA (2017). https://doi.org/10.1145/3120459.3120469
58. McConnell, S.: Technical debt. 10x software development. Blog] (2007). Available at: http://blogsconstruxcom/blogs/stevemcc/archive/2007/11/01/technical-debt-2aspx
59. Ning, C., You, F.: Optimization under uncertainty in the era of big data and deep learning: When machine learning meets mathematical programming. Comput. & Chem. Eng. **125**, 434–448 (2019). https://doi.org/10.1016/j.compchemeng.2019.03.034
60. O'Brien, F.A.: On the roles of or/ms practitioners in supporting strategy. J. of the Oper. Res. Soc. **66**(2), 202–218 (2015). https://doi.org/10.1057/jors.2013.130
61. Pascarella, L., Palomba, F., Di Penta, M., Bacchelli, A.: How is video game development different from software development in open source? In: IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), pp. 392–402 (2018)
62. Pehlivan, H.: Designing and interpreting a mathematical programming language. Sakarya University J. of Sci. **23**, 1027–1041 (2019). https://doi.org/10.16984/saufenbilder.494974
63. Pinto, G., Wiese, I., Dias, L.F.: How do scientists develop scientific software? an external replication. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp. 582–591. Campobasso, Italy (2018). https://doi.org/10.1109/SANER.2018.8330263
64. Pirabán, A., Guerrero, W., Labadie, N.: Survey on blood supply chain management: Models and methods. Comput. & Oper. Res. **112**, 104756 (2019). https://doi.org/10.1016/j.cor.2019.07.014
65. Potdar, A., Shihab, E.: An Exploratory Study on Self-Admitted Technical Debt. In: 2014 IEEE International Conference on Software Maintenance and Evolution, pp. 91–100 (2014)
66. Ralph, P., Tempero, E.: Construct validity in software engineering research and software metrics. In: 22nd International Conference on Evaluation and Assessment in Software Engineering, ACM, EASE'18, pp. 13–23. USA (2018)
67. Ranyard, J., Fildes, R., Hu, T.I.: Reassessing the scope of or practice: The influences of problem structuring methods and the analytics movement. European J. of Oper. Res. **245**(1), 1–13 (2015). https://doi.org/10.1016/j.ejor.2015.01.058
68. Redmiles, E.M., Acar, Y., Fahl, S., Mazurek, M.L.: A summary of survey methodology best practices for security and privacy researchers. Technical Report CS-TR-5055, UM Computer Science Department, (2017). https://doi.org/10.13016/M22K2W
69. Resnik, D.B., Shamoo, A.E.: Reproducibility and Research Integrity. Accountability in Res. **24**(2), 116–123 (2017). https://doi.org/10.1080/08989621.2016.1257387
70. Rios, N., de Mendonça Neto, M.G., Spínola, R.O.: A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. Inf. and Software Technol. **102**, 117–145 (2018). https://doi.org/10.1016/j.infsof.2018.05.010
71. Rios, N., Mendes, L., Cerdeiral, C., Magalhães, A.P.F., Perez, B., Correal, D., Astudillo, H., Seaman, C., Izurieta, C., Santos, G., Oliveira Spínola, R.: Hearing the Voice of Software Practitioners on Causes, Effects, and Practices to Deal with Documentation Debt. In: Madhavji, N., Pasquale, L., Ferrari, A., Gnesi, S. (eds.) Requirements Engineering: Foundation for Software Quality, pp. 55–70. Springer International Publishing, Cham (2020)
72. Rocha, J.C., Zapalowski, V., Nunes, I.: Understanding Technical Debt at the Code Level from the Perspective of Software Developers. In: Proceedings of the 31st Brazilian Symposium on Software Engineering, Association for Computing Machinery, SBES'17, pp. 64–73. USA (2017). https://doi.org/10.1145/3131151.3131164
73. Sharma, R., Kamble, S.S., Gunasekaran, A., Kumar, V., Kumar, A.: A systematic literature review on machine learning applications for sustainable agriculture supply chain performance. Comput. & Oper. Res. **119**, 104926 (2020). https://doi.org/10.1016/j.cor.2020.104926
74. Sierra, G., Shihab, E., Kamei, Y.: A survey of Self-Admitted Technical Debt. J. of Syst. and Software **152**, 70–82 (2019). https://doi.org/10.1016/j.jss.2019.02.056
75. Silva, V., Junior, H., Travassos, G.: A Taste of the Software Industry Perception of Technical Debt and its Management in Brazil. Journal of Software Engineering Research and Development **7**, 1:1–1:16 (2019). https://doi.org/10.5753/jserd.2019.19.https://sol.sbc.org.br/journals/index.php/jserd/article/view/19
76. Singer, J., Sim, S.E., Lethbridge, T.C.: Software Engineering Data Collection for Field Studies, pp. 9–34. Springer London, London (2008). https://doi.org/10.1007/978-1-84800-044-5_1

77. S. Sinkovits R., D. Soto O.: Introducing Computing and Technology through Problem-Solving in Discrete Mathematics. In: Practice and Experience in Advanced Research Computing, Association for Computing Machinery, PEARC '20, pp. 429–435. New York, NY, USA (2020). https://doi.org/10.1145/3311790.3396620
78. Steidl, D., Hummel, B., Juergens, E.: Quality analysis of source code comments. In: 21st International Conference on Program Comprehension (ICPC), pp. 83–92 (2013)
79. Storer, T.: Bridging the chasm: A survey of software engineering practice in scientific programming. ACM Comput. Surv. **50**(4), 1–32 (2017). https://doi.org/10.1145/3084225
80. Taylor, S.J.E., Eldabi, T., Monks, T., Rabe, M., Uhrmacher, A.M.: Crisis, What Crisis – Does Reproducibility in Modelling and Simulation Really Matter? In: 2018 Winter Simulation Conference (WSC), pp. 749–762 (2018). https://doi.org/10.1109/WSC.2018.8632232
81. Tseng, M.L., Tan, R.R., Chiu, A.S., Chien, C.F., Kuo, T.C.: Circular economy meets industry 4.0: Can big data drive industrial symbiosis? Resources, Conserv. and Recycl. **131**, 146–147 (2018). https://doi.org/10.1016/j.resconrec.2017.12.028
82. Vicente-Saez, R., Martinez-Fuentes, C.: Open science now: A systematic literature review for an integrated definition. J. of Business Res. **88**, 428–436 (2018). https://doi.org/10.1016/j.jbusres.2017.12.043
83. Vidoni, M., Cunico, L., Vecchietti, A.: An Empirical Framework to Applying Unit Testing in Operational Research. In: 48th International Conference on Computers and Industrial Engineering (CIE48), Auckland, New Zealand, code 144541 (2018)
84. Vidoni, M.: Self-admitted technical debt in r packages: An exploratory study. In: IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) (MSR), IEEE Computer Society, pp. 179–189. Los Alamitos, CA, USA (2021). https://doi.org/10.1109/MSR52588.2021.00030
85. Vidoni, M.: Beyond hard and soft or: Operational research from a software engineering perspective. J. of the Oper. Res. Soc. **0**(0), 1–23 (2021). https://doi.org/10.1080/01605682.2020.1865848
86. Vidoni, M., Cunico, L., Vecchietti, A.: Agile operational research. J. of the Oper. Res. Soc. **0**(0), 1–15 (2020). https://doi.org/10.1080/01605682.2020.1718557
87. Vieira, C., Magana, A.J., Falk, M.L., Garcia, R.E.: Writing in-code comments to self-explain in computational science and engineering education. ACM Trans. Comput. Educ. **17**(4), 1–21 (2017). https://doi.org/10.1145/3058751
88. Wang, B., Song, Y., Cui, X., Cao, J.: Mathematical programming for server consolidation in cloud data centers. In: 4th International Conference on Systems and Informatics (ICSAI), pp. 678–683 (2017)
89. Wehaibi, S., Shihab, E., Guerrouj, L.: Examining the Impact of Self-Admitted Technical Debt on Software Quality. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol 1, pp. 179–188 (2016)
90. Weibezahn, J., Kendziorski, M.: Illustrating the Benefits of Openness: A Large-Scale Spatial Economic Dispatch Model Using the Julia Language. Energies **12**(6), 1153 (2019)
91. Xavier, L., Ferreira, F., Brito, R., Valente, M.T.: Beyond the Code: Mining Self-Admitted Technical Debt in Issue Tracker Systems, Association for Computing Machinery, pp. 137–146. New York, NY, USA (2020).https://doi.org/10.1145/3379597.3387459
92. Yamashita, A., Moonen, L.: Do developers care about code smells? an exploratory survey. In: 20th Working Conference on Reverse Engineering (WCRE), pp. 242–251 (2013)
93. Yi, W., Chi, H.L., Wang, S.: Mathematical programming models for construction site layout problems. Autom. in Constr. **85**, 241–248 (2018). https://doi.org/10.1016/j.autcon.2017.10.031
94. Zazworka, N., Shaw, M.A., Shull, F., Seaman, C.: Investigating the Impact of Design Debt on Software Quality. In: Proceedings of the 2nd Workshop on Managing Technical Debt, Association for Computing Machinery, MTD '11, pp. 17–23. USA (2011). https://doi.org/10.1145/1985362.1985366