CASE STUDY

Automated assessment in a programming course for mathematicians

Henning Bostelmann, Department of Mathematics, University of York, York, UK.
Email: henning.bostelmann@york.ac.uk. https://orcid.org/0000-0002-0233-2928.

Abstract

The paper reports on a programming course for undergraduate Mathematics students in their 2nd year, with some parts compulsory for single-subject students. Assessment takes the form of several programming projects. Formative feedback as well as summative assessment is aided by automated unit tests, which allow for rapid and consistent marking, while focussing marker's time on students who require the most help.

Keywords: programming, automated assessment, unit tests, Java.

1. Introduction

Computer Science originates in Mathematics; computers are based on mathematical rules and, early on, programming was seen as a mathematical activity (Dijkstra 1974). In fact, fundamental mathematical concepts like sets, functions, their domains and codomains, have their counterpart in modern programming languages. Vice versa, computer technology plays a central role for Mathematics, its applications and its importance in society; Jaffe (1984) wrote that *"no reflection of mathematics about us is more striking than the omnipresent computer",* and this is even more valid in today's environment.

In that light, it is the author's opinion that in undergraduate Mathematics, the teaching of computer programming should be treated on equal footing to other core mathematical subjects, both in importance and level; though not everyone agrees.[1]

Certainly, there are substantial differences to usual Mathematics teaching. For one, the UK admissions process filters applicants by their mathematical abilities, but not their programming experience; hence a wide range of backgrounds needs to be catered for.

More importantly, in feedback and marking on programming tasks, an even more prominent focus than usual needs to be put on *outcomes:* computer programs need to produce the *correct* result, exactly adhering to the specified problem, and unlike perhaps in usual Mathematics assessment, there is typically no meaningful partially correct solution, unless it implements partial functionality. Without doubt, writing well-structured and well-documented programs is important (*"A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts [...]"* – Knuth 1996, p.2); but first and foremost, students as well as teachers need to verify whether code works correctly. To that end, they need to *test* it with input data; just reading the source

[1] *"I never wanted to study coding and won't need it in my future profession so I don't see the practicality and relevance of it. If I wanted to learn about coding, I would have studied computer science - this is a waste of my time."* – Anonymous student feedback 2018/19.

code is not a reliable way of verification. However, taking this in earnest for marking involves a large amount of tedious manual work, and might therefore be neglected for time reasons.

In the present case study, we report on a programming course for mathematics students in which feedback and marking is assisted with automated methods, more technically, *automated unit tests*. These are used both for giving rapid feedback to students in computer practicals, and for summative assessment of programming projects.

They also aid consistency of marking and remove bias, while focusing marker's time on important tasks such as written feedback.

2. Background

The module in question is an introductory programming course for Mathematics students in the second year. It has been taught, in slightly varying forms, at the University of York since the academic year 2013/14, with the author as module leader.

2.1. Module design

The 10 credit module is intended for single subject Mathematics students in their 2nd year, students on some joint programmes, as well as Natural Sciences students who may take it as an elective. It has no prerequisites beyond Calculus and Algebra at first-year level; in particular, no knowledge of programming is assumed, even if a proportion of the audience has varying levels of experience with programming in some context.

The syllabus is split into a basic part, introducing procedural programming (variables, expressions and assignments; data types, including floating point numbers; loops and conditional structures; functions; arrays; character strings; input/output), and an advanced part, including fundamentals of object-oriented programming (dynamic methods and inheritance). These are presented along *applications* from Mathematics, such as approximation algorithms, which are not systematically taught (beyond a brief discussion of roundoff errors in floating point arithmetic), but presented as examples in exercises or lectures, without formal justification. Besides programming techniques, the module also aims to teach associated skills, including documentation.

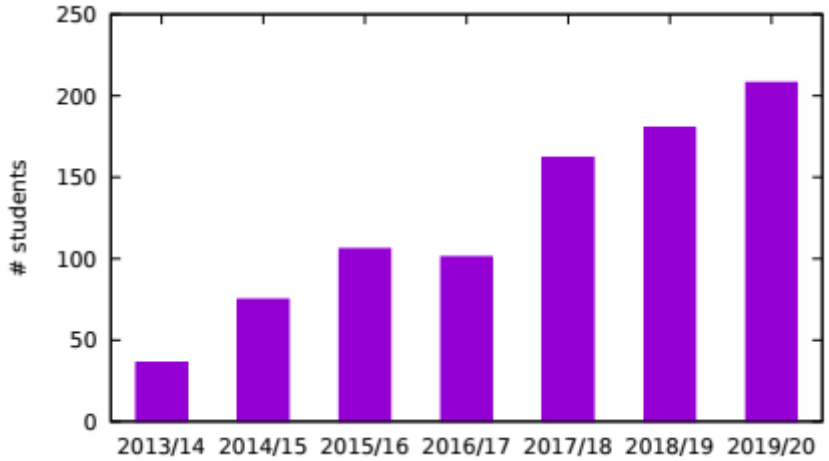

Figure 1: Participant numbers by academic year; until 2016/17: optional module 'Programming and Scientific Computing', from 2017/18: compulsory module 'Mathematical Skills 2'.

Until the year 2016/17, all of these topics were taught as one optional module 'Programming and Scientific Computing'. From 2017/18 on, the content was incorporated into a new compulsory module 'Mathematical Skills 2: Programming and Recent Advances': Here students are taught the basic parts of the programming syllabus (approximately 2/3 of the material) during the Autumn term; in the Spring term, students can choose between either the advanced programming syllabus or an essay topic in Pure Mathematics, Applied Mathematics or Statistics. (These choices will not be discussed further in this article.)

Participant numbers have been rising year on year (Fig. 1) and now exceed 200 students.

2.2. Technical setup

The module aims to teach foundational programming techniques, not (only) for use in mathematical applications, but as a general employability skill. To that end, a general purpose programming language is used, rather than an application-, product-, or vendor-specific system, which should demonstrate all conceptual aspects of a modern programming language (the author includes type strictness here). It is methodologically difficult to determine which languages are most frequently used in practice, in particular in an industry setting; that said, existing surveys indicate (TIOBE, 2020; Bissyandé et al., 2013) that languages of the C family (including C++, Java and vendor-specific derivatives) continue to be used widely, if not overwhelmingly. Of these, Java appears to be the best suited in a beginner's teaching setting.

As a development environment for Java, the module uses BlueJ, which was developed specifically for education purposes. In a comparatively simple user interface, BlueJ allows students to easily invoke individual functions of their programs, as well as offering a built-in debugger (also usable for demonstrations in lectures) and integrated unit test tools, which will become important below.

While BlueJ was originally intended for an 'Objects First' approach (Barnes and Kölling, 2016), the module uses it for a more traditional procedural approach for its basic part, along the first chapters of Nielsen (2009), with object-oriented techniques introduced only later.

Beyond the standard installation of BlueJ, the module uses – as an example for an external third-party library – the Apache Commons Math library (Apache 2020).

Teaching materials are provided to students via a Moodle-based VLE, which is also used for some randomised multiple choice quizzes (cf. Sec. 4).

2.3. Automated assessment

As a particular feature, and focus for this case study, the module aims to use automated assessment methods for rapid feedback to students as well as for summative assessment. Specifically, teacher-provided automated unit tests are employed for this purpose.

Unit tests are short pieces of program that run the developer's code with certain input data, and compare its output to expected values. They are a longstanding and commonly used tool in software development (see, e.g., Runeson 2006). Here we use the same techniques for checking that a student's work conforms to the specification given in an exercise description.

This allows student as well as teachers to verify rapidly whether a student's work is functionally correct. It is a common misconception that programming work (even of a simple kind) can reliably be checked by reading its source code – even missing out on small errors might make the reader assume that the program is 'correct', whereas it actually never performs the desired functionality. In other words, student work (as any other program) needs to be *tested* with relevant input data, rather

than cross-read by the teacher. Unit tests allow us to do this efficiently, freeing up teacher's time for other important aspects of the feedback process. Usage of these tests in formative and summative assignments is further described in Sec. 3 and 4 respectively.

Note that all unit test code in the module is provided by the *teacher* – students are not required to write unit tests or to understand the program code underlying them, they simply invoke them from a graphical user interface. Test code uses the unit test framework JUnit 4 (JUnit 2020), which is embedded into BlueJ.

3. Teaching and formative assessment

Teaching is centred around 9 computer practical (6 for the basic and 3 for the advanced part). Each practical is supported by 2 lectures which aim to introduce the relevant programming techniques and demonstrate examples.

Each practical comes with an exercise sheet that is released to students a few days in advance of the session, and a corresponding code template. During the session, students work independently on the exercises, but with a teacher present for help; the student/teacher ratio is approximately 15:1.

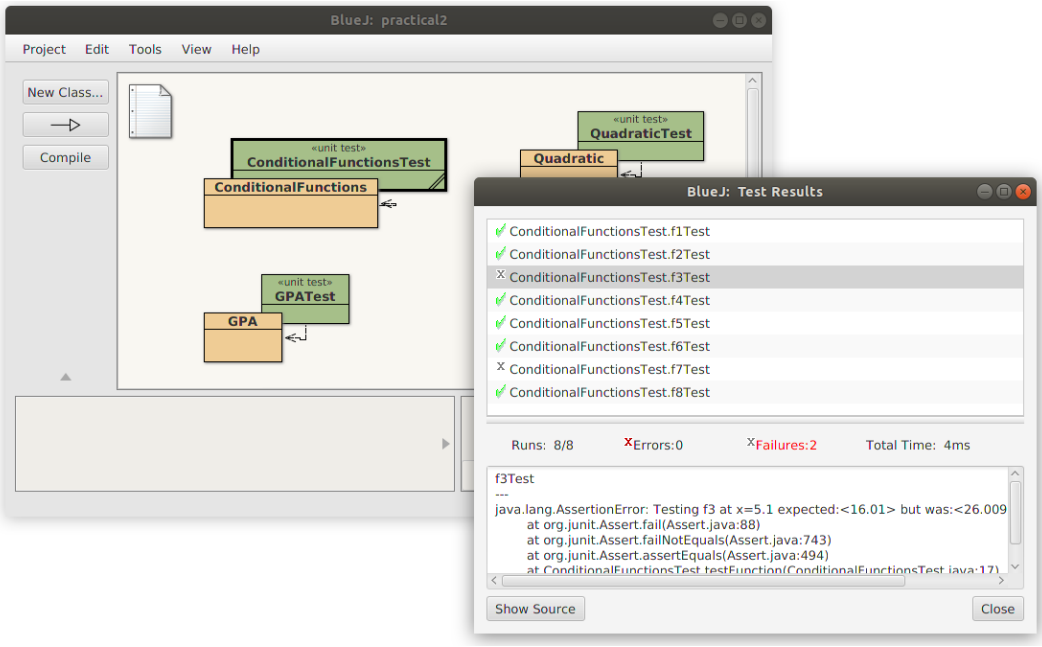

Figure 2: Unit tests within the practical materials. Screenshot from BlueJ

Unit tests are used in the practicals for rapid feedback. That is, unit tests are supplied with the code template for each exercise. Students will write code for the relevant exercise, and are requested to first run their code with relevant input data to verify that it works correctly. Once completed, they run the unit test and are presented with a 'traffic lights' result (Fig. 2). If the unit test passes, they continue to the next exercise; otherwise they can ask a teacher for assistance. (Failed unit tests will display a brief explanatory message, but this is often not detailed enough for students to localise the problem.)

Programming to pass these unit tests does require students to stick to the specifications on the exercise sheet very strictly, in particular to use the correct signatures (function or method names, input and output parameters and their data types), if not already given in the code template. This is in fact an intended learning outcome: in any collaborative programming setting – as students would encounter it in real-world applications – sticking to an agreed programming interface is crucial.

Each exercise sheet contains more exercises than can be solved in one hour (except by the most experienced students); the remaining exercises are left as homework, with no hand-in, but with feedback available via unit tests, and questions answered in the following practical session.

4. Summative assessment

The module is assessed on open assessments (Table 1), mainly consisting of 3 programming projects. These projects are marked semi-automatically; see Sec. 4.1 for details. Since the number of students on the module makes it impossible to set individual project topics, collusion may present an issue; we discuss this in Sec. 4.2.

Assessment component	time to work on assignment	weight in module	length (LOC)	proportion of automated marks	time from deadline to mark release	manual marking time per submission
Programming project 1	1 week	5%	50-100	100%	1 day	n/a
Programming project 2	6 weeks	25%	200	66%	3 weeks	20 min
Programming project 3	7 weeks	50%	300	50%	3 weeks	30 min
Online quizzes	1 week	15%		100%	immediate	n/a
Careers exercise	2 weeks	5%				

Table 1: Assessment components in Mathematical Skills 2, for students who choose the advanced programming part. Length of project given in Lines of Code (LOC) excluding documentation. All numbers except mark weights are approximate and may vary between years.

In addition to the projects, assessment includes a number of online quizzes based on multiple choice questions drawn from a random pool, which are presented and automatically marked via the Moodle VLE, and which students complete in their own time. These should be seen mostly as reading comprehension tests: they can be answered from absorbing the lecture material, without actually writing a program. The intention is to ease beginner students into the topic and allow them to collect marks for basic understanding. A careers exercise, which also counts towards the module mark, is not further discussed here.

4.1. Programming projects

In setting and marking the programming projects, assessment must be divided into two main parts: *functional* aspects (i.e., whether the code works correctly to specification) and *non-functional* aspects (whether the code is easy to read, well-structured and well-documented). In both aspects, student numbers require a distributed approach to marking, raising potential issues with marking consistency.

In all projects, functional aspects are marked with the aid of unit tests, which are prepared by the examiner in advance, but are not released to students. Student submissions are first verified against these unit tests, and a mark as well as an error report automatically generated. Marks are awarded entirely on the criterion whether the unit tests pass (typically 1 mark per test). The automated score for the functional part is only modified by the lead marker, and only under strict conditions (typically, when an entire part of the project fails to work due to a minor deviation, such as a mis-named function).

This semi-automatic marking process can be used in two ways: First, it is possible to set rapid feedback assignments (Project 1, cf. Table 1) which are marked *exclusively* on functional aspects, and almost fully automatically. Marks and automated error reports can then be released as little as 1 day after the deadline; only submissions with particularly low scores receive separate written feedback by the examiner. Verbal feedback to all students who request it is then given in the following practical session.

Second, for longer projects (Project 2 and 3), a distributed marking process is used: Automated reports are generated and shared with 3-4 markers; their role is then to determine *why* (not whether) the code fails to work, and to write corresponding feedback to students. This guarantees objectivity for this part of the assessment: the score is not subject to an individual marker's decision. It also allows markers to spend more time on feedback (rather than manual testing), and reduces possible oversights. In addition, markers assign scores for non-functional aspects along a structured marking guide and with brief per-item feedback; these items relate to the overall code structure, specific aspects of the code (e.g., expected function calls), adherence to code style conventions, and completeness of the documentation – students are typically asked to add Javadoc comments to their code.

We found the automated marking process to work smoothly in general, though some submissions need manual fixing – for example, where students use an incorrect directory structure in their submitted code. These errors typically occur for 1-2% of students and are corrected by the lead marker without penalty.

Use of automated marking in this way requires some care, because students can easily fail a large number of these tests by small omissions that may not immediately be obvious. Exact adherence to the given specification (in particular signatures) is required for tests to yield marks. To some extent, this is only a reflection of the reality of the subject: in software projects, specifications and conventions must strictly be followed to arrive at a working product.

On the other hand, some mitigation against such 'catastrophic failures' needs to be provided. To that end, the code template for every project contains a single unit test (the 'declaration test') that verifies not the functionality, but rather the function declaration and signatures in the student's code, using the Java Reflection API. Students are advised to use this test before submission to verify their basic code structure and fix any discrepancies.

4.2. Academic integrity

Project-based open assessment is an appropriate format for this module, since it comes fairly close to a realistic programming setting. While assessment by closed exam would be possible (and is sometimes used in similar situations, often with restriction on Internet access and availability of other resources), it has severe drawbacks: First, because of the limited time available, only the most elementary question can be asked, and aspects e.g. of code structure and documentation need to be dropped; second, it is a highly artificial setting which does not occur in real-world applications – no programmer would, in practice, do their work without Internet access.

However, since student numbers prohibit the setting of individual project topics, plagiarism and collusion between students is a significant concern, as probably with most other open assessments in Mathematics. In fact, it is quite common to see some students closely following their peer's solutions, with changes only in naming of variables, whitespace, etc. While rare, it has also occurred that students submitted almost literal copies of each other's work, sometimes in the form of binary identical files.

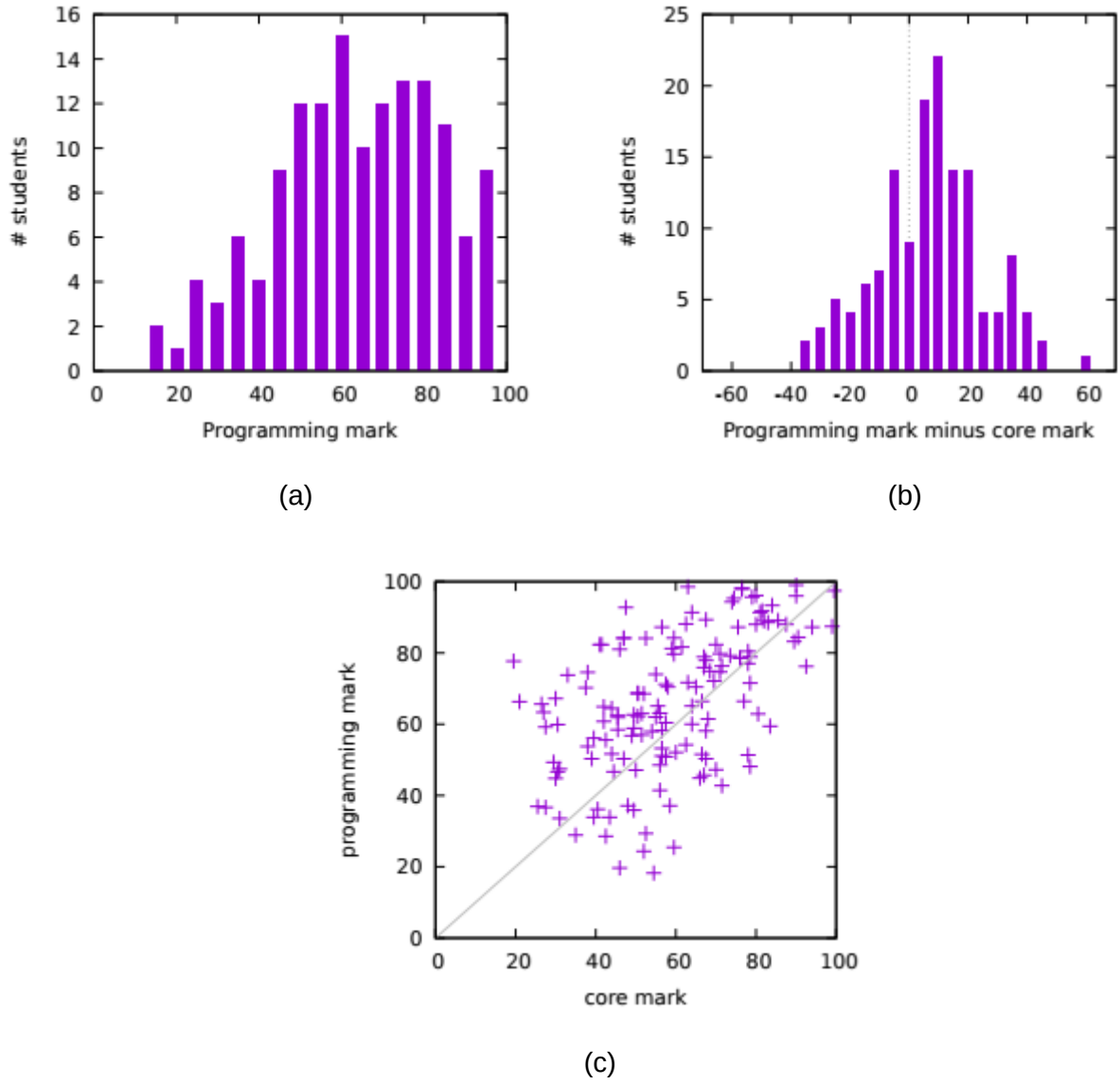

Figure 3: Programming marks vs. core marks in 2018/19. (a) Frequency of marks for the programming components; (b) frequency of differences between programming and core marks; (c) scatter plot of individual student's results.

In order to deal with this situation, marking of projects is assisted by an automatic similarity check. Specifically, we make use of the software JPlag (KIT, 2020; Prechelt et al., 2002) which is able to highlights similarities in code between student's submissions, ignoring trivial changes such as whitespace, comments, renaming of identifiers, and swapping of lines. Similarities identified by the tool are then investigated manually. We typically found relatively high numbers of similarities in Project 1 (sometimes affecting more than 5% of the submissions in various clusters), which are dealt with by means of mark reductions. After that, collusion cases in projects 2 and 3 are rare.

5. Student performance

Since programming appears to require somewhat different skills than typical Mathematics modules, and the module counts towards the degree classification, it is fair to ask how student results on the module compare with those in more traditional subjects.

To that end, we compare summative marks from the compulsory, 'basic' programming component with exam results from two other modules that most participants take in the same term, Vector Calculus and Linear Algebra; the average of these two will be referred to as the 'core' mark. Only students which took all three modules are included in the comparison; also, students who missed any of the assessments, or who had resits 'as if for the first time' approved for any of them, have been excluded from the analysis.

The results for 2018/19 are shown in Fig. 3. One notes that the mark distribution for the programming part alone (Fig. 3a) is within usual expectations, though possibly somewhat high in the 80-100 range, as is typically seen with coursework-based assessments. The differences between programming marks and core marks (Fig. 3b) show that there is a good correlation between general Mathematics performance and marks in programming, while there are a few wide outliers. This is confirmed by the scatter plot in Fig. 3c. In short, while there are some Mathematics students with good marks that struggle with programming tasks (and vice versa), these seem to be an exception rather than the rule.

6. Conclusions

The module discussed in this paper demonstrates that it is possible to integrate a generic programming course as a compulsory element into a Mathematics programme. It scales to the size of undergraduate cohorts, with student performance in line with expectations. Automated unit tests, when suitably set up, help with providing rapid feedback, ensuring consistency of marking, and making prudent use of staff resources.

7. References

Apache Software Foundation, 2020. *Commons Math library*. Available at: http://commons.apache.org/proper/commons-math/ [Accessed 24 February 2020].

Barnes, D.J. and Kölling, M., 2016. *Objects first with Java: a practical introduction using BlueJ*, 6th edition. London: Pearson.

Bissyandé, T.F., Thung, F., Lo, D., Jiang, L. and Réveillère, L., 2013. Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects*, IEEE 37th Annual Computer Software and Applications Conference, Kyoto*. pp. 303-312. http://doi.org/10.1109/COMPSAC.2013.55.

Dijkstra, E.W., 1974. Programming as a Discipline of Mathematical Nature. *The American Mathematical Monthly,* 81(6), pp.608-612. http://doi.org/10.1080/00029890.1974.11993624.

Jaffe, A., 1984. Ordering the Universe: The Role of Mathematics. *SIAM Review,* 26(4), pp.473-500.

JUnit project team, 2020. *JUnit 4*. Available at: https://junit.org/junit4/ [Accessed 19 February 2020].

Karlsruhe Institute of Technology (KIT), 2020. *JPlag - detecting software plagiarism.* Available at: https://jplag.ipd.kit.edu/ [Accessed 10 February 2020].

Knuth, D.E., 1996. *Selected Papers on Computer Science*. Chicago: University of Chicago Press.

Nielsen, F., 2009. *A Concise and Practical Introduction to Programming Algorithms in Java.* Springer: London. http://doi.org/10.1007/978-1-84882-339-6.

Prechelt, L., Malpohl, G. and Philippsen, M., 2002. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11), pp.1016-1038. http://doi.org/10.3217/jucs-008-11-1016.

Runeson, P., 2006. A survey of unit testing practices. *IEEE Software,* 23(4), pp.22-29. http://doi.org/10.1109/MS.2006.91.

TIOBE Software BV, 2020. TIOBE Index. Available at: https://www.tiobe.com/tiobe-index/ [Accessed 11 February 2020].