Abstract — Most Canadian engineering students take a computing course in their first year that introduces them to digital computation. The Canadian Engineering Accreditation Board does not specify the language(s) that can or should be used for instruction. As a result, a variety of languages are used across Canada. This study examines which languages are used in degree-granting institutions, currently and in the recent past. It also examines why institutions have chosen the languages that they currently use. In addition to the language used in instruction, the types and hours of instruction are also analyzed. Methods of instruction and evaluation are compared, as well as the pedagogical philosophies of the different programs with respect to introductory computing. Finally, a comparison of the expected value of this course to graduates is also presented.

We found a more diverse landscape for introductory computing courses than anticipated, in most respects. The guiding ethos at most institutions is skill and knowledge development, especially around problem solving in an engineering context. The methods to achieve this are quite varied, and so are the languages employed in such courses. Most programs currently use C/C++, Matlab, VB and/or Python.

Keywords: computing, programming language, course, introductory, pedagogy, Canadian

1. INTRODUCTION

Like many post-secondary institutions in Canada, Mount Royal University (MRU), in Calgary, Alberta, offers an engineering transfer program that allows students to finish their engineering degrees at another institution after the first year or two of study. Most MRU students transfer to the University of Calgary or to the University of Alberta. Recently, these two institutions diverged in their introductory computing courses, each utilizing a different language of instruction. MRU subsequently offered versions of the basic course in both languages. This sparked an internal debate as to which might be more pedagogically effective, and that led to the question “what language(s) do other Canadian programs employ?” This question was the genesis of this study.

From that initial motivation, the scope of the study grew to include logistical details for these courses, as well as pedagogical philosophies regarding desired outcomes, evaluation tools, and teaching techniques. The ultimate purpose of this study has become one of characterizing the Canadian engineering landscape in terms of first-year engineering computing courses. The reasons to do so are numerous. Best practices can be identified and shared. Norms can be determined. Potential weaknesses can be identified and rectified, and new and better ways of thinking about this curricular material can emerge.

In undertaking this study, the natural place to begin was the Canadian Engineering Accreditation Board (CEAB), of Engineers Canada. The CEAB sets standards for engineering degree-granting programs in Canada. They do not, however, set clear parameters for any introductory computing course. Indeed, “computers” and “computing” are almost never mentioned in the 2011 Accreditation Criteria and Procedures [2]. Under section 3.4.4.1 of this document (pp. 17), reference is made to 225 academic units of engineering science being required by graduation. Computer science is included in this category. Under section 3.5, notice is also made of the quality, suitability, and accessibility of computing facilities (pp. 20). Beyond these two passing mentions, the only other reference (not explicitly stated) is in section 3.1.5 under the use of engineering tools. There, the document describes how programs are required to expose students to modern engineering tools, to apply those tools to a range of activities, and to know their limitations (pp. 13). Computing would certainly qualify as such a tool. In summary, Canadian engineering programs have a lot of freedom as to how they integrate computing into their curriculum.

While curriculum defines content to be taught, pedagogy deals with the manner in which teaching and learning activities are designed in order to facilitate the achievement of learning goals. In recent surveys of the
literature on teaching introductory programming, three types of learning goals typical to programming courses have been described: problem solving, learning a particular programming language, and code/system production [8, 9]. However, there is no agreement about which of these should be the primary goal in an introductory course. One movement argues that “good pedagogy requires the instructor to keep initial facts, models and rules simple, and only expand and refine them as the student gains experience” [12] and that the targets of an introductory programming course should be transferable skills such as problem solving and algorithm development e.g. [5]. However, others have concluded that one cannot expect transfer of problem solving skills to happen in an introductory course, since the time students spend in practice is too limited e.g. [7]. One reason for the persistence of this debate is the prevalence of small-scale studies in specific instructional settings, making it difficult to draw generalizable conclusions about the underlying issues associated with learning programming.

Given the vagueness of the CEAB guidelines regarding curriculum, and a lack of agreement in the literature regarding pedagogy, the authors felt it would be worthwhile to characterize the current Canadian landscape in terms of introductory programming courses as thoroughly as possible. This has involved examining logistical issues, evaluation strategies, instructional techniques, graduate outcomes, pedagogical philosophies, and any and all languages employed.

The results were surprisingly varied in many respects. There is a large diversity of approaches across Canada, including the use of several different languages for instruction. Programs are often passionate about their choice of instructional language(s), and these choices are based on the differential weighting of several outcome criteria such as employability, problem solving abilities, learning core programming principles, and thinking algorithmically.

2. METHODS

2.1. Data Gathering

The CEAB maintains a list of all currently accredited Canadian engineering degree-granting institutions [3]. Program leaders (associate deans and directors) at all institutions on this list were contacted via email to solicit assistance in filling out a short survey (see Table 1). In most cases, course instructors and/or coordinators completed the surveys. The survey was also translated into French and this version was sent to all French language institutions.

Table 1: Survey questions.

| Q1. What is the name of your 1st year engineering computing course? |
| Q2. What is your program’s pedagogical philosophy towards this course i.e. what is it for, why do you have it, how should it be taught, what are the main goals, how should it benefit students, etc? |
| Q3. What language(s) does your program use in its 1st year engineering computing course? |
| Q4. What does your program consider to be the main advantages of using this language(s) versus others? |
| Q5. Does your program believe that there are any disadvantages to using this language(s)? |
| Q6. Why doesn’t the program use a language like (Matlab/C++)? |
| Q7. What language(s) did your program use in this course 5 years ago? |
| Q8. Is the program anticipating changing the languages(s) used in the 1st year engineering computing course in the next year or two? If so, what kind of language(s) are being considered? |
| Q9. How many hours of lectures, labs and tutorials do you have, per week, in the current course? |
| Q10. How does your program teach the application of the course’s content e.g. model problems/solutions, homework problems, projects, tests/quizzes? |
| Q11. How does your program evaluate the components of this course e.g. attendance, quizzes, tests, exam, assignments, projects, essays? |
| Q12. Five years after the students have taken this course, what does your program feel the main value of it will be to them? |

2.2. Analyses

Questions 1, 3, 7, 8 and 9 easily lent themselves to quantitative analysis. For the remaining questions, a mixed qualitative/quantitative approach was applied. First, all French responses were translated into English. Then, all answers for a given question were collated. A thematic analysis was carried out to identify underlying topics in each response. Responses for a similar topic were grouped. If responses were exactly or very highly redundant, they were consolidated in a single response with a noted weighting corresponding to the number of such responses. These consolidated response summaries were then analyzed for frequency with respect to the various topical themes and the specific responses within them. Also, the nature of the language used to express an idea was also examined, where relevant. For example, in some cases, opinions were stated assertively as facts.

Since this paper contains only synopses of the consolidated response summaries, readers are encouraged to contact the authors if they would like a copy of these consolidated responses.
3. RESULTS

3.1 Overview

There are 43 engineering degree-granting institutions in Canada [3]. A total of 37 took part in the study i.e. an 86% participation rate. From those 37 institutions, we used at least some information from 57 introductory computing courses, including 3 from Atlantic Canada, 14 from Quebec, 28 from Ontario and 12 from Western Canada. From within those numbers, four institutions had a total of 6 programs that did not actually have a first-year introductory computing course for engineers. Their introductory courses were offered in second-year. But in the spirit of the study, we included data from those courses in our dataset. Conversely, a few programs had more than one computing course in first-year. In those cases, we only used data from the first course i.e. the Fall course. A total of 11 institutions had more than one version of their core introductory computing course, with the largest number being 7 variants at one institution. Most of the 11 multi-version institutions had 2-3 variants.

3.2 Question 1 – Course Name

Many respondents provided the name of the course and its course code. The codes provided insights beyond the course names. While this course code dataset was only about 65% complete (36 responses), it was clear that first-year introductory computing courses are almost equally likely to be taught as computer science courses as they are to be taught as engineering or applied science courses (45% vs. 55%, respectively).

As for the course names, themselves, we took the nouns, adjectives and verbs of each title and computed relative frequencies. The results are well illustrated in a “word cloud” (see Figure 1, from using www.wordle.net) where the size of a word is proportional to the frequency of its occurrence.

![Word Cloud](image)

Fig. 1 – Word cloud for course names

In terms of some of the ideas that were present in course titles, there were 34 instances where a “level” was mentioned e.g. Introduction. The phrase “comp” appeared 33 times, while “eng” appeared 17 times, and “sci” appeared 8 times. Specific languages were mentioned in only 7 cases.

3.3 Question 2 – Pedagogical Philosophy

This pedagogical philosophy question garnered the largest number of comments of all the survey questions. The comments were grouped into 9 fairly distinct categories: skill development (~30% of the total comment volume), knowledge development (20%), pedagogical approach (11.5%), general prep (10%), specific prep (7.5%), discipline specific (7%), language specific (5%), use of labs/lectures (5%), and evaluation philosophy (3%).

The largest response category for Q2 was that of skill development. The skills that programs want students to learn are many and varied. However, a few skill sets clearly stand out. About one third of all comments focused on the idea of becoming skilled at developing algorithms and/or applying methodical, systems and/or structured thinking. In partnership with that theme, a large number of comments focused on developing an idea and following it through with effective code (including debugging and testing). Ultimately, this would be all directed at the third major response: problem solving skills. Secondary responses that were still fairly common included correct choice and use of data structures, solving numerical problems and performing complex calculations, and developing good programming techniques. Other noted skills included communications, modeling, efficiency evaluation, task decomposition, descriptions of program outputs, satisfaction of requirements, and use of the computer as a tool.

The second largest category of responses for Q2 was that of knowledge development i.e. learning facts and concepts, and developing understanding. There were a few key themes that emerged through these comments. About 20% of them spoke to the importance of understanding or knowing the basics, essentials and fundamentals of programming. Another large group focused on the idea of introducing students to computing and its importance as an engineering tool. It was deemed important to understand several topics including the how’s, why’s and limitations of computing, as well as the power, flexibility, value and importance of computing. As a group, there were a number of comments referring to the understanding of processes such as algorithmic thinking, problem solving, software development, program design, and engineering design and modeling. Other topics to know or understand included numerical methods, programming in a high-level language, programming in an object-oriented (OO) language, use of alternative computing methods, logic, and the various computing paradigms available today (procedural, functional, object oriented, event driven). A few mentions were made of more specific and basic programming knowledge such as loops, arrays, syntax, classes, types, usability, readability, and testing.
The third largest response category was that of pedagogical approach. These comments focused squarely on teaching and learning, whereas most of the comments in the other categories were more obliquely relevant to this category. There were few clear patterns in this dataset, except for the fairly common occurrence of two themes: teach programming in an engineering context, and this is an introductory experience so the basics are important. Beyond that, there was a wide array of ideas touching on many points relevant to teaching strategy e.g.

- show the breadth of engineering with this course
- use a language chosen by the specific discipline
- use a high-level structured language
- teach object-oriented fundamentals
- teach the essentials of procedural programming
- use other methods (spreadsheets) as appropriate
- develop structured thinking in students
- focus on individual skills/assignments
- focus on problem solving vs. coding (C/C++)
- use a systems approach
- teach engineering design methods
- actively engage students and limit class sizes
- start the term by focusing on task decomposition
- don’t waste time on pretty GUIs (C)

General preparation was the next most popular category. A number of clear themes emerged here. The most common theme (noted in about a 3rd of all responses) was that of establishing a solid foundation or framework for programming and/or problem solving using computers. Indeed, problem solving itself was quite a common theme, as was the understanding and use of computing tools. Another major theme was that of preparing for the needs of the future, whether that was in future course work and assignments, or in employment. One respondent called this course “indispensable” for meeting future needs. The other major theme focused on the idea of being able/capable of solving real problems and doing challenging calculations on one’s own, in the future. One respondent also noted the need to prepare students so that they could easily adapt to other languages as needed.

Specific preparation was the next biggest category. This category refers to the computing course fulfilling a specific need for some sort of student preparation. Half of all these comments referred to the need for competency in a specific language in future courses or course work. A few other comments referred to the need for numerical processing ability in the future. A couple of comments referred to low-level computing abilities for motors, instrumentation and sensors. And a few other responses referred to the development of very specific abilities i.e. FEA analyses, HMI development, and preparation for specific engineering competitions.

Discipline specific comments related to the different branches of engineering, and there were about a dozen comments relevant to this topic. A few ideas recurred in this dataset. Most prevalent (about 25% of the comments) was the idea that the course was not designed for engineering students, but rather for (computer) science students. Several other comments noted how specific courses were specifically designed or modified for electrical, mechanical or computer engineering students etc. Other comments made note of the fact that this introductory course was often the only computing course some students would ever take, and that there was no prerequisite for it. While it only appeared once in this section (it reappears later), one last comment seemed to sum up the dataset: this course serves different purposes for different programs and disciplines.

For use of labs/lectures almost all of the comments had a common theme i.e. labs and lectures reinforce each other with lectures introducing theory, while labs put theory into practice. Some comments stated that practice is very important and that lab exposure can increase participation and interest in lectures. A couple of comments suggested introducing problem solving early on, and making labs steadily more difficult over the term.

Several comments pertain to language specific goals and issues. About 75% of them related to C/C++, with a small minority relating to Matlab, and one to Python. The most prevalent goal was one of proficiency, competency and/or comfort with a specific language (usually C/C++). The other goal that was noted a couple of times was that of introducing students to a specific language.

Only a few comments related directly to evaluation philosophy. Two noted the importance of the quality of student work, and two stated that tests helped to ensure engagement and understanding.

### 3.4 Questions 3, 7 and 8- Languages

These questions focused on relatively simple, quantifiable responses having to do with programming languages. Collectively, they asked what language do you use in your first-year introductory computing course, what did you use 5 years ago, and do you anticipate changing to another language any time soon.

Table 2 shows the number of programs that use a given language today (keeping in mind that 11 of 57 or about 20% of courses use more than one of the languages). It also shows whether that language has been gaining (+) or losing (-) ground over the last 5 years.

It is clear that C and C++ remain dominant. However, they have both been losing ground recently to Matlab, Python, and VB/VBA/Excel. RobotC is also noteworthy for its emergence. In general, the “difficult” low-level languages seem to be the ones losing ground to less complex languages. Looking ahead 1-2 years, these
trends seem destined to continue. Six programs are considering switching to Matlab, while four consider Python and two consider VBA/Excel and C. One program each is considering switching to Ruby and C++. These potential changes are at the (potential) expense of C++ (in 7 cases), Matlab (in 3), VB (in 2) and C, C++, RAPTOR, CMAP, Python and C#, all in one case. A couple of programs noted that they regularly review the language(s) they use and will continue to do so.

**Table 2: Languages in current use.**

<table>
<thead>
<tr>
<th>Language</th>
<th>Current Frequency</th>
<th>Relative to 5 years ago</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>29</td>
<td>-6</td>
</tr>
<tr>
<td>Matlab</td>
<td>12</td>
<td>+5</td>
</tr>
<tr>
<td>VB/VBA/Excel</td>
<td>7</td>
<td>+2</td>
</tr>
<tr>
<td>Python</td>
<td>6</td>
<td>+4</td>
</tr>
<tr>
<td>C#</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>RobotC</td>
<td>2</td>
<td>+2</td>
</tr>
<tr>
<td>Java</td>
<td>2</td>
<td>-2</td>
</tr>
<tr>
<td>PLC</td>
<td>1</td>
<td>+1</td>
</tr>
<tr>
<td>RAPTOR</td>
<td>1</td>
<td>+1</td>
</tr>
<tr>
<td>CMAP</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>UNIX Script</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Fortran90</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C--</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Scratch</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Ruby</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>Assembler</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>BASIC</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

Interestingly, the distribution of languages used in Computer Science courses differed from those used in Engineering or Applied Science courses. For example, C/C++ was twice as likely to be used in ENG/APSC courses, and the languages used by only 1-2 programs were almost all used in ENGR/APSC courses. VB/VBA, Python and Matlab did not show any clear differences between CS and ENGR/APSC.

**3.5 Questions 4, 5 and 6 – Advantages and Disadvantages of Specific Languages**

Questions 4, 5 and 6 essentially looked at the perceived pros and cons of all the languages in use in first-year computing courses for Canadian engineers. Table 2 showed which languages are in current use. We will now examine the collective wisdom of the respondents on this topic by grouping comments into 6 sets: RAPTOR/PLC/C--/CMAP/Unix/Scratch, Java/C#/RobotC, Python, VB/VBA/Excel, Matlab and C/C++.

C and C++ were lumped together. They are indeed different languages, but their overlap and seamlessness led to this categorization in the survey although future work would do well to try and separate them for the purposes of this kind of study. The main advantages touted for C/C++ were generally quite technical. For example, there were many references to low-level application development such as with microcontrollers. Many respondents touted C/C++'s great language attributes i.e. you can learn all about programming with it, including things like procedural and OO programming, memory management, pointers, file handling, bit-wise operations, etc. C/C++ is flexible, powerful, universal and portable. In the words of one, perhaps representing the beliefs of many, it is “obviously the best, and must be learned.” Indeed, it is the standard for many industries and knowing it helps secure jobs. On the downside, it is hard to learn and even harder to master. Syntax is challenging and it often becomes the major pre-occupation of students, instead of problem solving. That said, several respondents (6) suggested that there were no downsides to C/C++. For many respondents, the strengths (language features) were also the weakness e.g. complex memory management, and challenges with I/O, strings and arrays. As one person said, “it’s too flexible.” “You get enough rope to hang yourself with it.” A couple pointed out that C/C++ was not designed for teaching, and that OO is maybe not best introduced in a first course. It was noted that many programs teach C first, and then maybe finish a course with C++. And some industries do not use C/C++ predominantly e.g. chemical. As with Matlab, some pointed out that C/C++ gets taught later anyways, so a case can be made to learn something else initially that is easier to learn and master for key concepts.

Matlab also received a lot of attention for and against due to the fact that it was explicitly mentioned in Q6. While Matlab is clearly quite popular, it does have a lot of detractors. On the plus side, it is highly applicable/useful to many common types of engineering problems such as in civil and mechanical. Given examples included various numerical analyses, statistics, algebra, data visualization, simulations, and more specific applications like truss analyses. Matlab is widely used in many engineering courses and in industry. It allows for a greater focus on problems, versus syntax. It can also relate very well to past courses (reinforcing them), current courses (deepening understanding of them) and future courses (enticing interest). It’s easy to learn and comprehensible, and has lots of great toolboxes. Someone thought that it was built around a similar philosophy to Python. Also, students tend to like it. On the downside, for those that thought there was any, many respondents (11) noted that it
gets used later in the curriculum so it was implied that one should learn another stronger language early on. Indeed, some explicitly felt that Matlab made for a very good second course in programming. As with Python, there was concern about using pre-defined functions blindly. Others felt that it was not a “real” language and was instead a glorified calculator. Matlab has limitations such as a lack of executables (generally) and difficulties with certain types of problems e.g. databases and GUIs. The unique syntax and UI was criticized, as was the cost of Matlab. Relative to other options, it can be more expensive for students and while toolboxes are great, they cost even more. There was also some sentiment that it is insufficient for certain disciplines (EE, SW, Mech). Others commented that learning other languages (like C) is difficult after learning Matlab, whereas learning Matlab is easy after learning C/C++. Some were concerned that it was not an OO and/or a compiled language. It’s too high-level. While some had touted its general applicability, others held the opposite view. One could perhaps agree that it is generally applicable within certain fields.

The Visual Basic family with associated applications like Access and Excel, was another fairly popular choice for first-year engineering computing courses, at least in some disciplines. There was definitely a bias for this language choice among disciplines that did not do a lot of programming in subsequent years. It’s easy to learn and use, and students can become quickly capable with it. There are lots of resource materials and the software is readily available. It relates intimately to macros in Excel and is good for HMIs and project based learning. Some felt there were no downsides to the VB family, assuming you weren’t going into serious programming. Others felt it was too high-level, hiding important details from users. And one can “cheat” between VBA and Excel macros, and code embedded in macros can be cumbersome. Also, VBA does not produce executables. Since this software is used later in many programs, some felt that it was worth learning something else initially since VB would be easy to pick up later.

Python is a relatively new language and was designed with teaching in mind [11]. It has become quite popular. According to its proponents in the survey responses, this is due to several factors. First, like C#, Python is free, and free libraries and development environments are available. The language has a small core but a rich library of functions. The language features a strong emphasis on code readability, which is good for teaching structured problem solving and pseudo code. It’s easy to learn, allows for rapid progression in competency, and focuses more on algorithms than syntax. It is an interpreted language, which may facilitate play and experimentation, but it does produce executables. It is getting very popular in industry, government and academia, belying its applicability to many domains. It’s a popular start for OO programmers (amongst CS people) but it can also facilitate initiation with procedural programming. It also balances low- and high-level programming, and thereby links well to most other languages. It provides a good foundation and jumping off point for learning other languages. It’s not messy with memory management, and it’s quite good for GUI and web development. Students like its “cool” applications. On the downside, if any, students can over-rely on canned functions, and the syntax with functions and classes can be confusing sometimes. Also, students who learn Python may find that Java would have better suited their needs. Overall, there is a clear sense that many Python users care about pedagogical issues and practicality for students while not sacrificing the positive qualities of more “serious” languages.

The second last set includes those languages currently used by two programs (Java/C# /RobotC). RobotC is used for programming Lego Mindstorm™ kits, an increasingly popular activity in engineering curriculum. Being similar to C/C++, it’s especially of interest to mechanical and mechatronics engineers as it facilitates use of motors, encoders and sensors. Java was touted for its increased use in industry and its more valid object-oriented (OO) design, versus C++. Several very good development environments are available for it and the language facilitates GUI and web design. It’s a full-fledged high-level programming language and at least one respondent felt that learning other high-level languages after Java was easy. On the downside, if any, it was suggested that Java has a fairly steep learning curve relative to Matlab, and it can take a while to develop applications with it. C# is a C variant with semantics similar to Java, and syntax similar to C/C++ but with more descriptive compiler messages. It was noted as being great for GUI development, and popular in the software development industry (and some later courses). However, other industries are not as familiar with it and few faculty know it/can teach it. A noted strength was its economic advantage. Compilers and development tools are free, and there are lots of reference materials and libraries available. It’s strong on type support and memory management, and it was suggested that learning other languages is easy after C#.

The last set (PLC/RAPTOR/C--/CMAP/Unix/Fortran/Scratch) includes all of those languages used in just one program. Commentary was generally limited, and was often provided by proponents. For example, Fortran90 was touted for its preparation for an FE exam, as well as being a good introductory high-level language. C-- is an in-house product at one institution that was touted as being a sub-set of C++ where syntax errors “were not possible” (students apparently quite like that). Flowchart RAPTOR is felt to help students visualize problem and solution structures and logical blocks, which can then be converted to functions in C++. It makes converting a big program into a well organized, structured program, much
easier. Scratch was noted for being a great language for novices, where the focus is on learning concepts, and not syntax. However, there are not many textbooks to draw upon as resources. CMAP was another language touted for its ease of learning, being a small subset of C with a simple user-interface and syntax. PLC (Ladder Logic) was introduced at the end of one programming course to offer students a bit more marketability for summer/co-op jobs, and to introduce digital logic for future courses. Unix scripting was not commented on specifically, being secondary to C++ in the course in question

3.6 Question 9 – Course Structure

The 9th question on the survey looked at how many hours of lectures, labs and/or tutorials are involved in the courses under examination, per week. In terms of lecture hours, the clear standard is three. However, there are a few courses that have two, and one program where there is one hour of programming instruction per week. Labs and tutorials were often not distinguished in terms of introductory computing courses. Combining them, about 35% of courses employ 2 hrs/week while another 35% employ 3 hrs/week. Seven use 1.5 hrs per week while 5 courses manage with just 1 hr/week. Another four courses make use of an open lab/tutorial model, in some cases combined with devoted lab time.

3.7 Question 10 – Teaching Content

The 10th question on the survey asked how programs taught the application of the course’s content. Responses were broken into 7 categories including labs/tutorials (≈23% of all responses), lectures (20%), homework and assignments (20%), tests/exams (19%), projects (6.5%), attendance (1%), and “other” (10%).

Labs/tutorials was one of two major categories of responses to Q10. These responses were tied to the lecture comments quite frequently, insofar as labs were the main way to apply what was learned in lectures. Labs would involve working through model problems, often emphasizing a variety of real-life engineering applications. Some suggested posting lab problems ahead of labs so that students could work on the design process over a period of time, and then they would demonstrate solutions during the actual labs. Many respondents referred to various ways in which evaluation took place during/out of labs. Some incorporated regular quizzes into labs (for knowledge reinforcement) while others integrated projects into the labs.

Lectures were a very popular mode of teaching, not surprisingly, although the question did say “teaching the application of the course’s content.” A popular response in this category was the idea of working through real problems in class, often with student input to the solutions. For others, lectures were noted as the starting point i.e. students would then proceed to homework and/or labs to apply what was learned in lectures. Lectures were for the presentation of theory, the modeling of problem solving approaches, and classes of solutions. Some respondents noted how they used blackboards, white boards, PowerPoint and/or in-class computer stations to teach although a couple of comments suggested in-class computer use could be negative as it was hard to follow for students. One respondent talked of using lectures for the discussion of concepts and course material, suggesting a very interactive model. A couple of comments described the use of partial notes where holes were left in course notes to be filled in during lectures.

The two basic concepts of homework and assignments were noted often, as was the idea of doing them weekly. How exactly this was done was the subject of many comments. For example, some instructors suggested exercises to complete. Others gave written procedures to students so they could go work with them on their own. A variety of assignment sizes were noted from short pass/fail exercises to longer multi-lab assignments involving documented design, coding and testing. The working of homework problems was a key mode of learning and understanding for many, although one respondent did not use homework because of the prospect of cheating. A couple of respondents referred to automated marking of small problems, and off-site assessment offering immediate performance feedback.

Tests/exams was a very popular response category, albeit with a rather limited range of distinct responses. Many respondents noted that they used tests, exams, quizzes, lab tests, and midterms. Quizzes were noted a few times as being useful to ensure that fundamental knowledge and concepts were assimilated.

Projects included both individual (400-600 lines) and team (1200-2000 lines) contexts. Small projects were noted to be complex problems while more challenging projects could last up to several weeks, with lab exercises being used to support them.

In terms of attendance, one respondent noted that lab attendance was mandatory as a method of ensuring that students were engaging with the material.

The other category was a catch-all for other ways of teaching application of content. There was quite a bit of variety in this category, including the following ideas:
- encouraging programming use in other courses
- peer discussions/cooperation
- multi-week guided exercises on specific topics
- talking about the machine and building from there
- working real/relevant problems with students
- posting model problems/solutions as well as the resources to help tackle them
- providing solid TA support
- showing modeling applications of systems
- teaching knowledge early, and application later
- don’t know (CS takes care of this)

3.8 Question 11 – Evaluation and Assessment

Question 11 complemented Question 10 by asking how coursework was evaluated, as opposed to how it was taught. Responses lent themselves to quantitative analysis, as most responses fell within a narrow range of options. Many respondents also provided assessment weights for various evaluation components, creating a richer picture of the evaluation models. Responses were grouped into 5 categories: weekly work, tests/exams, projects, attendance, and “other.”

Weekly work (homework and/or assignments) was a very common response from more than 2/3rds of all respondents. A few programs do such work in pairs, or in teams. At least 1/3rd of respondents noted that they evaluate weekly labs. The marking weights of these components varied considerably. For labs, weightings ranged from 10-30% while weekly homework/assignments varied from 5% to 65%. Team assignments were in the 25-30% range.

Not surprisingly, tests/exams were also very common. About 1/3rd of respondents utilized quizzes, worth 4-15%. A few programs employ practical, or lab, tests, with weightings varying from 10-45%. Midterms and regular tests were used by virtually all programs, with weightings ranging from 10-80% (most in the 20-25% range). Final exams were almost universal as well, but the range of weightings was unexpected, varying from 20-70% (although most were in the 40-60% range). Some respondents noted that they used online quizzes and tests.

Projects were also a fairly common mode of evaluation. About 15% of respondents reported using projects in their courses, with some doing them in pairs or teams. Weightings were in the 12-15% range.

In terms of attendance, only a couple of respondents noted that they track it and assign marks to it (ranging from 5-10%). However, a couple of programs require attendance and there are penalties for missed classes.

A few other items were also noted. iClickers™ were used by a couple of respondents. One had students contribute to a blog, and gave 8% for that work. Bonuses were used by another couple of respondents (worth 3-15%). Another noted that they gave bonus marks for class activity and use of programming in other courses.

3.9 Question 12 – Value to Graduates

The final question on the survey essentially asked about the expected value of this course to (recent) graduates, a very current topic for Canadian engineering schools. Many of the responses were reiterations of previous responses, and answers fell into four categories, two of which will be familiar. The biggest ones were skills and knowledge. Attitudes was another minor one, and logistical considerations was a small fourth category.

The largest response theme was that of skills and abilities that would be retained in the long term. There were a few major classes of abilities that appeared frequently. One was the ability to think in an algorithmic, logical, systematic, structured and methodical fashion. Another was centered on problem solving. This involved breaking problems down into parts, seeing the big picture of all the parts, thinking as an engineer, and using programming skills to help solve real-world problems. A proficiency in using computing tools to manage real problems was a strong theme overall. Other ideas that were presented included the ability to follow other people’s code, to assess the appropriateness of a computing solution, and to analyze problems and solutions. One person hoped that students would become adept at “the art of programming.”

The second biggest response theme (about 1/3rd of all responses) was knowledge. Five years after the course was over, many respondents wanted their students to know and/or remember key information, processes, concepts or ideas. While there were definitely some that wanted students to still know details about iteration, decision making structures, memory management and the like, others wanted them to retain higher-level key principles of programming such as the capabilities and limitations of computing tools and how one solves problems with them. The algorithmic approach was noted, as well as enough retained knowledge to work through problems. A couple of points suggested a desirable outcome was an awareness of how computers can be effective tools, and a recognition that an engineer must understand computing in order to avoid causing harm. Several hoped that foundational or basic knowledge would be retained so that one could relatively easily (re)learn languages as needed. Regarding specific language syntax, some hoped that there would be some basic retention while others felt there was no benefit in such retention.

The development of certain attitudes and feelings was an interesting and less traditional category of graduate outcomes. About 9% of the comments were relevant to this category. Appreciation, comfort/confidence, and perseverance were the attitudes that some respondents felt students would take away from the course in the long term. Appreciation referred to the programming process, the challenges involved, the importance of programming, and the relevance of it. Comfort and confidence referred to mastery of a language, the ability to learn another one, using computers, and solving problems with them. Perseverance presumably related to the challenges of programming.
In terms of logistical considerations, one respondent noted that the course would help them in industry, while another noted that the course may have helped some students avoid an unpleasant time in engineering i.e. if it filtered them out of the engineering program.

4. DISCUSSION

Given the limited knowledge about and sometimes faulty preconceptions of the state of introductory engineering computing courses in Canada, this relatively open-ended survey with a mixture of factual and opinion-based questions performed well. The landscape for these courses is quite diverse in terms of when the courses take place, who teaches them, what skills they impart, what languages they use, how they evaluate success, and what they are ultimately trying to achieve. Given the extent of standardization of the Canadian engineering curriculum, especially in the first year, this is somewhat surprising. This study has at least provided a rough but useful first image of this course material across Canada.

A familiar focal point of debate around these courses is “what language should we use?” The merits of C/C++, for example, have been debated in the literature, as in [4, 6]. A significant proportion of institutions seem to be answering this question on a discipline by discipline basis i.e. what is best for electrical engineers may not be what’s best for civil or mechanical engineers as they will i) have different upper-year courses, ii) different types of jobs/employers, and iii) different types of problems in their future years. However, what seems to be common across disciplines, and languages, are the key principles of solving problems with computing, the advantages and limitations in doing so, and the foundational skills and knowledge of programming.

On a personal note, the authors gained insights from this work that have led them to undertake changes in their own curriculum, and to reconsider the merits of other languages. If we are to improve, by definition, we must change. Our findings, based on the contributions of engineering educators across Canada, should give every instructor and coordinator of these courses pause. What is this course for, and who decides that answer? In whose best interest is the choice of language? How do you balance the vocational issues versus the pedagogical ones? Many questions have been raised for us by this work. Hopefully the reader will feel the same way. Devotion to all of the languages seems very strong, as demonstrated by the choices of words in many survey responses. But is that based more on faith, comfort, affinity, or merit, and what criteria are used to assess merit?

The large diversity of approaches in these courses across Canada, including the use of several different languages for instruction, is consistent with what is reported in the literature, mostly in American contexts [9]. Programs are often passionate about their choice of instructional language(s), and these choices are based on the differential weighting of several outcome criteria such as employability, problem solving abilities, learning core programming principles, and thinking algorithmically. The shift away from C/C++ to more specialized and/or pedagogical languages like VB, Matlab, and Python was an interesting result, since these languages are less complex than C/C++. This could indicate increasing attention to pedagogical issues. Indeed, Python was designed with educational purposes in mind [11]. Language choice will naturally also depend on the learning goals of a particular course and how that course fits within the broader curriculum of a program.

From the answers to various questions, it seems that these courses are primarily designed to meet traditional engineering learning outcomes of technical knowledge and application of material. However, there are other outcomes that instructors and program designers may want to address in their courses. Sheppard et al. [10] suggest that while technical skills are obviously important, non-technical professional skills and attitudes such as learning how to communicate, learning to work in teams, and learning to acquire attitudes of persistence, healthy skepticism, and optimism, are also important outcomes of an engineering education. One might argue that faculty are just not explicit about non-technical goals for student learning, but that such activities that practice such professional skills are integrated into many programming courses. Indeed, some responses to the question about the expected value of the course were more focused on non-technical outcomes. Sheppard et al, however, also argue for the importance of explicitly articulating to the learner what it is we are trying to teach and why we are teaching it, because “having no other experience, [students] take the classroom to represent the profession.” [10]

A somewhat surprising result in this study was the lack of responses that directly answered the questions regarding how application of course content is taught, and how student learning is assessed. Most comments that were classified in the “pedagogical approach” category in response to Q2 really only addressed pedagogy in superficial ways. For example, they did not describe whether teaching styles were more deductive (teacher transfers knowledge to student by explaining and demonstrating) versus inductive (teacher provides cases that students work on to help them develop an understanding of a phenomenon before a principle is introduced) nor what proportion of a typical class might be spent on learning activities. Similarly, responses to the question posed regarding evaluation of coursework created a picture of the types of assessments used (assignments, exams, etc.), but not the actual knowledge and skills being assessed, nor how they were assessed, nor
whether the types of given problems put subject matter in a real-world context.

One could speculate that the lack of detailed responses on teaching and assessment is reflective of a focus on content rather than pedagogy on the part of respondents. But it may also be due, at least in part, to flaws in the survey such as in the wording of some questions. Sample types of responses were provided for “assistance” but they may have only served to narrow the range of thinking about certain questions. If this kind of study was to be repeated, it could be more beneficial to ask specifically about learning outcomes of the course and how students are assessed on each outcome. That might give a richer description of the course rather than “10% for assignments”, “40% for midterms”, etc.

There were also several other limitations to the study. For example, the intent was to have survey respondents respond on behalf of programs. As such, administrators familiar with the courses were to be “ideal” respondents. Ultimately, in many cases, respondents were course instructors who may not have had the curriculum-level perspective that was desired, but conversely, some administrators did not have the necessary information to answer certain questions. Also, while response rates were high and completion rates on surveys were excellent, it was clear that some respondents put more effort into their answers than others. Since some institutions also had multiple versions of an introductory computing course, this resulted in multiple survey responses from some institutions and combined responses from others. These three factors led to some distortions in results since we looked at the frequency of occurrence of themes and ideas. Fortunately, as long as our conclusions are not overstated, these limitations do not diminish the key messages that can be taken away from the study.

To help address some of these issues, any future versions of this kind of study would be well advised to consistently acquire course outlines as part of data gathering, and to conduct a number of in-depth interviews. This method of qualitative research can produce deep insights without requiring all respondents to be interviewed. Interviews can provide a source of validity to a qualitative study by providing another data source for triangulation; a process of converging several sources of data [1].

5. CONCLUSIONS AND FUTURE WORK

The motivation for this study originated with a focus on programming languages in Canadian introductory engineering computing courses. Clearly, C/C++ are still dominant in the curriculum, as they have been for several years now. But they are gradually losing ground to other languages like Matlab, Python and VB/VBA, and that trend seems destined to continue. The increasing diversity seems to stem from increasingly discipline-specific needs as well as, possibly, more attention to pedagogical issues.

These computing courses are typically taught using 3 hrs of lectures a week as well as 2-3 hrs of lab/tutorial time. Evaluation of the courses typically involves weekly assignments, tests/exams/quizzes and sometimes projects. The relative weighting of these common elements varies enormously, however.

Philosophically, most course instructors/coordinators see the purpose of these courses and the expected value to graduates resting in the skills and knowledge acquired (especially regarding problem solving), as well as in the broad and specific preparation for employment in their discipline. Predominantly, they say they achieve these goals through labs, lectures and homework/assignments.

There will likely be good value in repeating this kind of study in 5-10 years, to track trends. If that is done, the survey questions and general methodology could be refined, as per the Discussion. It may also be worthwhile repeating this kind of study with other parts of the first-year curriculum, such as engineering design, mathematics, mechanics and chemistry. Alternatively, one could look at the whole core curriculum of Canadian programs on a specific topic such as computing, to get a more complete picture of the entire core training that a student experiences in a particular topical area.

Acknowledgements

The authors would like to thank all of the individuals from the engineering degree-granting programs that helped to provide input for this study. The value inherent in this work, whatever it may be, is a direct result of a strong community response to the request for feedback.

References


