University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Programming Language Translation

Scott Johnston
September 14, 2018

Abstract

With the number of different programming languages available on the market today, all offering different advantages, a programming language translator which would allow code to be converted from one language to another effectively is desirable. For this Honours project, I worked to develop a translation tool using the ANTLR framework, that would allow code in Java to be converted into Python. Despite their prevalence of these languages in modern programming, the development of this tool is relatively unique. The tool that shall be discussed will allow for code between the two languages to be translated with ease and with decent accuracy, and shows massive potential to be powerful in the future, which will also be discussed in this thesis.

Every abstract follows a similar pattern. Motivate; set aims; describe work; explain results.

"XYZ is bad. This project investigated ABC to determine if it was better. ABC used XXX and YYY to implement ZZZ. This is particularly interesting as XXX and YYY have never been used together. It was found that ABC was 20% better than XYZ, though it caused rabies in half of subjects."

Acknowledgements

I would like to thank my supervisor Derek Somerville for his guidance and support over the past year. I really valued the creativity Derek provided, as well as helping me through the unprecedented difficulties provided by the COVID-19 pandemic.

Education Use Consent

Contents

1 Introduction

1.1 Background Brief

1.1.1 ANTLR and StringTemplate

The Programming Language translator is built primarily using the ANTLR framework along with stringTemplate. ANTLR stands for "Another Tool for Language Recognition". The framework uses a user-generated grammar file to generate a parser which can then be used for reading, processing, executing, or in this case, translating structured text files.

StringTemplate is a java template engine which can be used to output source code.

These two softwares collaborated nicely due to stringTemplate powering ANTLR, and justification for how and why they were used can be found in the design and implementation section of this thesis.

1.1.2 Impact of COVID-19

The 2020-21 academic year has been gravely impacted by the ongoing global pandemic. This had an indirect yet significant affect on this honours project. Face-to-face communication has been extremely restricted throughout the year, meaning that meetings with my supervisor were limited to online calls via microsoft teams, which many would argue is harder to be engaged with.

On top of this the pandemic situation was anxiety inducing, which many recent studies have proven has lead to students struggling with motivation towards their studies. The university did however regularly update students with advice on coping throughout these troubling times.

Ultimately, this project was not majorly influenced by the global pandemic, as the software tool could be built and evaluated using my own technology. Perhaps things may have been done slightly differently, perhaps evaluating the product using colleagues own code, or using interviews to help evaluate the priorities associated with building said translator. Either way, the global pandemic should be noted in this thesis.

1.2 Problem Space

1.2.1 Java and Python

The decision to focus on a translator between Java and Python ultimately came from the fact that these are the two most prevalent programming languages in the Computing Science course at University of Glasgow. However there is strong justification for this language choice due to their relevance.

Python is a highly popular language for a number of reasons, most notably it is easy to read and write, it is an interpreted language, it is dynamically typed, and it possesses a rich base library and garbage collection feature. (https://python.land/python-tutorial/what-is-python) A paper by Fangohr detailed a comparison carried out between C, MATLAB and Python as teaching languages in engineering. This paper heavily supports that python has an easy-to-use syntax and highlights that python is here to stay in industry, due to it's intuitive nature. A paper titled "An Empirical Comparison of Seven Programming Languages" also mapped the advantages of python performance-wise using empirical data, as it performed amongst the best when put up against 6 other well known programming languages. This evidence of both strong performance and syntactical-ease are enough to justify the importance of Python, thus the decision to use it as the target language for the programming language translator was clear.

According to Oracle, 3 billion devices run on Java. This proves it's popularity, probably due to it's many advantageous factors. One major factor being the embrace of object-oriented programming (OOP) in contrast to the procedural programming approach associated with a lot of other languages. Other advantages include Java's platform-independency, it's support for a plethora of libraries being appealing to enterprise computing and Java's support for common programming key issues such as multi-threading and automatic memory management. Object-Oriented Programming is at the core of all Java programs, which is untrue for Python, despite having OOP capabilities. Java is restrictive and complex to design in comparison to the intuitive nature of python. Java is also statically typed (meaning that a variable's type must be defined), whilst Python is dynamically typed.

Evidently both Python and Java are powerful and popular for a reason, justifying their place as two of the top languages in the programming world. The differences that have also been highlighted are areas which will make the translation process interesting and justifies that there is enough of a difference between the two languages to make a translation tool worthwhile.

1.3 Motivation

Whilst it is not believed that Python will replace Java, given the benefit of some python features highlighted in section 1.2.1, there is definitely an existing argument as to why people may choose to code in python. Aside from this, Python is also closely related to newly emerging fields in computing science such as the Internet of Things (IoT).

In a 2019 article it was suggested by TIOBE that Python could overtake both Java and C to become the most popular programming language in the world. This was due to the "software engineering boom" which attracted newcomers to programming. To understand a simple program such as "hello world" a java user must understand classes, static methods and packages, whilst a C user is required to understand explicit memory management. A Python user however, simply must understand one single line. The article goes on to explain how Python is widely regarded the first programming language taught to university students, however not only due to it's simple high-level syntax. It is due to Python being the number one language in a number of different computing domains such as statistical, AI programming, scripting, system tests, web programming and scientific computing.

Figure 1.1 shows the TIOBE Programming Community index from February 2021. This index is used as indicator of the popularity of programming languages. The ratings use popular search engines to calculate the ratings, and is based on the number of skilled engineers world-wide, courses and third-party vendors. The graphic clearly indicates the rise of Python and relative fall from Java (Despite still being number 2 in the overall rankings). This evidence benefits what was stated in the 2019 article about Python's rise in the next few years.

This evidence combined makes Python a strong candidate to soon dominate the software market. This in turn could create the shift in many different industries from their current software (in a lot of cases a Java software) to one operated in Python. This could become a tedious job, and rewriting software in a new language from scratch would be both time consuming and expensive. Therefore, a mechanism that converts Java (a vastly popular programming language that is starting to wane) to Python (a language which is massively increasing in popularity) relatively accurately and automatically, could be imperative in the software development world in the near future.

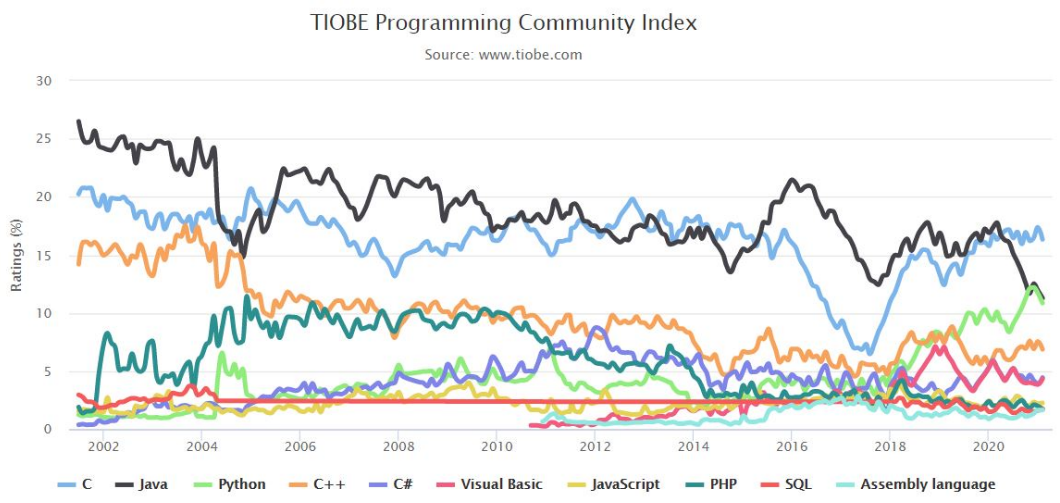


Figure 1.1: *TIOBE Programming Community Index from February 2021. Focus should be on the green line showing Python's strong growth between 2018-present.*

1.4 Research Questions

As briefly summarised in section 1.3, the overall goal was to try and create a tool which would convert a program written in Java to Python, without compromising accuracy or code quality. Aside from this overall goal, objectives were set out which were altered as the project went on to fit with what had been achieved.

Upon some background research which shall be touched upon in Section 2, Program conversion process has been placed among the top 10 challenges in the programming world (https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=895180), and the comparison of programming language conversion to "turning a sausage into a pig" from the same paper, shows that this was not going to be a straightforward task. A fully functioning translator with 100% accuracy was not achievable given the time frame and skill level, and some would even argue impossible to ever have such a powerful translator. An example of why this is impossible could be a package with certain capabilities existing in Java but not in Python.

Nevertheless, the following objectives were decided upon:

- 1) Ensure simple/common coding paradigms can be translated (explanation below)
- 2) Ensure the code is expandable for future work.
- 3) Ensure through evaluation that translator is at least as good as what is on the market
- 4) Keep translations as simple as possible
- 5)
- 6)

For the sake of clarity here is a more thorough description of each objective.

For objective **#1**, consideration was taken into what is the most useful and common coding patterns used in Java that should be implemented into the translator. Given my own coding experience accompanied by W3Schools.com (which offers tutorials for beginners), these were selected as follows:

Conditional statements in the form of if/else/ else if as well as logical conditions using mathematical symbols (e.g 'a>'b, 'a<b', 'a==b').

Switch statements, which are a form of conditional statement to select a block of code that executes if a case is met (switch/case/default)

While loops, and do/while loops where a block of code will be executed as long as a specified condition is reached.

For loops, where the number of times code will be looped is specified, rather than a condition waiting to be met. Also for-each loops can be used when looping through elements in a list-type structure such as an array.

Also included in this objective, would be converting the OOP structure required in Java into Python, Which meant creating classes and methods in the translation. This is not only good coding practice, but meant the code could be easily compared due to the shape remaining similar.

The ANTLR grammar file used for translation was also imperative in finding what areas of the Java programming language to focus on. This file will be discussed later in the design phase of this thesis.

Objective #2 was actually decided upon during my background research when looking for similar software tools that had been built. Whilst there was some helpful research, a lot of the tools were difficult to use any source code from, due to their unclear structure. This meant that even if my translator was just to be a "proof of concept", the way in which it should be built is so that it is extendable, such that the 'future work' set out in this thesis could possibly be built using the framework I have managed to design so far.

For Objective #3, there was a few translator tools on the market that did a similar thing to what this was intended to do, and decided that for this to be a success, the evaluator should perform at least as well as what is currently on the market. This meant testing my tool with the java2python test folder that can be found at https://github.com/natural/java2python/tree/master/test and I intended for my evaluation to be a success if my tool converted all these tests with accuracy and perhaps could improve upon this tool through a comparison. This objective will be discussed further in the evaluation phase of this thesis.

For Objective #4, I did not want to make translation overly complicated. A philosophy adopted with this tool from early on was that if the translation can be simplified, it has not been fully successful. This idea came from my evaluation of other translation systems, as I felt some conversions, whilst able to compile and run successfully, the translation looked over complicated and thus, may be too indecipherable and very difficult to maintain and add features to.

1.5 Contributions

1.6 Structure of Paper

1.6.1 Methodology Summary

To create this tool I first of all researched the current market of programming language translators. Inspiration from these tools was taken, however they were also criticized for what they lacked, and what could have been improved upon. This helped lead to the design process, as the criteria was able to be laid down for exactly what was required of the translator. Once the functional requirements were laid out, research was undertaken to identify what software was available on the market which would offer this functionality. Once ANTLR and StringTemplate were identified as reaching these requirements, GitHub was used to find any open source code which could be used for my project. Once this code was found, and I was familiar with how the software worked, I used this existing code to try and extend it to a translator. Evaluation was ongoing as I created the tool to ensure the objectives were stuck to closely.

1.6.2 What I have built

I have built a programming language translator tool using Java, and ANTLR and StringTemplate frameworks, which is able to translate from Java to Python. The tool is able to translate the majority of Java's in-built functionality to Python in the most minimalist form, so as not to confuse things and means the code is maintainable in the future. The way in which this tool has been built uses a Java Grammar from ANTLR's GitHub repository which I have extended slightly to add some functionality, and a StringTemplate file which was influenced by the StringTemplate GitHub however was primarily built from scratch. This technique of building was chosen due to it's extendability in the future, as this thesis will demonstrate how easily the tool could be expanded in future.

1.6.3 Thesis Outline

The rest of this paper is structured as follows: Chapter 2 analyses and critiques the current research in this field, and the current products on the market. Chapter 3 is a general summary of the method used to decide upon the criteria of the tool and it's evaluation. Chapter 4 is the analysis and the specification of what is required in a programming language translator. Chapter 5 details the design process of the translator before being built. Chapter 6 details the implementation of the translator and any setbacks faced in the building process. Chapter 7 will detail how the tool was evaluated and how well it performed in the evaluation process. Chapter 8 will then be the conclusion of the findings from this paper, reflection of the process, and any future work that could take place in this field.

TODO: Remove the guidance notes from your dissertation before submitting!

Why should the reader care about what are you doing and what are you actually doing?

1.7 Guidance

Motivate first, then state the general problem clearly.

1.8 Writing guidance

1.8.1 Who is the reader?

This is the key question for any writing. Your reader:

- is a trained computer scientist: *don't explain basics.*
- has limited time: *keep on topic.*
- has no idea why anyone would want to do this: *motivate clearly*
- might not know *anything* about your project in particular: *explain your project.*

- but might know precise details and check them: *be precise and strive for accuracy.*
- doesn't know or care about you: *personal discussions are irrelevant.*

Remember, you will be marked by your supervisor and one or more members of staff. You might also have your project read by a prize-awarding committee or possibly a future employer. Bear that in mind.

1.8.2 References and style guides

There are many style guides on good English writing. You don't need to read these, but they will improve how you write.

- *How to write a great research paper* Peyton Jones (2017) (**recommended**, even though you aren't writing a research paper)
- *How to Write with Style* Vonnegut (1980). Short and easy to read. Available online.
- *Style: The Basics of Clarity and Grace* Williams and Bizup (2009) A very popular modern English style guide.
- *Politics and the English Language* Orwell (1968) A famous essay on effective, clear writing in English.
- *The Elements of Style* Strunk and Whyte (2007) Outdated, and American, but a classic.
- *The Sense of Style* Pinker (2015) Excellent, though quite in-depth.

Citation styles

- If you are referring to a reference as a noun, then cite it as: "Orwell (1968) discusses the role of language in political thought."
- If you are referring implicitly to references, use: "There are many good books on writing (Orwell 1968; Williams and Bizup 2009; Pinker 2015)."

There is a complete guide on good citation practice by Peter Coxhead available here: `http://www.cs.bham.ac.uk/~pxc/refs/index.html`. If you are unsure about how to cite online sources, please see UNSW (2019). [1]

1.8.3 Plagiarism warning

WARNING

If you include material from other sources without full and correct attribution, you are commiting plagiarism. The penalties for plagiarism are severe. Quote any included text and cite it correctly. Cite all images, figures, etc. clearly in the caption of the figure.

[1] Specifying an online resource like `https://developer.android.com/studio` in a footnote sometimes makes more sense than including it as a formal reference.

1.8.4 Quoting text

If you are quoting a long passage, use a `quote` environment:

> If you scribble your thoughts any which way, your readers will surely feel that you care nothing about them. They will mark you down as an egomaniac or a chowderhead -or, worse, they will stop reading you. The most damning revelation you can make about yourself is that you do not know what is interesting and what is not.

(Vonnegut 1980)

If you are quoting inline, like Simon Peyton-Jones' following remark, use quotation marks "Conveying the intuition is primary, not secondary" (Peyton Jones 2017).

2 Background

What did other people do, and how is it relevant to what you want to do?

2.1 Guidance

- Don’t give a laundry list of references.
- Tie everything you say to your problem.
- Present an argument.
- Think critically; weigh up the contribution of the background and put it in context.
- **Don’t write a tutorial**; provide background and cite references for further information.

3 Analysis/Requirements

What is the problem that you want to solve, and how did you arrive at it?

3.1 Guidance

Make it clear how you derived the constrained form of your problem via a clear and logical process.

The analysis chapter explains the process by which you arrive at a concrete design. In software engineering projects, this will include a statement of the requirement capture process and the derived requirements.

In research projects, it will involve developing a design drawing on the work established in the background, and stating how the space of possible projects was sensibly narrowed down to what you have done.

4 Design

How is this problem to be approached, without reference to specific implementation details?

4.1 Guidance

Design should cover the abstract design in such a way that someone else might be able to do what you did, but with a different language or library or tool. This might include overall system architecture diagrams, user interface designs (wireframes/personas/etc.), protocol specifications, algorithms, data set design choices, among others. Specific languages, technical choices, libraries and such like should not usually appear in the design. These are implementation details.

5 Implementation

What did you do to implement this idea, and what technical achievements did you make?

5.1 Guidance

You can't talk about everything. Cover the high level first, then cover important, relevant or impressive details.

5.2 General guidance for technical writing

These points apply to the whole dissertation, not just this chapter.

5.2.1 Figures

Always refer to figures included, like Figure 5.1, in the body of the text. Include full, explanatory captions and make sure the figures look good on the page. You may include multiple figures in one float, as in Figure 5.2, using `subcaption`, which is enabled in the template.

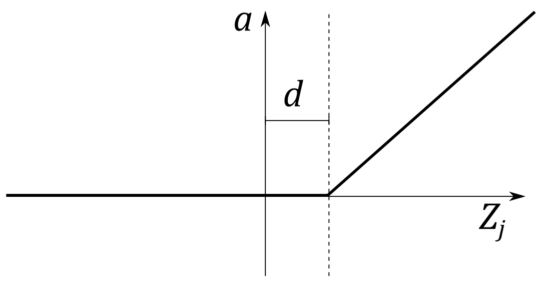
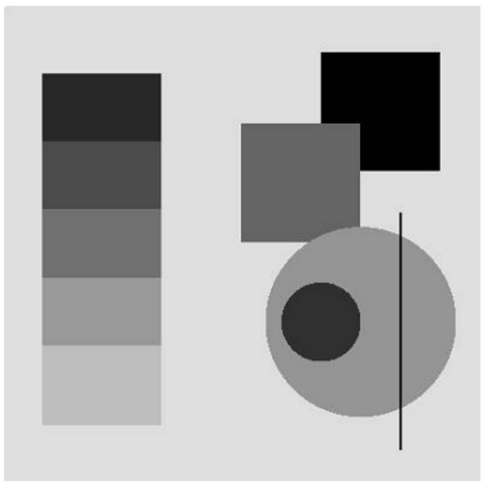
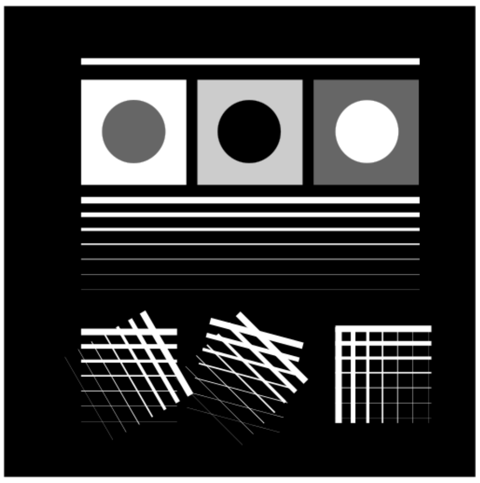


Figure 5.1: *In figure captions, explain what the reader is looking at: "A schematic of the rectifying linear unit, where a is the output amplitude, d is a configurable dead-zone, and Z_j is the input signal", as well as why the reader is looking at this: "It is notable that there is no activation* at all *below 0, which explains our initial results."* ***Use vector image formats (.pdf) where possible****. Size figures appropriately, and do not make them over-large or too small to read.*



(a) *Synthetic image, black on white.*



(b) *Synthetic image, white on black.*

Figure 5.2: *Synthetic test images for edge detection algorithms. (a) shows various gray levels that require an adaptive algorithm. (b) shows more challenging edge detection tests that have crossing lines. Fusing these into full segments typically requires algorithms like the Hough transform. This is an example of using subfigures, with* `subref`*s in the caption.*

5.2.2 Equations

Equations should be typeset correctly and precisely. Make sure you get parenthesis sizing correct, and punctuate equations correctly (the comma is important and goes *inside* the equation block). Explain any symbols used clearly if not defined earlier.

For example, we might define:

$$\hat{f}(\xi) = \frac{1}{2}\left[\int_{-\infty}^{\infty} f(x)e^{2\pi i x \xi}\right], \tag{5.1}$$

where $\hat{f}(\xi)$ is the Fourier transform of the time domain signal $f(x)$.

5.2.3 Algorithms

Algorithms can be set using `algorithm2e`, as in Algorithm 1.

```
Data: $f_X(x)$, a probability density function returing the density at $x$.
$\sigma$ a standard deviation specifying the spread of the proposal distribution.
$x_0$, an initial starting condition.
Result: $s = [x_1, x_2, \ldots, x_n]$, $n$ samples approximately drawn from a distribution with PDF $f_X(x)$.
begin
    $s \longleftarrow []$
    $p \longleftarrow f_X(x)$
    $i \longleftarrow 0$
    while $i < n$ do
        $x' \longleftarrow \mathcal{N}(x, \sigma^2)$
        $p' \longleftarrow f_X(x')$
        $a \longleftarrow \frac{p'}{p}$
        $r \longleftarrow U(0,1)$
        if $r < a$ then
            $x \longleftarrow x'$
            $p \longleftarrow f_X(x)$
            $i \longleftarrow i + 1$
            append $x$ to $s$
        end
    end
end
```

Algorithm 1: The Metropolis-Hastings MCMC algorithm for drawing samples from arbitrary probability distributions, specialised for normal proposal distributions $q(x'|x) = \mathcal{N}(x, \sigma^2)$. The symmetry of the normal distribution means the acceptance rule takes the simplified form.

5.2.4 Tables

If you need to include tables, like Table 5.1, use a tool like https://www.tablesgenerator.com/ to generate the table as it is extremely tedious otherwise.

5.2.5 Code

Avoid putting large blocks of code in the report (more than a page in one block, for example). Use syntax highlighting if possible, as in Listing 5.1.

Table 5.1: *The standard table of operators in Python, along with their functional equivalents from the* `operator` *package. Note that table captions go above the table, not below. Do not add additional rules/lines to tables.*

Operation	**Syntax**	**Function**
Addition	`a + b`	`add(a, b)`
Concatenation	`seq1 + seq2`	`concat(seq1, seq2)`
Containment Test	`obj in seq`	`contains(seq, obj)`
Division	`a / b`	`div(a, b)`
Division	`a / b`	`truediv(a, b)`
Division	`a // b`	`floordiv(a, b)`
Bitwise And	`a & b`	`and_(a, b)`
Bitwise Exclusive Or	`a ^b`	`xor(a, b)`
Bitwise Inversion	`~a`	`invert(a)`
Bitwise Or	`a | b`	`or_(a, b)`
Exponentiation	`a ** b`	`pow(a, b)`
Identity	`a is b`	`is_(a, b)`
Identity	`a is not b`	`is_not(a, b)`
Indexed Assignment	`obj[k] = v`	`setitem(obj, k, v)`
Indexed Deletion	`del obj[k]`	`delitem(obj, k)`
Indexing	`obj[k]`	`getitem(obj, k)`
Left Shift	`a <<b`	`lshift(a, b)`
Modulo	`a % b`	`mod(a, b)`
Multiplication	`a * b`	`mul(a, b)`
Negation (Arithmetic)	`- a`	`neg(a)`
Negation (Logical)	`not a`	`not_(a)`
Positive	`+ a`	`pos(a)`
Right Shift	`a >>b`	`rshift(a, b)`
Sequence Repetition	`seq * i`	`repeat(seq, i)`
Slice Assignment	`seq[i:j] = values`	`setitem(seq, slice(i, j), values)`
Slice Deletion	`del seq[i:j]`	`delitem(seq, slice(i, j))`
Slicing	`seq[i:j]`	`getitem(seq, slice(i, j))`
String Formatting	`s % obj`	`mod(s, obj)`
Subtraction	`a - b`	`sub(a, b)`
Truth Test	`obj`	`truth(obj)`
Ordering	`a <b`	`lt(a, b)`
Ordering	`a <= b`	`le(a, b)`

```python
def create_callahan_table(rule="b3s23"):
    """Generate the lookup table for the cells."""
    s_table = np.zeros((16, 16, 16, 16), dtype=np.uint8)
    birth, survive = parse_rule(rule)

    # generate all 16 bit strings
    for iv in range(65536):
        bv = [(iv >> z) & 1 for z in range(16)]
        a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p = bv

        # compute next state of the inner 2x2
        nw = apply_rule(f, a, b, c, e, g, i, j, k)
        ne = apply_rule(g, b, c, d, f, h, j, k, l)
        sw = apply_rule(j, e, f, g, i, k, m, n, o)
        se = apply_rule(k, f, g, h, j, l, n, o, p)

        # compute the index of this 4x4
        nw_code = a | (b << 1) | (e << 2) | (f << 3)
        ne_code = c | (d << 1) | (g << 2) | (h << 3)
        sw_code = i | (j << 1) | (m << 2) | (n << 3)
        se_code = k | (l << 1) | (o << 2) | (p << 3)

        # compute the state for the 2x2
        next_code = nw | (ne << 1) | (sw << 2) | (se << 3)

        # get the 4x4 index, and write into the table
        s_table[nw_code, ne_code, sw_code, se_code] = next_code

    return s_table
```

Listing 5.1: *The algorithm for packing the* 3×3 *outer-totalistic binary CA successor rule into a* $16 \times 16 \times 16 \times 16$ *4 bit lookup table, running an equivalent, notionally 16-state* 2×2 *CA.*

6 | Evaluation

How good is your solution? How well did you solve the general problem, and what evidence do you have to support that?

6.1 Guidance

- Ask specific questions that address the general problem.
- Answer them with precise evidence (graphs, numbers, statistical analysis, qualitative analysis).
- Be fair and be scientific.
- The key thing is to show that you know how to evaluate your work, not that your work is the most amazing product ever.

6.2 Evidence

Make sure you present your evidence well. Use appropriate visualisations, reporting techniques and statistical analysis, as appropriate. The point is not to dump all the data you have but to present an argument well supported by evidence gathered.

If you use numerical evidence, specify reasonable numbers of significant digits; don't state "18.41141% of users were successful" if you only had 20 users. If you average *anything*, present both a measure of central tendency (e.g. mean, median) *and* a measure of spread (e.g. standard deviation, min/max, interquartile range).

You can use `siunitx` to define units, space numbers neatly, and set the precision for the whole LaTeX document.

For example, these numbers will appear with two decimal places: 3.14, 2.72, and this one will appear with reasonable spacing 1 000 000.

If you use statistical procedures, make sure you understand the process you are using, and that you check the required assumptions hold in your case.

If you visualise, follow the basic rules, as illustrated in Figure 6.1:

- Label everything correctly (axis, title, units).
- Caption thoroughly.
- Reference in text.
- **Include appropriate display of uncertainty (e.g. error bars, Box plot)**

- Minimize clutter.

See the file `guide_to_visualising.pdf` for further information and guidance.

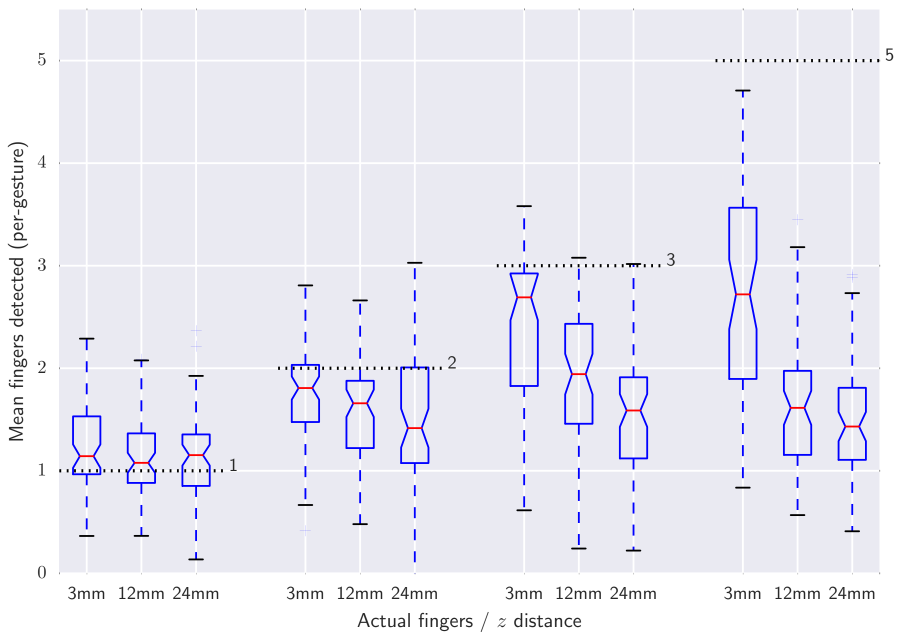


Figure 6.1: *Average number of fingers detected by the touch sensor at different heights above the surface, averaged over all gestures. Dashed lines indicate the true number of fingers present. The Box plots include bootstrapped uncertainty notches for the median. It is clear that the device is biased toward undercounting fingers, particularly at higher z distances.*

7 Conclusion

Summarise the whole project for a lazy reader who didn't read the rest (e.g. a prize-awarding committee). This chapter should be short in most dissertations; maybe one to three pages.

7.1 Guidance

- Summarise briefly and fairly.
- You should be addressing the general problem you introduced in the Introduction.
- Include summary of concrete results ("the new compiler ran 2x faster")
- Indicate what future work could be done, but remember: **you won't get credit for things you haven't done**.

7.2 Summary

Summarise what you did; answer the general questions you asked in the introduction. What did you achieve? Briefly describe what was built and summarise the evaluation results.

7.3 Reflection

Discuss what went well and what didn't and how you would do things differently if you did this project again.

7.4 Future work

Discuss what you would do if you could take this further – where would the interesting directions to go next be? (e.g. you got another year to work on it, or you started a company to work on this, or you pursued a PhD on this topic)

A Appendices

Use separate appendix chapters for groups of ancillary material that support your dissertation. Typical inclusions in the appendices are:

- Copies of ethics approvals (you must include these if you needed to get them)
- Copies of questionnaires etc. used to gather data from subjects. Don't include voluminous data logs; instead submit these electronically alongside your source code.
- Extensive tables or figures that are too bulky to fit in the main body of the report, particularly ones that are repetitive and summarised in the body.
- Outline of the source code (e.g. directory structure), or other architecture documentation like class diagrams.
- User manuals, and any guides to starting/running the software. Your equivalent of `readme.md` should be included.

Don't include your source code in the appendices. It will be submitted separately.

Bibliography

Orwell, G. (1968), Politics and the English language, *in* 'The collected essays, journalism and letters of George Orwell', Harcourt, Brace, Javanovich, pp. 127–140.

Peyton Jones, S. (2017), How to write a great research paper, *in* '2017 Imperial College Computing Student Workshop, ICCSW 2017, September 26-27, 2017, London, UK', pp. 1:1–1:1.

Pinker, S. (2015), *The sense of style: The thinking person's guide to writing in the 21st century*, Penguin Books.

Strunk, W. and Whyte, E. (2007), *The Elements of style*, Penguin.

UNSW (2019), 'How do i cite online sources?', `https://student.unsw.edu.au/how-do-i-cite-electronic-sources`. Last accessed: 2019-02-27.

Vonnegut, K. (1980), *How to write with style*, International Paper Company.

Williams, J. M. and Bizup, J. (2009), *Style: the basics of clarity and grace*, Pearson Longman.