Expressive Complexity of Computer Algebra Systems [Finalest Draft]

Robert Dougherty-Bliss
Oglethorpe University
Department of Mathematics and Computer Science

Contents

Contents i

Preface iii

1 Introduction 1

2 Expressive Complexity 5
- 2.1 The Halstead Metrics 6
- 2.2 Ambiguity 8

3 C-finite and Pisot Sequences 11
- 3.1 Definitions and preliminaries 11
- 3.2 Computational problems 13
- 3.3 Expressive complexity analysis 14
- 3.4 Qualitative discussion 14
- 3.5 Source code listings 21

4 Propositional Logic and Boolean Satisfiability 27
- 4.1 Definitions and preliminaries 27
- 4.2 Computational problems 29
- 4.3 Expressive complexity analysis 31
- 4.4 Qualitative discussion 31
- 4.5 Source code listings 38

5 Indefinite Summation 43
- 5.1 Definitions and preliminaries 44
- 5.2 Computational problems 46
- 5.3 Expressive Complexity analysis 47
- 5.4 Qualitative discussion 48
- 5.5 Source code listings 53

6 Conclusion 57

Bibliography 59

Preface

This thesis is, *de jure*, about the ergonomics of various computer algebra systems. It is, *de facto*, about interesting applications of computers to mathematics.

These topics grew out of a confluence of factors. First, I have always been a better programmer than mathematician. The opportunity to augment my mathematical powers with my programming powers was too good to ignore. Second, I discovered the wonderful book $A = B$ in the library at Oglethorpe University. This book is about proving, entirely automatically, summation identities. It introduced me to the philosophy that mathematics is about making complicated things routine, a philosophy that the computer serves as a hardy tool. I especially liked this book because it showed that intimidating and discouragingly-difficult identities were so "routine" that a computer could prove them—no genius required. Third, I have strong opinions about software. I believe that most popular computer algebra systems are complete slop. For far too long I have been forced to use Maple or Mathematica, and this thesis is partly a throwing off of my chains, such as they are.

In choosing the problems to study in this thesis, I was heavily influenced by the philosophy of $A = B$. Each problem is a case study in the "routinization" of a question in mathematics. We begin with a hard question, then show that it is actually not too hard for computers. We then show that one computer algebra system makes the problem easier than another.

This thesis is, on the one hand, submitted as a piece of original research to the Honors program at Oglethorpe University. On the other hand, it is an interesting tour through three applications of computer algebra to "serious" mathematics, which students so rarely see. Thus, the thesis could either be read as a means to evaluate my potential academic output, or as a way to learn at least three new things about computer algebra systems. The latter may be more fun for everyone involved.

The thesis is organized as follows. Chapter 1 is a lay-introduction to what I mean by "ergonomics." Chapter 2 states the Halstead metrics. Chapter 3 introduces C-finite and Pisot sequences then gives some recent results by Sloane and Zeilberger. Chapter 4 brings computer algebra to bear on propositional logic. Chapter 5 discusses Gosper's algorithm for hypergeometric indefinite summation. These three chapters can be tackled in any order. Chapter 6 gives some brief concluding remarks and points towards further research.

Acknowledgements

I had generous amounts of support while preparing this thesis. This is my first time running into the difficulty of writing acknowledgements, but I suppose that this is a good problem to have.

My advisor and committee chair, Dr. Brian Patterson, was immensely helpful and responsive throughout the entire process. The thesis is surely enhanced by his early guidance and detailed

feedback. Dr. Bradford Smith, the previous Honors program director, was instrumental in cracking his whip while writing the prospectus. Dr. Seema Shrikhande, the current Honors program director, did a great job overseeing the defense and other late-stage tasks. The remainder of my committee, Drs. Lynn Gieger, Philip Tiu, and John Merkel, helped to shape my early research questions into something manageable and interesting. Dr. Doron Zeilberger gave me his blessing and encouragement while translating his Maple code, which was useful motivation.

This thesis simply could not have been prepared without the excellent tools I had at my disposal. Typesetting was done with Leslie Lamport's LaTeX system [Lam94] and various standard packages for it. The actual writing was done with Bram Moolenaar's text editor Vim.

Chapter 1

Introduction

> [Ergonomics] is that branch of science which seeks to turn human–machine antagonism into human–machine synergy.
>
> Peter Hancock, *Essays on the Future of Human-Machine Systems* [Han97a]

Computer scientists often care about *computational complexity*. How long will an algorithm take to run? How will it perform on average? In essence, *how fast can we go?* Though this is an important consideration, it omits a crucial implementation detail: the human factor. Is the algorithm painful or tedious to write? Is it overly complicated for the average programmer? That is, *is it ergonomic to use?* It is easy to demand that performance trumps all, but this is a costly mistake. Tools and environments must account for human factors; programming languages are no exception.

We use the term *expressive complexity* to refer to the aggregate of human factors in a programming language or algorithm. This term is, in one sense, more literally accurate than computational complexity. We really do mean to measure how *complicated* programs are for humans to understand. This term is likely not well-understood by most mathematicians, so we will make a brief detour to explain it by way of analogy.

Ergonomics is the study of human relationship with work. It seeks to make necessary burdens easier and more enjoyable. This goal is based upon the observation that "long faces are not always efficient, nor are smiling ones idle" [Han97b]. Ergonomics considers physical and psychological *human factors*, such as comfort and stress, respectively. It may, for example, suggest appropriate levels and types of lighting for the workplace to improve morale, or recommend chairs with a certain amount of back support to avoid long-term injury. This is all to improve the human condition and workplace efficiency.

Though the term ergonomics may conjure images of comfortable chairs and standing desks, it is not constrained to physical factors. Beginning in the 1970s, researchers in ergonomics began to study *mental workload*. Roughly, this is how mentally taxing certain tasks are. If a worker's mental workload is too high, they are likely to make mistakes or "burnout" faster than a relaxed employee. The following is a more technical definition:

> [Mental workload is] the relation between the function relating the mental resources demanded by a task and those resources available to be supplied by the human operator.

[PSW08]

Mental workload can be a crucial design factor. Two studies in aviation accidents found that as much as 18% of pilot errors were due to confusing instrument design that made it difficult for pilots to understand their readouts [Sal12, pg. 244].

This is all to say that the tools we use and the tasks we complete should be easy to understand. We should not be satisfied that clear design happens by accident; we should deliberately strive for it. The consequences of ignoring this can range from decreased worker productivity and longevity, to grave, avoidable mistakes.

Consideration of mental workload is especially important in programming language design. Programming, more than other activities, is centered around thought. Its primary tool, the programming language, is a means to express computational thought in a way that the computer can understand. The task of the programmer is to mentally construct a solution to a problem, then translate this mental solution into a concrete programming language. Solving problems is already mentally taxing enough; we should not add to the mental workload by creating a confusing translation process.

As an example, consider a student beginning to learn programming. They must learn the mantra that computers are "stupid"; that they will only do exactly as they are told and no more. They must learn to translate their mental solutions into mechanical steps. That is, the student must learn to think like a machine. The successful student will overcome this initial hurdle, but the mental workload of translation is always present. The mental workload that remains is largely—I argue—a function of the programming language a programmer uses.

In the context of software, I call the contributions to mental workload *expressive complexity*, in opposition to traditional *computational complexity*. Expressive complexity, then, measures how complicated algorithms are to implement, how difficult a language is to use, and how much mental strain is imposed on a programmer by these objects.

Examples

Consider the following task: Sum the integers from 1 to 100. Here are three solutions:

Haskell

```
sum [1..100]
```

Python

```
sum(range(101))
```

C

```
int sum = 0;
for(int k = 0; k != 100; k++) {
    sum += k;
}
```

All three solutions have the same computational complexity. However, they clearly differ in their *expressive complexity*. The Haskell and Python solutions are almost exact 1-1 translations of the obvious solution: Just sum the integers from 1 to 100. In particular, the programmer does not have to think about *explicit iteration*, which is how computers think. Instead, they can essentially write down their mental solution.

In comparison, the C solution is very mechanical. It shows how the *computer* thinks of the process of summing integers, rather than how the *programmer* thinks of it. A separate `sum` variable must be accounted for, because computers must do such things. An explicit loop with a counter must be written, because computers must compute in terms of loops and counters. The programmer must consider these things, not because humans must, but because machines must. In short, C makes the programmer think like a machine rather than like a human.

This example shows that expressive complexity is not just a feature of a particular algorithm, but rather a feature of particular *languages*. We may thus compare languages by their expressive complexity and decide which best suit our purpose. This is our real goal of this thesis. To perform these comparisons we will need a suitable framework in which to measure expressive complexity. Fortunately, thanks to Maurice Halstead, such a thing already exists. We begin in the next chapter by describing this framework, known as the *Halstead* metrics.

Chapter 2

Expressive Complexity

> This finding, in turn, lends even greater weight to the previously mentioned observation that the human brain must follow, or be governed by, an interesting set of rules of which it has (or we have) heretofore been unaware.
>
> Maurice Halstead, *Elements of Software Science* [Hal77].

In this chapter we will discuss the *expressive complexity* of a program or algorithm. In particular, we will describe the Halstead metrics and their definitions pertaining to *effort* and *language level*. Since these concepts are likely foreign to most readers, let us take a moment to describe them.

First, expressive complexity is not *computational* complexity. Computational complexity is roughly how many steps an algorithm takes for an input of a given size. For example, the insertion sort algorithm can sort a list of n elements in $O(n^2)$ time. The mergesort algorithm can do the same in $O(n \log n)$ time. We are not, however, interested in how *long* an algorithm takes. Rather, we are interested in how *complicated* its expressions of computations are. For instance, while mergesort has better computational complexity than insertion sort, its more intricate splitting and recursive nature are more "complicated" to express.

As a more concrete example, consider the task of summing two numbers. In any reasonable language there are countably many ways to accomplish this task. In fact, to any degree desired, there is a solution that contributes arbitrarily many useless steps. Figure 2.1 shows two solutions, one simple and one needlessly complicated. The complicated one is longer, harder to read, and generally confusing. Yet, both solutions have the same *computational complexity*. Asymptotically, they may as well be the same thing. However, the simpler one has lower *expressive complexity*.

The fundamental observation about expressive complexity is that humans tend to minimize it when possible. Of the two programs in Figure 2.1, which would a programmer be likelier to write? Of course every experienced programmer would choose the simpler option, because it contributes no obviously useless steps. Indeed, programmers tend possess an innate desire for simplicity more generally. If obviously useless steps are observed, then they will be removed.

The first set of expressive complexity measurements were given by Maurice Halstead in [Hal77]. These measurements are now known as the *Halstead metrics*. Motivated by this fundamental observation, Halstead believed that "programmers remove useless steps" was a kind of natural law, and that this natural law would induce certain empirical relationships between his measurements

that could then be used to evaluate systems and reason about their expressive complexity. That is, Halstead argues that by merely measuring observable properties of code written as text, we can reason about how complicated a program is, how difficult it is to understand, and also derive various relationships between our measurements, just like a natural science. That is, programming, when linked with human thought, produces a system in which we can make systematic discoveries. Halstead refers to this system as the *Software Science*.

This conclusion sounds surprising. It states that the "unconstrained" system of human thought is somehow limited in what it can do by certain empirical relationships. "Intuition, however," as Halstead writes, "is far from trustworthy, as demonstrated when that ancient scientist dropped the wood and lead balls from the tower of Pisa." Halstead makes a convincing case that such relations exist. He provides logical derivations along with supporting data compiled from large corpora of programs. Following this argument, we shall base our study of expressive complexity on the Halstead metrics[1].

In the following sections we will define the Halstead metrics, present some empirical evidence in their favor, and discuss some technical ambiguity.

```
func f(x, y):
    return x + y
```

(a) Simple.

```
func f(x, y):
    sum := x + y
    sum2 := x + y + 100
    sum3 := sum2 - 100
    check := (sum3 = x + y)
    if check:
        return sum3
```

(b) Complicated.

Figure 2.1: Simple and complicated programs that accomplish the same task. The left snippet has lower *expressive complexity*.

2.1 The Halstead Metrics

In this section we state a slightly modified subset of the Halstead metrics. The interested reader is highly encouraged to seek out a copy of Halstead's original publication [Hal77]. Some concepts in the following definitions are taken as primitive, such as "statement," "operator," and so on. We trust that the reader is well-acquainted with these.

Definition 1 (Halstead metrics)**.** A *program* is a sequence of statements consisting of operands and operators applied to operands. Let η_1 and η_2 be the number of *distinct* operators and operands, respectively, and let N_1 and N_2 be the *total* number of operators and operands, respectively. The

[1] The Halstead metrics are not the sole way to understand expressive complexity. For instance, Thomas MaCabe introduced the *essential* and *cyclomatic* complexity measures in [Mac83]. Further, the Halstead metrics have some real flaws. See, for example, the analysis in [AA05]. Despite these objections, we focus on the Halstead metrics because they (attempt to) explicitly relate mental effort with the underlying language.

Halstead metrics are as follows.

$$\begin{aligned}
\text{Program vocabulary: } \eta &= \eta_1 + \eta_2 \\
\text{Program length: } N &= N_1 + N_2 \\
\text{Volume: } V &= N \log_2 \eta \\
\text{Program Level } L &= \frac{2}{\eta_1} \frac{\eta_2}{N_2} \\
\text{Difficulty: } D &= \frac{1}{L} \\
\text{Effort: } E &= DV
\end{aligned}$$

We shall now give intuitive interpretations for select metrics.

Volume The quantity $\log_2 \eta$ is the minimum number of bits needed to represent every operator and operand in a program. Since there are N such elements used, the quantity $V = N \log_2 \eta$ is the minimum number of bits needed to represent the entire program. (Rather, $\lceil V \rceil$ is the minimum number. This approximation is off by at most 1, and is quite common in asymptotic analysis.)

Here is an amusing interpretation of V: Consider the task of writing a program of length N with vocabulary η. When writing each element, a programmer must search through the η possible elements in their mind to discover which to use. If we suppose that there is some ordering to these elements, then we could use a comparison-based sort to find the correct element to use. The best comparison-based sort on a list of size η has runtime asymptotic to $\log_2 \eta$ [Cor+09, ch. 8]. Thus this search takes roughly $\log_2 \eta$ "mental comparisons" when choosing an operator or operand. Since there are N choices to make total, the quantity $V = N \log_2 \eta$ tells us roughly how many mental comparisons were used to write a program. This interpretation is dubious, to say the least.

Program Level and Difficulty These metrics are intended to measure how "high level" a program is, which is closely correlated to how difficult it is to understand. Halstead vigorously handwaves during this derivation, but still produces a lengthy argument. We will gloss over many of the details here.

Halstead argues that the highest level language possible would *have* an operator for every task, while the lowest level would need to *implement* an operator for every task. Therefore, he posits that the "level" of an implementation is inversely proportional to the number of unique operators in the language, or

$$L \sim \frac{\eta_1^*}{\eta_1},$$

where η_1^* is the minimum number of unique operators needed in a theoretical "best" implementation. In fact, we may take $\eta_1^* = 2$, since the procedure to be implemented would already exist in the best implementation.

Next, Halstead argues that the more an implementation repeats its operands, the lower level it is. From this, he derives

$$L \sim \frac{\eta_2}{N_2}.$$

To combine these measurements, we define

$$L = \frac{2}{\eta_1} \frac{\eta_2}{N_2}.$$

It is clear that both factors are in $[0, 1]$, so multiplying them gives a "joint" measurement of both in that interval.

Difficulty is merely the inverse of program level, $D = 1/L$. The higher level a program is, Halstead argues, the easier it is to write or understand it. This is a tenuous point—consider the difficulty of beginners in learning an unfamiliar high-level language like Haskell—but we shall take it. In particular, Halstead interprets D as the average number of "mental discriminations" necessary in each "mental comparison." These terms lack rigor, but we shall use them to describe the "effort" metric.

Effort If D average mental discriminations needed to make a mental comparison, and V mental comparisons are needed for a program, then $E = DV$ measures how many mental discriminations are needed to write a given program.

Verifying the metrics

Despite Halstead's logical derivations, there is little reason to believe that his metrics measure anything without empirical evidence to support them. After all, why should these seemingly arbitrary metrics have any sway over the expressive complexity of a program? No amount of hand-waving can free us from the burden of evidence.

Halstead himself presents some impressive empirical evidence for his metrics in [Hal77], but we shall reproduce some here for a modern audience. The Python tool `halstead`[2][Dou19] uses the library `radon` [Lac18] to analyze the Halstead metrics of Python git repositories.

In Figures 2.2 to 2.4, we have plotted the metric N against the *computed length* metric, $\hat{N}$. The computed length is defined by

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2.$$

Halstead argues that N should be directly propositional to N—indeed, even that $\hat{N} = N$—which may seem surprising at first glance. On the surface, they seemingly have no relationship to one another. Nevertheless, as our figures show, there is a strong linear linear relationship between the two. Halstead provides data such as this to argue for his metrics.

2.2 Ambiguity

In defining the Halstead metrics we explicitly assumed that the concepts "operator" and "operand" were understood. Given the broad spectrum of programming languages and the modes of thought surrounding them, this allows for a substantial amount of ambiguity. What exactly is an operator? What exactly is an operand? To what extent can we *reliably* compute the Halstead metrics? It turns out that these questions do not have good answers.

Halstead himself is silent on this matter, but critics have pointed out this flaw. From [AA05]: "Halstead has not explicitly described the generic measurable concepts of operators and operands."

[2]Disclosure: I created `halstead`.

Further, "there is no general agreement among researchers on the most meaningful way to classify and count these tokens." Halstead's "Software Science" does not quite live up to its name.

To be intellectually honest, we should temper our conclusions with some doubt. We shall use two different implementations of the Halstead metrics written in different languages designed for different purposes. Only one of these implementations are open for inspection. It is difficult to guarantee that our measurements are exactly equal across languages. We must plead that our measurement errors wash out in the average. This is the unfortunate state of affairs.

In fact, this situation is not even unique to the Halstead metrics. Computational complexity analysis is often performed under the assumption that some operations are "basic," and should not really count as operations. For example, addition is often taken to be constant time, while addition in a basic for-loop is linear time. If we took a different set of operations to be basic, then we would have different results.

We offer the following, final disclaimer:

The Halstead metrics are defined by the tool used to measure them.

For Python programs, we use the library `radon`[3]. For Maple, we use the builtin library `SoftwareMetrics`. Both libraries include definitions of the Halstead metrics and various other complexity measures. Maple's `SoftwareMetrics` is essentially not open source, while `radon` is hosted on GitHub. See [Lac18] for more information on `radon` and [Map11] for `SoftwareMetrics`.

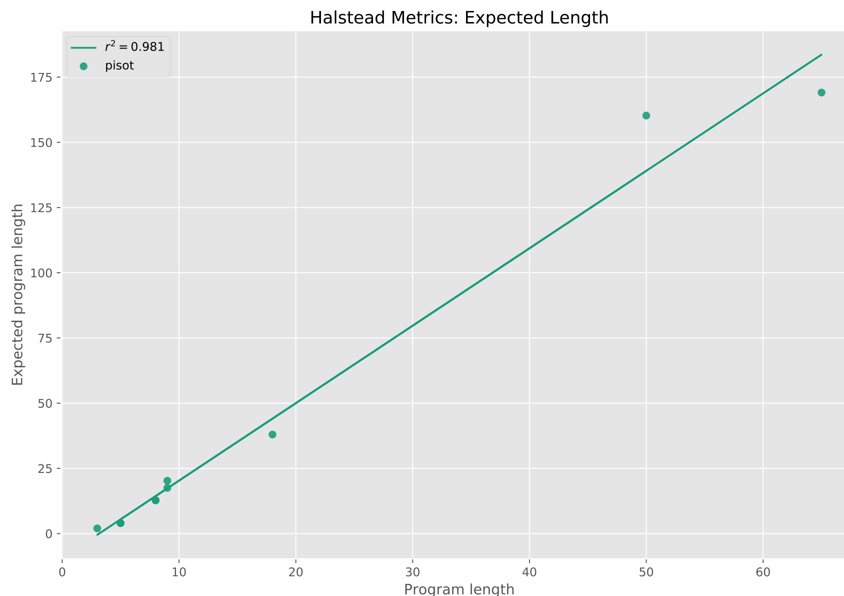


Figure 2.2: `halstead` applied to rwbogl/pisot.

[3]Disclosure: I contributed to the `radon` library in the course of his work.

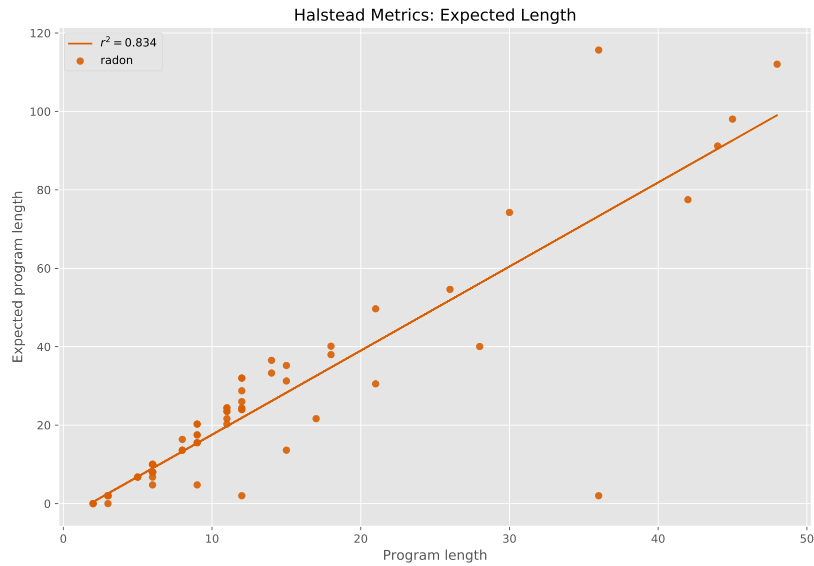


Figure 2.3: **halstead** applied to rubik/radon.

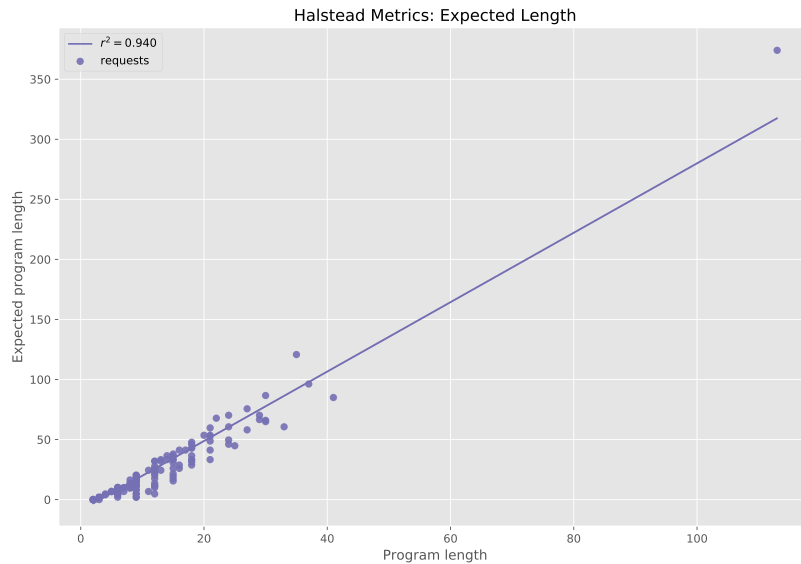


Figure 2.4: **halstead** applied to requests/requests.

Chapter 3

C-finite and Pisot Sequences

> In my eyes, a piece of human mathematics is interesting and important if and only if it can give us a clue how to *generalize* it and *teach* the method to the computer.
>
> Doron Zeilberger, Opinion 72 [Zei16].

In this chapter we will discuss a recent result of Neil Sloane and Doron Zeilberger on the *Pisot* sequences. The Pisot sequences are a class of sequences defined by a somewhat complicated recurrence involving floors and squares. In 1938 Charles Pisot demonstrated some connections between them, algebraic number theory, and the Pisot–Vijayaraghavan (PV) numbers [Pis38]. We are not especially interested in these connections, but rather the remarkable decision procedure described by Sloane and Zeilberger in [ESZ16] to determine if a Pisot sequence satisfies a "simple" recurrence. In doing so Sloane and Zeilberger make extensive use of Maple, which provides us an excellent example of computer algebra in the wild and under the reins of experienced mathematicians.

3.1 Definitions and preliminaries

Definition 2 (C-finite sequences)**.** A sequence $\{a_n\}$ is C-*finite* iff it satisfies a finite linear recurrence relation with constant coefficients. That is, iff

$$a_{n+d} = \sum_{0 \leqslant k < d} c_k a_{n+k} \qquad (n \geqslant 0)$$

for some fixed positive integer d and constants $c_1, \dots, c_d$.

The class of C-finite sequences occur often in combinatorics and related fields. For example, the Fibonnaci numbers $\{F_n\}$ are C-finite from the recurrence

$$F_{n+2} = F_{n+1} + F_n, \qquad n = 0, 1, 2, \dots.$$

If S_n denotes any set with n elements and $\mathcal{P}(S_n)$ its powerset, then $|\mathcal{P}(S_n)|$ is C-finite with

$$|\mathcal{P}(S_{n+1})| = 2 \cdot |\mathcal{P}(S_n)|.$$

The C-finite sequences have a simple characterization: A sequence $\{a_k\}$ is C-finite iff its generating function $A(x) = \sum_{k\geqslant 0} a_k x^k$ is rational. By partial fractions, it follows that every C-finite sequence can be expressed in a closed-form way completely algorithmically. For details and further properties of C-finite (and related) sequeneces, see [KP11].

Definition 3 (Pisot sequences)**.** Denote by $E_r(x, y)$ the integer sequence defined by the recurrence

$$\begin{aligned} a_0 &= x \\ a_1 &= y \\ a_n &= \left\lfloor \frac{a_{n-1}^2}{a_{n-2}} + r \right\rfloor, \quad n = 2, 3, \ldots \end{aligned}$$

where $0 < x < y$ are integers and r is a real constant in $[0, 1]$. The class of sequences $E_r(x, y)$ so defined are the *Pisot sequences*, defined by Charles Pisot in 1938 [Pis38; Boy78; ESZ16]. In their original formulation, the parameter r was fixed as $1/2$. In this case, the definition of a_n is synonymous with the integer nearest to a_{n-1}^2/a_{n-2}, rounding up in the case of a tie.

The Pisot sequences have some interesting properties and connections to other areas, but we are interested in the following fact: *Some* Pisot sequences are C-finite. For instance, the sequence $E_{1/2}(5, 17)$ satisfies

$$a_n = 4a_{n-1} - 2a_{n-2}, \qquad n \geqslant 2.$$

However, some Pisot sequences merely *seem* to be C-finite, but actually are not! For example, $E_{1/2}(30, 989)$ satisfies

$$a_n = 33a_{n-1} - 2a_{n-2} + 30a_{n-3} - 11a_{n-4}$$

for $4 \leqslant n \leqslant 15888$, but the equality breaks down at $n = 15889$ (!). Before the results in [ESZ16], explaining this surprising behavior was an open question. The answer is neatly contained in the following theorem.

Theorem (Sloane and Zeilberger)**.** *Let $\{a_n\}_{n\geqslant 0}$ be a C-finite sequence. If the largest root of the characteristic equation of $\{a_n\}$ is the only root outside of the unit circle, then $\{a_n\}$ satisfies the Pisot recurrence*

$$a_{n+1} = \left\lfloor \frac{a_{n-1}^2}{a_{n-2}} + r \right\rfloor \quad (0 < r < 1)$$

asymptotically. That is, it satisfies the recurrence for all $n \geqslant n_0$, where n_0 is some computable constant. In particular, to check if $E_r(x, y) = \{a_n\}$, it suffices to check the finitely many terms with $n \leqslant n_0$.

Before moving on, let us explore an interesting connection between the Fibonacci numbers and the Pisot sequences. The Fibonacci sequence $\{F_n\}$ has characteristic equation

$$x^2 - x - 1 = 0.$$

There are two roots, only one of which is outside of the unit circle. It follows that $\{F_n\}$ is *asymptotically* a Pisot sequence. In particular, working through the proof in [ESZ16], we can see that it

works for $n \geqslant 4$ when $r = 1/2$. Therefore, $\{F_n\}$ consists of the initial terms $\{0, 1, 1, 2\}$ prepended to the Pisot sequence $E_{1/2}(3, 5)$. It is easy to observe this computationally:

$$F_n : [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \dots]$$
$$E_{1/2}(3, 5) : [3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \dots].$$

We can also classify Pisot sequences that *seem* to be C-finite. Essentially, if the characteristic equation has two roots outside of the unit circle, and one is very close to the circle, then the relation will break down after many terms. The exact n where the relation breaks down can be predicted. For more details, see [ESZ16].

3.2 Computational problems

We want to know if a given Pisot sequence is C-finite. There is a simple algorithm to do this, sketched in [ESZ16]. We break the algorithm into two parts:

1. Guess a possible linear recurrence.
2. Verify that the conjectured recurrence holds asymptotically, and check the finitely many terms before the recurrence begins to hold.

Guessing linear recurrences

To determine if a Pisot sequence is C-finite, we must first determine what C-finite sequence it might be. Fortunately, we do not need to be clairvoyants to accomplish this. A simple linear algebra routine will suffice.

Suppose that the sequence $\{a_n\}_{n \geqslant 0}$ satisfies the d-order recurrence

$$a_n = \sum_{k=1}^{d} c_k a_{n-k}.$$

Then, the following system in $\{c_k\}_{k=1}^{d}$ would have a solution:

$$\begin{aligned}
a_{d-1}c_1 + a_{d-2}c_2 + \cdots + a_0 c_d &= a_d \\
a_d c_1 + a_{d-1}c_2 + \cdots + a_1 c_d &= a_{d+1} \\
&\vdots \\
a_{2d-2}c_1 + a_{2d-3}c_2 + \cdots + a_{d-1}c_d &= a_{2d-1}.
\end{aligned} \tag{3.1}$$

All we need to do is solve this system of equations for the coefficients. This is entirely routine, in the sense that a computer can do them.

Therefore, if a sequence $\{a_n\}_{n \geqslant 0}$ is C-finite, then we can determine the coefficients by computing and solving systems of the form (3.1) for finitely many d. If we *think* that a sequence satisfies such a recurrence, then we can check as many values of d as desired. Should one yield a solution, then we have a conjectured recurrence. This algorithm cannot *disprove* a linear recurrence, but it can give a *lower bound* for the order that such a linear recurrence must have.

Pisot algorithm

Armed with the tools to guess linear recurrences, the procedure to determine whether or not a Pisot sequence is C-finite is simple. First, we determine a canditate C-finite sequence. Then, following the results in [ESZ16], we check to see if only one root of the resulting characterstic equation is outside of the unit circle. If this is the case, then we know that the C-finite sequence is *eventually* a Pisot sequence. We need only check that sufficiently many terms of the C-finite sequence agree with the Pisot sequence. That is, we may prove this *by example*.

Despite the technical arguments used to develop it, this procedure is quite simple. It distills to computing the roots of a polynomial and making comparisons. Any decent computer algebra system can compute roots or approximate them accurately, so the task is more about evaluating programming capabilities than mathematical capabilities.

3.3 Expressive complexity analysis

For our complexity analysis we shall compare two bodies of code: Doron Zeilberger's `Pisot` Maple package, and a bundle of Python code from various sources to solve the Pisot problem. This bundle includes SymPy's `sequences` module and the author's own `pisot` module.

The method taken to analyze the code is as follows: Concatenate all relevant source code into a single file, then apply the language specific program to evaluate the Halstead metrics on the combined source file. Both programs run at the procedure level, so every procedure has its Halstead metrics evaluated individually. The results of this operation are fed into a `pandas` dataframe for further analysis and plotting. The statistical summary of this analysis is in Tables 3.1 and 3.2.

A cursory glance at our statistical summary makes it clear that Python has superior scores in nearly every Halstead metric. In particular, note that the *maximum* observed difficulty in the Python procedures is only just beyond the 50th percentile observed for the Maple procedures for the same metric. For the effort metric, Python's maximum is only just beyond the 75th percentile of observed difficulties in Maple. The situation is similar for the remaining metrics.

In addition to tables summarizing the datasets, Figures 3.1 and 3.2 provide some graphical perspective. In them, we see that most Python procedures we analyzed are relatively short. The shortest Maple procedures match these in effort and difficulty, but as length grows the Python looks like it just *barely* begins undercutting the Maple dataset. Though tempting, we should refrain from making the conclusion that this pattern holds generally; at this stage of our venture we simply do not have enough data to support it. Our figures do, however, strongly motivate choosing problems with lengthier procedures and larger code bases.

Finally, to drive home the results, we have displayed histograms of log-effort and log-difficulty in Figures 3.3 and 3.4. The log scale makes it easier to compare the relative magnitude of effort and difficulty across the two languages. From the figures, we see that the analyzed Maple procedures tend to have effort and difficulty many times that of their Python counterparts.

3.4 Qualitative discussion

For our qualitative discussion, we shall focus on an important subroutine of the Pisot algorithm. During the course of its execution, the Pisot algorithm must determine the second largest root of the characteristic equation of a C-finite sequence. In Zeilberger's `Pisot.txt`, this is accomplished in the procedure `Pis`. In the author's `pisot.py`, this is accomplished in `pisot_root`. We shall

	vocabulary	length	volume	difficulty	effort
count	45	45	45	45	45
mean	27.18	50.60	265.64	7.76	3300.40
std	18.55	50.32	325.41	5.39	5756.45
min	3.00	3.00	4.75	0.50	2.38
25%	11.00	15.00	53.77	3.33	182.59
50%	25.00	38.00	167.18	7.64	1254.19
75%	33.00	59.00	289.51	10.11	4317.97
max	95.00	283.00	1859.27	23.47	29980.71

Table 3.1: Summary of Maple expressive complexity statistics.

	vocabulary	length	volume	difficulty	effort
count	44	44	44	44	44
mean	9.59	13.18	54.42	1.61	241.68
std	10.34	20.39	115.60	1.50	887.33
min	2.00	2.00	2.00	0.50	1.00
25%	3.00	3.00	4.75	0.50	2.38
50%	6.50	6.50	17.58	1.10	21.25
75%	11.00	12.50	43.02	1.94	97.86
max	52.00	121.00	689.75	8.37	5774.68

Table 3.2: Summary of Python expressive complexity statistics.

compare and contrast language features qualitatively, in an effort to determine *why* our quantitative measurements are as they are.

Source listings for these procedures are given in Section 3.5. The important four are Listing 3.1, Listing 3.2, Listing 3.3, and Listing 3.5. Listing 3.1 and Listing 3.3 are the implementations of the basic Pisot algorithm in Maple. They are the logic and root finding procedures, respectively. Listing 3.2 and Listing 3.5 serve the same purpose for Python.

Complexity encapsulation

First, we should address the input types of our procedures. We have stripped all comments from the source listings for brevity, so it may not be clear what is being passed around. The Maple procedure `Pis` represents C-finite sequences as a list of lists with two elements. The first list contains the coefficients of the recurrence, and the second list contains the initial values. For instance, the Fibonacci sequence could be represented as

$$[[1, 1], [0, 1]].$$

The Python function `pisot_root` expects a `CFinite` class, a class which implements various useful functions on C-finite sequences. For instance, the class has `degree` and `coeffs` attributes and can be sliced to get values, as in

```
fib[:6] = [0, 1, 1, 2, 3, 5].
```

This asymmetry in input types may seem unfair, but it reflects the asymmetry in how embedded classes are in the respective languages. Python classes are “first class” citizens; everything in Python

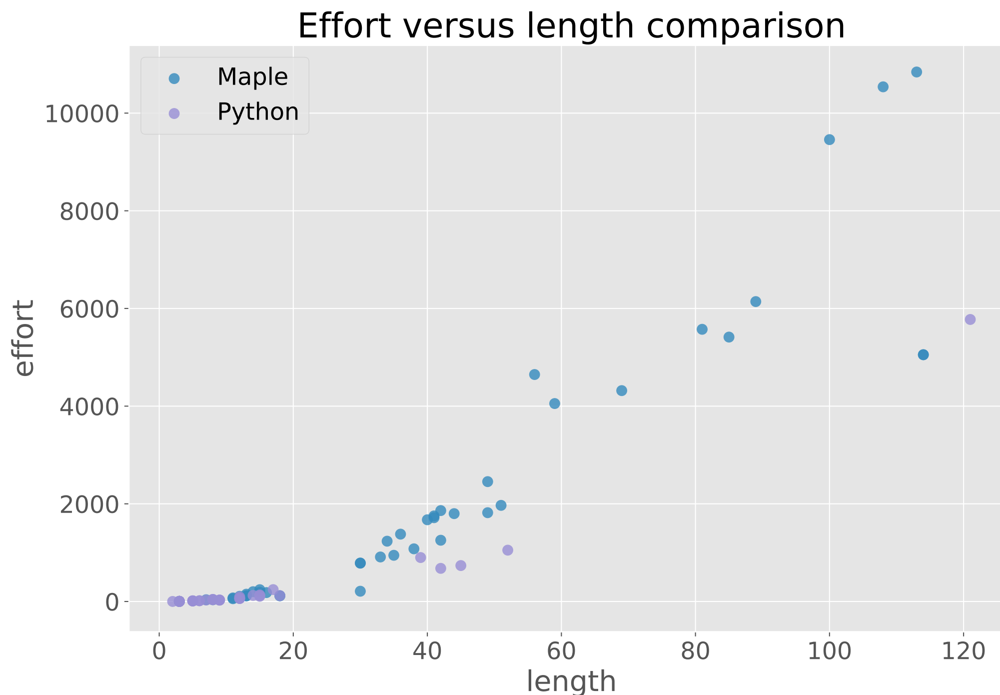


Figure 3.1: Scatterplot comparing effort against length for Maple and Python code to solve the Pisot problem. Python has clusters of low length, but it is hard to discern a relative pattern for longer length procedures. (Two outliers of the Maple dataset are omitted for clarity. They lie far up and to the right of the displayed points.)

is a class or an instance of a class, and implementing new objects is often the most straightforward way to abstract behavior. In Maple, classes are "second class" citizens; Maple classes were only implemented in 2012 with the release of Maple 16, and they are largely relegated to the background of the documentation. The result of this relegation, as we can see through Zeilberger's code, is that complex data types with shared behavior are considered on an ad-hoc basis. This is one area where Python is the clear favorite for expressive complexity.

For example, consider this operation from the first line of `Pis`:

```
[solve(x^nops(C[2])-add(C[2][i]*x^(nops(C[2])-i),i=1..nops(C[2]))]:
```

The variable `C` is the input list representing a C-finite sequence. To understand this line, a programmer must understand the structure of `C`. A comment could explain it or a programmer could memorize it, but these solutions take time and ignore the role that language design can play. This is exactly the problem that a record type could solve, but Maple's record system is particularly weak. Here, an experienced user has opted to create a complicated, list-based solution.

Contrast this situation with line 2 of Listing 3.5:

```
c_seq.characteristic_roots()
```

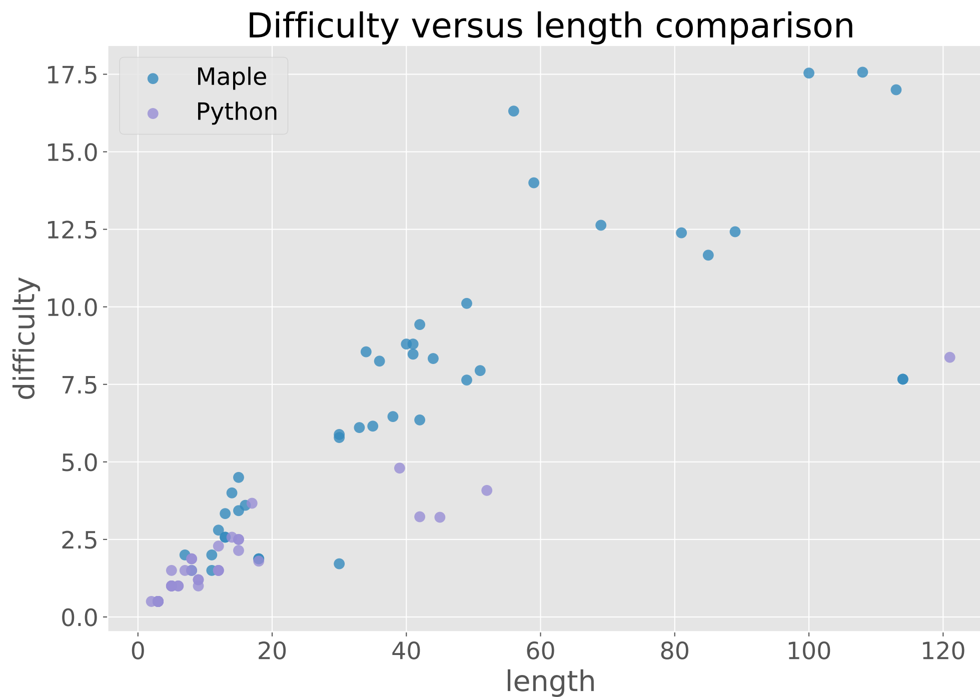


Figure 3.2: Scatterplot comparing difficulty against length for Maple and Python code to solve the Pisot problem. Python and Maple begin on a similar trajectory, but it appears that Python begins to undercut Maple for longer lengths. This figure, along with Figure 3.1, suggests that we should analyze *longer* Python programs to see if this pattern continues.

Using the `CFinite` class, the root-finding operation is abstracted into a single method call which clearly indicates its purpose. The complicated line in the Maple snippet still exists, but it has been abstracted by the `CFinite` class where the user will not have to think about it.

Of course, *in principle* the complicated Maple line 2 in `Pis` could have have been abstracted into a function call or a Maple class. However, it was not. *In principle* a programmer could always make optimal design choices, but they often do not. Programming languages should encourage and lower the barrier to making good design choices. Here, Maple does not.

Fundamental datatypes

A language's built-in datatypes, when properly designed and used, can immensely reduce the expressive complexity of programs. We previously used Listing 3.3 as a springboard to discuss Maple's lack of coherent classes and records, but its *built-in* datatypes mostly match Python's, with some peculiarities. In particular, minor modifications to the Maple Listing 3.3 can make it essentially match the Python Listing 3.5.

The fundamental container datatypes for Maple are lists and sets; on top of these, Python adds a dictionary datatype. Maple lacks a strict dictionary *datatype*, but allows arbitrary indexing of

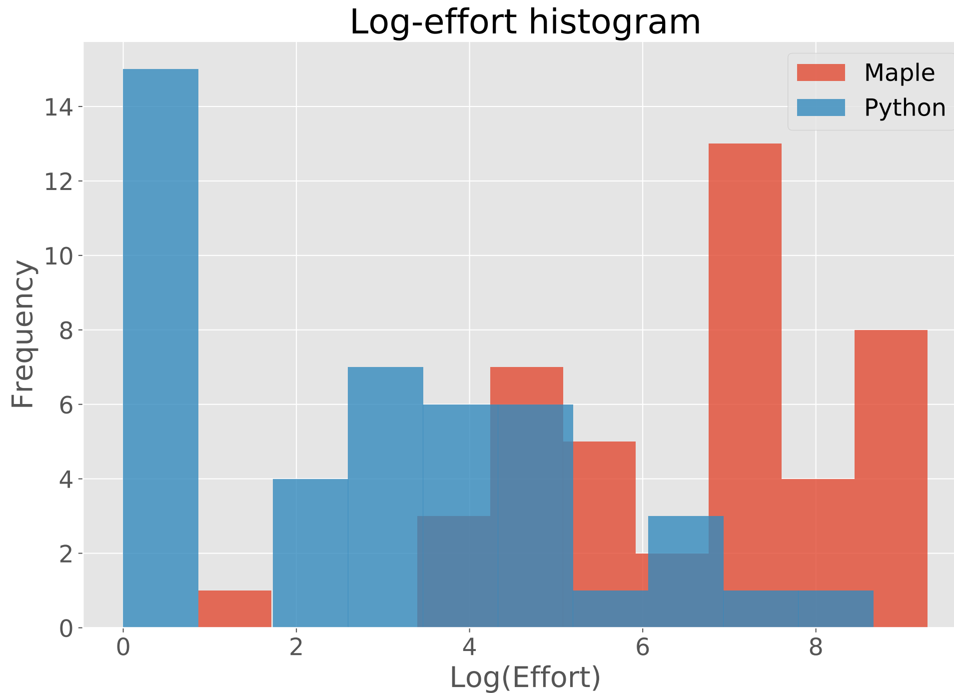


Figure 3.3: Histogram of log-effort comparing Python and Maple. This demonstrates the relative difference between the two values. That Python's histogram is concentrated further to the left than Maple's signifies that Python generally has lower effort.

symbolic variables. For example:

```
1 x["foo"] := 1:
2 x["foo"]; # Evaluates to 1.
```

This feature does not feature prominently in the official Maple documentation, so it is rarely used. Nonetheless, Maple *can* mimic dictionary datatypes.

The decision algorithm sketched in [ESZ16] is only suitable for certain kinds of Pisot and C-finite sequences. In particular, if 1 is a root of the characteristic equation, then it should be ignored. If it is the *only* root, then the procedure breaks down. Thus, having the roots, we must remove 1 and check if it was the only root. Three snippets to accomplish this are shown in Figure 3.5.

Looking at the snippets in Figure 3.5, it would seem that Maple has some bizarre hoops to jump through for list operations, which may Python an edge in expressive complexity. However, this is actually just the consequence of working with the wrong datatype. The code from [ESZ16] chose to keep its roots as a list when a set would be more beneficial. This small change results in a much simplified procedure, as shown in Listing 3.4.

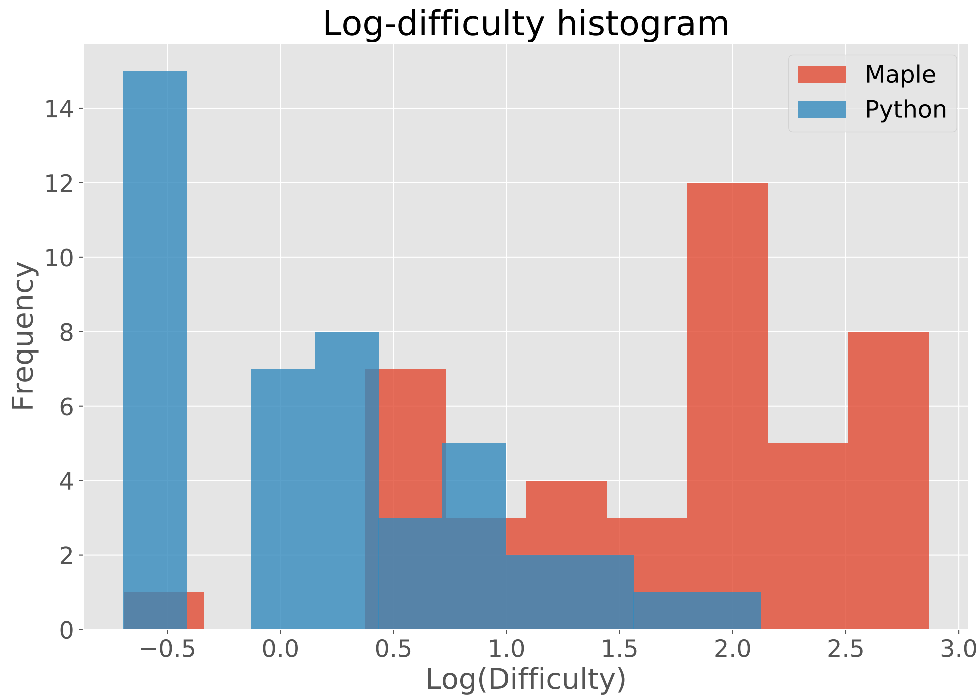


Figure 3.4: Histogram of log-difficulty comparing Python and Maple. That Python's histogram is concentrated further to the left than Maple's signifies that Python generally has lower effort.

Symbolic quality

Though expressive complexity is an important consideration when choosing a language for symbolic computation, it cannot be the only one. In particular, the quality of the symbolic computation should be considered. Thus, in this subsection we will describe the symbolic capabilities of Maple and Python.

First, it should be obvious that Python is imbued with symbolic capabilities by the library `SymPy` rather than having them built-in. This is not an excessive hurdle to overcome with modern package managers, but it does require some familiarity with computer systems. Maple has baked-in symbolic capabilities. In fact, variables in Maple are assumed to be symbolic by default, as opposed to Python where they must be declared as such with the `SymPy` library. This adds a further slight inconvenience to Python.

The bulk of the computer algebra in this problem comes in the root finding procedure. Both computer algebra systems handle this in fundamentally the same way. Maple has a built-in `solve` function, while `SymPy` implements a `solve` function, which are almost identical. Maple has no syntactical advantage over Python in this regard. The quality and speed of symbolic solutions and numerical approximations are roughly the same. This is to be expected; any self-respecting computer algebra system can handle polynomials.

```
if 1 in roots:
    del roots[1]

if not roots:
    return None
```

(a) Python.

```
lu := convert({op(lu)} minus
    {1},list):

if lu = [] then
     RETURN(FAIL):
fi:
```

(b) Maple from [ESZ16].

```
while member(1, lu, 'index')
    do
     subsop(index=NULL, lu):
od

if lu = [] then
     RETURN(FAIL):
fi:
```

(c) Alternative Maple code without type conversion. The while-loop is necessary to remove all instances of 1 from `lu`.

Figure 3.5: Comparison of the "deletion" operation in Python dictionaries and Maple lists. Though the Python operations are clearly simpler here, the Maple operations can easily compete with Python when the program is properly designed. This is demonstrated in Listing 3.4.

Expressive complexity summary

Maple's lack of a coherent class or record system makes it difficult to abstract complexity in complicated expressions. The language does not encourage sensible design decisions when it should. However, aside from some peculiarities, Maple's built-in datatypes match closely with Python, to the point that its operations could be put in 1-1 correspondence. Maple and Python seem evenly matched for basic symbolic computations like root finding.

3.5 Source code listings

```
PtoR := proc(x,y,r,L,K) local C,gu1,gu2,i,ku1,ku2:
    if r <= 0 or r >= 1 then
        print("r must be strictly between 0 and 1 for this procedure"):
        RETURN(FAIL):
    fi:

    C := GuessRecAd(PS(x,y,r,2*L+10),L):
    if C = FAIL then
        RETURN(FAIL):
    fi:

    gu1 := SeqFromRec(C,K):
    gu2 := PS(x,y,r,K):

    if gu1 <> gu2 then
        for i from 1 to K while gu1[i]=gu2[i] do od:
        RETURN([FAIL,C,i,Pis(C)]):
    fi:

    ku1 := evalf(PisInd(C,r,1,K),10):
    ku2 := evalf(PisInd(C,r,trunc(K/2),K),10):

    if C[2] = [2,-1] then
        [C, ku1, ku2, LINEAR]:
    elif Pis(C) > 1 then
        [FAIL, C, FirstDev(x,y,r,C), Pis(C)]:
    else
        [C, ku1, ku2, Pis(C)]:
    fi:
end:
```

Listing 3.1: A cleaned and simplified version of PtoR from [ESZ16].

```
def pisot_to_cfinite(pisot, guess_length, check_length):
    if pisot.r <= 0 or pisot.r >= 1:
        raise ValueError("r must be strictly between 0 and 1 for this
            procedure")

    c_seq = pisot.find_cfinite_recurrence(guess_length)

    if not c_seq:
        return None

    if pisot[:check_length] != c_seq[:check_length]:
        return None

    root_abs = pisot_root(c_seq)

    if root_abs > 1:
        return None

    return c_seq
```

Listing 3.2: A cleaned version of `pisot_to_cfinite`.

```
1  Pis:=proc(C) local x,lu,i,aluf,mu:
2      lu := [solve(x^nops(C[2])-add(C[2][i]*x^(nops(C[2])-i),i=1..nops(C[2]))
           )]:
3
4      if nops(lu) <> nops(C[1]) then
5          RETURN(FAIL):
6      fi:
7
8      if member(1,lu) then
9          lu := convert({op(lu)} minus {1},list):
10         if lu=[] then
11             RETURN(FAIL):
12         fi:
13     fi:
14
15     aluf:=1:
16     for i from 2 to nops(lu) do
17         if abs(evalf(lu[i])) > abs(evalf(lu[aluf])) then
18             aluf := i:
19         fi:
20     od:
21
22     mu := evalf([op(1..aluf - 1, lu),op(aluf+1..nops(lu), lu)]):
23
24     max(seq(abs(mu[i]), i=1..nops(mu))):
25 end:
```

Listing 3.3: A cleaned and simplified version of `Pis` from [ESZ16].

```
Pis:=proc(C) local x,lu,i,aluf,mu:
    lu := [solve(x^nops(C[2])-add(C[2][i]*x^(nops(C[2])-i),i=1..nops(C[2]))
       )]:

    if nops(lu) <> nops(C[1]) then
        RETURN(FAIL):
    fi:

    lu := {op(lu)} minus {1}:

    if lu = {} then
        RETURN(FAIL):
    fi:

    lu := {seq(evalf(abs(root)), root in lu)}:
    max_k := max[index](lu):
    subops(max_k=NULL, lu):

    max(lu):
end:
```

Listing 3.4: A "complexity optimized" version of `Pis` from [ESZ16].

```
def pisot_root(c_seq):
    roots = c_seq.characteristic_roots()

    if sum(roots.values()) != c_seq.degree:
        return None

    if 1 in roots:
        del roots[1]

    if not roots:
        return None

    norms = [re(abs(root).evalf()) for root in roots.keys()]

    max_index = norms.index(max(norms))
    del norms[max_index]

    return max(norms)
```

Listing 3.5: A cleaned version of `pisot_root`. The underlying `CFinite` class removes much of the mystery of Listing 3.3. Python's cleaner list handling eliminates loops and verbose type conversions.

```
1 def characteristic_poly(self, var=sympy.Dummy("r")):
2     d = self.degree
3     return (var**d
4               - sum(var**(d - k - 1) * self.coeffs[k]
5                      for k in range(self.degree)))
6
7 def characteristic_roots(self):
8     return sympy.solve(self.characteristic_poly(), dict=True)
```

Listing 3.6: A cleaned version of the characteristic equation handling from the `CFinite` class.

Chapter 4

Propositional Logic and Boolean Satisfiability

> Automated theorem proving has two goals: (1) to prove theorems and (2) to do it automatically. Over the years experience has shown these goals are incompatible.
>
> Melvin Fitting, *First-Order Logic and Automated Theorem Proving* [Fit12].

Many important mathematical statements can be translated into the form, "If condition A is true, then condition B must also be true." Or, for those so inclined, "$A \implies B$." This is a particular kind of statement in *propositional logic*, a logic which combines elementary *propositions* to construct more complex statements. Propositional logic is introduced to mathematics students fairly early in their education to familiarize them with manipulating logical structure. Once the basics are covered (the contrapositive, de Morgan's laws, etc.), it is usually abandoned for more useful topics such as sets, induction, and so on. This is a real shame, because propositional logic is a ripe target for *automated theorem proving*. In particular, every statement of the form $A \implies B$ can be checked completely automatically in propositional logic. We shall compare implementations of propositional logic in Maple and SymPy.

4.1 Definitions and preliminaries

Our main question is this: Given a propositional hypotheses S and a propositional conclusion X, can we decide if X follows from S? Written another way, if $S = \{A_1, \ldots, A_n\}$, is the statement

$$A_1 \wedge A_2 \wedge \cdots \wedge A_N \implies X \tag{4.1}$$

a tautology?

Consider some examples: Does the statement $(A \implies B) \implies A$ always imply A? Or, in other words, is

$$((A \implies B) \implies A) \implies A \tag{4.2}$$

a tautology? Do the statements $A \implies P$, $B \implies Q$, and $A \vee B$ imply $P \vee Q$? That is, is

$$(A \implies P) \wedge (B \implies Q) \wedge (A \vee B) \implies P \vee Q \tag{4.3}$$

a tautology? Further consider the following formula:

$$(P \implies (Q \implies R)) \implies ((P \implies Q) \implies (P \implies R)). \tag{4.4}$$

Is *this* a tautology? It is evident that such formulas become difficult to verify as their complexity increases[1].

There are fairly efficient, mechanical procedures to check the validity of most propositional formulas. To adequately describe these procedures, we must first give an adequate definition of propositional logic.

Briefly, propositional logic is the familiar boolean algebra formed with the usual "logical and," "logical or," and "negation" operators over the set $\{t, f\}$. Its variables are *propositions*, implicitly interpreted as "basic statements" which can be either true or false. Its sentences look like this:

$$A \wedge B, \quad (\neg P_1 \wedge P_2) \vee P_3, \quad \neg A \vee A, \quad A \wedge \neg A. \tag{4.5}$$

We would say that the first two are true *sometimes*, the third is *always* true, and the last is *always* false.

A more formal definition of propositional logic requires a bit of work. The *syntax* of propositional logic—the underlying language—and the *semantics* of propositional logic—truth, operations and so on—must both be defined. We shall now give these definitions.

The "properly formed sentences" in propositional logic are the *propositional* formulas.

Definition 4 (Propositional formulas)**.** Our language is built from two parts:

1. A countable set of distinct symbols, $\{P_1, P_2, \ldots\}$, and two constants $\top$ and $\bot$. The former are called the *propositional letters*, while the latter two are thought of as *true* and *false*, respectively. These values together are the *atomic formulas*.

2. A set of binary connectives ($\wedge$, $\vee$, and so on) and the unary connective $\neg$.

The *propositional formulas* constitute the smallest set $\mathbf{P}$[2]such that:

1. $\mathbf{P}$ contains all atomic formulas.

2. If $X \in \mathbf{P}$, then $\neg X \in \mathbf{P}$.

3. For any binary connective $\circ$, if $X, Y \in \mathbf{P}$, then $X \circ Y \in \mathbf{P}$.

For example, every sentence in (4.5) is a propositional formula.

The definition of $\mathbf{P}$ as the *smallest* such set lets us prove various things about $\mathbf{P}$ by induction on the size of formulas, the foremost application of which is to prove that propositional formulas can be uniquely parsed. We omit this proof, trusting that the reader is sufficiently familiar with propositional logic to continue.

[1] These formulas *are* tautologies. They are, respectively, Peirce's Law [Pei85], a simple argument, and Frege's second propositional axiom [Bus98].

[2] At least one set—the set of all strings formed from the given symbols—satisfies the stated properties, and the intersection of such sets also satisfies the stated properties. The propositional formulas are the intersection of all such sets.

Definition 5 (Boolean valuations)**.** A *boolean valuation* is a map $\nu \colon \mathbf{P} \to \{\mathbf{t}, \mathbf{f}\}$ such that the following properties hold:

1. $\nu(\top) = \mathbf{t}$; $\nu(\bot) = \mathbf{f}$, where $\top$ and $\bot$ are the propositional constants true and false. (The values $\mathbf{t}$ and $\mathbf{f}$ are formally different from $\top$ and $\bot$, but largely for historical reasons.)

2. $\nu(\neg X) = \neg\nu(X)$ for all propositional formulas X, where $\neg$ is the negation operator.

3. $\nu(X \circ Y) = \nu(X) \circ \nu(Y)$ for all propositional formulas X and Y and all binary connectives $\circ$.

Note that at this point we further require all connectives in the language of propositional logic to be operators on the set $\{\mathbf{t}, \mathbf{f}\}$. For instance, the binary connective $\wedge$ is an operator defined as the "logical and." That is, $x \wedge y = \mathbf{t}$ iff both x and y are $\mathbf{t}$. Thus there are actually only finitely many binary connectives to consider.

Being recursively defined, boolean valuations are entirely determined by their values on the propositional letters. That is, they amount to deciding which of the variables in an expression are "true" and which are "false." We are are especially interested in those propositional formulas which *must* be true because of their structure. The following definition captures this idea, and connects our syntax with our semantics.

Definition 6. A propositional formula X is a *tautology* iff $\nu(X) = \mathbf{t}$ for all boolean valuations ν.

This is equivalent to the common "truth table" definition of a tautology. As we might expect, the propositional formulas $P \vee \neg P$ and $\neg\neg\top$ are tautologies.

Definition 7. A propositional formula X is a *propositional consequence* of a set of propositional formulas S iff $\nu(X) = \mathbf{t}$ for every boolean valuation ν that is true for every formula in S. We also write $S \models X$, and say that S *entails* X. (Note that X is a tautology iff $\emptyset \models X$.)

Propositional consequence is our real goal. Propositional formula (4.1),

$$A_1 \wedge A_2 \wedge \cdots \wedge A_n \implies X,$$

is a tautology iff

$$\{A_1, A_2, \ldots, A_n\} \models X.$$

Thus we begin the sequel with a discussion of how this decision problem might be solved. In that discussion, the following definition will be crucial.

Definition 8. A set of propositional formulas S is *satisfiable* iff $\nu(S) = \{\mathbf{t}\}$ for some boolean valuation ν.

4.2 Computational problems

Given a set of propositional formulas S and a specific propositional formula X, we want to know if $S \models X$. There are many ways to do this, but we must be careful to not fall into a trap. The statement $S \models X$ is true if and only if $\nu(X) = \mathbf{t}$ for every boolean valuation ν such that $\nu(S) = \{\mathbf{t}\}$. This is equivalent to stating that the set $S \cup \{\neg X\}$ is *not satisfiable*. That is, no boolean valuation can map all of its members to $\mathbf{t}$. The problem of checking a propositional consequence is therefore

a special case of the boolean satisfiability problem (SAT), famously the first problem proven to be NP-complete[3].

NP-completeness is often taken as stating that a problem is "very difficult." This is not entirely accurate, but it is true that naïve solutions to the SAT problem will likely perform poorly. Fortunately, SAT remains a popular research area due to its simple statement and many applications in computer science, engineering, and logic. More sophisticated methods developed in the past century have led to SAT solvers that can often solve "real world" SAT problems fairly quickly—despite its pernicious "NP-complete" label.

As a foil for the more sophisticated SAT solvers, consider the "truth table" method for tautology checking. Given a propositional formula X with n variables in it, there are 2^n different possible assignments (boolean valuations) that will affect its truth value. By systematically searching through all 2^n of these, we can check if X evaluates to true under all of them. If it does, then X is satisfiable; if it does not, then X is not satisfiable. The 2^n different valuations are often written in a table alongside X, hence the name. This is clearly a correct algorithm with exponential runtime. The reader is invited to check all of (4.2), (4.3), and (4.4) using this method, or to examine the truth table proof of (4.2) in Table 4.1.

A	B	((A	$\implies$	B)	$\implies$	A)	$\implies$	A
T	T	T	T	T	T	T	T	T
T	F	T	F	F	T	T	T	T
F	T	F	T	T	F	F	T	F
F	F	F	T	F	F	F	T	F

Table 4.1: Truth table proof of Peirce's law (4.2). Note that the *main connective* column, the column which determines the truth value of the statement, contains only T's, so the statement must be a tautology.

The basis for many more sophisticated SAT solvers is the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [ZS00]. In particular, the SAT solver used in SymPy is based directly on the DPLL procedure. The initial method (the DP method) is due to Davis and Putnam [DP60]. It was then refined by Davis, Logemann, and Loveland [DLL62].

The basic idea of the DPLL algorithm is to assign propositional letters truth values until a contradiction occurs, then backtrack to the problematic assignment. Backtracking alone is often a superior search method—though with $O(2^n)$ worst-case runtime—but the DPLL also includes numerous heuristics to improve its performance. For example, if a propositional letter is in a clause by itself, then it can be assigned a final value immediately. For example, to satisfy

$$A \wedge (B \vee C),$$

it is clear that A must be assigned $\mathbf{t}$. Such heuristics, simple though they may be, make a large difference.

Interesting advances have been made in SAT solvers in recent years. For example, the *MapleSAT* solver has employed machine learning heuristics along with other modern SAT techniques to achieve a high rate of success. MapleSAT has placed highly among the various annual SAT competitions,

[3] That is, any polynomial time solution to SAT would imply a polynomial time solution to every NP problem. This is the famous Cook–Levin theorem; see [Coo71].

	vocabulary	length	volume	difficulty	effort
count	26	26	26	26	26
mean	18.154	32.654	168.791	5.470	2382.534
std	20.372	45.321	279.649	6.037	5503.900
min	2.000	2.000	2.000	0.500	1.000
25%	2.000	2.000	2.000	0.500	1.000
50%	13.500	17.000	61.913	3.398	260.740
75%	22.500	34.000	153.801	9.131	1440.492
max	75.000	162.000	1009.069	22.337	22539.100

Table 4.2: Summary of Maple expressive complexity statistics.

and is now the deault SAT solver employed in Maple itself. See the thesis of Jia Liang [Lia18] for more details.

4.3 Expressive complexity analysis

For our complexity analysis we shall compare two bodies of code: SymPy's `logic` module and Maple's `Logic` package. The method to analyze the code is mostly identical to the previous chapter. For Python, we concatenate all relevant source code into a single file, then apply the `radon` program to evaluate the Halstead metrics on the single source file. For Maple, we simply apply the `SoftwareMetrics[HalsteadMetrics]` procedure to the `Logic` package. As before, both programs run at the procedure level, so every procedure has its Halstead metrics evaluated individually. The results of this operation are fed into a `pandas` dataframe for further analysis and plotting. The statistical summary of this analysis is in Tables 4.2 and 4.3.

As before, Python has superior scores. The maximum Python scores are sometimes orders of magnitude lower than Maple's maximums. As before, we have given graphical evidence in Figures 4.1 to 4.4. The conclusions drawn from those are roughly the same as before.

Note that we are once again lacking Python procedures long enough to compare with long Maple procedures, but that this is now a different situation. Before, we were comparing user-written Python code with user-written Maple code. We are now comparing library Python code with library Maple code. That is, the logic modules and packages we are comparing were written by the respective computer algebra system maintainers rather than their users. The problem has persisted across those two different scenarios, which might suggest that Maple as a language favors—for some reason—longer procedures. Longer procedures generally obtain higher Halstead metrics, which could explain the large differences seen here.

4.4 Qualitative discussion

For our qualitative discussion we shall focus on three short programs to apply the *resolution rule.* The resolution rule states that we can *resolve out* complementary propositional letters in disjunctions. That is, for propositional letters a_k, x, and b_k,

$$\{a_1 \vee \cdots a_n \vee x, \quad b_1 \vee \cdots b_m \vee \neg x\} \models a_1 \vee \cdots a_n \vee b_1 \vee \cdots \vee b_m.$$

	vocabulary	length	volume	difficulty	effort
count	56	56	56	56	56
mean	9.536	13.679	50.045	2.032	174.881
std	6.655	11.543	54.930	1.552	304.693
min	2.000	2.000	2.000	0.500	1.000
25%	4.000	6.000	12.000	0.667	8.000
50%	7.000	9.000	26.898	1.688	42.730
75%	14.000	19.500	73.682	2.821	232.972
max	26.000	54.000	247.588	7.206	1784.090

Table 4.3: Summary of Python expressive complexity statistics.

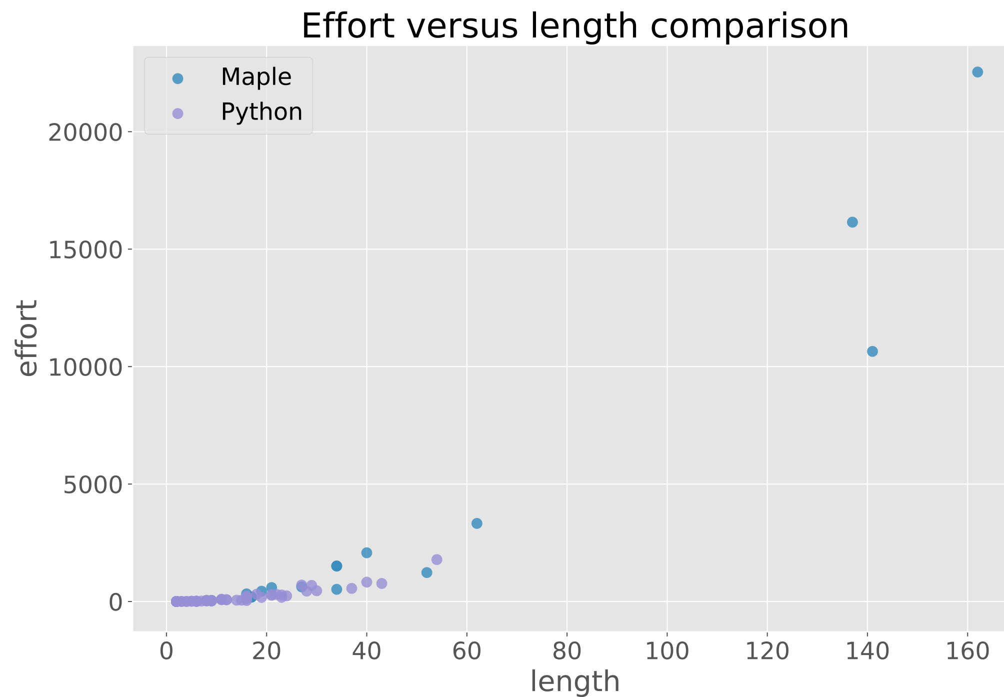


Figure 4.1: Scatterplot comparing effort against length for Maple and Python code for propositional logic. While Python has clusters of low length, it is hard to tell if it undercuts Maple in any real way without more data.

In other words, given two clauses that are disjunctions, if any propositional letter appears negated in the other, then it can be removed and the resulting clauses joined under a disjunction. Repeated application of this rule forms basis of many automated theorem provers. See, for instance, [Fit12].

Maple and Python programs to apply the resolution rule are presented in Listings 4.1 and 4.2. The solutions take fundamentally different approaches to representing logical operators. As an interesting foil, Listing 4.3 contains a Haskell implementation of the resolution rule similar to

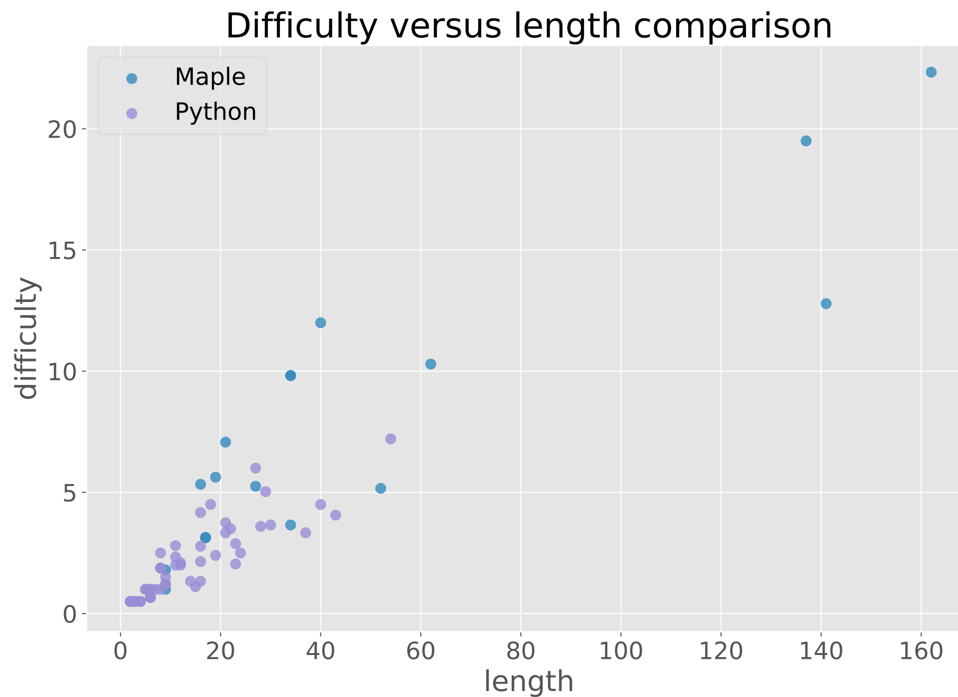


Figure 4.2: Scatterplot comparing difficulty against length for Maple and Python logic code. For lower lengths, it seems that both languages lie on the same trajectory. It is hard to tell if there is any divergence between the languages for longer programs. This figure, along with Figure 3.1, suggests that we should analyze longer programs.

Maple's approach. Simple examples of propositional logic in Maple and Python are presented in Listings 4.4 and 4.5.

Length and recursive data

The first striking difference between Listings 4.1 to 4.3 is the length of the programs. The Maple version clocks in at 36 lines, while the Python version is less than half that at 13 lines. Haskell, with no computer algebra support, yields a solution in 22 lines. Line count is not an exact representation of the Halstead "length" metric, but it is a starting point. The Maple program is significantly longer than its Haskell and Python counterparts. This lends some credence to a conjecture in the previous section; Maple has higher Halstead metrics because its programs tend to be longer.

The primary reason for the difference in lengths is the structural design of logic in each language. In SymPy, the `Or` class serves as a general `n`-ary disjunction with arguments kept in a tuple. This is possible because disjunction is associative and commutative, and therefore easily generalizes to any finite number of arguments. In Maple, the disjunction is implemented as the binary `&or` operator. A series of disjunctions in Maple is actually a series of *nested* disjunctions, not one disjunction applied to many arguments. That is, the list of arguments is essentially stored in a recursive

Figure 4.3: Histogram of log-effort comparing Python and Maple.

datatype. A rough graphical approximation of this is illustrated in Figure 4.5. This difference means that working with boolean objects in Maple beyond the superficial level brings hurdles to overcome, as our attention is taken away from the logic, and into the underlying structure.

Looking at Figure 4.5, it is tempting to think that Maple's choice to treat operators as nested leads to more complicated programs, but this is unfair. Recursive datatypes are not necessarily more complicated. They have been implemented ergonomically in many languages, particularly functional languages. One such language is Haskell [Jon03]. Listing 4.3 contains disjunction datatype as a binary data constructor implemented in Haskell, mirroring the spirit of Maple's disjunction. This will allow us to compare the expressive complexity of Maple's solution with a language that takes roughly the same approach to this problem.

Haskell is especially well equipped to handle recursive data types. This is largely due to pattern matching, a common feature in functional languages. Pattern matching allows (requires, even) users to create structured datatypes which are accessed syntactically. That is, datatypes, even user-created ones, are automatically embedded into the syntax of the language; they are married together. This makes working with structured datatypes exceptionally clear in languages like Haskell.

Maple, on the other hand, does not use pattern matching. Most structured datatypes—lists, binary expressions, and so on—are accessed using the polymorphic `op` function. The `op` function is difficult to describe, because it operates differently based on what its arguments are. On a list, it acts as an accessor by index; on a matrix, it both gives size information and elements; on

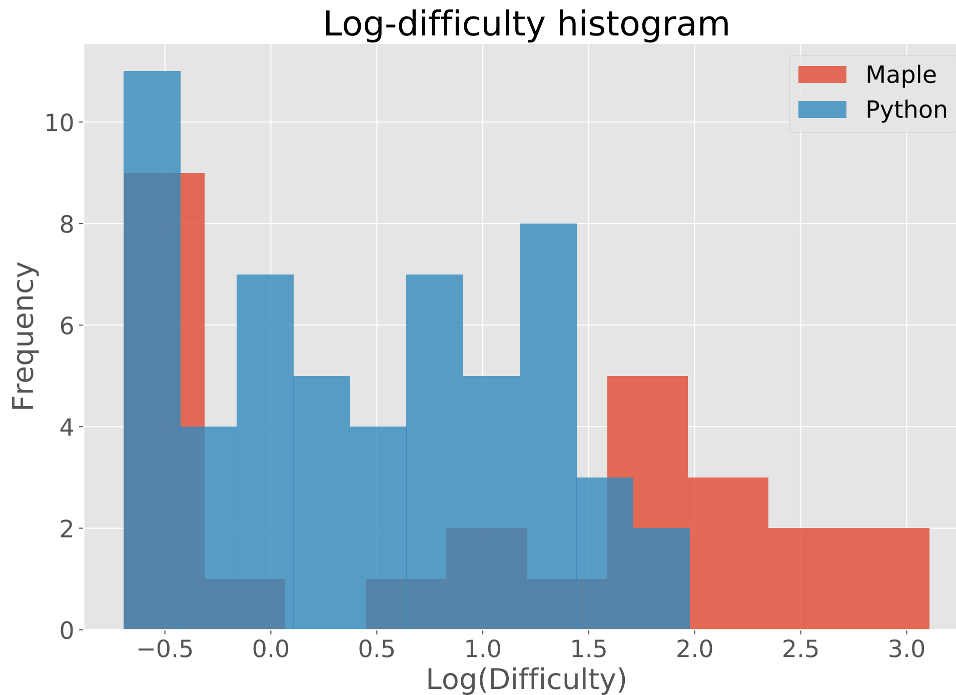
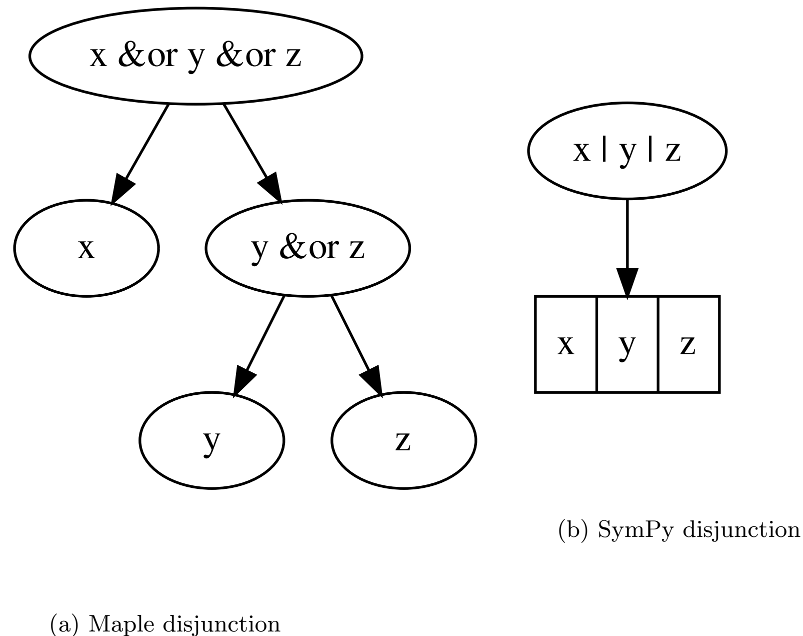


Figure 4.4: Histogram of log-difficulty comparing Python and Maple.

a binary expression, it parses the expression into arguments. The downfall of `op` is precisely its flexibility. When encountering an application of the `op` function, the programmer must try to figure out how the `op` function will behave on a particular datatype. There is no way to determine this by inspection, only a manual or some guesswork. Thus the much overloaded `op` ruins any hope of joining syntax and semantics.

In particular, compare the `unfold_dis` procedure in Listing 4.1 with the `toList` procedure in Listing 4.3. The Maple procedure checks the number of operands in the expression as a base case. "Number of operands in expression" is semantically separate from the actual datatype under consideration. The *goal* is to determine whether or not we are looking at a disjunction or a (possibly negated) variable. Maple's syntax provides no built-in way to handle this. The programmer must simply trust that "one operand in expression" is synonymous with "possibly negated variable." Haskell, on the other hand, forces the programmer to specify which case they are considering through pattern matching. There is (in this case) no other way to handle data in Haskell. Thus, rather than relying on "number of operands in expression," the Haskell program specifically asks "is this a disjunction or a variable?" That is, the syntax and semantics are tied together. This is an excellent example of good design.

In short, SymPy provides its logical objects in a convenient form. Maple provides its logical objects in a recursive form, but then fails to provide robust methods to handle them.



(b) SymPy disjunction

(a) Maple disjunction

Figure 4.5: Maple and SymPy disjunction representation.

Symbolic quality

Maple does not perform obvious simplifications when it would be appropriate. Consider the following example. The function `evalb` in Maple evaluates a given expression to a boolean for use in conditional statements. For any symbol `x`,

```
evalb(&not (&not x) = x);
```

evaluates to `false`. Thus Listing 4.1 applies the `BooleanSimplify` function whenever two propositional formulas are compared to ensure that equality is correctly checked. SymPy, on the other hand, makes such simplifications automatically. For instance, `~(~x)` evaluates directly to `x`. This allows Listing 4.2 to avoid any tedious calls to a simplification routine.

It is possible that avoiding such simplification would lead to more flexibility in transforming logical expressions. However, as seen in the previous section, this is already a chore in Maple.

Aside from the listings given here, Maple does benefit from having a cutting-edge SAT solver. SymPy would likely lag behind in performance unless it adopts an external SAT solver.

Expressive complexity summary

The design of nested expressions in Maple force programmers to grapple with the over-overloaded `op` function. As a result, handling logical expressions can be obnoxious in user-defined functions, as syntax and semantics are divorced. Maple could learn from languages—such as Haskell—with more

robust structured datatypes. SymPy provides convenient and consistent ways to inspect nested expressions in general.

4.5 Source code listings

```
with(Logic):

unfold_dis := proc(disjunction)
    if nops(disjunction) = 1 then
        return [disjunction];
    fi:

    left := unfold_dis(op(1, disjunction)):
    right := unfold_dis(op(2, disjunction)):

    return [op(left), op(right)]:
end proc:

resolve := proc(clause_1, clause_2)
    args_1 := unfold_dis(clause_1):
    args_2 := unfold_dis(clause_2):

    resolvants := []:

    for x in op(clause_1) do
        for y in op(clause_2) do
            neg_x_simp := BooleanSimplify(&not x);
            x_simp := BooleanSimplify(x);
            y_simp := BooleanSimplify(y);
            if neg_x_simp = y_simp then
                new_1 := select(c -> BooleanSimplify(c) <> x_simp, args_1):
                new_2 := select(c -> BooleanSimplify(c) <> y_simp, args_2):

                new := [op(new_1), op(new_2)]:
                resolvants := [op(resolvants), new]:
            fi:
        od:
    od:

    return resolvants:
end proc:

# Example usage.
resolve(x &and (&not y), y)
```

Listing 4.1: Implementation of the resolution rule in Maple with the `Logic` package. Since Maple lacks built-in pattern matching, the most straightforward way to "unroll" a sequence of disjunctions is a recursive function that terminates on expressions with a single operand. The mechanism to do this obscures the operation's semantics and hinders readability.

```
from sympy.logic.boolalg import Or


def resolve(clause_1, clause_2):
    args_1 = clause_1.args if clause_1.args else (clause_1,)
    args_2 = clause_2.args if clause_2.args else (clause_2,)

    for x in args_1:
        for y in args_2:
            if ~x == y:
                new_1 = [c for c in args_1 if c != x]
                new_2 = [c for c in args_2 if c != y]
                yield Or(*(new_1 + new_2))

# Example usage.
resolve(x & ~y, y)
```

Listing 4.2: Implementation of the resolution rule in Python with the SymPy package.

```haskell
import Data.List

data Term = Dis Term Term
            | Var String
            | NegVar String
            deriving (Eq, Show)

neg :: Term -> Term
neg (NegVar s) = Var s
neg (Var s) = NegVar s
neg _ = error "Only literal negations are implemented!"

toList :: Term -> [Term]
toList (Dis x y) = toList x ++ toList y
toList x = [x]

resolve :: Term -> Term -> [[Term]]
resolve clause_1 clause_2 = map resolveOut matchingLiterals
    where args_1 = toList clause_1
          args_2 = toList clause_2
          matchingLiterals = map neg args_2 `intersect` args_1
          resolveOut x = (args_1 \\ [x]) ++ (args_2 \\ [neg x])

-- Example usage.
x = Var "x"
y = Var "y"
main = print $ resolve (Dis x (neg y)) y
```

Listing 4.3: Implementation of the resolution rule in Haskell from scratch. Note how straightforward accessing the elements of a binary datatype through pattern matching is. In particular, note the effectiveness of pattern matching on the binary data constructor `Dis` in the `toList` function. Save for the standard library `Data.List` import, this is a self-contained program. (A Haskell expert could likely improve this implementation by a wide margin, exploiting the various abstract data classes meant for recursive datatype iteration.)

```
with(Logic):

peirce := ((A &implies B) &implies A) &implies A:
resolution := ((A &implies P) &and (B &implies Q) &and (A &or B)) &implies
    (P &or Q):
frege := (P &implies (Q &implies R)) &implies ((P &implies Q) &implies (P &
    implies R)):

print(Tautology(peirce));
print(Tautology(resolution));
print(Tautology(frege));
```

Listing 4.4: Checking Equations (4.2) to (4.4) with Maple's `Logic` package.

```
from sympy.logic.inference import valid
from sympy.abc import A, B, P, Q, R

peirce = ((A >> B) >> A) >> A
resolution = ((A >> P) & (B >> Q) & (A | B)) >> (P | Q)
frege = (P >> (Q >> R)) >> ((P >> Q) >> (P >> R))

print(valid(peirce))
print(valid(resolution))
print(valid(frege))
```

Listing 4.5: Checking Equations (4.2) to (4.4) with SymPy's `logic` module.

Chapter 5

Indefinite Summation

> Civilization advances by extending the number of important operations which we can perform without thinking about them. Operations of thought are like cavalry charges in a battle—they are strictly limited in number, they require fresh horses, and must only be made at decisive moments.
>
> Alfred North Whitehead, *An Introduction to Mathematics* [Whi17].

Sums are ubiquitous in mathematics. They are—quite unfairly—relegated to the dregs of analysis for most students, where they serve as a lifeless prop to develop the Riemann integral or some other theory. Summation is an important tool in its own right. Numerous combinatorial quantities are expressible via summations, sums over divisors are common in number theory, and—of course—errors in analysis can often be broken up into sums. Some sums are well known, such as

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \quad \text{and} \quad \sum_{k=1}^{n} \binom{n}{k} x^k = (1+x)^n.$$

More difficult ones are not:

$$\sum_{1 \leqslant k < n^2} \left\lfloor \sqrt{k} \right\rfloor = \frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n.$$

One way to evaluate $\sum a_k$ is to find d_k such that

$$\Delta d_k = d_{k+1} - d_k = a_k,$$

because then $\sum a_k = \sum (d_{k+1} - d_k)$ telescopes; that is,

$$\sum_{a \leqslant k < b} (d_{k+1} - d_k) = d_b - d_a.$$

Finding such a $\{d_n\}$ is the process of *indefinite summation*, a discrete analogue to indefinite integration. There are algorithms to automatically handle the definite and indefinite summation of wide classes of sequences. Here we shall discuss the indefinite case and compare implementations of these algorithms in Maple and SymPy.

5.1 Definitions and preliminaries

We will assume that everyone is familiar with basic summation notation, sequences, and so on. We mostly use the summation notation from *Concrete Mathematics* [GKP88]. That is, we will write

$$\sum_{P(k)} f(k)$$

to mean the sum of $f(k)$ over all integers k satisfying some property P. We will generally assume that only finitely many integers satisfy P, but some results can be extended to handle infinite sums as well.

The basic problem in automatic summation this: Can we "automatically" evaluate sums of the form

$$\sum_{a \leqslant k < b} f(k)$$

for arbitrary functions $f(k)$ and limits a and b[1]? The answer is a resounding yes, given a suitable definition of "arbitrary." Namely, we need to have the summand be "nice" in a certain way which we shall describe later. Before we get there we must discuss some preliminaries.

Definition 9. Given a sequence $\{f(n)\}$, the *forward difference* of f is the new sequence

$$\Delta f(n) = f(n+1) - f(n).$$

If $F(n)$ is such that $\Delta F(n) = f(n)$, then we call $F(n)$ an *antidifference* of $f(n)$.

The forward difference is like a "discrete" difference quotient. That is, it is like the quotient

$$\frac{f(n+h) - f(n)}{h},$$

where $h = 1$ is the smallest we can go.

The key property of forward differences is that summing over them yields a telescoping sum which is trivial to evaluate. As an example, any two antidifferences differ by a constant in n, because summing over $\Delta F = \Delta G$ from 1 to $n-1$ yields $F(n) - G(n) = F(1) - G(1)$.

This telescoping property is what makes antidifferences crucial. One way to evaluate the sum $\sum_{a \leqslant k < b} f(k)$ is to find an antidifference of $f(k)$. Indeed, if $f = \Delta F$, then

$$\sum_{a \leqslant k < b} f(k) = \sum_{a \leqslant k < b} \Delta F(k) = F(b) - F(a). \tag{5.1}$$

The process of finding the antidifference of a given sequence is called *indefinite summation.*

Once a suitable difference is in hand, formidable sums can be handled quite mechanically. For instance, it is routine to check the identity

$$\Delta\left(\frac{n(n-2)(n-1)}{3} + \frac{n(n-1)}{2}\right) = n^2.$$

[1] Note that the upper limit of the sum is excluded. This is not important; it is only a convention to make later statements cleaner.

It follows that

$$\sum_{1\leqslant k<n} k^2 = \frac{n(n-2)(n-1)}{3} + \frac{n(n-1)}{2} - 0 \\ = \frac{n(n-1)(2n-1)}{6}, \tag{5.2}$$

which is the familiar evaluation.

The whole point of summation is to evaluate sums in "closed form." Generally, this means something like this: Give an expression equal to $\sum_k f(k)$ which does not include a summation sign. This is a little vague, but fortunately we have a better way forward. By "closed form," we mean *hypergeometric*, defined next.

Definition 10. A term $f(n)$ is *hypergeometric* iff

$$\frac{f(n+1)}{f(n)} = \frac{P(n)}{Q(n)} z^n$$

for specific polynomials P and Q and some value z. A series

$$\sum_{k\geqslant 0} t_k z^k$$

is *hypergeometric* if its coefficients t_n are hypergeometric terms.

The hypergeometric functions and their sums cover almost every function encountered in practice. For example, the following functions are all hypergeometric:

$$n \qquad n! \qquad \frac{n^2-1}{n+2}\frac{n!}{(2n+3)!} \qquad \binom{2n}{n}.$$

Each of the following series is easy to see as hypergeometric:

$$e^z = \sum_{k\geqslant 0} \frac{z^k}{k!}$$

$$\sin z = \sum_{k\geqslant 0} (-1)^k \frac{z^{2k+1}}{(2k+1)!}$$

$$\ln(1+z) = \sum_{k\geqslant 1} (-1)^{k+1} \frac{z^k}{k}$$

So, more or less, most things that we run into are either hypergeometric terms or sums of them.

Indefinite summation is not the only way to evaluate sums. Many summands do not have closed-form antidifferences, yet their *definite* sums are easy to evaluate. For instance, the sum $\sum_{k=0}^{x} \binom{n}{k}$ is difficult (read: impossible) to evaluate in closed form for arbitrary x. Yet $\sum_{k=0}^{n} \binom{n}{k} = 2^n$ is a well-known definite sum. We will not go further here, but we refer the interested reader to [PWZ96], particularly Chapter 6.

5.2 Computational problems

We have already laid out the fundamental problem for consideration. We want to evaluate the sum

$$\sum_{a \leqslant k < b} f(k)$$

by finding an antidifference of f; that is, a function F such that $f = \Delta F$. An application of (5.1) will finish the job.

Finding an antidifference for *hypergeometric* f is a solved problem. The solution is given by *Gosper's algorithm.* If $f(k)$ is a hypergeometric term, then Gosper's algorithm will either output a hypergeometric antidifference, or state that none can exist. In this case, the sum cannot be expressed in "closed form," i.e., as the difference of two hypergeometric terms. We shall give a sketch of Gosper's algorithm, then demonstrate its application to the nice special case of sums of polynomial terms.

In the sketch we omit important details such as existence results, subalgorithms, and so on. Let us agree that when we are tacit on such things, the skeptical reader should consult Chapter 5 of [PWZ96] for a full explanation.

Sketch of Gosper's Algorithm

Given a hypergeometric term t_n, the goal of Gosper's algorithm is to find a hypergeometric term z_n such that

$$z_{n+1} - z_n = t_n. \tag{5.3}$$

The big idea is that this problem can be reduced from finding *hypergeometric* solutions, to rational solutions, and then to *polynomial* solutions of progressively simpler and simpler recurrences.

Gosper's algorithm runs as follows:

1. Since t_n is hypergeometric, $r(n) = t_{n+1}/t_n$ is some rational function of n. Factor $r(n)$ as

   $$r(n) = \frac{a(n)}{b(n)} \frac{c(n+1)}{c(n)}$$

   for some polynomials a, b, and c, subject to the condition that $\gcd(a(n), b(n+h)) = 1$ for all nonnegative integers h. This can be done automatically.

2. Find all polynomial solutions $x(n)$ to the recurrence

   $$a(n)x(n+1) - b(n-1)x(n) = c(n).$$

   This can be done by checking a few initial conditions and then solving a system of linear equations for coefficients. If no polynomial solutions exist, then the sum cannot be expressed as a hypergeometric term plus a constant.

3. The general antidifference z_n is given by

   $$z_n = \frac{b(n-1)}{c(n)} x(n) t_n.$$

Example

Gosper's algorithm is more technical than anything we have considered in detail thus far, so we will give an example of its application in an especially simple case. Earlier, in (5.2), we evaluated $\sum_{k=0}^{n} k^2$ based on indefinite summation. Here, we will show how to do this automatically.

Following Gosper's algorithm, set $t_n = n^2$. Then,

$$r(n) = \frac{t_{n+1}}{t_n} = \frac{(n+1)^2}{n^2}.$$

It it clear that we can take $a(n) = b(n) = 1$, and $c(n) = n^2$. We now look for polynomial solutions to $x(n+1) - x(n) = n^2$. Both sides are polynomials, so they must have the same degree. The leading terms on the left-hand side cancel, meaning that $\deg x = \deg n^2 + 1 = 3$. Thus,

$$x(n) = \alpha n^3 + \beta n^2 + \gamma n + \xi$$

for some undetermined constants α, β, γ, and ξ. Plugging this into our equation for $x(n)$ and equating coefficients gives us the equations

$$\begin{aligned} 3\alpha - 1 &= 1 \\ 3\alpha + 2\beta &= 0 \\ \alpha + \beta + \gamma &= 0. \end{aligned}$$

These have the unique solution $(\alpha, \beta, \gamma) = (1/3, -1/2, 1/6)$, with ξ arbitrary. If we take $\xi = 0$, this gives

$$x(n) = \frac{n^3}{3} - \frac{n^2}{2} + \frac{n}{6} = \frac{n(n-1)(2n-1)}{6}.$$

Therefore, our solution is

$$z_n = \frac{b(n-1)}{c(n)} x(n) t_n = \frac{n(n-1)(2n-1)}{6}.$$

We have "mechanically" discovered the identity

$$\Delta\left(\frac{n(n-1)(2n-1)}{6}\right) = n^2,$$

which yields the evaluation

$$\sum_{0 \leqslant k < n} k^2 = \frac{n(n-1)(2n-1)}{6}$$

that we proved earlier.

5.3 Expressive Complexity analysis

For our complexity analysis we shall compare two bodies of code: SymPy's `concrete` module and Maple's `SumTools` package. The method to analyze the code is identical to the previous chapters. For Python, we concatenate all relevant source code into a single file, then apply the

	vocabulary	length	volume	difficulty	effort
count	41	41	41	41	41
mean	44.20	171.85	1216.02	19.54	164289.43
std	61.17	477.82	4066.32	36.16	932942.16
min	2.00	2.00	2.00	0.50	1.00
25%	14.00	25.00	92.51	5.73	464.14
50%	26.00	54.00	255.00	8.68	2526.49
75%	46.00	143.00	830.45	22.74	19896.48
max	368.00	3056.00	26048.01	229.86	5987521.23

Table 5.1: Summary of Maple expressive complexity statistics for the summation problem.

	vocabulary	length	volume	difficulty	effort
count	52	52	52	52	52
mean	24.62	41.12	221.92	4.16	1934.36
std	24.05	52.76	348.37	3.35	4362.43
min	2.00	2.00	2.00	0.50	1.00
25%	9.00	9.75	30.91	1.50	46.53
50%	15.50	20.50	84.48	3.71	284.56
75%	32.50	49.50	246.17	5.66	1535.47
max	110.00	260.00	1692.03	13.06	22103.44

Table 5.2: Summary of Python expressive complexity statistics for the summation problem.

`radon` program to evaluate the Halstead metrics on the single source file. For Maple, we simply apply the `SoftwareMetrics[HalsteadMetrics]` procedure to the `SumTools` package. As before, both programs run at the procedure level, so every procedure has its Halstead metrics evaluated individually. The results of this operation are fed into a `pandas` dataframe for further analysis and plotting. The statistical summary of this analysis is in Tables 5.1 and 5.2.

As we have seen in the previous two chapters, Python has a superior score in nearly every Halstead metric, though the difference is less pronounced here. As before, we have given graphical evidence in Figures 5.1 to 5.4. The conclusions drawn from those are roughly the same as before.

Here we finally obtain evidence that Python difficulty and length might generally undercut Maple's. Figures 5.1 and 5.2 show that Maple's effort and difficulty, as a function of procedure length, seem to grow much faster than Python's. We suspected this conclusion in the previous two chapters, but refrained from this conclusion since we lacked sufficiently long Python programs. We now have a more complete picture.

5.4 Qualitative discussion

For our qualitative discussion we shall focus on two short programs to compute antidifferences of polynomials. That is, given a polynomial $p(n)$, we will compute a polynomial $x(n)$ such that $\Delta x(n) = p(n)$. Maple and Python programs are given in Listings 5.1 and 5.2. Simple examples of Gosper's algorithm in both systems are presented in Listings 5.3 and 5.4.

What we have gained in interesting quantitative data—slightly longer Python programs—we

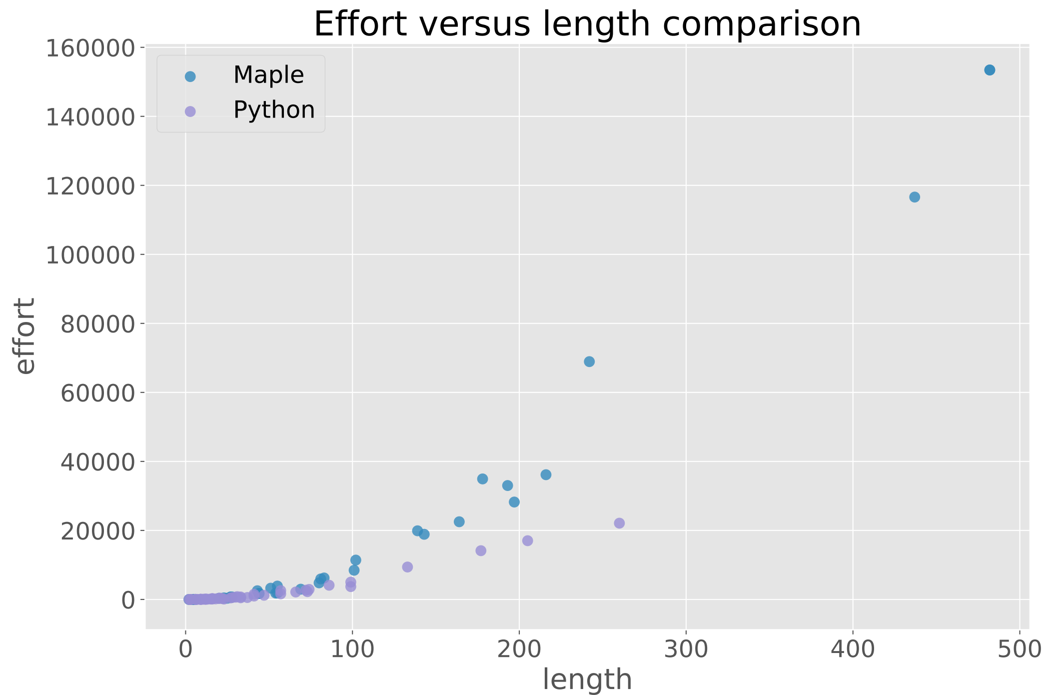


Figure 5.1: Scatterplot comparing effort against length for Maple and Python code to solve the summation problem. Python seems to undercuts Maple in effort at longer length procedures. (An outlier in the Maple dataset has been omitted for clarity. It lies far up and to the right of all displayed points.)

have lost in interesting qualitative data. Our discussion here is brief. The most notable feature of the two implementations is that they are remarkably similar. Despite being a general purpose programming language, Python has not lost much to Maple in this problem.

Since Python puts emphasis on its class system, the SymPy `Poly` class implements various nice features, such as polynomials being callable. That is, given `p = Poly(x**2 + x + 1, x)`, we can write things such as `p(1)` or `p(n + 1)`. This lets us avoid the function call `Translate(p, x, 1)` in the Maple solution.

One feature of Maple is that its variables are symbolic by default. Thus, while SymPy must declare the symbolic variables `c[k]` with `symbols(...)`, the Maple solution can merely use the variables `c[k]` without defining anything. This could be viewed as a strength for interactive use, but a weakness for writing programs, where clarity is more important than convenience.

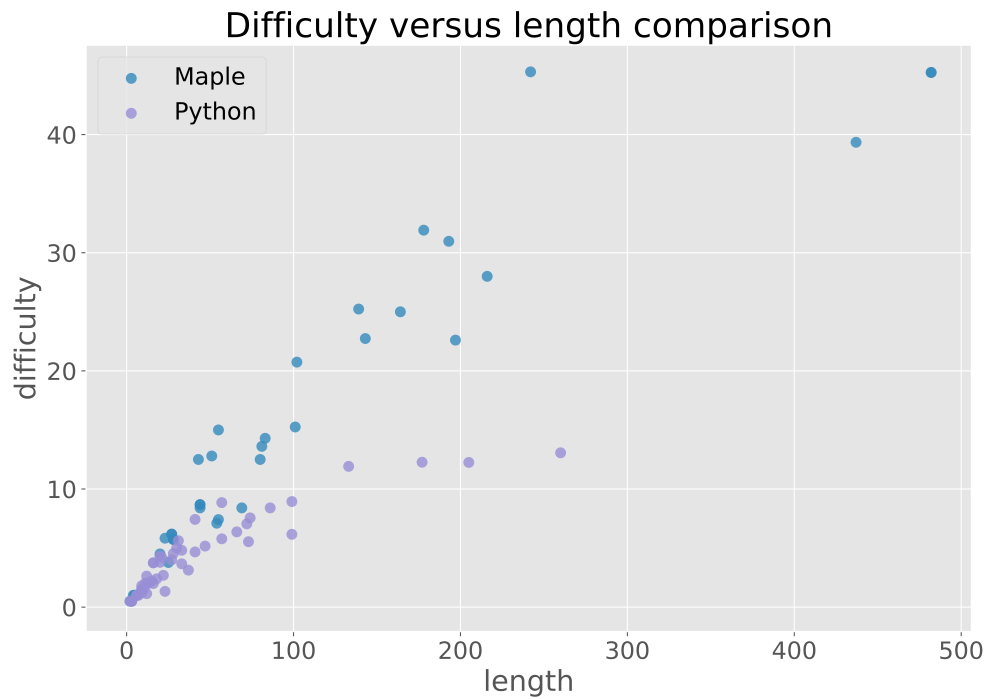


Figure 5.2: Scatterplot comparing difficulty against length for Maple and Python code to solve the summation problem. Python and Maple begin on a similar trajectory, but Python then sharply undercuts Maple for longer lengths.

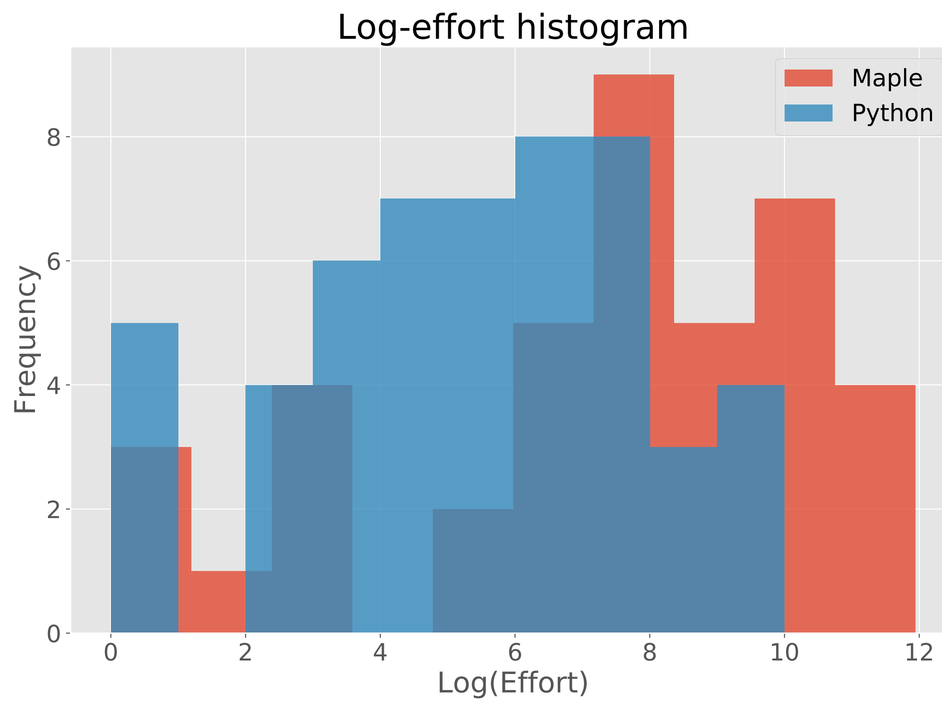


Figure 5.3: Histogram of log-effort comparing Python and Maple. Differences in this plot show differences in orders of magnitude.

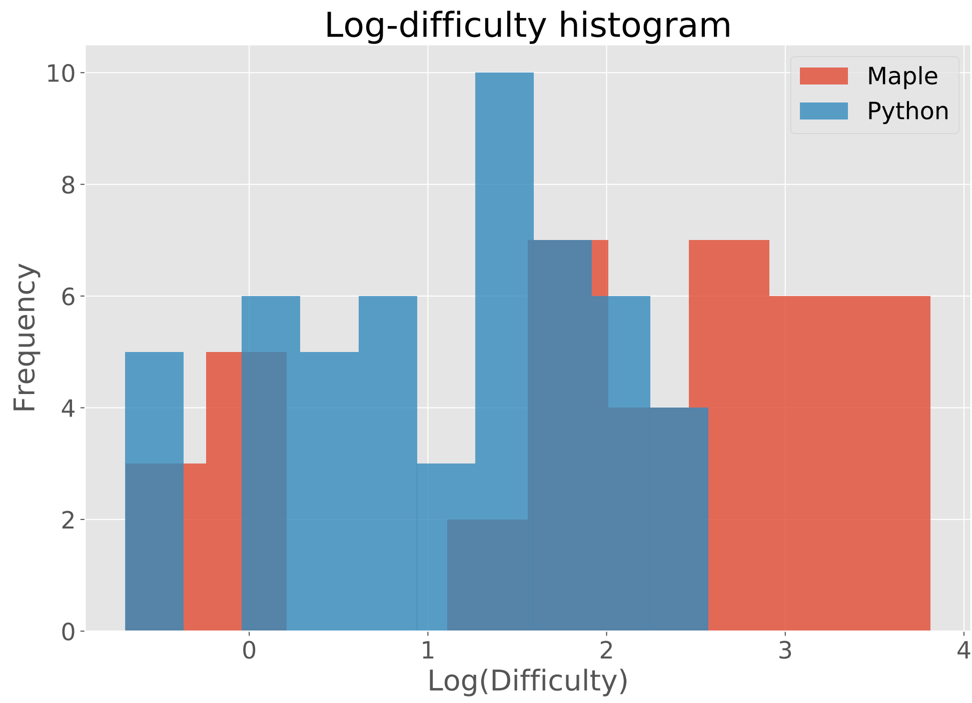


Figure 5.4: Histogram of log-difficulty comparing Python and Maple. Differences in this plot show differences in orders of magnitude.

5.5 Source code listings

```
import sympy as sp
from sympy import Dummy
from sympy.abc import n


def gosper_polynomial(p):
    p = sp.poly(p)
    p_var = p.gens[0]
    x_degree = p.degree() + 1
    cs = sp.symbols("c0:{}".format(x_degree + 1))

    x = sp.Poly(reversed(cs), p_var)

    n = Dummy("n")
    soln = sp.solve(x(n + 1) - x(n) - p(n), cs)

    return x.as_expr().subs(soln)


# Prints: n * (n - 1) * (2n - 1) / 6.
print(gosper_polynomial(n**2).subs("c0", 0).factor())

# Prints: n**9/9 - n**8/2 + 2*n**7/3 - 7*n**5/15 + 2*n**3/9 - n/30
print(gosper_polynomial(n**8).subs("c0", 0))
```

Listing 5.1: Finding antidifferences of polynomials with SymPy.

```
with(PolynomialTools):


gosper_polynomial := proc(p :: polynom)
    p_var := indets(p)[1]:
    x_degree := degree(p) + 1:

    vars := {seq(c[k], k=0..x_degree)}:
    x_poly := add(c[k] * p_var^k, k=0..x_degree):

    diff := Translate(x_poly, p_var, 1) - x_poly - p:
    eqns := CoefficientList(diff, p_var):

    soln := solve(eqns, vars):

    return subs(soln, x_poly);
end proc:


# Prints: n (2n - 1) (n - 1) / 6.
print(factor(subs(c[0] = 0, gosper_polynomial(n^2))));

# Prints: 1/9 n^9 - 1/2 n^8 + 2/3 n^7 - 7/15 n^5 + 2/9 n^3 - 1/30 n/30
print(subs(c[0]=0, gosper_polynomial(n^8)));
```

Listing 5.2: Finding antidifferences of polylnomials with Maple.

```
> with(sumtools):
> sum(k^2, k=0..n-1);
                         3         2
                    1/3 n  - 1/2 n  + 1/6 n

> gosper(k^2, k);
                      (k - 1) k (2 k - 1)
                      -------------------
                               6

> gosper(k^2, k=0..n-1);
                      (n - 1) n (2 n - 1)
                      -------------------
                               6
```

Listing 5.3: Checking Equations (4.2) to (4.4) with Maple's `sumtools` package.

```
1  >>> import sympy as sp
2  >>> import sympy.concrete.gosper as gosper
3  >>> from sympy.abc import k, n
4
5  >>> sp.summation(k**2, (k, 0, n - 1)).factor()
6  n*(n - 1)*(2*n - 1)/6
7
8  >>> gosper.gosper_sum(k**2, k)
9  k*(k - 1)*(2*k - 1)/6
10
11 >>> gosper.gosper_sum(k**2, (k, 0, n - 1))
12 n*(n - 1)*(2*n - 1)/6
```

Listing 5.4: Checking Equation (5.2) with SymPy's `concrete` module.

Chapter 6

Conclusion

Un auteur ne nuit jamais tant à ses lecteurs que quand il dissimule une difficulté.
(An author never does more damage to his readers than when he hides a difficulty.)

Évariste Galois, *Manuscrits de Évariste Galois* [Gal08].

Like a carpenter with a favorite hammer, every programmer has a preferred language. Sometimes this preference is quite nebulous—nothing more than a gut feeling—other times there are arguments to back it up. These arguments often focus on how the preferred language solves such and such problem elegantly, while other languages require clunky, heavy-handed solutions. It may seem plausible to attribute this preference to personal tastes. After all, with dozens of languages to suit every possible fancy, who are we to say what is truly better? This relativism is a mistake. At some point we must call a fool the carpenter who uses a screwdriver on nails. That is the spirit of what we have attempted here.

We have provided a series of quantitative measurements using the Halstead metrics to demonstrate that one language—Python—is simpler than another—Maple—in the area of computer algebra. Though Python seems to outperform Maple, the quantitative difference is not always significant. The qualitative difference seems generally striking, but could be viewed differently by different readers. That is, the results here must be considered *suggestive* rather than *prescriptive*. This should be viewed as the starting point of a more serious investigation, not the end of one.

What should a more serious investigation look like? There are two major improvements in our methodology that would strengthen our conclusions: (1) A larger sample size, and (2) reduced measurement ambiguity.

We examined three nontrivial problems in computer algebra. This gave us interesting things to talk about, but a small body of code to compare. Once Halstead metric computation is automated, there is no reason to limit ourselves in such a fashion. An obvious next step would be to gather and compare enormous corpora of code in both languages. Both Maple and Python are immensely popular in their areas of specialty. There is an abundance of freely available and easily accessible code to analyze.

We gathered the Halstead metrics with two language specific tools. These were Python's `radon` library and Maple's `SoftwareMetrics` package. Only `radon` was truly open for examination, so it

was impossible to determine how exactly Maple was computing the Halstead metrics. It is possible that the two libraries were computing slightly different metrics. In fact, both libraries make questionable choices were computing the metrics, but it is likely infeasible to patch `SoftwareMetrics`. A superior option would be to create a language-agnostic tool to compute the Halstead metrics.

Despite these problems, this technique of evaluating programming languages remains promising. Programming language design is driven by a desire to expend less energy. We have not marched from assembly language to the present out of mere curiosity. It is easy to declare our current languages superior in comparison to the primitive languages of the past; it is harder to look at modern languages and make the same conclusion. In any event, we can surely do better than shrug and give up. We have just made the first stab for mathematicians.

Bibliography

[AA05] Rafa E Al Qutaish and Alain Abran. "An Analysis of the Design and Definitions of Halstead's Metrics". In: *15th Int. Workshop on Software Measurement (IWSM'2005). Shaker-Verlag*. 2005, pp. 337–352.

[Boy78] David W Boyd. "Pisot and Salem numbers in intervals of the real line". In: *Mathematics of Computation* 32.144 (1978), pp. 1244–1260.

[Bus98] Samuel R Buss. "An Introduction to Proof Theory". In: *Handbook of Proof Theory* 137 (1998), pp. 1–78.

[Coo71] Stephen A Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the third annual ACM symposium on Theory of computing*. ACM. 1971, pp. 151–158.

[Cor+09] Thomas H Cormen et al. *Introduction to Algorithms*. MIT Press, 2009.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. "A Machine Program for Theorem-Proving". In: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[Dou19] Robert Dougherty-Bliss. *halstead*. https://github.com/rwbogl/halstead. 2019.

[DP60] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *Journal of the ACM (JACM)* 7.3 (1960), pp. 201–215.

[ESZ16] Shalosh B Ekhad, NJA Sloane, and Doron Zeilberger. "Automated Proof (or Disproof) of Linear Recurrences Satisfied by Pisot Sequences". In: *arXiv preprint arXiv:1609.05570* (2016).

[Fit12] Melvin Fitting. *First-order logic and automated theorem proving*. Springer Science & Business Media, 2012.

[Gal08] Évariste Galois. *Manuscrits de Évariste Galois*. Gauthier–Villars, 1908.

[GKP88] Ronald Graham, Donald Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 1988.

[Hal77] Maurice H Halstead. *Elements of Software Science*. Elsevier North-Holland, 1977.

[Han97a] Peter A. Hancock. *Essays on the Future of Human-Machine Systems*. Human Factors Research Laboratory, University of Minnesota, 1997.

[Han97b] Peter A. Hancock. "On the Future of Work". In: *Ergonomics in design* 5.4 (1997), pp. 25–29.

[Jon03] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[KP11] Manuel Kauers and Peter Paule. *The Concrete Tetrahedron: Symbolic Sums, Recurrence Equations, Generating Functions, Asymptotic Estimates*. Springer Science & Business Media, 2011.

[Lac18] Michele Lacchia. *radon*. https://github.com/rubik/radon. 2018.

[Lam94] Leslie Lamport. *LaTeX: A Document Preparation System: User's Guide and Reference Manual*. Addison-wesley, 1994.

[Lia18] Jia Hui Liang. "Machine Learning for SAT Solvers". PhD thesis. University of Waterloo, Dec. 2018.

[Mac83] Thomas J MacCabe. *Structured Testing*. 6. IEEE Computer Society Press, 1983.

[Map11] Maplesoft. *Overview of the SoftwareMetrics Package*. Accessed 2018-04-01. 2011. URL: https://www.maplesoft.com/support/help/Maple/view.aspx?path=SoftwareMetrics.

[Pei85] Charles Sanders Peirce. "On the Algebra of Logic: A Contribution to the Philosophy of Notation". In: *American journal of mathematics* 7.2 (1885), pp. 180–196.

[Pis38] Charles Pisot. "La répartition modulo 1 et nombres algébriques." French. In: *Ann. Sc. Norm. Super. Pisa, II. Ser.* 7 (1938), pp. 205–248.

[PSW08] Raja Parasuraman, Thomas B Sheridan, and Christopher D Wickens. "Situation Awareness, Mental Workload, and Trust in Automation: Viable, Empirically Supported Cognitive Engineering Constructs". In: *Journal of Cognitive Engineering and Decision Making* 2.2 (2008), pp. 140–160.

[PWZ96] Marko Petkovsek, Herbert S Wilf, and Doron Zeilberger. $A = B$. A K. Peters, 1996.

[Sal12] Gavriel Salvendy. *Handbook of Human Factors and Ergonomics*. John Wiley & Sons, 2012.

[Whi17] Alfred North Whitehead. *An Introduction to Mathematics*. Courier Dover Publications, 2017.

[Zei16] Doron Zeilberger. *Opinion 72: The Next Term in the Sequence: [Dog, Human, Mathematician, ...] is "Computer-Programmer for Computer-Generated Mathematics"*. Accessed 2019-02-05. 2016. URL: http://sites.math.rutgers.edu/~zeilberg/Opinion72.html.

[ZS00] Hantao Zhang and Mark Stickel. "Implementing the Davis–Putnam Method". In: *Journal of Automated Reasoning* 24.1-2 (2000), pp. 277–296.