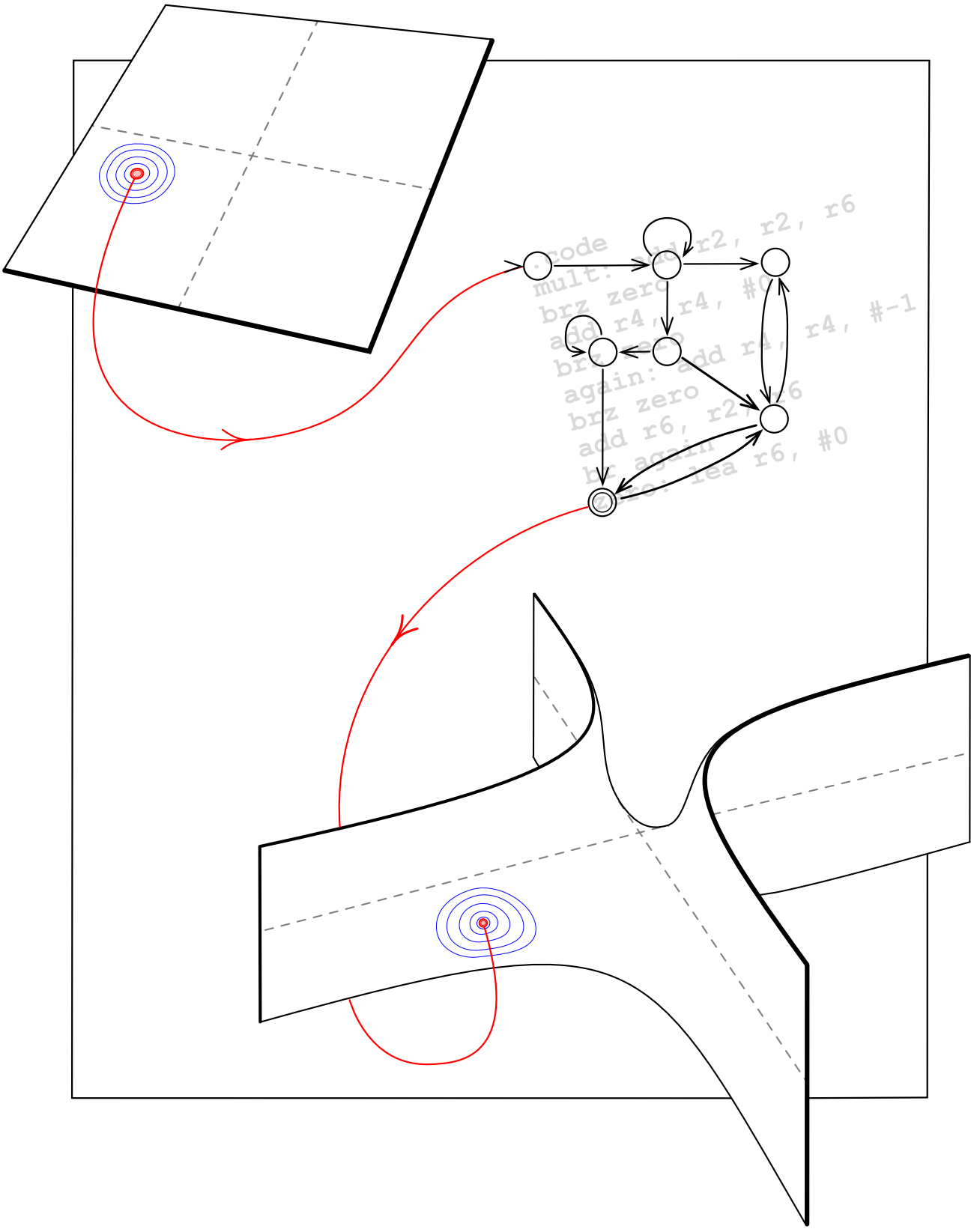
Complexity in the Real world

Chris Heunen



Abstract

Whereas Turing Machines lay a solid foundation for computation of functions on countable sets, a lot of real-world calculations require real numbers. The question arises naturally whether there is a satisfying extension to functions on uncountable sets. This thesis states and discusses such a generalization, based on previous research. It also discusses higher order functions, e.g. differentiation. In contrast to preceding works, however, the focus is on complexity – after computability, of course. By giving a different perspective on Weihrauch's excellent definition of computability in the uncountable case, we show that this theory indeed admits a useful notion of complexity. Various examples are given to demonstrate the theory, including an application to distributions, also called generalized functions, as a form of 'stress-test'.

The cover image is an illustration of multiplication:
In the upper-left plane ($\mathbb{R}^2$), a point (x,y) is input into the upper-right Turing Machine by approximation. The Turing Machine then outputs approximations to $x \cdot y$, in the lower surf ace, which is precisely the graph of the function multiplication $\{(x,y,z) \in \mathbb{R}^3 \ : \ z = x \cdot y\}$.

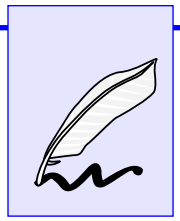
Complexity in the Real world

Chris Heunen

February 2005

Master of Science thesis no. 536
Supervised by dr. H. Geuvers and dr. D.C. van Leijenhorst

Computing Science Institute
Radboud University Nijmegen
The Netherlands



Voorwoord

Deze scriptie gaat over *complexiteit van reële functies*. Grof gezegd komt dat neer op het bekijken hoe moeilijk een gegeven opdracht is om uit te rekenen, en óf het überhaupt wel kan. Laat ik eerst maar eens uitleggen wat alle begrippen inhouden. Het lijkt misschien kinderlijk, maar zo worden de hoofdzaken tenminste zeker duidelijk, en de verschillen daartussen.

Een *functie* is iets, dat op een magische manier dingen van andere dingen maakt. Je kunt je een functie ongeveer voorstellen als een fabriekje: als je er iets in stopt, komt er ook weer iets uit. Dezelfde invoer levert wel steeds dezelfde uitvoer: van rode verf maakt het fabriekje bijvoorbeeld altijd blauwe, maar van blauwe verf weer groene. En, heel belangrijk, *je weet niet hoe het fabriekje dat doet*; je weet alleen dat als je rood invoert, er blauw terug komt. Meestal bekijk je functies over wat abstractere dingen. Bijvoorbeeld *natuurlijke getallen* – dat zijn de getallen 1, 2, 3, en zo verder (en dus niet 0, -1, of $\frac{1}{2}$, en al helemaal geen $\sqrt{2}$ of zulke enge dingen[1]). Een functie van natuurlijke getallen naar natuurlijke getallen is dus een soort rekenmachientje, dat bij invoer n bijvoorbeeld $n+3$ oplevert. En net als bij een rekenmachientje, weet je als gebruiker niet *hoe* het antwoord precies wordt uitgerekend.

Om precies te maken hoe die berekening nu wordt gedaan, hebben we het begrip *Turing machine*. Een Turing machine is een denkbeeldig machientje, met een oneindige rol tape – ongeveer zoals een cassettebandje dat nooit ophoudt – en daarop een lees/schrijfkop. (Omdat oneindige tapes natuurlijk niet bestaan, 'bestaat' een Turing machine ook niet 'echt'. Maar wel dingen die er heel veel op lijken, zoals onze computers.) Verder is een Turing machine altijd in een bepaalde toestand, ongeveer zoals mensen boos, blij, verdrietig of bang kunnen zijn. Maar in tegenstelling tot mensen kan een Turing machine alleen heel simpele acties uitvoeren: afhankelijk van wat de kop leest en in welke toestand het machientje is, kan het naar een andere toestand gaan en de kop één hokje verplaatsen of iets schrijven. Te allen tijde 'ziet' het machientje dus maar één hokje van die hele, oneindige, tape. In appendix A vind je een uitgebreidere beschrijving, met voorbeeld, van een Turing machine.

Bekijk bijvoorbeeld die functie van straks, $f(n) = n+3$. Een Turing machine voor f krijgt dus als invoer n streepjes op de invoer-tape, en dient als uitvoer $n+3$ streepjes te leveren. Een Turing machine die dat doet is dus bijvoorbeeld de machine die achteraan de invoer gaat staan met z'n kop, dan drie streepjes schrijft, en tenslotte stopt. (Maar er zijn natuurlijk meer manieren om f uit te rekenen.) Hiermee is het probleem van *berekenbaarheid* al opgelost, want er is een beroemde stelling die zegt dat alles wat redelijkerwijs (door mensen) uitgerekend kan worden, ook door een Turing machine uitgerekend kan worden[2].

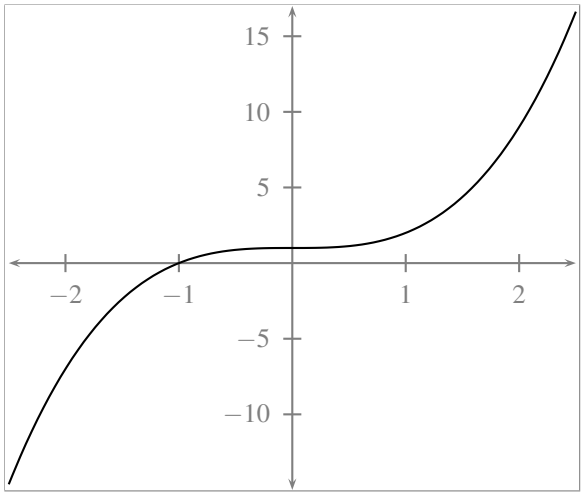
Dan is er nog het probleem van *complexiteit*: hoe moeilijk is die berekening dan? Wel, als we dus een Turing machine hebben die iets uitrekent, kunnen we gewoon het aantal stapjes tellen dat die machine nodig heeft! Zo zie je ook meteen dat de complexiteit niet van de gegeven functie zelf afhangt, maar van de manier waarop je hem uitrekent: een machine kan best 'dom' te werk gaan en zo stapjes verspillen. De f van straks wordt bijvoorbeeld ook berekend door een machine die vier streepjes zet en er weer eentje weghaalt.

[1] Of $3+4i$, voor degenen die dat wat zegt.

[2] Nou ja, het is eigenlijk maar een vermoeden, dat filosofisch is van aard en dus niet formeel bewezen kan worden. Maar sinds 1935 is er nog niemand geweest die er iets op aan te merken had: praktisch mag je er wel vanuit gaan dat de stelling klopt.

Goed, dat is het klassieke geval, daarbij weten we al hoe het ineen steekt. Maar deze scriptie ging dus over reële functies. Dat zijn functies, niet over natuurlijke getallen, maar over *reële getallen*. Reële getallen zijn 'alle getallen': 1, 2 en 3, maar ook 0, -1, $\frac{1}{2}$ en $\sqrt{2}$ et cetera[3]. Ze heten reëel, omdat onze intuitie over tijd en ruimte er zo over denkt: op een willekeurige rechte lijn vinden we dat elk punt een getal voorstelt, hoe dicht het ook bij een ander punt ligt. Elk reëel getal kun je als kommagetal schrijven, zoals je hebt geleerd op de basisschool, maar het stuk achter de komma hoeft niet per se op te houden – denk aan $\frac{1}{3} = 0,333333\cdots$. Je ziet dus al meteen dat er veel meer reële getallen zijn dan natuurlijke.

Een *reële functie* kun je je dus voorstellen als een grafiek: bij elk punt van de horizontale as als invoer, maakt het fabriekje als uitvoer het punt wat erboven ligt, afgelezen op de verticale as. Dit is bijvoorbeeld de grafiek van de functie $f(x) = x^3 + 1$:

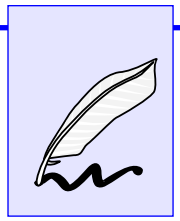


Willen we van zo'n functie nu bekijken hoe we hem precies berekenen en 'hoe duur' die berekening dan is, dan zitten we meteen in de problemen – tenminste, als we nog steeds Turing machines willen gebruiken. Immers, ten eerste moeten we een reëel getal op de tape zetten als invoer. Nu, dat kan nog wel als kommagetal, want de tape was toch oneindig lang. Maar de *uitvoer is ook oneindig*, dus het machientje stopt sowieso nooit! Hoe kunnen we dan ooit stapjes gaan tellen?

Over dat probleem gaat deze scriptie dus. We geven gepaste uitbreidingen van de begrippen 'berekenbaar' en 'Turing machine', en zullen zien dat die ook uitstekend geschikt zijn om complexiteitsvraagstukken mee aan te pakken. Om dat te demonstreren, geven we diverse toepassingen en voorbeelden. Eén daarvan is bijvoorbeeld het oprekken van het begrip 'functie', een ander is om de afspraak te veranderen hoe we een reëel getal op de tape zetten; maakt het wat uit als we dat op andere manieren doen dan als kommagetal?

Waar dat goed voor is? Wel, de huidige computers rekenen niet met 'echte' reële getallen, maar slechts met benaderingen daarvan. En dat betekent dat ze met afrondfouten te maken hebben. Onze computers zijn sowieso maar benaderingen van een 'echte' Turing machine (ze hebben immers maar eindig veel geheugen). Als we nu een model hadden voor hoe je 'echt' met reële getallen om zou moeten gaan, konden we er op zijn minst voor zorgen dat onze computers zich daaraan houden, net zoals de huidige computers zoveel mogelijk op een echte Turing machine proberen te lijken. En, zoals altijd, is het gemakkelijker werken met dingen die goed in elkaar zitten. Je moet zogezegd geen hamer op een schroef gebruiken.

[3] Maar nog steeds niet $3+4i$; en dat wist je ook al, als je weet had van $3+4i$.



Preface

This thesis deals with *complexity of real functions*. That amounts roughly to investigating how hard a given problem is to calculate, and whether that is possible in the first place. Let me first explain all the notions involved. It might seem childish, but at least the main points and their differences will become clear this way.

A *function* is something that magically produces things from other things. You could envision a function like a small factory: if you put something in, then something will come out again. Equal input will yield equal output every time: red paint gets processed into, say, blue paint by our factory, but blue transforms into green. Most importantly, *you don't know how the factory works*; all you know is that red input yields blue output. Mostly, one considers functions on somewhat more abstract things. For example *natural numbers* – those are the numbers 1, 2, 3 and so on (and not 0, -1, or $\frac{1}{2}$, and most certainly not something scary like $\sqrt{2}$.[4]). So, a function from natural numbers to natural numbers is like a simple calculator. When given input n, it could for example output $n+3$. And just like with a desk calculator, you don't know exactly *how* the answer is calculated as user.

To specify more precisely how those calculations are performed, there is the notion of a *Turing machine*. A Turing machine is a hypothetical machine, with an infinite roll of tape – like a cassette tape that never ends – and a read/write-head on it. (Since infinite tapes don't really exist, a Turing machine doesn't 'really exist'. But there are things that resemble it very much, like our computers.) Furthermore, a Turing machine is always in a certain state, just like people can be angry, happy, sad or afraid. But unlike people, a Turing machine can only perform very simple actions: depending on what the head reads and in what state the machine is, it can change state and either move the head one cell, or write something. At all times, the machine can only 'see' one cell of that infinite tape.

For example, consider that function we had earlier, $f(n) = n+3$. A Turing machine for f gets n dashes on the input-tape, and ought to output $n+3$ dashes. An example of a Turing machine which does just that is one that moves its head to the end of the input, write three dashes, and then stops. (But there are of course more ways to compute f.) This already solves the problem of *computability*, because there is a famous theorem saying that everything that can reasonably be computed (at all, i.e. by humans) can be computed by a Turing machine[5].

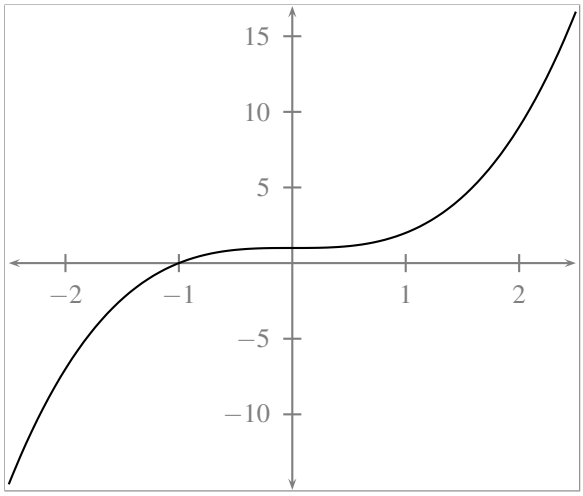
And then there is the problem of *complexity*: how hard is that computation? Well, if we have a Turing machine that computes something, we can simply count the number of steps it needs! So we immediately see that complexity does not depend on the given function itself, but rather on the way in which you compute it: a machine could act 'stupidly' by wasting steps. Our f of just now is also computed by a machine that puts four dashes and then erases one again.

[4] Nor $3+4i$, for those in the know.

[5] Well, actually, this is only a conjecture of philosophical nature, which thus cannot be proven formally. But there has been no-one since 1935 objecting it: practically, we can safely assume it is correct.

All right, that is the classical case, in which we already know how it fits together. But this thesis was about real functions. Those are functions on real numbers instead of on natural numbers. Real numbers are 'all numbers': 1, 2, and 3, but also 0, -1, $\frac{1}{2}$ and $\sqrt{2}$ et cetera[6]. They are called real because our intuitions about time and space regard them so: on an arbitrary line, we feel that every point represents a number, however close to another point. Every real number can be written as a decimal fraction, like you learned in primary school, but the part behind the decimal point need not necessarily end – think of $\frac{1}{3} = 0.333333\cdots$. You immediately see that there are much more real numbers than natural ones.

You can imagine a *real function* as a graph: given any point of the horizontal axis as input, our little factory produces the point right above it, read off on the vertical axis. For example, this is the graph of the function $f(x) = x^3 + 1$:

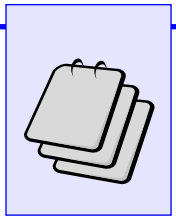


If we want to know how to compute such a function precisely, and 'how much' that computation 'costs', we are immediately in a scrape – at least, if we still want to use Turing machines. After all, we first have to put a real number on tape as input. Well, we can do that as a decimal fraction, because the tape was infinitely long anyway. But *the output is also infinite*, so the machine will never halt! How can we ever count steps?

That is the problem this thesis discusses. We give a befitting extension of the notions 'computable' and 'Turing machine', and we will see that those are perfectly well suited for handling complexity with. To demonstrate that fact, we give various applications and extensions. One of those is to stretch the notion 'function', and another is to change the agreement how we put a real number on tape; does it matter if we do this in other ways than as a decimal fraction?

What's the use? Well, current computers do not compute with 'real' real numbers, but only with approximations of those. And that means they encounter roundoff-errors. Our computers are only approximations of a 'real' Turing machine anyhow (their memory is finite). If we had a model telling us how to 'really' handle real numbers, we could at least ensure that our computers obey that, just like current computers try to resemble a real Turing machine as closely as possible. As always, it is easier to use things well-designed for the task at hand. You should not use a hammer to drive in a screw, so to speak.

[6] But still not $3 + 4i$; and you already knew that if you were in the know of its existence.



Contents

CHAPTER

1 Prologue

The scientific method has oozed into the very fabric of our society. Almost everything we use in our daily lives has been calculated through and through: how fast our cars should go, how much power the light bulb consumes, what frequency our telephone calls use, how much weight an elevator can take, how much wind our houses can withstand, but also how much our insurance premium is, how much our bread costs, or the mixture of our medicine. Engineers, medics and statisticians develop elaborate models to calculate numerous properties to ensure that their designs are safe, viable or profitable. In short:

Everybody uses scientific calculation. ***(I)***

Furthermore, one of the most fundamental cornerstones of modern science is the real number, especially in the calculus of differentiation and integration, widely spread among engineers, biologists and the like. We can safely say:

The real numbers are fundamental to scientific calculation. ***(II)***

All these calculations are practicable because of the vast computational power available to us. Whereas it was not abnormal to spend most of one's life to calculate 15 digits of π in the 16^{th} century, nowadays nobody would even seriously consider waiting more than a year for a simple, mechanical, calculation for practical use. We state:

Scientific calculation is performed on computers. ***(III)***

Now, the following consequence of (I), (II) and (III) is the justification for the research this thesis hopes to contribute to:

Therefore, it is in everybody's interest that computers handle real numbers well.

What's wrong with the current situation? 1.1

As the previous conclusion suggests, in the current situation *computers don't handle real numbers well*. Since a computer only has a finite memory, it can only use a finite set of numbers to calculate with. Sure, we can choose how much precision we want. But as you can clearly see in figure 1.1, the real numbers (the paper) are incredibly more dense than these so-called floating point numbers (the dots), no matter how big the chosen precision.

The following ghastly example by (Muller, 1989) illustrates just how awful the situation can get when using floating point numbers instead of 'real' real numbers. Consider the numbers $a_1, a_2, \ldots$, defined by

$$a_0 = \frac{11}{2}, \quad a_1 = \frac{61}{11}, \quad a_{n+1} = 111 - \frac{1130 - \frac{3000}{a_{n-1}}}{a_n}$$

The result when calculating these a_n using floating point numbers is listed in the following table.

n	2	5	6	7	8	10	11	12
a_n	5.6	5.6	4.3	-29.0	125.7	100.1	100.0	100.0

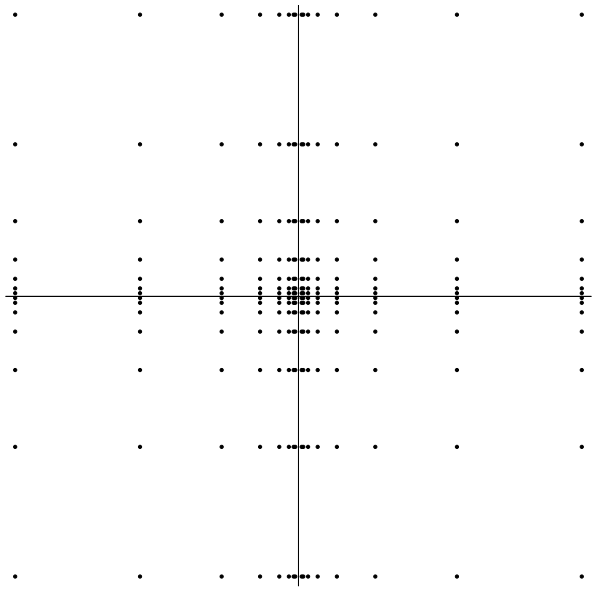


Figure 1.1: All pairs of floating-point numbers in base 2, and with mantissa and exponents of 2 digits. A typical set of computer floating-point numbers has mantissae ≤ 64 and exponents ≤ 14, like the set $\{m \cdot 2^{e-64} \mid m,e \in \mathbb{Z}, -2^{64} < m < 2^{64}, (2^{14}-1) < e < 2^{14}\}$

Using floating point, the sequence converges to 100.0. But we can easily prove that

$$a_n = \frac{6^{n+1}+5^{n+1}}{6^n+5^n}$$

from which we immediately see that the actual limit is 6! The result 100.0 isn't even close to the actual limit. The actual values of a_n, rounded to 1 decimal, are

n	2	5	6	7	8	10	11	12
a_n	5.6	5.7	5.7	5.8	5.8	5.9	5.9	5.9

So, for $n > 5$, the floating point approximation is completely wrong, even while calculating a_n takes very few operations ($2n$ divisions and $2n-2$ subtractions). But is gets worse. Even higher precision floating point arithmetic is not good enough:

precision	a_{10}	a_{20}	a_{30}	a_{40}	a_{50}
10	147.26	100	100	100	100
20	5.86	99.7	100	100	100
30	5.86	5.97	100	100	100
40	5.86	5.97	5.994	100	100
50	5.86	5.97	5.996	5.92	100

No matter how high we fix the precision, we will never be able to even approximate $\lim_{n\to\infty} a_n$ with floating point arithmetic. We trust that this example efficiently demonstrates the flaws of floating point arithmetic, and thus the need for exact computation.

An anecdote tells that Alston Householder, mathematician, would even be nervous to fly on airplanes since he knew that they are designed using floating point arithmetic.

Alas! As mentioned before, computers are only finite and hence simply cannot handle real numbers well. If we slightly misuse the words of Theodore Postol:

"The real world is complex and sometimes none of the choices we have are good ones."

But that is not the last word. The least we can do is make sure we have a valid theoretical model for computations with real numbers. Just like computers aim to be as much like the hypothetical Turing machine as possible, we could impose an extended model to aim for.

The aim of this thesis 1.2

Turing himself already proposed something like computable real numbers (Turing, 1936). After that, several replacements and improvements have been suggested, but there has been no end-all, like the Turing machine showed to be for discrete functions. In this thesis, we state and study such a proposal, mainly based on (Weihrauch, 2000).

Next to the obvious computability, our focus will be on the aspect of computational complexity, which is unusual: mostly, complexity in this setting is seen as a by-product of computability, and not studied much deeper. We feel complexity is a fine 'measure' of 'good' definitions in this context. We will mostly focus on time-complexity – other variants are anologous.

After stating the definitions, we derive various properties, work out some examples, and apply the theory to a generalized notion of function from analysis, called distribution, as a form of 'stress-test'. Throughout all this, the central theme of the similarities and differences between various representations of real numbers plays a large role.We also compare our definitions to a previous suggestion by (Ko, 1991). All this is done in order to show that this theory is viable.

Although a lot of things can be stated vastly more general, we have opted for another approach. Though the first priority is of course clarity, the style of this thesis is less that of a research report than that of an undergraduate math textbook, since we feel a thesis should be accessible to fellow students. Therefore, we usually prefer specific cases over more general ones, trying to convey as much intuition as possible, while not comprising on formality. We try to guide the reader gently over one specific path through the subject, which does not necessarily lead directly to that mountain ridge providing broad overviews, but which avoids crevasses and stays in sight of the summit.

Conventions and notations

Our choice for simplicity has a few consequences. First, we employ the usual nonchalance regarding Turing machines: we let a description suffice, and hardly ever give the actual program (as described in appendix A). From time to time, we will even disregard some details of the implementation, and state our description in a high-level language. The reason for this is that anyone who has ever had a first course covering Turing machines will know how to fill in the details and even find a detailed description a bother, whereas the reader who has never heard of Turing machines before today will have no use for details either.

Furthermore, we carefully avoid language-decisions. Though that is the usual approach to problems with Turing machines, this thesis deals primarily with functions and their computation. Since there is enough to define as is, we feel that a more rigorous approach detracts more than it would clarify. This also means that our P and NP, which are usually sets of languages, are now sets of functions (usually called FP and such).

As for the mathematics, we mostly apply the standard notations. We require little mathematical prerequisites – most are used in examples that can be skipped without losing focus – but some knowledge of basic analysis and topology will definitely benefit the reader. We do expect, however, a fair degree of familiarity with formal notation – but nothing a first or second year student wouldn't know. Yes, the notions employed are sometimes advanced, but we strive to introduce them all in such a way that most students will be able to understand. A notable exception is chapter 4, which necessarily digs deeper, because it contains the main contribution to the research field. Even there, we will try only to give the necessary formal definitions; for example, Weihrauch's notions of $\leq$ and $\leq_t$ are very useful, but we avoid them.

We use the word 'function' throughout this thesis, where sometimes one might expect the more general 'mapping'. (The word function is usually reserved for mappings to real numbers, and a mapping can have any set as its domain or range.) The rationale behind this is again to simplify things.

The one thing that is not standard is our notation of partial functions. With $f : (\subseteq\mathbb{R}) \to \mathbb{R}$, we mean to express that there is a part U of $\mathbb{R}$, on which f is defined, and takes values in $\mathbb{R}$. We write $\mathrm{Dom}(f)$ for U.

By convention we will denote by $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$, the natural numbers (including zero), the integers, the fractions, and the real numbers, respectively.

What is this thing called $\mathbb{R}$? 1.3

As this thesis is about functions from $\mathbb{R}$ to $\mathbb{R}$, let us first consider this $\mathbb{R}$ we are talking about. When dealing with real numbers, we are mostly not concerned with what they actually *are*, but only how to work with them. One person may think of a real number as a point on a line, another as an infinite sequence of digits. It does not matter, as long as both can derive the same properties.

The mathematicians answer to this problem is an *axiomatization*; a list of statements about the real numbers that are acceptable to all of us, whatever idea of a real number we have. We consider a statement about the real numbers true, (only) if it follows from these axioms. If we all agree upon the axioms, then we will all accept (precisely) the same consequences, and that's that.

Of course, we still need to know that *something* actually *exists* that obeys our axioms, or we could be doing all our work in vain. That calls for a *construction* of $\mathbb{R}$. We will meet several such constructions in chapter 3. But first the axioms – we will define three of them[1].

1.1 **Definition (Field)** A field is a triple $(R,+,\cdot)$ of a set R and two functions $+,\cdot : R \to R$, such that:

1. $a+(b+c)=(a+b)+c$ for all $a,b,c \in R$ ($+$ *is associative*)
2. $a+b=b+a$ for all $a,b \in R$ ($+$ *is commutative*)
3. there is an element $0 \in R$ such that $a+0=a$ for all $a \in R$ (*zero*)
4. for every $a \in R$, there is a $-a \in R$ with $a+(-a)=0$ (*opposite*)
5. $a\cdot(b\cdot c)=(a\cdot b)\cdot c$ for all $a,b,c \in R$ ($\cdot$ *is associative*)
6. $a\cdot b=b\cdot a$ for all $a,b \in R$ ($\cdot$ *is commutative*)
7. $a\cdot(b+c)=a\cdot b+a\cdot c$ for all $a,b,c \in R$ ($\cdot$ *distributes over* $+$)
8. there is an element $1 \in R$, $1 \neq 0$, such that $a\cdot 1=a$ for all $a \in R$ (*one*)
9. for every $a \in R$, $a \neq 0$, there is an $a^{-1} \in R$, with $a\cdot a^{-1}=1$ (*inverse*)

1.2 **Definition (Totally ordered field)** A totally ordered field is a tuple $(R,+,\cdot,\leq)$ such that $(R,+,\cdot)$ is a field, and $\leq$ a binary relation on R with

1. $x \leq x$ for all $x \in R$ ($\leq$ *is reflexive*)
2. $x \leq y$ and $y \leq x$ imply $x=y$ for all $x,y \in R$ ($\leq$ *is anti-symmetric*)
3. if $x \leq y$ and $y \leq z$ then also $x \leq z$ for all $x,y,z \in R$ ($\leq$ *is transitive*)
4. $x \leq y \Rightarrow x+z \leq y+z$ for all $x,y,z \in R$ ($\leq$ *respects* $+$)
5. $(x \leq y \text{ and } 0 \leq u) \Rightarrow xu \leq yu$ for all $x,y,u \in R$ ($\leq$ *respects positive* $\cdot$)

That numbers form (totally ordered) fields is something we are all used to; the definitions are precisely the rules of arithmetic we learned in primary school. The relation $x<y$ is defined (only) if $x \leq y$, but $x \neq y$. This means that exactly one of the following three is true: $x<y$, $x=y$ or $x>y$.

Axiom (I): $\mathbb{R}$ is a totally ordered field

The next axiom tells us something about the relation between the real numbers $\mathbb{R}$ and the integers $\mathbb{Z}$: every real number can be 'rounded down'. It was first attributed to Eudoxos by Archimedes, and is therefore often named after them.

Axiom (II): For every $x \in \mathbb{R}$, there is an integer $n \in \mathbb{Z}$ such that $n \leq x < n+1$

Using only axioms (I) and (II), the rational numbers $\mathbb{Q}$ are still in the race; they too obey both axioms. The third axiom is what separates $\mathbb{R}$ from $\mathbb{Q}$. For example, it forces $\sqrt{2}$ to be in $\mathbb{R}$, whereas $\sqrt{2}$ is definitively no fraction. Effectively, axiom (III) forces $\mathbb{R}$ to be *complete*.

[1] There are several equivalent axiomatizations, too. Here, we follow (van Rooij, 1986).

1.3 **Definition (Upper bound, supremum)** Let $S \subset R$ be a non-empty subset of a totally ordered field $(R, +, \cdot, \leq)$. We say that $M \in R$ is an *upper bound* for S, and that S is *bounded above*, if $x \leq M$ for all $x \in S$.

We say that M is the *supremum* of S, notation $M = \sup V$, if M is an upper bound for S, and when M' also is an upper bound for S, then $M \leq M'$. Because of its nature, a supremum is also called a *least upper bound*. (Analogously, we can define a *lower bound* and an *infimum*).

Axiom (III): Every non-empty subset of $\mathbb{R}$ that is bounded above has a supremum.

Axiom (III) is often called the supremum-axiom.

Notice that we have assumed that we *already have* $\mathbb{Z}$ (in the axiom of Archimedes and Eudoxos). We could also have axiomatized or explicitly constructed $\mathbb{Z}$ as well, but that is not of much concern for us.

This is a trend we will follow throughout this thesis: in a certain sense, $\mathbb{N}$, $\mathbb{Z}$ and even $\mathbb{Q}$ are 'easy', whereas $\mathbb{R}$ is not – a priori. For example, natural numbers, integers and fractions can be written down on a finite piece of paper, but real numbers cannot (in general).

CHAPTER

2 Computability

Models of computation 2.1

Classically, there are several conceptual models of computation, although they have all been proven equivalent: in all models, precisely the same functions are computable. For example, λ-calculus, μ-recursiveness, and Turing Machines are all different but equivalent approaches to computability.

This means there are several ways we can try to approach computability in the real case. Perhaps the model of the register-machine with a few standard constructs, μ-recursiveness, is the easiest to generalize. This is the way early generalizations used (Blum et al., 1998). However, we choose for the Turing Machine, because it allows easy definitions of computational complexity which actually have some sort of physical meaning: the time complexity of a Turing machine corresponds to the execution time of a computer program, whereas it is unclear what exactly to count in a λ-calculus-term when asked for its complexity.

The problem 2.2

Fix a finite set Σ and call it the alphabet. With Σ^n we denote the Cartesian product $\{a_1 a_2 \cdots a_n : a_i \in \Sigma\}$ of all words of length n of letters in this alphabet. We define the set of words Σ^* over the alphabet Σ as $\bigcup_{n\in\mathbb{N}} \Sigma^n$, words of any length. Note that every word in this set Σ^* still has finite length!

In the classical theory of computation, a function $f : \Sigma^* \to \Sigma^*$ is computable when there is a Turing machine[1] M that computes it. That is, given a finite input $w \in \Sigma^*$, M calculates for a finite amount of time, and then halts on (finite) output $f(w)$.

This mechanism is great for computation on countable sets, like functions $\mathbb{N} \to \mathbb{N}$. But what if we want to calculate real functions $\mathbb{R} \to \mathbb{R}$? After all, $\mathbb{R}$ is uncountable, so there cannot exist an alphabet Σ with which to represent $\mathbb{R}$, since Σ^* would be only countable.

The above definition does not generalize to this situation; it is essentially a finite notion. For if we put, in some form, a real number $x \in \mathbb{R}$ on the input tape, and try to 'compute' $f(x)$ for a real function $f : \mathbb{R} \to \mathbb{R}$ in this fashion, the output would be infinite. So, the machine will never halt!

A representation of $\mathbb{R}$ 2.3

Let us first specify how we can "put a real number on an input tape". We already saw that we cannot represent $\mathbb{R}$ with finite words over any alphabet. Since we want to stick to finite alphabets, the only option is to allow infinite words. We define the set of infinite words over an alphabet Σ as $\Sigma^\omega = \{a_0 a_1 a_2 \cdots : a_i \in \Sigma\}$.

Now, we could of course take $\Sigma = \{0, 1, 2, \ldots, 9\}$ and represent $x \in \mathbb{R}$ with one of its decimal expansions. But, as we will see in chapter 3, this is not good enough. We cannot go any further into this now, but

[1] We assume the reader is familiar with the concept of a Turing machine. The basics are summarized in appendix A.

one can intuitively see that the decimal expansion representation does not allow for corrections 'to the left' once a decimal has been written down – which must happen after only a finite prefix of the input has been inspected – there is no going back. So what if the next decimal of input tells us that the decimal already written down is wrong?

We will use a representation loosely originating from decimal expansions which does not suffer from this disadvantage. Let $x \in \mathbb{R}$. Because there are fractions arbitrarily close to x, we can make whole sequences of fractions $a_1 \leq a_2 \leq \ldots \in \mathbb{Q}$ and $b_1 \geq b_2 \geq \ldots \in \mathbb{Q}$ such that $a_i < x < b_i$ and $b_i - a_i \leq 2^{-i}$ and $x = \bigcap_{i\in\mathbb{N}}[a_i, b_i]$. So we can represent x with the sequence of segments $([a_1,b_1],[a_2,b_2],\ldots)$. Note that we can actually write this sequence as an infinite word, because every fraction can be represented finitely. We call this the representation of nested intervals, and we will make it more precise in chapter 3.

Just as with decimal expansions, where $\texttt{0.99999}\cdots = \texttt{1.00000}\cdots$, a real number can have multiple nested interval representatives. For example, the familiar real number π can be represented by $([3,4],[3.1,3.2],[3.14,3.15],[3.141,3.142],\ldots)$, but also by $([3,4],[3,3.5],[3,3.25],[3.125,3.25],\ldots)$.

The definition 2.4

Though there is no once and future definition yet, (Weihrauch, 2000) offers a plausible extension of the computability concept to real functions, based on work dating back to Grzegorczyk and Lacombe.

We call a real function $f : (\subseteq\mathbb{R}) \to \mathbb{R}$ computable, if a Turing machine maps any representative of any $x \in \mathrm{Dom}(f)$ to some representative of $f(x)$. As we already saw, the input and output tapes of the Turing machine should hold 'infinite words'.

The output, a representative of $f(x)$, is infinite, thus the machine never halts. So we cannot even be sure that the first k symbols on the output tape are correct, since the machine might still change them. Therefore, we must furthermore demand that the head on the output tape can only move onwards.

So, (Weihrauch, 2000) arrives at a definition '*A function $f : (\subseteq\mathbb{R}) \to \mathbb{R}$ is computable if there is a machine M such that $f(x) = y$, $x \in \mathrm{Dom}(f)$, if and only if M computes forever on any representative of x, writing some representative of y on the output tape*' for 'machines' of which the output head only moves onwards. This enables him to circumvent having to be precise about the number of steps involved. We state our (equivalent) definition a bit more precisely.

2.1 **Definition (Computability of real function with precision k)** Let $f : (\subseteq\mathbb{R}) \to \mathbb{R}$ be a real function, and $k \in \mathbb{N}$ a natural number. We call f computable with precision k, if and only if

- there is a Turing machine M with (one-way) infinite input and output tapes, of which the output tape's head can only move right,
- when given a representative $\bar{x}$ of $x \in \mathrm{Dom}(f)$ on the input tape,
- after a finite number ($N_{\bar{x}}$) of steps of calculation,
- a prefix of M's output tape coincides with that of a representative of $f(x)$ up to precision k (i.e. the corresponding real numbers differ at most 2^{-k}).

The notion of representation, used in this definition, will be made rigorous later. For now, the nested interval representation, loosely defined in the previous section, will suffice.

2.2 **Notation** Just like $N_{\bar{x}}$ in definition 2.1 denotes the number of steps needed, we write $I_{\bar{x}}$ for the number of input precision needed, i.e. the maximum i such that (a_i, b_i) is needed in the computation. Then $I_{\bar{x}}$ is bounded by $N_{\bar{x}}$.

2.3 **Definition (Computable real function)** Let $f : (\subseteq\mathbb{R}) \to \mathbb{R}$ be a real function. We call f computable if and only if there is a Turing Machine M such that for all $k \in \mathbb{N}$, M computes f with precision k.

Intuitively, f is computable by M if M computes it to arbitrary precision. For real functions of n variables, $f : (\subseteq\mathbb{R}^n) \to \mathbb{R}$, we consider a Turing machine with n input tapes. (Which is equivalent to a Turing machine with 1 input tape, see appendix A.)

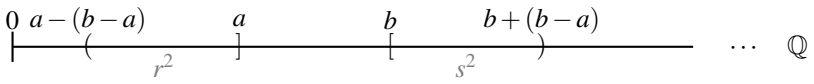


Figure 2.1: Illustration of example 2.5.

2.4 **Example (Multiplication is computable)** The function $\mathbb{R}^2 \to \mathbb{R} : (x,y) \mapsto x \cdot y$ is computable.
PROOF Let representatives $I_1, I_2, \ldots$ of x and $J_1, J_2, \ldots$ of y be given on the input tapes, I_i and J_i segments with rational endpoints. Now define $I \cdot J = \{x \cdot y \mid x \in I, y \in J\}$. There is a Turing Machine that multiplies two fractions by the school method. So there is also a machine that computes $I_1 \cdot J_1, I_2 \cdot J_2, \ldots$ from the input. And that is a representative of $x \cdot y$! □

2.5 **Example (Taking square roots is computable)** The function $\mathbb{R}^+ \to \mathbb{R} : x \mapsto \sqrt{x}$ is computable.
PROOF For every $a, b \in \mathbb{Q}$ with $0 \leq a < b$, there are fractions $r, s \geq 0$ with $a - (b-a) < r^2 \leq a < b \leq s^2 < b + (b-a)$. Hence we can find, for example by an exhaustive search, a rational segment $[r,s]$ such that $[a,b] \subseteq [r^2, s^2]$ and $|r^2 - s^2| < 3|a-b|$. Let us call the computable function thus described f.

Furthermore, there is a computable function g that maps $[a;b]$ to $[\max\{0,a\};b]$. If $(I_1, I_2, \ldots)$ is a representative of $x \geq 0$, then $\bigcap_{n \in \mathbb{N}} (f \circ g)(I_n) = \sqrt{x}$.

Finally, since the function h that maps a sequence of rational segments $I_1, I_2, \ldots$ to a nested sequence of rational segments $I_1, I_1 \cap I_2, I_1 \cap I_2 \cap I_3, \ldots$ is computable, we see that taking square roots is computable. □

In fact, one of our reasons to state definitions 2.1 and 2.3 more precisely than (Weihrauch, 2000) is that they immediately remind a lot of Cauchy's definition of continuity of a real function (and we will shortly see a connection between the two, in theorem 2.13). The order of the quantors therefore begs the question of uniformity. It is unreasonable to demand that $N_{\bar{x}}$ is independent of $\bar{x}$: for the vast majority of computations the length will depend on their input, since the computations themselves depend on their input – other than constant functions. But if we limit the amount of precision needed to something independent of $\bar{x}$, we are left with a useful class of computable functions.

2.6 **Definition (Uniformly computable real function)** Let $f : (\subseteq \mathbb{R}) \to \mathbb{R}$ be computable and $k \in \mathbb{N}$. The real function f is called uniformly computable with precision k if and only if it is computable with precision k, and moreover, $I_{\bar{x}}$ can be bounded independently of $\bar{x}$: "there is one I for all $\bar{x}$".

Likewise, f is called uniformly computable if and only if there is a Turing machine M such that for all $k \in \mathbb{N}$, M uniformly computes f with precision k: "there is one I for every k".[2]

2.7 **Example (Linear functions are uniformly computable)** Let $p, q \in \mathbb{Q}$. Then $f(x) = px + q$ is uniformly computable.
PROOF Let a representative $\bar{x} = ([a_1,b_1],[a_2,b_2],\ldots)$ of x be given on the input tape. Then we can certainly compute $([f(a_1), f(b_1)], [f(a_2), f(b_2)], \ldots)$, a representative of $f(x)$. From $x - a_i \leq 2^{-i}$ we infer that $f(x) - f(a_i) = (px+q) - (pa_i + q) \leq |p| 2^{-i}$. Hence output precision k is reached when computing $f(a_I)$ for $I = \lceil \log_2 |p| \rceil + k$. Since this only depends on p and k, it is in particular independent of $\bar{x}$, and we conclude that f is uniformly computable. □

An example of a computable function that is not uniformly computable is the inversion $f : (0,1) \to \mathbb{R}$ defined by $f(x) = \frac{1}{x}$. We could show this directly from definition 2.6, but it is an easy consequence of theorem 2.16 to come, so we delay the proof of this statement to corollary 2.17. Intuitively, however, this should already be clear: whatever N we try, there is always an x closer to 0 that takes more time to invert up to the desired precision.

[2] One could imagine a further than computability and uniform computability, where the I depends only on x, instead of all representatives $\bar{x}$'s of that x. This notion seems to be less relevant, however. To be truthful, uniform computability is not encountered in the literature, nor will it feature in prominent theorems in this thesis – the relevancy of the entire notion could be debated. We give it here as an attempt to explain the structure of computable functions further, relating it (in theorem 2.16) to known matters.

Since a real number can be regarded as a constant function of zero arguments, we obtain the following definition of a computable real number.

2.8 **Corollary (Computable real number)** *Let $x \in \mathbb{R}$ be a real number. We call x computable if and only if there is a Turing machine as in definition 2.1, which computes a representative of x on no input.*
Notice that any computable real number is automatically uniformly computable.

2.9 **Example (Every fraction is computable)** A fraction $q \in \mathbb{Q}$ can be represented finitely. So, we can simply construct a Turing machine that consecutively outputs the segments $[q-2^{-i}, q+2^{-i}]$.

2.10 **Example ($\sqrt{2}$ is computable)** This follows immediately from example 2.5, but we can also eliminate some overhead for a more direct method.

Define $f : \mathbb{N} \to \mathbb{N}$ by $f(n) = \min\{k \in \mathbb{N} \mid k^2 < 2n^2 \leq (k+1)^2\}$, $J_0 = [1,2]$ and $J_n = [\frac{f(n)}{n}, \frac{f(n)+2}{n}]$. Then f is computable (by exhaustive search for k), so there is a machine that computes $J_0, J_1, \ldots$. Furthermore $\sqrt{2} \in J_n$ for all n and $\lim_{n\to\infty} \text{length}(J_n) = 0$, so $\{\sqrt{2}\} = \bigcap_{n\in\mathbb{N}} J_n$.

Again, the sequence is not nested, but we can fix that with the same method as in example 2.5.

Properties 2.5

A property that almost immediately comes to mind is the behavior of composition. Actually, we have already used this property before.

2.11 **Lemma (Composition respects (uniform) computability)** *The classes $\mathscr{C}$ of computable real functions and $\mathscr{C}_u$ of uniformly computable real functions are closed under composition.*
PROOF Let $f, g \in \mathscr{C}$. Then there are Turing Machines M_f, M_g that compute f and g. We construct a Turing Machine M to compute $g \circ f$ to show that also $g \circ f \in \mathscr{C}$. Basically, we just 'interleave' both machines, feeding one the output of the other when available. Because we have stated definitions 2.1 and 2.3 more precisely than Weihrauch, we have to be more elaborate about proving that this M actually computes $g \circ f$.

Let $\bar{x}$ be a representative of $x \in \text{Dom}(f)$, and let $k \in \mathbb{N}$. Let $N^g_{k,\bar{x}} \in \mathbb{N}$ be the number of steps that M_g needs to reach output precision k upon input $M_f(\bar{x})$. Within these $N^g_{k,\bar{x}}$ computational steps, M_g has used some finite number of symbols of the input $M_f(\bar{x})$; we call this number $I^g_{k,\bar{x}}$. Let $N^f_{k,\bar{x}}$ be the number of steps M_f needs to reach output precision $I^g_{k,\bar{x}}$ upon input $\bar{x}$, and denote by $I^f_{k,\bar{x}}$ the number of input symbols used. Then M outputs an approximation of precision k in the (finite) number of steps it takes to simulate $N^g_{k,\bar{x}}$ computational steps of M_g and $N^f_{k,\bar{x}}$ steps of M_f, and needs $I^f_{k,\bar{x}}$ input symbols of $\bar{x}$ for that. Hence M indeed computes $g \circ f$.

If both M_f and M_g are uniform, $I^g_{k,\bar{x}}$ and $I^f_{k,\bar{x}}$ will be independent of $\bar{x}$. Hence then M will compute $g \circ f$ uniformly. □

The property of real functions that is by far used most in everyday scientific life is continuity (Burden and Faires, 2001). If $f : (\subseteq\mathbb{R}) \to \mathbb{R}$ is continuous and $\lim_{n\to\infty} x_n = x$, then also $\lim_{n\to\infty} f(x_n) = f(x)$; an approximation of the input gives an approximation of the output. Therefore, one might guess that continuous functions are computable, or at least that there is a connection between the two.

2.12 **Definition (Continuous function)** A function $f : (\subseteq\mathbb{R}) \to \mathbb{R}$ is called continuous at $x \in \text{Dom}(f)$ if and only if for every $\varepsilon \in \mathbb{R}^+$, there is a $\delta \in \mathbb{R}$ such that $|x-y| \leq \delta \Rightarrow |f(x)-f(y)| \leq \varepsilon$ for all $y \in \text{Dom}(f)$. It is called continuous if and only if it is continuous at every point of its domain. Note that δ can depend on x as well as ε. This function $\delta : \text{Dom}(f) \times \mathbb{R}^+ \to \mathbb{R}^+$ is called a modulus of continuity.

As it turns out, the converse is true: computable functions are necessarily continuous! This result dates back to (Grzegorczyk, 1955), and is true because every finite prefix of a representative induces a open set, and in finite time, a Turing machine can only read a finite prefix of its input.

The proof we give here is applicable more generally, in any topological space. A function on a topological space is called continuous, if the pre-image of any open set is open. (In $\mathbb{R}$, the open sets are generated by open intervals.)

2.13 **Theorem (Every computable function is continuous)** *Let $f:(\subseteq\mathbb{R})\to\mathbb{R}$ be computable. Then f is continuous.*

PROOF Let M be a machine computing f and let $x\in\mathrm{Dom}(f)$. To prove that f is continuous at x, we need to show that for every open set $V\subseteq\mathbb{R}$ with $f(x)\in V$, there is an open set $U\subseteq\mathrm{Dom}(f)$ with $x\in U$ and $f(U)\subseteq V$. So, let V be open and containing $f(x)$.

Let $([a_0,b_0],[a_1,b_1],\ldots),a_i,b_i\in\mathbb{Q}$ with $a_i<a_{i+1}<b_{i+1}<b_i$ and $\bigcap_{i\in\mathbb{N}}[a_i,b_i]=x$ be a representative of x. Likewise, let $([c_0,d_0],[c_1,d_1],\ldots)$ be the output of M on this input; $([c_0,d_0],[c_1,d_1],\ldots)$ is a representative of $f(x)$. There is an $n\in\mathbb{N}$ such that $[c_n,d_n]$ is contained in V.

For producing this initial part of the output, M takes N steps, for some $N\in\mathbb{N}$. In these N steps, M can read at most the first N symbols from the input. So, choose U to be the interval of real numbers (a_N,b_N). Obviously, we then have $x\in U$ and U open.

Now assume that $x'\in U$. Then x' has a representative with the same first N symbols as x. On this input, M has to write the same output, because it only reads the first N digits. Thus $f(x')\in[c_n,d_n]$, and so we conclude $f(U)\subseteq[c_n,d_n]\subseteq V$. □

That discontinuous functions are not computable is not all that remarkable, because equality is not even decidable:

2.14 **Example (Equality is not decidable)** $f:\mathbb{R}^2\to\mathbb{R}$ defined by $f(x,y)=\begin{cases}1 & \text{if } x=y\\ 0 & \text{if } x\neq y\end{cases}$ is not computable.

Of course, this follows from theorem 2.13, but we can also see it more directly. For, to write the first symbol of the output, a machine has to compare *all* symbols of the representatives of x and y. But this can of course not be done in a finite number of steps. So f can not even be computable with precision 1!
Likewise, the $\leq$ and $\geq$-relations are not decidable. Yet, these basic operations are widely used in computer applications (de Berg et al., 2000). So, the obvious way to compute standard discontinuous functions like $f(x)=\begin{cases}1 & \text{if } x\geq 0\\ 0 & \text{if } x\leq 0\end{cases}$ immediately fails.

There is a uniform counterpart to theorem 2.13. This is a consequence of the fact that our definition of (uniform) computability resembles that of (uniform) continuity so closely. This time, we use Cauchy's original ε,δ-definition 2.12 of continuity, since topologically we need more structure to consider uniform continuity than just any topological space[3].

2.15 **Definition (Uniformly continuous function)** A function $f:(\subseteq\mathbb{R})\to\mathbb{R}$ is called uniformly continuous if and only if it is continuous, and a modulus of continuity δ can be chosen independent of x. In other words, f is uniformly continuous if for every $\varepsilon\in\mathbb{R}^+$, there is a $\delta\in\mathbb{R}$ such that $|x-y|\leq\delta\Rightarrow|f(x)-f(y)|\leq\varepsilon$ for all $x,y\in\mathrm{Dom}(f)$.

2.16 **Theorem (Every uniformly computable function is uniformly continuous)** *Let $f:(\subseteq\mathbb{R})\to\mathbb{R}$ be uniformly computable. Then f is uniformly continuous.*

PROOF Let $\varepsilon\in\mathbb{R}^+$ be given. Pick $k\in\mathbb{N}$ such that $2^{-k}\leq\varepsilon$. Since f is computable, there is a machine M that computes it. Since it is uniformly computable, there is an $I\in\mathbb{N}$ such that for any representative of any point of its domain as input, M reaches output precision k using only I input symbols. Put $\delta=2^{-I}$.

Let $x,y\in\mathrm{Dom}(f)$ be such that $|x-y|\leq\delta$. We can pick representatives $\bar{x}$ of x and $\bar{y}$ of y that have the same first I symbols. Hence, writing $M_I(\bar{x})$ for the output of M on input $\bar{x}$ after examining the input up to precision I (and reaching output precision k), we have $M_I(\bar{x})=M_I(\bar{y})$. (Notice that $M_I(\bar{x})$ represents a segment of real numbers.) Since now both $f(x)\in M_I(\bar{x})$ and $f(y)\in M_I(\bar{y})=M_I(\bar{x})$, we have $|f(x)-f(y)|\leq\sup\{|a-b|:a,b\in M_I(X)\}=2^{-k}=\varepsilon$. □

[3] A forteriori, the spaces we use, with the Cantor topology, do not have this extra property. They are not *uniform spaces*.

2.17 **Corollary (Inversion is not uniformly computable)** *The function $f : (0,1) \to \mathbb{R}$ defined by $f(x) = \frac{1}{x}$ is not uniformly computable.*

Thus we see that the class $\mathscr{C}_u$ of uniformly computable functions is a (relatively) small part of the class of all computable functions $\mathscr{C}$, which was to be expected. After all, one would expect there to be much more 'chaotic' algorithms than the 'orderly' uniform ones which faithfully process the input up to the needed precision only.

Lifting the notion 2.6

On top of real functions $(\subseteq\mathbb{R}) \to \mathbb{R}$, we would like to look at real functional operators like differentiation or integration. These are operations that map functions to numbers, or functions to other functions.

2.18 **Definition (Real function space)** If X and Y are representable spaces, we write $\mathscr{F}(X,Y)$ for the set of all computable functions $X \to Y$. If $X \subseteq \mathbb{R}$ and $Y = \mathbb{R}$, we abbreviate this to $\mathscr{F}(X)$.

2.19 **Definition (Real function operator)** Let $X \subseteq \mathbb{R}$ and $Y \subseteq X$. A real function operator F is a function $F : (\subseteq\mathscr{F}(X)) \to \mathscr{F}(Y)$.

Representation of functions

We would like to define a similar notion of computability on real function operators, e.g. without extending the Turing Machine definition 2.3 too much. But this time, the input is not a real number $x \in \mathbb{R}$, but an entire function $f \in \mathscr{F}(X)$ for some $X \subseteq \mathbb{R}$. So first we need some way to represent such a function on the input tape, just as we needed to represent a real number on tape.

2.20 **Definition (Rational polygon)** We call a function $p : (\subseteq\mathbb{R}) \to \mathbb{R}$ a simple rational polygon if there are $n \in \mathbb{N}$ and vertices $x_1, \ldots, x_n \in \mathbb{Q}$ such that $x_1 < \ldots < x_n$, $\mathrm{Dom}(p) = [x_1, x_n]$ and p is linear on $[x_i, x_{i+1}]$ and $p(x_i) \in \mathbb{Q}$ for every $1 \leq i < n$. A rational polygon is a finite union of simple ones that still is a function.

2.21 **Lemma (Rational polygons are uniformly computable)** *Any rational polygon p is uniformly computable.*

PROOF It suffices to consider only simple rational polygons p. Suppose p has vertices $x_1, \ldots, x_n$. Then p is defined by $p(x) = p(x_i) + \frac{x - x_i}{x_{i_1} - x_i}(p(x_{i+1}) - p(x_i))$ between x_i and x_{i+1}. Since the coefficients are rational, example 2.7 tells us that p is uniformly computable on each segment $[x_i, x_{i+1}]$, say by Turing machine M_i. We combine these machines into one.

Let a representative $([a_1,b_1],[a_2,b_2],\ldots)$ of $x \in \mathrm{Dom}(p)$ be given on the input tape. We may assume, without loss of generality, that $[a_1,b_1]$ is contained in two fixed segments $[x_i, x_{i+2}]$: if $[a_1,b_1]$ overlaps with more than one $[x_i, x_{i+2}]$, we can simply skip to $[a_l,b_l]$, where we pick l such that $2^{-l} \leq \min\{x_{i+1} - x_i \mid i = 1, \ldots, n-1\}$. And if $[a_l,b_l]$ is contained in one fixed segment $[x_i, x_{i+1}]$, then we can simply use M_i to compute p (uniformly). Thus we are left with the case that $[a_l,b_l]$ is contained in precisely two fixed segments. We now distinguish four cases.

If $p(x_i) \leq p(x_{i+1}) \leq p(x_{i+2})$, then we transform $[a_k,b_k]$ into $[p(a_k), p(b_k)]$ using M_i and M_{i+1}.
If $p(x_i) \geq p(x_{i+1}) \geq p(x_{i+2})$, then we transform $[a_k,b_k]$ into $[p(b_k), p(a_k)]$.
If $p(x_i) \leq p(x_{i+1}) \geq p(x_{i+2})$, then we transform $[a_k,b_k]$ into $[\min\{p(a_k), p(b_k)\}, p(x_{i+1})]$.
If $p(x_i) \geq p(x_{i+1}) \leq p(x_{i+2})$, then we transform $[a_k,b_k]$ into $[p(x_{i+1}), \max\{p(a_k), p(b_k)\}]$.

Notice that this yields a sequence of nested segments. By furthermore computing and taking into account the maximum gradient $\Delta = |\max\{\frac{p(x_{i+1}) - p(x_i)}{x_{i+1} - x_i} \mid i = 1, \ldots, n-1\}|$, we can ensure that the segments we output are of length at most 2^{-k}, by skipping $\lceil \Delta^{-1} k \rceil$ input segments. This computation gives a representative of $p(x)$. It is uniform, because the finite number of computations of M_i are. □

2.22 **Definition (Function ball)** Let $\varepsilon \in \mathbb{R}$, $\varepsilon > 0$, and let p be a rational polygon. We call the set $B(p,\varepsilon) = \{f \in \mathscr{F}(\mathrm{Dom}(p)) : \|f - p\| \leq \varepsilon\}$ a function ball with center p and radius ε. Here, $\|f\| = \sup_{x\in\mathrm{Dom}(f)} |f(x)|$ is the supremum norm, and for $f \in \mathscr{F}(X)$ on a compact set $X \subseteq \mathbb{R}$ this equals $\max_{x\in\mathrm{Dom}(f)} |f(x)|$ by theorem 2.13.

2.23 **Definition (Representation of $\mathscr{F}(X)$)** Let $X \subseteq \mathbb{R}$ be a finite union of rational segments, and let $f \in \mathscr{F}(X)$. We call $(p_0, p_1, p_2, \ldots)$ a representative of f if for all $n \in \mathbb{N}$, p_n is a rational polygon on X such that $f \in B(p_n, 2^{-n})$.

Note that a rational polygon can be coded in a self-delimiting, finite way, such that we can indeed write a representative $(p_0, p_1, \ldots)$ of f on a Turing Machine's tape. So, like our definition of a computable real number, definition 2.8, computability of a function representative also follows directly from definition 2.3.

There are other representation systems of $\mathscr{F}(X)$, but this one is the "weakest", in the sense that the most functions have a representative, while the evaluation function is still computable.

A representative $(B_0, B_1, \ldots)$ of $f \in \mathscr{F}(X)$ encloses f arbitrarily narrowly. This enables one to compute the application or evaluation function operator, since computing comes down to approximating.

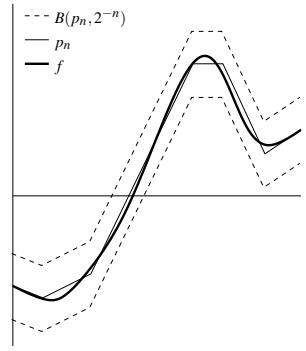


Figure 2.2: A function f in a function ball with radius 2^{-n} around a rational polygon p_n.

2.24 **Lemma (Evaluation is uniformly computable)** *Let $X \subseteq \mathbb{R}$ be a finite union of rational segments. Define the function Apply : $\mathscr{F}(X) \times X \to \mathbb{R}$ by Apply$(f,x) = f(x)$. Then Apply is uniformly computable.*

PROOF Let representatives $(B_0, B_1, \ldots)$ of f and $\bar{x}$ of x be given on two infinite input tapes, with $B_n = B(p_n, 2^{-n})$. Without loss of generality, we need only consider the situation where X is a rational segment and every p_n is a simple rational polygon (with $\mathrm{Dom}(p_n) = X$).

Using lemma 2.21, we now compute p_0 on input $\bar{x}$, until it outputs the first approximation $[c_0, d_0]$. Now, assuming we already have $[c_n, d_n]$, we go on to compute p_{n+1} on input $\bar{x}$, until its $n+1$st approximation $[c_{n+1}, d_{n+1}]$. We have

$$\begin{aligned} \|f(x) - [c_n, d_n]\| &\leq |f(x) - p_n(x)| + \|p_n(x) - [c_n, d_n]\| \\ &\leq \|f - p_n\| + \|p_n(x) - [c_n, d_n]\| \\ &\leq 2^{-n} + 2^{-n} = 2^{-(n-1)}, \end{aligned}$$

and conclude that this procedure computes Apply. This computation is uniform because we know in advance how much input symbols to use of $\bar{x}$ and $(B_0, B_1, \ldots)$. It might seem strange that we can uniformly compute the evaluation of a non-uniformly computable function, but consider that constructing a representative of that function might take a long time. □

A representative of a function can be computable, just as a real number could be (cf. definition 2.8): it is computable if and only if there is a machine that computes that representative upon empty input. So a function $f : X \to \mathbb{R}$ can have a computable representative, and it can be computable. These two properties are equivalent.

2.25 **Lemma (Modulus of continuity is computable)** *Let $X \subseteq \mathbb{R}$ be a finite union of rational segments. Then we can effectively modify a machine M_f computing $f \in \mathscr{F}(X)$ into a machine M'_f that computes a modulus of continuity δ for f. (That is, there is a Turing machine N that, given a code of M_f for an f, computes a code of M'_f.)*

PROOF We only give a sketch of the proof. For more details see (Weihrauch and Zheng, 1999).

Step 1: Denote by K_X the collection of all compact subsets of X. The metric $d : X \times K_X \to \mathbb{R}$ defined by $d(x,C) = \max\{|x-y| \ : \ y \in C\}$ is computable, so K_X is representable (see section 3.3).

Step 2: Since f is computable, it is continuous, and thus maps compact sets into compact sets. We can effectively find a function $F : K_X \to K_X$ such that $f(C) = F(C)$ for any compact $C \subseteq X$.

Step 3: X is effectively locally compact. That is, there is a computable $\beta : X \times \mathbb{R} \to K_X$ such that $\beta(x,\delta)$ is a compact of radius $\gamma(x,\delta) \neq 0$ contained in $[x-\delta, x+\delta]$, and γ is non-decreasing in δ.

Step 4: The function $h : X \times \mathbb{R}^+ \to \mathbb{R}^+$ defined by $h(x,\delta) = \max\{|f(x)-y| \ : \ y \in f(\beta(x,\delta))\}$ is computable, and non-decreasing in δ. So the function $g(x,\delta) = h(x,\delta) + \delta$ is computable and strictly increasing in δ. Hence the function $\hat{\delta}_1(x,\varepsilon) = \min\{\delta \ : \ \varepsilon = g(x,\delta)\}$ is computable. So finally the function $\hat{\delta}(x,\varepsilon) = \gamma(x,\hat{\delta}_1(x,\varepsilon))$ is computable, and moreover satisfies $|x-x'| < \hat{\delta}(x,\varepsilon) \Rightarrow |f(x) - f(x')| < \varepsilon$. Thus we have found a modulus of continuity $\hat{\delta}$ for f in a computable way.

□

2.26 **Theorem (A function is computable iff it has a computable representative)** *Let $X \subseteq \mathbb{R}$ be a finite union of rational segments, and let $f \in \mathscr{F}(X)$. Then f has a computable representative (cf. definition 2.23) if and only if f is computable (cf. definition 2.3).*

PROOF

($\Rightarrow$) Assume f has a computable representative. Then there is a Turing machine M that computes that representative on no input. We construct a machine to compute f as follows. Let x be given on the input tape. We use M to generate rational polygon approximations of f on a second tape. Finally we run Apply to write a representative of $f(x)$ on the output tape. So, by lemma 2.11 and lemma 2.24, f is computable.

($\Leftarrow$) Assume that f is computable. By theorem 2.13, we know that f must be continuous. And since X is compact, f is even uniformly continuous. This means

$$\forall_{\varepsilon>0}\exists_{\delta>0}\forall_{x,y\in X}[|x-y| < \delta \Rightarrow |f(x)-f(y)| < \varepsilon];$$

there is one δ that works for all x (if f is only continuous, δ can depend on x).

By lemma 2.25, this one δ enables us to choose n rational points, say $x_1, \ldots, x_n \in X$, such that $\forall_{x\in X}\exists_{i\leq n}[|x-x_i| < \delta]$. For example $n = \lceil \frac{(\max X)-(\min X)}{\delta} \rceil$ with $x_i = (\min X) + i\frac{(\max X)-(\min X)}{n}$ if X is connected (i.e. a segment) – and it suffices to only consider the connected components. Hence we can construct a rational polygon p_ε, namely with vertices $(x_i, [f(x_i)])$, where we round $f(x_i)$ to a fraction, such that f is contained in $B(p_\varepsilon, \varepsilon)$.

So if we let a machine consecutively execute this procedure with $\varepsilon = 2^{-n}$ and write $B(p_\varepsilon,\varepsilon)$ to the output tape, we see that f has a computable representative. □

Computability of real function operators

Of course, we can now simply lift definition 2.3 to define computable real functionals: we call a real functional $F : (\subseteq \mathscr{F}(X)) \to \mathscr{F}(Y)$ computable, if there is a Turing Machine as in definition 2.1 that maps any representative of any $f \in \mathscr{F}(X)$ to any representative of $F(f)$.

Generalizing the notion 2.7

The concept of computability just given can be vastly generalized. We discuss two examples of generalization. First we apply the definition of computability to objects other then functions. This might not seem such a big deal, but that will change in chapter 4. Second, we abstract from real functions to mappings on uncountable sets in general.

Sets

Like in the discrete case, we can apply the notion of computability to a host of objects, like graphs or sets, provided we choose a suitable representation.

A set $X \subseteq \mathbb{N}^n$ is called decidable, if its characteristic function $\chi_X : \mathbb{N}^n \to \{0,1\}$, which we define by $\chi_X(x) = \begin{cases} 0 & \text{if } x \notin X \\ 1 & \text{if } x \in X \end{cases}$, is computable. If we would try to extend this definition to subsets of $\mathbb{R}^n$, we would immediately stumble on a problem: according to the continuity theorem 2.13, no set X whatsoever would then be decidable, since χ_X is not continuous, unless $X = \emptyset$ or $X = \mathbb{R}^n$.

So, we have to use a continuous function to take over the role of χ_X, which vanishes outside X, and is non-trivial within. Or the other way around; all that matters is that we can effectively decide, not whether we prefer yes or no when the queried point is inside or outside of the set.

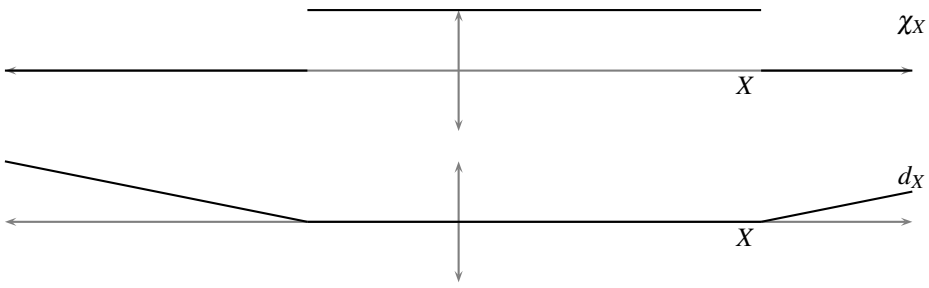


Figure 2.3: Graphs of the characteristic and distance functions χ_X and d_X of a set $X \subseteq \mathbb{R}$.

2.27 **Definition (Decidable set)** Let $n \in \mathbb{N}$. We call a set $X \subseteq \mathbb{R}^n$ decidable, if its distance function $d_X : \mathbb{R}^n \to \mathbb{R}$ defined by $d_X(x) = \inf_{y \in X} \|y - x\|$ is computable.

Easy examples of decidable sets in $\mathbb{R}^2$ are (open) discs and (open, filled) polygons. An example of a non-decidable set in $\mathbb{R}^2$ is the Mandelbrot set (Blum et al., 1998).

Functions on uncountable spaces

We have given a definition of computability for real functions, but why be content with that? The idea of computation on 'infinite words' can just as well be done on other objects than real numbers, as long as they can be represented suitably. So let us assume a finite alphabet Σ, and consider the set of 'infinite words' over this alphabet, Σ^ω.

2.28 **Definition (Infinite words)** Fix a finite set Σ, called the alphabet. We define the set of infinite words over Σ as $\Sigma^\omega = \{a_0 a_1 a_2 \cdots \mid a_i \in \Sigma\} = \{p : \mathbb{N} \to \Sigma\}$.

2.29 **Definition (Prefix)** For an alphabet Σ and $k \in \mathbb{N}$, we define the prefix relation $\simeq_k$ on infinite words as follows. For $u, v \in \Sigma^\omega$, we can write $u = a_1 a_2 \cdots, v = b_1 b_2 \cdots$ with $a_i, b_i \in \Sigma$. Define $u \simeq_k v$ if and only if $a_i = b_i$ for all $i = 1, \ldots, k$. If we are not interested in the length k of the prefix, we will frequently drop the index k and simply write $\simeq$.

2.30 **Definition (Computable string function)** Let Σ and Σ' be alphabets and $k \in \mathbb{N}$. A string function $f : (\subseteq \Sigma^\omega) \to \Sigma'^\omega$ is called computable with precision k if and only if

- there is a Turing machine M with (one-way) infinite input and output tapes, of which the output tape's head can only move right,
- when given a $u \in \Sigma^\omega$ on the input tape,

- there is an $N_u \in \mathbb{N}$, such that
- after at most N_u steps of calculation,
- $M's$ output tape holds a word v (ending in infinitely many blanks) such that $v \simeq_k f(u)$.

If N_u can be chosen independently of u, f is said to be uniformly computable with precision k.

Finally, f is called (uniformly) computable if it is (uniformly) computable with precision k, by the same M, for all $k \in \mathbb{N}$.

2.31 **Definition (Representation)** Fix an alphabet Σ. A representation[4] of a set X over Σ is a surjective function $\rho : (\subseteq \Sigma^\omega) \twoheadrightarrow X$. We call X representable over Σ if such a representation exists.

Most representations will be 'separable', in the sense that there is a distinct countableness involved. For example, the Nested Interval representation we used in this chapter approximates (with rational segments) in a countable number of steps: in $([a_0,b_0],[a_1,b_1],\ldots)$, the segments are indexed by the countable set $\mathbb{N}$. We do not pursue this more precisely, however.

2.32 **Definition (Computability)** We call a function $f : (\subseteq X) \to Y$ computable, if X and Y are representable over Σ and Σ' by some representations $\rho_X : (\subseteq \Sigma^\omega) \twoheadrightarrow X$ and $\rho_Y : (\subseteq \Sigma'^\omega) \twoheadrightarrow Y$ and there is a computable string function $g : (\subseteq \Sigma^\omega) \to \Sigma'^\omega$ such that $f \circ \rho_X = \rho_Y \circ g$.

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ {\scriptstyle\rho_X}\uparrow & & \uparrow{\scriptstyle\rho_Y} \\ \Sigma^\omega & \xrightarrow{g} & \Sigma'^\omega \end{array}$$

This g is called a (ρ_X,ρ_Y)-*realisation* of f.

2.33 **Remark (Cantor topology)** If we take $\Sigma = \{0,1\}$, we can make Σ^ω into a topological space, by using definition 2.29, called the *Cantor space*, or the *Cantor topology* on Σ^ω. A subbasis for this topology is given by the sets $\mathcal{O}_w = \{u \in \Sigma^\omega \mid w \simeq u\}$. That is, the open sets in Cantor space are combinations of infinite unions and finite intersections of sets of infinite words with the same finite prefix. See (Kelley, 1955) for more details.

We can visualize Cantor space as a binary tree, with edges labeled `0` and `1`, in which every infinite branch corresponds to an infinite word, and every finite branch to a finite word.

(Notice that Cantor space is *not* connected; it is homeomorphic to the Cantor discontinuum.)

[4] There is a pitfall in the literature here: in the Computable Analysis-camp, a representation 'assigns an object to a name': it is a function $(\subseteq \Sigma^\omega) \to X$. But in Representation Theory, as used in e.g. Lie Groups and Quantum Mechanics, it is precisely the other way around: a representation assigns a 'name' (a matrix, or something else which calculation is easy with) to an actual object (element of the Lie group).

CHAPTER

3 Representations

Using Turing Machines to define computability on real functions forces us to interpret the input and output via representations (cf. definition 2.31). This immediately raises the question whether different representations yield different behavior in computability (or complexity, which we will study in the next chapter). In this chapter, we study several commonly used representations of real numbers and real functions, to see if our definitions hold up, and if not, how their behavior varies. We will illustrate each representation by exemplifying the magical constant π. Furthermore we look at arithmetic one each representation: the very least thing we want to be computable are the basic rules of calculation.

Real numbers 3.1

Since this thesis is about computation on real functions, the first thing we need to consider is of course a real number. How do we write a real number on a Turing Machine tape? There might be differences in complexity or even computability when using different representations; for example, $\frac{1}{2}$, $\frac{2}{4}$ and even $\frac{1{,}000{,}000}{2{,}000{,}000}$ refer to the same real number – they have exactly the same mathematical or *extensional* properties. But from a computing science point of view, they differ, since storing $\frac{1{,}000{,}000}{2{,}000{,}000}$ costs more bits than storing $\frac{1}{2}$ (Stoelinga, 1997). So we see that unlike proofs, which are about abstract properties like $x \leq x+1$, computations, like $x+y$, are about representations.

Coding prerequisites

In the following, we will frequently use several standard representations, for example of $\mathbb{N}$ and $\mathbb{Q}$. Let us define these first, once and for all. They are mere formalities, and you should not let these technical definitions cloud your view of the bigger picture. For more details, see (Veldman, 1987), or, to a lesser extent, (Weihrauch, 2000).

We will construct a function $\ll\cdot\gg : \{0,1\}^\omega \to X$ for various X: the same notation is used for the representation of various 'basic' sets. We will also use the notation $\ll\cdot\gg_X$ for clarity, but if the context renders it clear which version of this overloaded representation is meant, the subscript X is frequently omitted.

3.1 **Definition (Some standard representations)** Recall that a representation of a set X over the alphabet $\{0,1\}$ is a surjective function $(\subseteq\{0,1\}^\omega) \twoheadrightarrow X$.

Natural numbers are represented by $\ll\cdot\gg_\mathbb{N}$, defined by $\ll b_0 b_1 \dots b_k\gg_\mathbb{N} = \sum_{i=0}^{k} 2^i \cdot b_i$, where $b_i \in \{0,1\}$. So, natural numbers are represented by their binary notation.

Integers are represented by $\ll\cdot\gg_\mathbb{Z}$, defined by $\ll b_0 b_1 \dots b_{k+1}\gg_\mathbb{Z} = \begin{cases} \ll b_0 b_1 \dots b_k\gg_\mathbb{N} & \text{if } b_{k+1} = 0 \\ -\ll b_0 b_1 \dots b_k\gg_\mathbb{N} & \text{if } b_{k+1} = 1 \end{cases}$. So, the rightmost bit represents the sign of the integer.

Tuples of integers can be represented over $\{0,1\}$ using the fundamental theorem of arithmetic. Denote the primes by $p_0, p_1, \dots$: so $p_0 = 2, p_1 = 3, p_2 = 5, p_3 = 7$, etc. We can now define a bijective

function code : $\bigcup_{k\in\mathbb{N}}\mathbb{Z}^k \to \mathbb{Z}$ by $\text{code}(x_0,\dots,x_k) = \prod_{i=0}^{k} p_i^{x_i+1}$. The convention is that for the empty tuple () we have code(()) = 0. We can now represent k-tuples of integers by $\ll b_0 b_1 \cdots b_k \gg_{\mathbb{Z}^k} = \text{code}^{-1}(\ll b_0 b_1 \cdots b_k \gg_{\mathbb{Z}})$.

Fractions are of course tuples of integers: we define $\ll b_0 b_1 \cdots b_k \gg_{\mathbb{Q}} = \frac{p}{q}$ where $\ll b_0 b_1 \cdots b_k \gg_{\mathbb{Z}^2} = (p,q)$.

Notice that the above representations are all finite, i.e. their range is actually limited to $\{0,1\}^*$. Moreover, they are even bijective when restricted to $\{0,1\}^*$!

Sequences of objects are infinite, and hence require the whole of $\{0,1\}^\omega$. We define them by concatenation, but we need a way of delimiting the individual objects to do that. Therefore, we define $\ll 1^k 0 b_0 b_1 \cdots b_k w \gg_{X^\infty} = \ll b_0 b_1 \cdots b_k \gg_X \circ \ll w \gg_{X^\infty}$, where $b_i \in \{0,1\}$, $w \in \{0,1\}^\omega$, and $\circ$ denotes concatenation. Notice that though it seems cyclic, this is well-defined!

If the individual objects in the sequence are themselves infinite, more advanced measures have to be taken. We then define $\ll w \gg_{X^\infty} = (u_0, u_1, u_2, \dots)$, where $w_{\ll i,j \gg_{\mathbb{N}^2}} = (u_i)_j$.

Since every $\ll\cdot\gg_X$ defined above is in fact bijective when restricted to a proper subset of X, we can write $\ll\cdot\gg^{-1}$ for the inverse when the input is known to be in this restricted domain. Using these standard representations over the alphabet $\{0,1\}$, we can build more complicated ones.

Nested Intervals[1] 3.2

In section 2.3 we loosely defined the representation of nested intervals, and have used it ever since, for example when proving computability-related theorems. We will shortly give a justification for this. Let us first define this representation thoroughly.

3.2 **Definition (Nested Interval representation)** We define a representation $\rho_{\text{NI}} : (\subseteq \{0,1\}^\omega) \twoheadrightarrow \mathbb{R}$ by

$$\rho_{\text{NI}}(\ll((p_0,q_0,c(0)),(p_1,q_1,c(1)),\dots)\gg^{-1}) = x$$

if
- $p_i, q_i \in \mathbb{Q}$ for all $i \in \mathbb{N}$,
- $p_i \le p_{i+1}$ and $q_i \ge q_{i+1}$ for all $i \in \mathbb{N}$,
- $p_i < x < q_i$ for all $i \in \mathbb{N}$, and
- $\bigcap_{i\in\mathbb{N}}[p_i,q_i] = \{x\}$ (This is equivalent to $\lim_{i\to\infty} p_i - q_i = 0$ and $\lim_{i\to\infty} p_i = \lim_{i\to\infty} q_i = x$.), and
- there is a computable function $c : \mathbb{N} \to \mathbb{Q}$ such that $q_k - p_k \le c(k)$; c is called the *modulus of convergence*.

So a real number is represented by a sequence of rational segments that enclose it ever narrower, with a computable margin of error at each step. Notice that ρ_{NI} is surjective, thus indeed a representation. We shall also loosely write $(([p_0,q_0],c(0)),([p_1,q_1],c(1)),\dots)$ or even $([p_0,q_0],[p_1,q_1],\dots)$ for a representative.

One could demand more of ρ_{NI} if this comes in handy: for example, p_i, q_i could be taken to be fractions of the form $\frac{a}{2^b}$ with $a \in \mathbb{Z}$ and $b \in \mathbb{N}$. At present, however, we do not require these extra properties.

3.3 **Example (Pi)** A nested interval representation for π might be $(([\frac{31}{10},\frac{32}{10}],\frac{1}{10}),([\frac{314}{100},\frac{315}{100},\frac{1}{100}]),([\frac{3141}{1000},\frac{3142}{1000}],\frac{1}{1000}),\dots)$.

[1] The naming is somewhat unfortunate. We will stick to the convention that an *interval* is an open set $\{x \in \mathbb{R} : a < x < b\}$, whereas a *segment* is a closed set $\{x \in \mathbb{R} : a \le x \le b\}$.

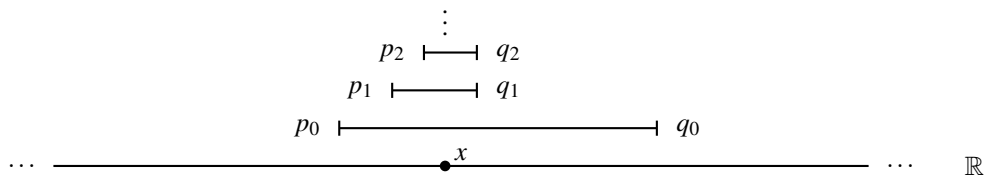


Figure 3.1: Illustration of the Nested Interval representation

3.4 **Remark (Arithmetic on Nested Intervals)** Naturally, 'pointwise' addition and multiplication defines the same operations on two real numbers in nested intervals representation. And arithmetic on $\mathbb{Q}$ is computable, for example by the school methods. We saw this already in example 2.4.

Cauchy sequences 3.3

A Cauchy sequence is a sequence whose terms become arbitrarily close to each other as the sequence progresses; it is a sequence which 'ought to' converge (and in fact does, because $\mathbb{R}$ is complete by the supremum-axiom). Since equivalence classes of Cauchy sequences over $\mathbb{Q}$ are a common way to 'define' $\mathbb{R}$ (Bishop and Bridges, 1985), it is not surprising that one can *represent* $\mathbb{R}$ using Cauchy sequences over $\mathbb{Q}$.

3.5 **Definition (Cauchy sequence over $\mathbb{Q}$)** A sequence $a_1, a_2, a_3, \ldots$ in $\mathbb{Q}$ is called a Cauchy sequence over $\mathbb{Q}$ if there are computable functions $c : \mathbb{N} \to \mathbb{N}$ and $d : \mathbb{N} \to \mathbb{Q}$ such that $\forall_{k\in\mathbb{N}} \forall_{m,n\geq c(k)} [\, |a_m - a_n| < d(k) \,]$ and $\lim_{k\to\infty} d(k) = 0$. The functions c and d are called *modulus of convergence*. Usually, $d(k) = 2^{-k}$.

> We introduced the moduli of convergence into the definition to eliminate the existential quantor. Actually knowing the function c brings the enormous benefit that it enables actual computations over the Cauchy sequence. (If one did not have c, one would only know that some N existed such that the Cauchy property holds for larger numbers. But which one?) Almost all representations include such a computability-demand.

From analysis, we know that every Cauchy sequence over $\mathbb{Q}$ has a limit (in $\mathbb{R}$), and that every real number is the limit of a (Cauchy) sequence over $\mathbb{Q}$ (van Rooij, 1986). The former property is called '$\mathbb{R}$ is *complete* over $\mathbb{Q}$', the latter is called '$\mathbb{Q}$ is *dense* in $\mathbb{R}$'.

3.6 **Definition (Cauchy sequence representation)** We define a representation $\rho_{\mathrm{CS}} : (\subseteq \{0,1\}^\omega) \twoheadrightarrow \mathbb{R}$ by

$$\rho_{\mathrm{CS}}(\ll a_1, c(1), d(1), a_2, c(2), d(2), a_3, c(3), d(3), \ldots) \gg^{-1}) = \lim_{i\to\infty} a_i$$

if $(a_1, a_2, \ldots)$ is a Cauchy sequence over $\mathbb{Q}$ with moduli of convergence c, d.

3.7 **Example (Pi)** We can use the fact that π is defined as twice the smallest positive zero of the cosine to devise a Cauchy sequence for π. Take $a_0 = 0$ and $a_{i+1} = a_i + \lfloor \cos(a_i) \rfloor$ for $i \in \mathbb{N}$, where we round the cosine down to a suitable fraction. Then $a_i < a_{i+1}$ for all i; $(a_1, a_2, \ldots)$ is a strictly increasing sequence. It is even a Cauchy sequence, and its modulus of convergence is easily found, but not very interesting (Bishop and Bridges, 1985). So, we see that π is represented by the Cauchy sequence $(2a_1, 2a_2, 2a_3, \ldots)$.

3.8 **Remark (Arithmetic on Cauchy sequences)** If we add, subtract, multiply and divide two Cauchy sequences pointwise, we obtain another Cauchy sequence.

Metric spaces

The Cauchy representation is applicable more generally than for $\mathbb{R}$: the construction holds on metric spaces in general.

3.9 **Definition (Metric space)** Let R be a set, and $\delta : R \times R \to \mathbb{R}$ a real-valued function on R^2. We call δ a metric, if it has the following three properties:

- $\delta(x,y) = \delta(y,x)$ for all $x,y \in R$ *(Symmetry)*
- $\delta(x,z) \leq \delta(x,y) + \delta(y,z)$ for all $x,y,z \in R$ *(Triangle inequality)*
- $\delta(x,y) \geq 0$ for all $x,y \in R$, and $\delta(x,y) = 0$ only if $x = y$ *(Positivity)*

A space R together with a metric δ on it, (R,δ), is called a metric space.

3.10 **Example** There are several metrics on $\mathbb{R}$:

- We can endow any set R with the discrete or trivial metric defined by $\delta(x,y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$.
- The most familiar metric on $\mathbb{R}$ is by far the Euclidean one, defined by $\delta(x,y) = |x-y|$.
- But there are other metrics on $\mathbb{R}$, like $\delta(x,y) = |\arctan(y) - \arctan(x)|$.

3.11 **Definition (Cauchy Sequence representation of a metric space)** Let (R,δ) be a metric space, and Q a countable dense subset of R.

A sequence $a_1, a_2, a_3, \ldots$ in Q is called a Cauchy sequence over Q if there are computable functions $c : \mathbb{N} \to \mathbb{N}$ and $d : \mathbb{N} \to \mathbb{Q}$ such that $\forall_{k\in\mathbb{N}} \forall_{m,n \geq c(k)} [\delta(a_m, a_n) < d(k)]$ and $\lim_{k\to\infty} d(k) = 0$.

Since Q is countable, it has a representation $\ll\cdot\gg$, composed of the ones we saw in the beginning of this chapter. Then we can also define a representation $\rho_{\mathrm{CS}} : (\subseteq \{0,1\}^\omega) \twoheadrightarrow R$ of R by

$$\rho_{\mathrm{CS}}(\ll(a_1, c(1), d(1), a_2, c(2), d(2), a_3, c(3), d(3), \ldots)\gg^{-1}) = x$$

if $(a_1, a_2, a_3, \ldots)$ is a Cauchy sequence over Q with moduli of convergence c, d, converging to x, that is, $\lim_{k\to\infty} \delta(a_k, x) = 0$.

Base-B expansions 3.4

Perhaps you are by now wondering what all the fuss is about: why not use the familiar representation of $\mathbb{R}$ everyone uses in daily life to denote real numbers? Well, both Nested Intervals and Cauchy Sequences offered a way to 'compensate'. In a finite prefix of a representation, there is still 'enough room' (but not 'too much'): with Nested Intervals, one can always choose a smaller segment that still contains problematic cases, and for a Cauchy Sequence it does not matter if the first few items behave erratically, as long as the tail obeys the Cauchy property. We will first define the base-B expansion rigorously, and then illustrate the problem with this representation.

3.12 **Definition (Base-B expansion representation)** Let $B \in \mathbb{N}$, $B \geq 2$. We define the base-B expansion representation $\rho_{\mathrm{B}B} : (\subseteq \{0,1\}^\omega) \twoheadrightarrow \mathbb{R}$ by

$$\rho_{\mathrm{B}B}(\ll(n, b_n, b_{n-1}, \ldots, b_0, b_{-1}, b_{-2}, b_{-3}, \ldots)\gg^{-1}) = \sum_{i=-\infty}^{n} b_i \cdot B^i$$

if $b_i \in \{0,1,\ldots,B-1\}$ for $i \leq n$. The number B is called the *base* or *radix*. The usual notation for a B-ary expansion of x is $[x]_B$.

In case $B = 10$, this representation is the intimately familiar decimal expansion. Notice that a real number can have more representations in this system; for example, $2.0000\cdots = 1.9999\cdots$.

3.13 **Example (Pi)** Of course, a base-10, or decimal, expansion representation of π is $3.141592653589\cdots$.

As we mentioned before, $\rho_{\mathrm{B}B}$ does not 'leave enough room' (Di Gianantonio, 1991):

3.14 **Example (Arithmetic on base-B expansions)** Multiplication is not computable under $\rho_{\mathrm{B}B}$.
PROOF Let $f : \mathbb{R}^2 \to \mathbb{R} : (x,y) \mapsto x \cdot y$, and suppose $\rho_{\mathrm{B}B} \circ f \circ \rho_{\mathrm{B}B}^{-1}$ were computable. Now consider $x = \frac{1}{3}, y = 3$ with representation $x = \mathtt{0.33333}\cdots$ and $y = \mathtt{3.00000}\cdots$. Then, surely, there would be a number $N \in \mathbb{N}$ such that a machine M computes the first digit of the output in at most N steps. But after N steps, M can only have read the first N digits of x. So M cannot output a `0` as the first digit, because the $N+1$-st digit of x can be a 4. Neither can it output a `1`, because the $N+1$-st digit can be a 2. But this is a contradiction! Hence f is not computable under the representation of base-B expansions. □

Note that the continuity theorem 2.13 nevertheless holds!

Though the base-B expansion representation is not desirable, since even simple functions like multiplication are not computable, it still plays a very large role in thinking about real computation. For our trained minds, it is comfortable to think about the input and output tapes of the Turing Machines as if they held (infinite) decimal expansions. 'Higher precision', for us, almost coincides with 'more digits'.

Variations

Luckily, the deficiencies of the base-B expansion representation can be repaired (Di Gianantonio, 1991). For example, we can introduce negative digits:

3.15 **Definition (Base-B negative digit expansion representation)** Let $B \in \mathbb{N}$, $B \geq 2$. We define a representation $\rho_{\mathrm{B}-B} : (\subseteq\{\mathtt{0},\mathtt{1}\}^\omega) \twoheadrightarrow \mathbb{R}$ by

$$\rho_{\mathrm{B}-B}(\ll(n,b_n,b_{n-1},\ldots,b_0,b_{-1},b_{-2},b_{-3},\ldots)\gg^{-1}) = \sum_{i=-\infty}^{n} b_i \cdot B^i$$

if $b_i \in \{-(B-1),-(B-2),\ldots,-1,0,1,\ldots,B-1\}$ for $i \leq n$. We often abbreviate the digit `-d` by $\overline{\mathtt{d}}$.

(This representation with $B = 2$ is called the *signed digit representation* by Weihrauch, and we will meet it again in chapter 4.) Or we can use non-integer bases B:

3.16 **Definition (Base-B non-integer expansion representation)** Let $B \in \mathbb{R}/\mathbb{Z}$, $B > 1$ be a computable real number, and $D \in \mathbb{N}$ such that $1 < B < D$. We define a representation $\rho_{\mathrm{B}B,D} : (\subseteq\{\mathtt{0},\mathtt{1}\}^\omega) \twoheadrightarrow \mathbb{R}$ by

$$\rho_{\mathrm{B}B,D}(\ll(n,b_n,b_{n-1},\ldots,b_0,b_{-1},b_{-2},b_{-3},\ldots)\gg^{-1}) = \sum_{i=-\infty}^{n} b_i \cdot B^i$$

if $b_i \in \{0,1,\ldots,D-1\}$ for $i \leq n$.

Both variations 'leave more room', since small corrections are still possible after the n^{th} digit has been given. They are far less intuitive, though. For example, with negative digits in base 2, the number 3 could be written as $[1,1]$ or $[1,0,-1]$.

Notice that in this case it is not necessary to give a modulus of convergence , since it is easily expressible in terms of B: if needed, we can always use $c(k) = B^{-k}$.

Dedekind cuts 3.5

Like Cauchy Sequences, Dedekind cuts were first introduced as a way to construct $\mathbb{R}$ from $\mathbb{Q}$ (Dedekind, 1963). The main intuition is that a real number x, intuitively, is completely determined by the fractions strictly smaller than x and those strictly larger than x. To quote Dedekind himself:

> If all points of the straight line fall into two classes such that every point of the first class lies to the left of every point of the second class, then there exists one and only one point which produces this division of all points into two classes, this severing of the straight line into two portions.

He goes on by defining that "a point produces the division of the real line if this point is either the least or greatest element of either one of the classes mentioned above". Thus, a real number is nothing more than a 'cut' of $\mathbb{Q}$.

3.17 **Definition (Dedekind cut)** A Dedekind cut is a subset $x \subseteq \mathbb{Q}$ such that:

- x is not empty.
- $\mathbb{Q} \backslash x$ is not empty.
- x contains no greatest element.
- For $p, q \in \mathbb{Q}$, if $p \in x$ and $q < p$, then $q \in x$ as well.

Some authors state an extra demand to ensure some constructive proofs (Bishop and Bridges, 1985). Though we do not need it here, we state it for completeness:

- for all $p, q \in \mathbb{Q}$, if $p < q$, then $q \geq x$ or $\exists_{s \in x}[s > p]$.

3.18 **Definition (Dedekind cuts as real numbers)** $\mathbb{R}$ is the set of Dedekind cuts, ordered by set-theoretic inclusion: $x < y :\Leftrightarrow x \subsetneqq y$ and $x = y \Leftrightarrow x \subseteq y$ and $y \subseteq x$.

One readily verifies that this construction obeys the axioms for $\mathbb{R}$, in particular the supremum axiom. Therefore, this construction naturally yields a representation of $\mathbb{R}$.

3.19 **Definition (Dedekind cut representation)** We define a representation $\rho_{\mathrm{DC}} : (\subseteq \{0,1\}^{\omega}) \twoheadrightarrow \mathbb{R}$ by

$$\rho_{\mathrm{DC}}(\ll (a_1, c(1), a_2, c(2), a_3, c(3), \ldots) \gg^{-1}) = \sup\{a_i : i \in \mathbb{N}\}$$

if $a_i \in \mathbb{Q}$ with $a_i < a_j$ whenever $i < j$, and c is a computable function (called modulus of convergence) such that $|a_k - \sup\{a_i : i \in \mathbb{N}\}| \leq c(k)$. (Notice that every such set of a_i's determines a set that satisfies the requirements for a Dedekind cut.)

As with Cauchy sequences, we can also define the algebraic operations (addition, multiplication, division) and constants (0 and 1) for Dedekind cuts in a constructive fashion.

3.20 **Definition (Arithmetic on Dedekind cuts)** Given two Dedekind cuts (real numbers) x and y, we define

- $0 = \{x \in \mathbb{Q} : x < 0\}$
- $1 = \{x \in \mathbb{Q} : x < 1\}$

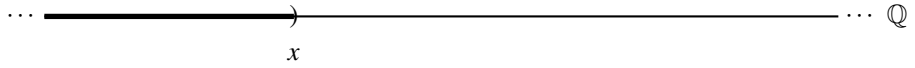


Figure 3.2: Illustration of a Dedekind cut

- $x+y=\{p+q : p\in x, q\in y\}$
- $-x=\{p\in\mathbb{Q} : -p\notin x$, but $-p$ is not the least element of $\mathbb{Q}\backslash x\}$
- $|x|=\begin{cases} x & \text{if } x\geq 0 \\ -x & \text{if } x\leq 0 \end{cases}$
- If $x>y$, then $x\cdot y=\{r\in\mathbb{Q} : r\leq 0$ or $r=p\cdot q$ for some $p\in x, q\in y$ with $p,q>0\}$.
  In general, $x\cdot y=\begin{cases} 0 & \text{if } x=0 \text{ or } y=0 \\ |x|\cdot|y| & \text{if } x>0, y>0 \text{ or } x<0, y<0 \\ -(|x|\cdot|y|) & \text{if } x>0, y<0 \text{ or } x<0, y>0 \end{cases}$
- For $x>0$, $\frac{1}{x}=\{p\in Q : p\leq 0$ or $p>0$ and $\frac{1}{p}\notin x$, but $\frac{1}{p}$ is not the least element of $\mathbb{Q}\backslash x\}$.
  If $x<0$, $\frac{1}{x}=-(\frac{1}{|x|})$

With these operations, $(\mathbb{R},<,=,+,\cdot,0,1)$ is indeed a complete ordered field: the required properties mostly follow from the properties of $\mathbb{Q}$ as an ordered field[2].

Continued fractions 3.6

A *continued fraction* is of the form

$$a_0+\cfrac{b_1}{a_1+\cfrac{b_2}{a_2+\cfrac{b_3}{a_3+\ldots}}}$$

where $a_i, b_i\in\mathbb{Z}$. Every continued fraction can be simplified to a normal form, and every real number can be written as such a simple continued fraction!

3.21 **Definition (Simple continued fraction)** A real number x is called a simple continued fraction if $a_i\in\mathbb{Z}$ exist such that

$$x=a_0+\cfrac{1}{a_1+\cfrac{1}{a_2+\cfrac{1}{a_3+\ldots}}}$$

or, informally, $x=[a_0,a_1,\ldots]$.

3.22 **Proposition (Every real number is a continued fraction)** *Let $x\geq 0$ be a real number. Define a_i inductively by*

$$x_0=x,\ \ a_n=\lfloor x_n\rfloor,\ \ x_{n+1}=\begin{cases} \frac{1}{x_n-a_n} & \text{if } a_n\neq x_n \\ 0 & \text{otherwise} \end{cases}$$

Then $x=[a_0,a_1,a_2,\ldots]$
PROOF by simply verifying through calculation. See (Olds, 1963) for a spelled-out calculation. □

In fact, a continued fraction $[a_0,a_1,\ldots,a_n]$ for x is finite if and only if $x\in\mathbb{Q}$. We call the n^{th} terms $[a_0,\ldots,a_n]$ of a continued fraction $[a_0,a_1,\ldots]$ its n^{th} *convergent*.

These facts lead to another representation of $\mathbb{R}$.

3.23 **Definition (Continued fraction representation)** We define a representation $\rho_{\text{CF}} : (\subseteq\{0,1\}^\omega)\twoheadrightarrow\mathbb{R}$ by

$$\rho_{\text{CF}}(\ll a_0,c(0),a_1,c(1),a_2,c(2),a_3,c(3),\ldots)\gg^{-1})=a_0+\cfrac{1}{a_1+\cfrac{1}{a_2+\cfrac{1}{a_3+\ldots}}}$$

[2] In two steps however, namely when proving that inverses and opposites are properly defined, we require an extra property of $\mathbb{Q}$: the Archimedean property. Therefore, Dedekind cuts are not desirable as a representation of the p-adic numbers.

for $a_i \in \mathbb{Z}$, and a computable function c such that $|[a_0,\ldots,a_k] - [a_0,a_1\ldots]| \leq c(k)$. We will also informally write $x = [a_0,a_1,\ldots]$.

Continued fractions yield very elegant formulas, and are therefore very powerful in number theory.

3.24 **Example (Pi)** Two continued fraction expansions involving π are (Olds, 1963)

$$\frac{4}{\pi} = 1 + \cfrac{1^2}{2+\cfrac{3^2}{2+\cfrac{5^2}{2+\cfrac{7^2}{2+\ldots}}}} \qquad \text{and} \qquad \frac{\pi}{2} = 1 - \cfrac{1}{3-\cfrac{2\cdot3}{1-\cfrac{1\cdot2}{3-\cfrac{4\cdot5}{1-\cfrac{3\cdot4}{3-\cfrac{6\cdot7}{1-\ldots}}}}}}.$$

(The first formula was found by Brouckner about 1658, the second by Stern in 1833.) A (simple) continued fraction representation of π is $\pi = [3,7,15,1,292,1,1,1,2,1,3,1,14,2,1,1,2,2,2,2,\ldots]$, which was found by Lambert in 1770.

3.25 **Remark (Arithmetic on continued fractions)** Continued fractions allow numerical calculations a little at a time without ever introducing any error (roundoff or truncation). This is the reason why ρ_{CF} leaves 'enough room', where ρ_{BB} didn't. However, basic arithmetic on continued fractions is not that simply expressed. We refer to (Potts, 1998) for a highly generalized approach; actual algorithms following Potts for arithmetic operations on numbers in continued fraction representation can be found in (Beeler et al., 1972). (They led to a world record of 51,539,600,000 digits of π in 1997!)

3.26 **Remark (Topological properties of representations)** As we saw in chapter 2, topology plays a big role in computability aspects. In particular, a 'good' representation ρ should have at least the following topological properties (Müller, 1986).

- ρ is a continuous function (with respect to the Cantor topology on Σ^ω and the Euclidean topology on $\mathbb{R}$).
- For any topological space (M,τ) and any $H : (\subseteq\mathbb{R}) \to M$, if $H \circ \rho$ is continuous, than so is H itself.
- For any compact $K \subseteq \mathbb{R}$, the set $\rho^{-1}(K)$ is compact.

As we will see chapter 4, we need to make some more demands if we are interested in complexity as well as computability. Special attention has to be paid to 'efficiency' of a representation.

Equivalent representations 3.7

Although they might not look it, all of these representations are quite alike – with the exception of the base B expansion representation, in which even simple arithmetic is uncomputable.

3.27 **Definition (Equivalence of representations)** Let X be a set and Σ an alphabet. We call two representations $\rho,\rho' : (\subseteq\Sigma^\omega) \twoheadrightarrow X$ equivalent if there is a bijective computable string function $f : \Sigma^\omega \to \Sigma^\omega$ that maps any ρ-representative of any $x \in X$ to a ρ'-representative of x, more precisely

$$\forall_{x\in X}\left[\,\bar{x} \in \rho^{-1}(x) \;\Rightarrow\; f(\bar{x}) \in (\rho')^{-1}(x)\,\right],$$

and moreover does so *in polynomial time*.

For now, we will disregard the hypothesis *in polynomial time*, since we have not even defined it yet. In chapter 4, we will come back to this matter. Therefore, the proof of the following theorem is not complete, but will be completed by its counterpart, theorem 4.22, in chapter 4.

3.28 **Theorem (Representation equivalence theorem)** *The representations ρ_{NI}, ρ_{CS}, ρ_{DC} and ρ_{CF} are all equivalent.*

PROOF We build a cycle of transformations:

"$\rho_{NI} \Rightarrow \rho_{DC}$" Given a Nested Interval representation $((a_0,b_0,c(0)),(a_1,b_1,c(1)),\ldots)$ with $a_i < a_{i+1} < b_{i+1} < b_i$, $a_i, b_i \in \mathbb{Q}$ and $\bigcap_{i\in\mathbb{N}}[a_i,b_i] = x \in \mathbb{R}$, we can simply give a Dedekind Cut-representation of the same number x by $(a_0,c(0),a_1,c(1),a_2,c(2),\ldots)$, since $sup\{a_i : i \in \mathbb{N}\} = x$, and c surely is a modulus of convergence for ρ_{DC} too[3].

"$\rho_{DC} \Rightarrow \rho_{CS}$" Let a Dedekind Cut-representation for $x \in \mathbb{R}$ be given by $(a_0,c(0),a_1,c(1),a_2,c(2),\ldots)$, where of course $sup\{a_i : i \in \mathbb{N}\} = x$. We define a Cauchy-sequence for x simply by $a_0,a_1,\ldots$, with modulus of convergence $d(k) = 2c(k)$ and $c'(k) = k$, both of which are computable.

"$\rho_{CS} \Rightarrow \rho_{CF}$" This is the hardest transformation, and we need lemma 35 from (Ménissier-Morain, 1994, page 114) to do it: if $\frac{p_n}{q_n}$ is the n^{th} convergent of a continued fraction for $x \in R$, then $|x - \frac{p_n}{q_n}| \leq \frac{1}{2^{2n+1}}$.

Now, let a Cauchy sequence $a_1,a_2,\ldots$ converging to x with modulus c be given, so we have that $\forall_k \forall_{m,n\geq c(k)}[|a_m - a_n| < 2^{-k}]$. Then certainly $\forall_k \forall_{m\geq c_k}[|a_m - x| < 2^{-k}]$. We construct a continued fraction $[b_0,b_1,\ldots]$ for x, with modulus of convergence c', inductively. First, take $b_0 = \lfloor a_{c(1)} \rfloor$.

Next, from the lemma, we know that if $b_0,\ldots,b_n$ are fixed, then $|y - [b_0,\ldots,b_n]| \leq \frac{1}{2^{2n+1}}$, where y is the limit of our continued fraction under development: of course our goal is to define the b_i such that $y = x$. So assume $|x - [b_0,\ldots,b_n]| \leq \frac{1}{2^{2n+1}}$ in order to choose b_{n+1}. Well, we already have $|a_{c(2n+1)} - x| \leq \frac{1}{2^{2n+1}}$, so we are done once $[b_0,\ldots,b_{n+1}] = a_{c(2n+1)}$.

Conclusion: choose $b_{n+1} = \cfrac{1}{\cfrac{1}{\vdots}{a_{c(2n+1)} - b_0} - b_n}$, and $c'(k) = 2^{-k}$.

"$\rho_{CF} \Rightarrow \rho_{NI}$" Given a continued fraction $[a_0,a_1,\ldots]$ for $x \in \mathbb{R}$ with modulus of convergence c, we need to construct a sequence of nested intervals enclosing x. But the convergents $\frac{p(n)}{q(n)}$ of a continued fraction more or less *are* rational segments: for example, take the nth segment I_n to be $[\frac{p(n)-1}{q(n)}, \frac{p(n)+1}{q(n)}]$. Now we only need to make these I_i nested. But that is easy: take $J_n = \bigcap_{i=1}^{n} I_i$, and you have a nested interval representation $(J_0,J_1,\ldots)$. For the modulus of convergence, we can then lazily take $c'(k) = |\frac{p(k)-1}{q(k)} - \frac{p(k)+1}{q(k)}| = \frac{2}{q(k)}$. □

Notice that all these transformations are not too bad: intuitively, no information is lost in the process, and mostly they feel as if linear in time, although we do not yet know what that means precisely. For example, in the conversion of ρ_{NI} to ρ_{DC}, all we do is leave out half of the description, and in $\rho_{CF} \Rightarrow \rho_{NI}$, all we do is some additions and subtractions.

3.29 **Proposition** *Let $\rho,\rho' : (\subseteq\Sigma^\omega) \twoheadrightarrow X$ be two representations of some set X over the alphabet Σ, and let $f : (\subseteq X) \to X$ be a function on X. If ρ and ρ' are equivalent, then f is ρ-computable if and only if it is ρ'-computable.*

This corollary again indicates that the base-B representation is inadmissible: since multiplication is computable in all our other representations, but not in base-B expansions, they do not have the same computability behavior. Hence they are not equivalent, otherwise the corollary would be contradicted. The fact that all those representations have the same computability-behavior, and the base B representation does not, leads to the following notion, which is a specific case of (Weihrauch, 2000)'s more general definition.

3.30 **Definition (Admissible representation)** Let ρ and τ be representations of a set M over the alphabet Σ. We say that ρ is *admissible with respect to* τ, if they are equivalent as representations.

We say that a representation $\rho : (\subseteq\Sigma^\omega) \twoheadrightarrow \mathbb{R}$ is *admissible* if it is admissible with respect to the nested interval representation ρ_{NI}.

[3] We see here explicitly that ρ_{DC} is actually nothing more than 'the lower half of ρ_{NI}'.

So this is the justification for the fact that we have used the Nested Interval-representation in previous chapters without even bothering: every other (admissible!) representation we would have considered would have yielded the same results.

It is amusing to note that Turing himself (Turing, 1936) first tried to define computability on real numbers with the decimal expansion representation. A year later, he recognized the deficiency and published another article that favors the nested interval representation.

3.31 **Example (Several admissible representations)** As we saw in our equivalence theorem 3.28, the representations of Nested Intervals, Cauchy Sequences, Dedekind Cuts and Continued Fractions are all admissible. But, as example 3.14 taught us, the representation of base B expansions is *not* admissible. Its variation with negative digits is, however.

3.32 **Remark** Non-admissibility boils down to $\rho^{-1}(x)$ being finite. A nice example of this is the base-B expansion representation, in which every real number has at most two representations. If it is irrational, it has a unique decimal expansion. If it is rational, the decimal expansion is either eventually periodic or finite. The former is unique, while the latter has precisely two possibilities: it can end in trailing zeroes or trailing nines. For example, $1 = \rho_{\mathrm{B}10}(\mathtt{1.0000}\cdots) = \rho_{\mathrm{B}10}(\mathtt{0.9999}\cdots)$. The base-$B$ expansion representation is not admissible. We will prove this shortly.

On the other hand, consider the (admissible) base-B negative digit expansion representation. In that case, every real number has (countably) infinitely many representations. For example, for all $n \in \mathbb{N}$, we have $0 = \rho_{\mathrm{B}-10}(\mathtt{0}^n\overline{\mathtt{1}}\mathtt{99999}\cdots)$.

3.33 **Lemma (Finite representations are not admissible)** *Let $\rho : (\subseteq \Sigma^\omega) \twoheadrightarrow \mathbb{R}$ be a representation of $\mathbb{R}$. If $\rho^{-1}(x)$ is finite for some $x \in \mathbb{R}$, then ρ is not admissible.*

PROOF Suppose there was a bijective computable string function $f : \Sigma^\omega \to \Sigma^\omega$ that maps any ρ-representative of any number in $\mathbb{R}$ to a ρ_{NI}-representative of it. Then f would give a one-to-one correspondence between the finite set $\rho^{-1}(x)$ and $\rho_{\mathrm{NI}}^{-1}(x)$. But the latter is always infinite! After all, if we have one $([a_0,b_0],[a_1,b_1],\ldots) \in \rho_{\mathrm{NI}}^{-1}(x)$, then also $([a_n,b_n],[a_{n+1},b_{n+1}],\ldots) \in \rho_{\mathrm{NI}}^{-1}(x)$ for every $n \in \mathbb{N}$, and since ρ_{NI} is surjective, every real number does have at least one representative, and thus automatically infinitely many. □

3.34 **Corollary ($\rho_{\mathrm{B}B}$ is not admissible)** *The base-B expansion representation is not admissible.*

Real functions 3.8

Similar to real numbers, we could represent real functions in different ways than with converging rational polygons. (By the way, we can recognize the representation with rational polygons as a generalization of the Cauchy sequences representation of real numbers.)

For example, instead of a function ball, we could use a union of open rectangles that cover the function, illustrated in the figure to the right on this page.

But those are, of course, only minor variations. However, in retrospect, all those representations of reals in the previous section were also minor variations of each other.

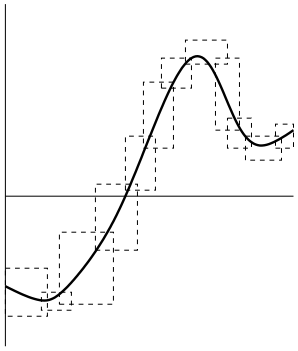


Figure 3.3: A function f can be represented by sequence of a finite number of squares, enclosing f ever narrower.

How about a major variation? What if we could get rid of the restriction that we can only represent functions on a bounded part of their domain?

Let us presuppose that we need function evaluation to be computable (just like we wanted basic arithmetic to be computable on representations of real numbers).

Three ways to overcome this restriction come to mind. The first is a fairly straightforward, but still effective(!) approach, the second is drawn from functional analysis, and the third is inspired by the method with which Markov tackled the problem of computability. We will only mention them, and not prove any kind of equivalence theorem.

Naive but effective 3.9

Let $f : (\subseteq\mathbb{R}) \to \mathbb{R}$ be computable, and have an unbounded domain. For simplicity, let us just assume that $\mathrm{Dom}(f) = \mathbb{R}$.

On every (bounded!) segment, we can represent f with rational polygons. Say that between $-n$ and n, f is represented by rational polygons $p_{n,1}, p_{n,2}, p_{n,3}, \ldots$ such that $|f(x) - p_{n,k}(x)| \leq 2^{-k}$ for $-n \leq x \leq n$. Then the sequence $(p_{n,k})_{n,k\in\mathbb{N}}$ is still countable, and hence we can use it to represent f.

3.35 **Definition (Rational polygon representation of $\mathscr{F}(\mathbb{R})$)** Let $f : \mathbb{R} \to \mathbb{R}$ be a computable function. Then there are rational polygons $p_{n,k}$ such that $|f(x) - p_{n,k}(x)| \leq 2^{-k}$ for $-n \leq x \leq n$. The rational polygon representation $\rho_{\mathrm{RP}} : (\subseteq\{0,1\}^\omega) \twoheadrightarrow \mathscr{F}(\mathbb{R})$ is defined by

$$\rho_{\mathrm{RP}}(\langle\!\langle p_{\varphi(0)}, p_{\varphi(1)}, p_{\varphi(2)}, \ldots\rangle\!\rangle) = f,$$

where $\varphi : \mathbb{N} \to \mathbb{N}\times\mathbb{N}$ is the standard bijection given by $\varphi^{-1}(n,m) = \frac{1}{2}(m+n)(m+n+1)+m$.

In general, we require $\mathrm{Dom}(f)$ to be the limit of finite unions of segments, in order to approximate f with rational polygons.

Functional analysis 3.10

The functions we are interested in representing, the computable ones, are all continuous. (One cannot reasonably expect non-computable functions to be representable effectively.) For representations, that is a very useful fact, because we can use other branches of mathematics that also deal with spaces of continuous functions to draw knowledge from.

Functional analysis is concerned with spaces of continuous functions. One major result is that all continuous functions on a given field, $\mathscr{F}(K)$, form a vector space (although infinite-dimensional). A forteriori: that vector space has a basis, for example of Lagrange, Bernstein or Chebyshev-polynomials when $K = \mathbb{R}$ (Burden and Faires, 2001).

In the following we will simply assume a basis $\mathscr{P}$ of $\mathscr{F}(X)$, consisting of all polynomials with rational coefficients. Notice that $\mathscr{P}$ is dense in $\mathscr{F}(X)$, but still countable.

3.36 **Proposition (Restriction of a representation)** *Suppose $\rho : (\subseteq\Sigma^\omega) \twoheadrightarrow X$ is a representation of X, and $Y \subseteq X$ is a subset of X. Then $\rho\big|_Y : (\subseteq\Sigma^\omega) \twoheadrightarrow Y$, the restriction of ρ to Y, is a representation of Y.*

3.37 **Definition (Polynomial representation of $\mathscr{F}(\mathbb{R})$)** By the continuity theorem 2.13, we know that $\mathscr{F}(\mathbb{R})$ is a subset of $C(\mathbb{R})$, the space of all continuous real functions. We can define a metric on $C(\mathbb{R})$ by (Weihrauch and Zhong, 2003)

$$\delta(f,g) = \sum_{k=1}^{\infty} 2^{-k} \frac{\|f-g\|_k}{1+\|f-g\|_k}, \quad \text{where } \|h\|_k = \sup_{|x|\leq k} |h(x)|.$$

We can also find a countable dense subset of $C(\mathbb{R})$, namely $\mathscr{P}$, the space of all polynomials with rational coefficients. So definition 3.11 gives a representation $\rho_{C(\mathbb{R})}$ of $C(\mathbb{R})$.

We define $\rho_{\mathrm{PO}} : (\subseteq\Sigma^\omega) \twoheadrightarrow \mathscr{F}(\mathbb{R})$ as the restriction of $\rho_{C(\mathbb{R})}$ to $\mathscr{F}(\mathbb{R})$.

When using such a basis $\mathscr{P}$ to actually write down representations for a given function, particular polynomials prove handy. For example, Lagrange polynomials are interpolating polynomials: they go through a set of given points, and are of low degree. Because we have a dense but countable subset of $\mathbb{R}$, namely the dyadic fractions $\frac{n}{2^m}$, we can interpolate ever more points to approximate the given function.

The situation with Bernstein and Chebyshev polynomials is not interpolating, but the idea remains the same. The Chebyshev polynomials are optimal, in the sense that the approximation error is smallest (Burden and Faires, 2001), but the Lagrange polynomials are easiest to define (and work with).

For an analyst, the polynomial representation is very natural, because it is just a concrete version of the Weierstrass Approximation Theorem.

À la Markov 3.11

There is another way to overcome the burden of only representing a function on a bounded part of its domain. Instead of enlarging the bounded part step by step, we can also try to describe the function on the entire domain more precisely.

As we have seen, for every computable function exists a Turing machine that computes it. This Turing machine itself can be represented finitely! After all, it is nothing more than a finite number of states and transitions (see appendix A). Equivalently, we can represent it by its input-output pairs: its graph.

Now, using the famous Universal Turing Machine (UTM)-theorem[4], we see that given such a representation of a computable function, we can indeed compute the evaluation (and (thus) much more). This is the algorithmic approach taken by Markov and Kolmogorov, although there is an important difference: the 'Russion school' only looks at the *computable* real numbers, $\mathbb{R}_c$, and functions $(\subseteq\mathbb{R}_c)\to\mathbb{R}_c$ thereof. We named this section after them because of the idea that a computable function can be represented by an encoding of its computing Turing machine. We will now look at the situation where we use the graph to represent a Turing machine.

3.38 **Proposition (Approximate word function of a string function)** *Suppose $f:(\subseteq\Sigma^\omega)\to\Sigma^\omega$ is a computable string function. Then there is a computable word function $f^*:(\subseteq\Sigma^*)\to\Sigma^*$ such that $f(x_1x_2\cdots)=\lim_{n\to\infty}f^*(x_1\cdots x_n)$ with $x_i\in\Sigma$ when $x_1x_2\cdots\in\mathrm{Dom}(f)$. f^* is called the* approximate word function *of f.*

PROOF Define $f^*(\mathtt{x}_1\cdots\mathtt{x}_n)=\mathtt{y}_1\cdots\mathtt{y}_m$ with $\mathtt{x}_i,\mathtt{y}_i\in\Sigma$, where

$$\forall_{w\in\Sigma^\omega}\exists_{v\in\Sigma^\omega}\left[f(\mathtt{x}_1\cdots\mathtt{x}_nw)=\mathtt{y}_1\cdots\mathtt{y}_mv\right],$$

and m is the largest such integer. This is welldefined because we know that the output head of a machine computing f can only move right: there cannot be two different $\mathtt{y}_1\cdots\mathtt{y}_m$ to one $\mathtt{x}_1\cdots\mathtt{x}_n$, except when one is a prefix of the other, which we have excluded by choosing m maximal.

Notice that f^* is monotonous, in the sense of prefixes: for $x,y\in\Sigma^*$ we have $x\simeq y\Rightarrow f^*(x)\simeq f^*(y)$. Thus the limit of f^* is indeed welldefined. □

3.39 **Definition (Representation of computable string functions)** We define the representation $\eta:(\subseteq\Sigma^*)\twoheadrightarrow\mathscr{F}(\subseteq\Sigma^\omega)$ of the computable string functions by $\eta(g)=f$, where $f\in\mathscr{F}(\subseteq\Sigma^\omega)$ is the function given by $f(\mathtt{x}_1\mathtt{x}_2\cdots)=\lim_{n\to\infty}f^*(\mathtt{x}_1\cdots\mathtt{x}_n)$, if $g\in\Sigma^*$ is an encoding of $\{(x,f^*(x))\ :\ x\in\mathrm{Dom}(f^*)\}$.

Notice that the graph of f^* is countable since f^* is a word function (Σ^* is countable), so it can indeed be encoded in Σ^*. We also denote $\eta(w)$ by η_w.

We recall two important properties of η from recursion theory.

[4] This is an example where we benefit from not having altered the Turing Machine: the machines we use to compute real functions are ordinary Turing Machines, hence we do not have to prove a new UTM theorem.

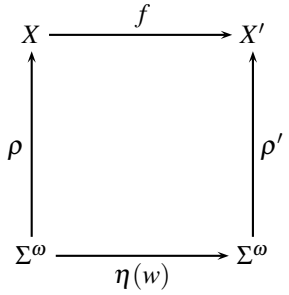
3.40 **Proposition (UTM)** *By the UTM-theorem, there is a computable function $u_\eta : (\subseteq\Sigma^\omega)\times\Sigma^\omega \to \Sigma^\omega$ such that $u_\eta(w,v) = \eta_w(v)$ for all $w \in \mathrm{Dom}(\eta)$ and $v \in \Sigma^\omega$.*
PROOF See (Weihrauch, 2000, section 2.3). □

3.41 **Proposition (SMN)** *By the S(m,n)-theorem, for every computable function $h : (\subseteq\Sigma^\omega)\times\Sigma^\omega \to \Sigma^\omega$ there is a (total) computable function $s : \Sigma^\omega \to \Sigma^\omega$ such that $h(w,v) = \eta_{s(w)}(v)$.*
PROOF See (Veldman, 1987). □

3.42 **Definition (Representation of computable functions)** Suppose $\rho : (\subseteq\Sigma^\omega) \twoheadrightarrow X$ and $\rho' : (\subseteq\Sigma^\omega) \twoheadrightarrow X'$ are representations. We define a representation $[\rho \to \rho']$ of $\mathscr{F}(X,X')$ by

$$[\rho \to \rho'](w) = f \quad \text{iff} \quad \eta(w) \text{ is a } (\rho,\rho')\text{-realisation of } f.$$

Thus $[\rho \to \rho'](w) = f \Leftrightarrow f(\rho(v)) = \rho'(\eta(w)(v))$ for all $v \in \mathrm{Dom}(f\circ\rho)$.



3.43 **Corollary (Representation of computable real functions)** *$[\rho_{CS} \to \rho_{CS}]$ is a representation of $\mathscr{F}(\subseteq\mathbb{R})$.*

3.44 **Remark** What we are trying to do here is to extend representations of X and Y to a representation of $\mathscr{F}(X,Y)$ or $[X \to Y]$ in general. This is called *higher type construction*, and is an intricate problem. It is not yet fully understood how the represented space $[X \to X]$ relates to the well known spaces of analysis (Normann, 2002).

On representations of functions, there is a similar notion of admissibility as with real numbers, given in definition 3.30.

CHAPTER

4 Complexity

Complexity is a natural extension of computability: once we know exactly *how* to compute something, isn't the next question to wonder *how long* this computation will take? Since we have defined computability using Turing machines, we can simply count the number of steps the machine is going to need. Unfortunately, with infinite input and output, the machine is never going to halt! Therefore, we are going to need a more elaborate method to count steps.

In appendix A we describe how to count steps in machines that do halt, in a very general way. In this chapter, we massage this definition (without altering it too much) to cover our needs. A central theme in this chapter will be to choose the right representation; 'not too big', nor 'not too small'.

Time and Lookahead 4.1

Complexity theory is about counting costs, naming prices, if you will. Typical 'costs' are the amount of time a computation takes, or the amount of tape cells used. But, in theory, we could use anything to count costs. Such a price-labeling function is called a *resource measure*. In this section, we define two very widely used ones.

Consider the following proposed definition:

4.1 **Example (A useless Time function)** Let $f : (\subseteq\mathbb{R}) \to \mathbb{R}$ be computable. Then there is a machine M that computes f. For $x \in \mathrm{Dom}(f)$ and precision $k \in \mathbb{N}$, we define $\mathrm{Time}_M(x,k)$ as the number of steps M needs to produce an output (prefix) y such that $|y - f(x)| \leq 2^{-k}$.

This definition wouldn't work because it is highly dependent on the *representative* of x, not on x itself. For example, take f to be the identity, and $x = 0$. Then M could get as input the representative $([-2^{-0},2^{-0}],[-2^{-1},2^{-1}],\ldots)$, or it could get the more efficient $[-2^{-0^2},2^{-0^2}],[-2^{-1^2},2^{-1^2}],\ldots)$. In the former case, it would take k time steps to reach an output precision of 2^{-k}, so $\mathrm{Time}_M(x,k) = k$, while in the latter case, $\mathrm{Time}_M(x,k) = \lceil\sqrt{k}\rceil$. Hence this Time_M is not well-defined. (This dependence on representation is already apparent in the classical case, as we will see in example 4.23).

(Weihrauch, 2000, example 7.2.1) shows that it is useless to demand that there *exists* one good representative of x such that Time_M is minimal. If we would take that as definition, everything will be linear in Time! Nor is it useful to demand that Time_M is minimal *for all* representatives of x. For then Time_M would always be ∞: for given $N \in \mathbb{N}$, one can always make a representative of x that takes more than N steps to compute up to precision 2^{-k} output for.

Instead, (Weihrauch, 2000) defines the resource measures directly on string functions, and so prevents the problem that Time depends on the representative of x. Then he looks at complexity classes induced by representations. Not all representations are suitable for useful notions of complexity (as we already saw in chapter 3): the main criterion for a representation to be useful is that is has 'not too many names for a single real number'. A bit more precisely, that the set of all strings which represent a real number is not too big, nor too small. (Analogously to the useless definitions with $\exists$ (too small) and $\forall$ (too big) above). In (Weihrauch, 2000), the story is then restricted to a single representation, the signed digit representation

(called the base-2 negative digit expansion representation in the previous chapter), for which this pre-image is of 'precisely the right size' (namely compact in the Cantor topology).

However, what we actually want is a 'complexity measure for real functions', not one for string functions which happen to represent a real function. What we would really like is to have (bounds on) a complexity of e.g. the sine, not of one particular way to compute the sine.

After all, isn't a theorem like "it takes *at least* $n \cdot \log(n)$ comparisons to sort n items in situ" much more appealing than one like "this or that implementation of a sorting algorithm takes *at most* n^2 comparisons"?

Let us try a different approach, inspired by Cauchy's definition of limit, of which our computability notion reminds strongly anyway, with its ever more closely approximated (input and) output. Instead of picking one special 'effective' representation, we let the definitions depend on the 'effectiveness' of a representation. This has the advantages that it is more direct for $\mathbb{R}$, hence easier to determine the complexity of a given example, and moreover, that the dependence on representations is made more explicit, while at the same time weakening the strong reliance on the single representation of 'signed digits', which might not be the most suitable one for all problems.

First we need to clarify a notation-related issue we ignored in example 4.1, as we will heavily use this shorthand later.

4.2 **Remark (Segment notation)** If $\rho : (\subseteq \Sigma^\omega) \twoheadrightarrow \mathbb{R}$ is a representation, $\bar{x} \in \mathrm{Dom}(\rho)$ a representative and $k \in \mathbb{N}$, we denote by $\bar{x}\big|_k$ the prefix of $\bar{x}$ of length k. Moreover, by $\rho(\bar{x}\big|_k)$ we denote the segment of all real numbers x which have a ρ-representative starting with $\bar{x}\big|_k$.

This should be interpreted right – a prefix of length k does not necessarily mean k letters of Σ. We could have stated this more rigidly, but that would not clarify much; the right interpretation is rather natural since a representation is always countable. A few examples are in order.

For example, if ρ is the representation of nested intervals, and $\bar{x} = \ll((p_0,q_0,c(0)),(p_1,q_1,c(1)),\ldots)\gg$, then $\bar{x}\big|_k = ((p_1,q_1,c(1)),\ldots,(p_{k-1},q_{k-1},c(k-1))$, and thus we interpret $\rho(\bar{x}\big|_k)$ as the segment $[p_{k-1},q_{k-1}]$.

If ρ is the representation of base-B expansions, and $\bar{x} = \ll(b_n,b_{n-1},\ldots,b_0,b_{-1},b_{-2},\ldots)\gg$, then $\rho(\bar{x}\big|_k) = [\,\sum_{i=n-k}^{n} b_i \cdot B^i\,,\,\sum_{i=n-k}^{n} b_i \cdot B^i + (B-1)\cdot B^{n-k-1}\,]$.

4.3 **Definition (Precision measure)** A *precision measure* of a representation $\rho : (\subseteq \Sigma^\omega) \twoheadrightarrow \mathbb{R}$ is a function $e : (\subseteq \Sigma^\omega) \times \mathbb{N} \to \mathbb{R}$, such that

$$\forall_{k\in\mathbb{N}} \forall_{\bar{x}\in\mathrm{Dom}(\rho)} \forall_{y\in\rho(\bar{x}\big|_k)} \left[\, |\rho(\bar{x}) - y| \leq e(\bar{x},k) \,\right]$$

With $\rho(\bar{x}\big|_k)$, we mean the prefix of $\bar{x}$ of length k, regarded as a segment of real numbers.

To finish the first example of remark 4.2, where ρ was the representation of nested intervals: we have $\rho(\bar{x}\big|_k) = [p_{k-1},q_{k-1}]$, and thus $e(\bar{x},k) \leq \max\{|x-p_k|,|x-q_k|\}$.

A precision measure is not precisely a modulus of convergence of a representation, since it is $\mathbb{R}$-valued, which makes a big difference, as we will see later. For example, had a precision measure not been $\mathbb{R}$-valued, the unicity in the following lemma 4.4 would not have been true.

Note that a precision measure is not actually a measure; though Σ^ω comes equipped with the Cantor topology, and thus has a Borel σ-algebra, (for k fixed) e works on points in that topological space, not on subsets.

The thing it resembles most is a norm on Σ^ω. It is positive-definite and has a triangle inequality. The underlying space Σ^ω here is just not precisely a vector space.

4.4 **Lemma (Existence, uniqueness and computability of precision measures)** *There always exists a precision measure for a given representation ρ, and there is a unique minimum precision measure. Moreover, this precision measure is computable if the representation is (computable, as a function, with respect to ρ_{NI}).*

PROOF Existence is trivial: we can simply write down the well-defined $e(\bar{x},k) = \sup_{y \in \rho(\bar{x}|_k)} |\rho(\bar{x}) - y|$. This e is automatically unique as the minimum precision measure. Assume now that ρ is computable, as a function, by the Turing machine M_ρ. We construct a machine M that computes e. Let $\bar{x} \in \Sigma^\omega$ and $k \in \mathbb{N}$ be given on two input tapes. First, we let M_ρ compute $\rho(\bar{x}|_k)$ on a work tape – that is, we give M_ρ the finite input $\bar{x}|_k$, and let it run until it has processed that finite input: that takes a finite amount of steps, and yields a prefix of a representative of $\rho(\bar{x})$, which can be regarded as a rational segment. Second, we compute (by M_ρ) $\rho(\bar{x})$ on a second work tape. Third, we intertwine these two computations with M_{subtract} to compute the difference of $\rho(\bar{x})$ and the left and right endpoints of $\rho(\bar{x}|_k)$. Thus M computes e. □

In a sense, a precision measure establishes an equilibrium between input precision and output precision: it tells us how much output is needed to get at least as much information as is present in a specified part of the input. Therefore, we can now measure 'at the input side', and that is what makes the following definition possible.

4.5 **Definition (Time function)** Let $f : (\subseteq \mathbb{R}) \to \mathbb{R}$ be computable by M. For $\bar{x} \in \rho^{-1}(\mathrm{Dom}(f))$, a precision measure e of ρ, and a precision $k \in \mathbb{N}$, we define $\mathrm{Time}_M(\bar{x},k)$ as the number of steps M needs to produce an output (prefix) Y upon input $\bar{x}$ such that $|y - f(\rho(\bar{x}))| \leq e(\bar{x},k)$ for all $y \in \rho(Y)$.

So, intuitively, $e(\bar{x},k)$ measures 'how close $\bar{x}$ is to x, up to precision k', and $\mathrm{Time}_M(\bar{x},k)$ is the number of steps needed to compute output of the same k-th precision as the input. Note that for Weihrauch's signed digit representation, $e(\bar{x},k) = 2^{-k}$ is a *uniform* precision measure.

Another important factor that matters is the so-called lookahead, the number of input symbols needed to produce an output (of a certain precision). It is defined analogously to the time function.

4.6 **Definition (LookAhead function)** Let $f : (\subseteq \mathbb{R}) \to \mathbb{R}$ be computable by M. For $\bar{x} \in \rho^{-1}(\mathrm{Dom}(f))$, a precision measure e of ρ, and a precision $k \in \mathbb{N}$, we define $\mathrm{LookAhead}_M(\bar{x},k)$ as the number of input cells M has visited before producing an output (prefix) Y upon input $\bar{x}$ such that $|y - f(\rho(\bar{x}))| \leq e(\bar{x},k)$ for all $y \in \rho(Y)$.

> Note that this situation reminds a lot of Kolmogorov complexity (Staiger, 2002), since we naturally want the precision measure to be 'as small as possible'. When we mentioned earlier that it would be nice to have a theorem like "it takes *at least* $n \cdot \log(n)$ comparisons to sort n items in situ" instead of "this or that implementation of a sorting algorithm takes *at most* n^2 comparisons", we grazed over this fact. Complexity inherently depends on an algorithm, not on the function it computes. When we try to alter this, for example by defining the complexity of a function to be the infimum of the complexities of all algorithms computing it, we pass over to the realm of Kolmogorov complexity. This is a very interesting realm, but just not concretely computable: we can hardly quantize anything computable about 'all algorithms computing a function'.
>
> In a sense, this is also what makes the $\mathbb{R}$-valued minimum precision measure of lemma 4.4 not computable in general. A $\mathbb{Q}$-valued approximation would be computable, but not the infimum anymore – it would be 'one algorithm', instead of the infimum of 'all algorithms', computing a function. In the next few pages, we concretize this line of thought, though we will not use it later on, since the emphasis of this thesis is on computability.

To get rid of the $\bar{x}$ in favor of x, we would very much like to define something like

$$\mathrm{Time}_f(x,k) = \inf_{\bar{x} \in \rho^{-1}(x)} \mathrm{Time}_M(\bar{x},k)$$

for $x \in \mathrm{Dom}(f)$, and even $\mathrm{Time}_f(x) = \lim_{k \to \infty} \mathrm{Time}_f(x,k)$. But then we would hardly ever be able to actually compute a complexity of a given function, so that would make a pretty useless resource measure.

Since Σ^ω is countable, so is $\rho^{-1}(x)$. So we could try 'averaging':

$$\mathrm{Time}_f(x,k) = \lim_{n \to \infty} \frac{1}{n} \sum_{i=0}^{n} \mathrm{Time}_M(\overline{x_i},k),$$

where $\rho^{-1}(x) = (\overline{x_0}, \overline{x_1}, \ldots)$. But this is still useless. First, we have to calculate $\mathrm{Time}_M(\bar{x}, k)$ for every possible representative $\bar{x}$ of x, so it is still abstract. Moreover, it is a priori not guaranteed that this sequence even converges!

This is true for any operation F for which we try

$$\mathrm{Time}_f(x,k) = \lim_{n\to\infty} F(\mathrm{Time}_M(\overline{x_0},k), \ldots, \mathrm{Time}_M(\overline{x_n},k)).$$

A priori, why on earth would this converge? Why would different representatives $\bar{x}$ of x have some general invariant for all functions f into which you input them? The only thing they all have in common is that they represent x, but that guarantees nothing about similarities of the operation of an arbitrary function on two of them in terms of 'effectiveness'.

Therefore, it seems natural to restrict matters, in order to be able to state a good definition of $\mathrm{Time}_f(x,k)$. And that is precisely what Weihrauch does.

> Notice that this restriction to the signed digit representation is where the compactness (in-the-Cantor-topology) demand comes in: compactness is the usual trick to get convergence of subsequences of arbitrary sequences.

4.7 **Conjecture (This definition of Time_M is the strongest possible)** *Let $\rho : (\subseteq \Sigma^\omega) \twoheadrightarrow \mathbb{R}$ be an admissible representation, and let M compute $f : (\subseteq \mathbb{R}) \to \mathbb{R}$. There exists no canonical definition of $\mathrm{Time}_f : \mathrm{Dom}(f) \times \mathbb{N} \to \mathbb{N}$ in terms of $\mathrm{Time}_M : \rho^{-1}(\mathrm{Dom}(f)) \times \mathbb{N} \to \mathbb{N}$, without putting a restriction on ρ.*

> We cannot prove this conjecture, because its nature is more philosophical than mathematical. How can we ever prove things about all possible (sensible) definitions, if we don't actually define them? In that respect, it resembles the Church-Turing-thesis, which tries to state something about all possible (sensible) definitions of computation humans are capable of. However, the above discussion is a convincing indication of the truth of this conjecture.

Although Weihrauch's 'signed digit complexity' cannot be generalized, we did get another perspective out of this discussion than the dictatorial demand to only use the signed digit representation; now we can understand better why this is useful. In our language, the signed digit representation means a constraint on the precision measure. So we come to:

4.8 **Definition (Time function)** Let $f : (\subseteq \mathbb{R}) \to \mathbb{R}$ be computable by M via an admissible representation ρ. Let e a precision measure for ρ, $x \in \mathrm{Dom}(f)$ and $k \in \mathbb{N}$. We define

$$\mathrm{Time}_{M,\rho}(x,k) = \inf\{\mathrm{Time}_M(\bar{x},k) \ : \ \rho(\bar{x}) = x,\ e(\bar{x},k) \le 2^{-k}\}^{1},$$

$$\mathrm{Time}_{f,\rho}(x,k) = \inf\{\mathrm{Time}_{M,\rho}(x,k) \ : \ M \text{ computes } f\}.$$

Since $\mathrm{Time}_M(\bar{x},k)$ takes values in $\mathbb{N}$, which is well-ordered, both these infimums are actually minimums.

Furthermore, for $X \subseteq \mathrm{Dom}(f)$, we denote by $\mathrm{Time}^X_{f,\rho}(k)$ the number of steps M takes to compute up to precision k for any $x \in X$:

$$\mathrm{Time}^X_{f,\rho}(k) = \sup_{x\in X} \mathrm{Time}_f(x)(k),$$

if it exists. Notice that if X is compact, this supremum always exists since f is computable by M, and we have $\mathrm{Time}^X_M(k) = \max_{x\in X} \mathrm{Time}_M(x)(k)$.

We will abbreviate this to $\mathrm{Time}_{f,\rho}(k)$ in the case that $X = \mathrm{Dom}(f)$, and to $\mathrm{Time}_f(k)$ if the representation is clear from the context.

Notice that in general Time_f is not computable, since then equality would be decidable (by the demand $\rho(\bar{x}) = x$), which contradicts example 2.14. Of course, we expected Time_f not to be computable.

[1] It could happen that for certain ρ and x there is no representative $\bar{x}$ such that $e(\bar{x},k) \le 2^{-k}$. In that case we say $\mathrm{Time}_{M,\rho}(x,k) = \infty$. However, ρ must be a bit awkward not to allow close reprentatives, e.g. the nested interval representation with the extra demand that the length of the kth segment is 2^{1-k}. All representations we studied in chapter 3 in fact allow infinitely many close representatives.

4.9 **Definition (LookAhead function)** Let $f : (\subseteq\mathbb{R}) \to \mathbb{R}$ be computable, ρ a representation, e a precision measure for ρ, $x \in \mathrm{Dom}(f)$ and $k \in \mathbb{N}$. We define

$$\mathrm{LookAhead}_{M,\rho}(x,k) = \min\{\mathrm{LookAhead}_M(\bar{x},k) \ : \ \rho(\bar{x}) = x,\ e(\bar{x},k) \le 2^{-k}\},$$
$$\mathrm{Time}_{f,\rho}(x,k) = \min\{\mathrm{LookAhead}_{M,\rho}(x,k) \ : \ M \text{ computes } f\},$$
$$\mathrm{LookAhead}^X_{f,\rho}(k) = \sup_{x\in X} \mathrm{LookAhead}_M(x,k),$$

for $X \subseteq \mathrm{Dom}(f)$. We will abbreviate this to $\mathrm{LookAhead}_{f,\rho}(k)$ in the case that $X = \mathrm{Dom}(f)$, and to $\mathrm{LookAhead}_f(k)$ if the ρ is clear from the context.

Notice that one reason why we used two infimums (over machines M and over representatives $\bar{x}$) in definitions 4.8 and 4.9 is that this way, we know there is one actual Turing Machine that computes a function within resources arbitrarily close to T. If there were only one infimum, it could happen that machine M_1 performs really well on representative $\bar{x_1}$, but really bad on $\bar{x_2}$, whereas M_2 consumes less resources on $\bar{x_2}$ than on $\bar{x_1}$; hence there would not be one machine that actually achieved a complexity bound arbitrarily close to T.

Basically, we have restricted what representatives to look at for complexity counts: the ones that are not too inefficient. This is a slight improvement over Weihrauch's approach, since we can now use any admissible representation to model our function with. While they all give the same results, some functions are expressible much easier in e.g. Nested Interval representation than in signed digit representation.

Between Time and Lookahead 4.2

With a bit of common sense, we can extract some properties about Time, LookAhead and relations between them almost immediately from the definitions.

4.10 **Corollary (Basic properties)** *Let M be a machine computing $f : (\subseteq\mathbb{R}) \to \mathbb{R}$. Then:*

(a) The function $(\bar{x},k) \mapsto \mathit{Time}_M(\bar{x},k)$ is bounded by a computable function.

(b) The function $(\bar{x},k) \mapsto \mathit{LookAhead}_M(\bar{x},k)$ is bounded by a computable function.

(c) The set $\{(\bar{x},k,t) \in \mathrm{Dom}(f)\times\mathbb{N}\times\mathbb{N} \mid \mathit{Time}_M(\bar{x},k) = t\}$ is semi-decidable, in the sense that follows from (a).

(d) The set $\{(\bar{x},k,t) \in \mathrm{Dom}(f)\times\mathbb{N}\times\mathbb{N} \mid \mathit{LookAhead}_M(\bar{x},k) = t\}$ is semi-decidable.

(e) For every $\bar{x} \in \mathrm{Dom}(f\circ\rho)$ and $k \in \mathbb{N}$ we have $\mathit{LookAhead}_{M,\rho}(\bar{x},k) \le \mathit{Time}_{M,\rho}(\bar{x},k)$, hence also $\mathit{LookAhead}^X_M(k) \le \mathit{Time}^X_M(k)$ for every $X \subseteq \mathrm{Dom}(f)$ if well-defined.

PROOF The functions in parts (a) and (b) themselves need not be computable since we cannot decide whether $|y - f(\rho(\bar{x}))| \le e(\bar{x},k)$. However, we can simply overestimate e by the computable modulus of convergence c of the representation. Therefore, we can simply run M on input $\bar{x}$, and count steps and input cells visited respectively, until an output prefix has been written that satisfies c up to kth precision. Part (c) follows easily by simulating t steps of M.

If we try the same approach on part (d), a priori we do not know how many steps to simulate. All we can do is wait until our simulation visits tape cell t, and answer positively if it does. However, we know that after a finite number of computational steps, M will have reached output precision k. And then we can simply compare t to the number of input cells our simulation inspected.

Part (e) is trivial: if $\mathrm{LookAhead}_{M,\rho}(\bar{x},k)$ is approximated for some representatives $\overline{x_1},\overline{x_2},\ldots$, then certainly M needs at the very least $\mathrm{LookAhead}_{M,\rho}(\overline{x_i},k)$ computational steps to visit those input cells. Hence $\mathrm{Time}_{M,\rho}(\overline{x_i},k)$ bounds $\mathrm{LookAhead}_{M,\rho}(\overline{x_i},k)$. Taking limits on both sides proves (e). □

One glaring difference of corollary 4.10 with Weihrauch's lemma 7.1.2 is that in (a) and (b), Weihrauch's lemma states 'computable' rather than 'bounded by a computable function'. This is caused by the fact that Weihrauch looks at the exact number of input and output cells, whereas we are interested in the precision of the input and output. Another illustrating difference is part (d), which is decidable for us, but undecidable for Weihrauch, because we limited the representatives to consider.

A machine that computes a function $f : (\subseteq\mathbb{R}) \to \mathbb{R}$ can work economically by looking ahead only as far as is necessary: read only as much input as is needed to specify the output up to desired precision. Or, it can read many more symbols than is necessary. However, every computable function can be computed by a machine with minimum lookahead. We cite the following theorem from (Gordon and Shamir, 1983), adapted to our language.

4.11 **Theorem** *For every computable function $f : (\subseteq\mathbb{R}) \to \mathbb{R}$ and for every admissible representation ρ, there exists a Turing machine M computing f, such that for every (other) Turing machine M' also computing f, for every $\bar{x} \in \rho^{-1}(\mathrm{Dom}(f))$ and every $k \in \mathbb{N}$, we have $\mathit{LookAhead}_M(\bar{x},k) \leq \mathit{LookAhead}_{M'}(\bar{x},k)$.*

However, for some functions there is a trade-off between Time and LookAhead: decreasing LookAhead might considerably increase Time.

4.12 **Example** Let ρ an admissible representation, $f : (\subseteq\mathbb{R}) \to \mathbb{R}$ be computable by M with respect to ρ, and $k \in \mathbb{N}$ a precision. We define the *dependence* on $x \in \mathrm{Dom}(f)$ by

$$\mathrm{Dep}_M(x,k) = \min\{n \in \mathbb{N} \;:\; \forall_{\bar{x}\in\rho^{-1}(x),\bar{y}}\big[\,\bar{x} \simeq_n \bar{y} \;\Rightarrow\; M(\bar{x}) \simeq_k M(\bar{y})\,\big]\}.$$

(Recall that $\simeq$ denotes the prefix-relation of definition 2.29.) By definition 2.1, such n always exist, so that this is well-defined. Furthermore, from the proof of theorem 4.11 in (Gordon and Shamir, 1983) it is clear that Dep_M is computable.

We now construct a Turing machine M' that also computes f, but with $\mathrm{Dep}_M = \mathrm{LookAhead}_{M'}$, by re-evaluating the entire input up to precision $\mathrm{Dep}_M(x,k)$ (and no further!) for producing the k^{th} symbol of output. Note that we need not go any further than $\mathrm{Dep}_M(x,k)$ cells on the input tape because more input precision will not effect the output of symbol k.

In the worst case scenario, we see that an upper bound on $\mathrm{Time}_{M'}$ in terms of Time_M is given by $\mathrm{Time}^X_{M'}(k) = \sum_{i=1}^{k} \mathrm{Time}^X_M(i)$.

So we see there is a trade-off between LookAhead and Time, and algorithms that actually need this awful upper bound on Time for minimal LookAhead really exist. An example of such a ghastly algorithm is given in (Borwein and Borwein, 1987, page 170) to calculate π: for the next approximation, it needs to 're-evaluate all the previous ones'. (This method was used in the 1986 world record-computation of 29,360,000 digits of π by D.H. Bailey. So it might not be a smart algorithm in this context, it certainly does work. Another interesting and related algorithm is that of (Bailey et al., 1997), which can compute any digit in the hexadecimal expansion of π, without using any other digits!)

Complexity classes 4.3

Now we have a resource measure, the next step is to divide all 'problems' on feasibility; to divide all (computable) functions into classes which cost as much resources (as in Wilf, 1986 and Papadimitriou, 1994). These classes are called *complexity classes*, and the theory from here does not vary much from the classical case (appendix A).

Complexity class hierarchy

Let K be the class of Turing machines as in definition 2.1, and $\mu : K \times (\subseteq\Sigma^\omega) \times \mathbb{N} \to \mathbb{N}$ a *resource measure*. A *resource function* is a function $\mathbb{N} \to \mathbb{N}$, the thing to be measured to. Because we have chosen our definition of the standard resource measures Time and LookAhead so carefully, the definition of a complexity class is now pretty easy, almost natural.

4.13 **Definition (Complexity class)** Let an admissible representation ρ, a resource measure $\mu : K \times (\subseteq \Sigma^\omega) \times \mathbb{N} \to \mathbb{N}$, and a resource function $T : \mathbb{N} \to \mathbb{N}$ be given. The *complexity class* of T by K is

$$[T] = \{ f \in \mathscr{C} \mid \exists_{M \in K} \big[M \text{ computes } f, \text{ and } \forall_{\overline{u} \in \rho^{-1}(\mathrm{Dom}(f))} \forall_{k \in \mathbb{N}} \big[\mu(M, \overline{u}, k) \leq T(k) \big] \big] \}$$

Intuitively, $[T]$ is the class of all functions that are computable within resources T. Officially, we should speak of $[T]_\mu$, but mostly it will be clear whether $[T]$ means $[T]_{\mathrm{Time}}$, $[T]_{\mathrm{LookAhead}}$, or something else.
When $\mathscr{T}$ is a set of resource functions, we will also speak of $[\mathscr{T}] = \{f \in \mathscr{C} \mid \exists_{T \in \mathscr{T}} [f \in [T]]\} = \bigcup_{T \in \mathscr{T}} [T]$.

One difference with Weihrauch's definition of complexity is that he does not bound Time by $T(k)$ directly, but rather by $c \cdot T(k) + d$ for some constants $c, d \in \mathbb{N}$, so as to keep the robustness of the Turing Machine model. We will cover this by using the $\mathscr{O}$-symbol, introduced in the next section.

As a last note in this subsection, we cite from (Weihrauch, 2000, example 7.2.9) that there are computable functions with arbitrarily high complexity bounds: so certainly our theory, which slightly generalizes Weihrauch's, is not trivial.

4.14 **Remark (A computable function exceeding a complexity bound)** Let $t : \mathbb{N} \to \mathbb{N}$ be a complexity bound. There is a computable real number x that is not computable in Time t.

Notice that a resource measure is finite by definition, so this example implies that a complexity class is never all of $\mathscr{C}$! (After all, if M computes f, and $k \in \mathbb{N}$, then there is an $N \in \mathbb{N}$ such that after N steps, M has produced the first k symbols of $f(x)$ for every $x \in \mathrm{Dom}(f)$, see definition 2.1; for the LookAhead, apply corollary 4.10(e).)

$\mathscr{O}$-notation 4.4

In almost all discussions about complexity theory, you will come across Landau's $\mathscr{O}$-symbol and kindred. Informally, $f = \mathscr{O}(g)$ expresses that f is less complex than g in the long run. For example, $f = \mathscr{O}(g)$ and $g = \mathscr{O}(f)$ says that f is about as complex as g (and vice versa). Essentially, these notations grant us the ease of not having to count exactly; to say that some computation requires $\mathscr{O}(n^2)$ steps does not need such an elaborate proof as for exactly $3n^2 + n + 59824$ steps.

First, we define the order symbol for resource functions. That is, for functions $\mathbb{N} \to \mathbb{N}$.

4.15 **Definition (Order, Order of complexity)** Let S and T be resource functions $\mathbb{N} \to \mathbb{N}$. We say that S *is of order at most that of* T and write $S = \mathscr{O}(T)$ if

$$\exists_{N \in \mathbb{N}} \exists_{C \in \mathbb{N}} \forall_{n \geq N} [S(n) \leq C \cdot T(n)].$$

$S = \mathscr{O}(T)$ amounts to $\lim_{n \to \infty} \frac{S(n)}{T(n)} \leq C$ if defined.

Intuitively, $S = \mathscr{O}(T)$ when S is not larger than T in the long run. Next, we can 'overload' this $\mathscr{O}$-notation in a thousand and one ways, so we can not only speak of orders of resource functions, but also of orders of computable functions or orders of complexity classes.

4.16 **Definition (Order)** Let μ be an arbitrary resource measure.

(a) For a computable function $f : (\subseteq \mathbb{R}) \to \mathbb{R}$ and a resource function $T : \mathbb{N} \to \mathbb{N}$, we define
$$f = \mathscr{O}(T) \;\Leftrightarrow\; \exists_{S : \mathbb{N} \to \mathbb{N}} \big[f \in [S]_\mu \text{ and } S = \mathscr{O}(T) \big].$$
This expresses that f is computable within resources T.

(b) For a computable function $f : (\subseteq\mathbb{R}) \to \mathbb{R}$ and a set of resource functions $\mathscr{T}$, we define
$$f = \mathscr{O}(\mathscr{T}) \;\Leftrightarrow\; \exists_{T\in\mathscr{T}}\big[f = \mathscr{O}(T)\big].$$
This expresses that f is computable within resources $\mathscr{T}$.

(c) For two computable functions $f, g : (\subseteq\mathbb{R}) \to \mathbb{R}$, we define
$$f = \mathscr{O}(g) \;\Leftrightarrow\; \forall_{S:\mathbb{N}\to\mathbb{N}}\big[g = \mathscr{O}(S) \Rightarrow f = \mathscr{O}(S)\big].$$
This expresses that f is computable with less (or just as much) resources than g.

(d) For two complexity classes $[C]_\mu, [D]_\mu$, we define
$$[C]_\mu = \mathscr{O}([D]_\mu) \;\Leftrightarrow\; \forall_{f\in[C]_\mu, g\in[D]_\mu}\big[f = \mathscr{O}(g)\big].$$
This expresses that every function in $[C]_\mu$ is computable in less (or just as much) resources than any function in $[D]_\mu$.

(e) For a complexity class $[C]$ and a resource function $T : \mathbb{N} \to \mathbb{N}$, we define
$$[C]_\mu = \mathscr{O}(T) \;\Leftrightarrow\; \forall_{f\in[C]_\mu}\big[f = \mathscr{O}(T)\big].$$
This expresses that every function in $[C]$ is computable within resources T.

(f) For a complexity class $[C]_\mu$ and a set of resource functions $\mathscr{T}$, we define
$$[C]_\mu = \mathscr{O}(\mathscr{T}) \;\Leftrightarrow\; \exists_{T\in\mathscr{T}}\big[T = \mathscr{O}(\mathscr{T})\big].$$
This expresses that every function in $[C]$ is computable within resources $\mathscr{T}$.

4.17 **Example (Some trivial examples)** Consider the functions $S(n) = n^2 + n + 4$ and $T(n) = n^3 - 10$. Then $S = \mathscr{O}(T)$. Let $f(x) = 1$ and $g(x) = x^2$. Then $f = \mathscr{O}(S)$, $f = \mathscr{O}(T)$, $f = \mathscr{O}(\{S,T\})$, $f = \mathscr{O}(g)$, $[S] = \mathscr{O}([\{S,T\}])$, et cetera.

We will shamelessly abuse these notations at will, and hope the intention is clear after this. For example, we will speak of the complexity class $[f]$ of a computable function f, or even $[\mathscr{O}(f)]$. With $\mathscr{O}(n^2)$, where n is a free variable, we mean $\mathscr{O}(S)$, where S is the function $n \mapsto n^2$.

4.18 **Remark** Although functions of two variables should have two independent precision measures in the definition of complexity, it is customary to assume that both have the same precision. If this is not the case, we can pad the more efficient one until it is just as inefficient as the other one. For example multiplying two polynomials using the Fast Fourier Transform only works on polynomials whose coefficients are of length a power of two. Of course, the polynomials with this property are a minority. However, the usual trick is to just pad them with leading zeroes until there is a power of two of coefficients.

Just as in classical complexity theory (Papadimitriou, 1994), we can now derive a thousand and one lemmata about complexity classes, their completion, their behavior under various operations, and several important standard classes like P and NP. Because this is not of particular interest to us, we merely give an example.

Looking at the previous definitions, we see that $[C \cup D] = [C] \cup [D]$ (that is why we chose the existential quantor in definition 4.16(b)), and $f \in [C] \Rightarrow f = \mathscr{O}([C])$. Because of this second property, we have a way to close a complexity class under the '$\mathscr{O}$-operation'.

4.19 **Definition (Closure of a complexity class)** Let $C \subseteq RF$ be a set of resource functions. We say that a complexity class $[C]$ is closed, if $[C] = \mathscr{O}([C])$. We denote the closure of $[C]$ as $\overline{[C]}$.

Representations 4.5

So far, we have looked at Turing machines which receive input in the same representation as they output. What if we have a Turing machine that computes a function, but in which the input is interpreted with a different representation than the output?

Conversion between representations is basically a *string function*: it is about a machine that handles concrete sequences of symbols. That those symbols happen to correspond to abstract objects (a real number) is something that is only interpreted so by the user. Hence, naively, we do not need the elaborate definitions to talk in terms of $\mathbb{R}$ instead of in strings (representations of $\mathbb{R}$). However, we do have to worry about whether this conversion preserves the 'efficiency' of a represented object, since our definition 4.5 and definition 4.6 of resource measures hinges hereupon.

Upon careful rereading of definition 4.5, we see that there is only one precision measure involved, for the input representation, because the output-side is measured *in* $\mathbb{R}$, not in the output representation. But we can easily restate definition 4.5 for the case that the input and output representations differ.

4.20 **Definition (Output precision measure)** An *output precision measure* of a representation $\rho : (\subseteq\Sigma^\omega) \twoheadrightarrow \mathbb{R}$ is a function $e : (\subseteq\Sigma^\omega) \times (\subseteq\Sigma^\omega) \times \mathbb{N} \to \mathbb{R}$, such that

$$\forall_{k\in\mathbb{N}}\forall_{\bar{x}\in\mathrm{Dom}(\rho)}\forall_{\bar{y}\in\mathrm{Dom}(\rho)}\forall_{y\in\rho(\bar{y}|_k)}\left[\ |\rho(\bar{x})-y| \le e(\bar{x},\bar{y},k)\ \right]$$

4.21 **Definition (Time function)** Let $\rho,\sigma : (\subseteq\mathbb{R}) \twoheadrightarrow \mathbb{R}$ be two representations of $\mathbb{R}$, and let d be a precision measure of ρ, and e an output precision measure of σ. Suppose $f : (\subseteq\mathbb{R}) \to \mathbb{R}$ is computable by a Turing machine M, in which the input is to be interpreted with ρ, and the output with σ. For $\bar{x} \in \rho^{-1}(\mathrm{Dom}(f))$ and a precision $k \in \mathbb{N}$, we define $\mathrm{Time}_M(\bar{x},k)$ as the number of steps M needs to produce an output (prefix) $\bar{y}$ upon input $\bar{x}$ such that $e(\bar{y},M(\bar{x}),k) \le d(\bar{x},k)$.

> Notice, just like in the remark after definition 4.3, that the $\rho(\bar{y}|_k)$ in an output precision measure should be interpreted right; for all practical purposes, we can assume that elements of $\mathbb{Q}$ can be stored in a single tape cell. Though this would of course require an infinite alphabet, it is of no use to clutter the definitions further.

Other resource measures, like LookAhead, can be restated analogously.

Now we know what it means, we can think about the complexity of the representation conversions in theorem 3.28. Basically, what we are asking for is the Time-complexity of the identity function, in which the input is interpreted in one representation, and the output in another.

4.22 **Theorem (Complexity of representation conversion)** *The representations ρ_{NI}, ρ_{DC}, ρ_{CS}, and ρ_{CF} are all convertible into each other in polynomial Time.*

PROOF We verify that all the transformations from the cycle in the proof of theorem 3.28 are polynomial in Time. We spell out one verification in detail, and treat the rest more coarsely.

"$\rho_{\mathrm{NI}} \Rightarrow \rho_{\mathrm{DC}}$" Given input $\ll((a_0,b_0,c(0)),(a_1,b_1,c(1)),\ldots)\gg$, all this transformation does, is output $\ll(a_0,c(0),2^{-0},a_1,c(1),2^{-1},\ldots)\gg$.

An input precision measure d for ρ_{NI} is $d(\bar{x},k) = c(k) \ge \max\{|a_k - \rho_{\mathrm{NI}}(\bar{x})|, |b_k - \rho_{\mathrm{NI}}(\bar{x})|\}$, where $\bar{x} = \ll((a_0,b_0,c(0)),(a_1,b_1,c(1)),\ldots)\gg$. An output precision measure e for ρ_{CS} is $e(\bar{x},\bar{y},k) = c(k) \ge |a_k - \rho_{\mathrm{CS}}(\bar{x})|$, where $\bar{x} = \ll(a_0,c(0),a_1,c(1),\ldots)\gg$.

Substituting, we see that to meet the condition

$$e(\ll(a_0,c(0),\ldots,a_n,c(n))\gg,\ll(a_0,c(0),\ldots)\gg,k) \le d(\ll((a_0,b_0,c(0))\ldots)\gg,k),$$

we need $c(n)$ to be smaller than $c(k)$. That is, $\mathrm{Time}_M(\bar{x},k)$ is the number of steps it takes to copy $a_0,c(0),\ldots,a_k,c(k)$ from the input tape to the output tape and skip $b_0,\ldots,b_k$: those are $3k$ steps. It is true that these a_i, b_i and $c(i)$ are elements of $\mathbb{Q}$, so there is no knowing how many actual steps it takes to copy one of them. But because of our definition of precision measure (see the remarks after definition 4.3 and definition 4.20), for all practical purposes it is all right to pretend elements of $\mathbb{Q}$ can be stored in one tape cell.

All in all, we see that $\mathrm{Time}_M(k) = 3k = \mathcal{O}(k)$ is surely polynomial.

"$\rho_{DC} \Rightarrow \rho_{CS}$" The same line of reasoning as in "$\rho_{NI} \Rightarrow \rho_{DC}$" shows that to output as much information as in the first k parts of the input, $(a_0, c(0), \ldots, a_k, c(k))$, is to output $(a_0, 0, 2c(0), \ldots, a_k, k, 2c(k))$. That is, we need k multiplications by 2, we need to copy k fractions from input to output, and we need to write k constants. In the next chapter, we will see these multiplications cost 2 steps each (we are pretending that any fraction can be represented by a single letter of the alphabet, remember?). In total, we have $\text{Time}_M(k) = 4k$, which is polynomial.

"$\rho_{CS} \Rightarrow \rho_{CF}$" For the kth precision of this transformation, we need to compute $2^{-0}, \ldots, 2^{-k}$ as the c'; this takes k multiplications by a constant, which is polynomial. Furthermore, we are required to compute $\frac{1}{a_{c(1)} - b_0}$ up to $\cfrac{1}{\cfrac{1}{\vdots} - b_k}$. Doing this recursively, that costs one subtraction and one inversion for the first stage, and two inversions and one subtraction for all the other stages. That makes a total of $3k - 1$ arithmetic operations, which is polynomial.

"$\rho_{CF} \Rightarrow \rho_{NI}$" This time, we need to output $\frac{2}{q_0}, \ldots, \frac{2}{q_k}$, which means k divisions by a constant. Furthermore, we need to compute $\frac{p_n \pm 1}{q_n}$ k times, which means k subtractions by a constant and k additions by a constant. Finally, we need to nest the intervals, which we can accomplish recursively: we only need to intersect $[\frac{p_n - 1}{q_n}, \frac{p_n + 1}{q_n}]$ with the previous $[\frac{p_{n-1} - 1}{q_{n-1}}, \frac{p_{n-1} + 1}{q_{n-1}}]$, which comes down to computing $k - 1$ minimums and $k - 1$ maximums. In the next chapter, we will see that, like arithmetic, this is also a fast operation. We conclude that the time complexity of this transformation is polynomial. □

This is an indication that we are on the right track: in the classical theory, the representation of the problem is an essential part of its complexity, as is evident from the following example. Apparently, we have allowed and disallowed the right representations for a uniform theory of computational complexity.

4.23 **Example (Knapsack problem)** The knapsack problem is, when given a sum b and a set of weights $a_1, \ldots, a_n$, to find the weights $x_1, \ldots, x_n$ that were used to generate the sum $a_1x_1 + \cdots + a_nx_n = b$ (Wilf, 1986). This problem can be solved in the dumb way, by trying all possible x_i. This will, in the worst case, take about 2^n tries. When, as usual, we are required to construct a Turing machine that computes the x_i upon input $b, a_1, \ldots, a_n$ in binary representation, this method thus costs $\mathcal{O}(2^n)$.

However, (Garey and Johnson, 1979, page 96) learns that although the Knapsack problem is NP-complete, it is not *strongly NP-complete*, meaning that when we represent the input unary rather than binary, it is computable by a machine that works in polynomial time!

In our definitions, we have tried to work around this by looking at 'the information stored in a prefix', rather than the length of a prefix. In the knapsack problem with unary input representation, the information stored in those 2^n tape cells is still n – in our language, a precision measure for the unary representation would be $e(\bar{x}, k) = \log_2(k)$, although that would somewhat be abuse of the language.

4.24 **Corollary (Admissible representations induce similar complexity classes)** *Let C be a class of functions $\mathbb{N} \to \mathbb{N}$, including the polynomials, and $\rho : (\subseteq \Sigma^\omega) \twoheadrightarrow X$ and $\rho' : (\subseteq (\Sigma')^\omega) \twoheadrightarrow X$ be admissible representations (with respect to each other). Then the complexity class $[C]_\rho$ under one representation is exactly the complexity class $[C]_{\rho'}$ under the other representation.*

PROOF Suppose that a computable function $f : X \to X$ is in the complexity class $[C]_\rho$. Because ρ is admissible with respect to ρ', there is a polynomial-time computable function $p : \Sigma^\omega \to (\Sigma')^\omega$ that transforms ρ into ρ'. Since composition respects computability (by lemma 2.11), we see that $p \circ f \circ \rho$ is in $[C]_{\rho'}$.

The other way around is, of course, analogous. □

CHAPTER

5 Examples

We can use concrete algorithms (Burden and Faires, 2001, for example) to obtain (very concrete) time limits on some functions or operators. Since this would imply making all the gory details of the Turing Machine and the representation of the input and output explicit in order to count the exact number steps, this job is a very tiresome one.

Since we have eliminated this need to be overly precise with our complexity classes and $\mathcal{O}$-notation, we can now determine the complexity class of a specific method more or less fluently. This gives an upper bound for the intrinsic complexity.

(Weihrauch, 2000) proves the computability of numerous real functions, like addition, subtraction, multiplication, division, sine (and cosine), taking roots. Furthermore minimalization, maximalization, integration are computable real function operators (but differentiation not just like that). Let us take a few of these examples to determine an upper bound on the complexity thereof.

Arithmetic 5.1

The most basic things in real computation are arithmetic operations. We determine the complexity of addition and multiplication, using the base-2 negative digit expansion representation. In this case, that representation seems the most appropriate, since the algorithm we all learned in primary school to add two numbers works 'digit wise'. Since this is the first real example of the theory, we spell it out in some detail.

Addition

First, we consider a machine M to add two real numbers between 0 and 1. Later, we enlarge this to addition on all of $\mathbb{R}$. Let us have a look at a sample calculation first.

		-1	1	2	1		-2	-2	-1	-2	1				
0	.	1	1	$\overline{1}$	0	0	1	1	$\overline{1}$	$\overline{1}$	0	$\overline{1}$	1	$\cdots$	
0	.	1	$\overline{1}$	0	$\overline{1}$	0	0	1	$\overline{1}$	$\overline{1}$	$\overline{1}$	1	0	$\cdots$	+
1	.	0	$\overline{1}$	0	1	1	0	$\overline{1}$	$\overline{1}$	0	$\overline{1}$	$\cdot$	$\cdot$	$\cdots$	

This signed digit representation has the advantage that we can add left to right: the carries (the small gray numbers at the top) ripple through to the right. This is a consequence of the fact that this representation is admissible (it 'leaves enough room to compensate') – note that this is not the only possible calculation: $1.0\overline{1}1\overline{1}10\overline{1}\overline{1}0\overline{1}$ would also have been a correct sum. Notice that the above calculation is correct: in decimal notation, the summands are 0.642578125 and 0.18896484375, and the sum we calculated is 0.8310546875, which is indeed equal to the actual sum, 0.83154296875, when rounded to a fraction with denominator 1024. Now, let us look at how this algorithm works exactly, in order to determine its Time complexity.

Let two numbers x and y be given on two input tapes, with representatives $x = \rho_{B-2}(\ll(0, x_{-1}, x_{-2}, \ldots)\gg)$ and $y = \rho_{B-2}(\ll(0, y_{-1}, y_{-2}, \ldots)\gg)$, with $x_i, y_i \in \Sigma = \{\overline{1}, 0, 1\}$.

The first thing M does is to check whether $x+y \geq 1$, by simply looking at the two foremost digits of x and y. If we write $\bar{x} = 0.x_{-1}x_{-2}\cdots$ and $\bar{y} = 0.y_{-1}y_{-2}\cdots$, then the sum $0.x_{-1}x_{-2}+0.y_{-1}y_{-2}$ is at most $\frac{1}{2}$ from the nearest integer. Hence we can write that sum as $z_0.0+0.r_1r_2$ for some z_0 and r_1, r_2.

More precisely, M determines a $z_0 \in \Sigma$ and a carry $r_0 \in \{-2,-1,0,1,2\}$ such that

$$\begin{aligned} \rho_{B-2}(\ll(1,z_0,0,0,0,\ldots)\gg) + r_0 \cdot 2^{-2} &= \rho_{B-2}(\ll(0,x_{-1},x_{-2},0,0,0,\ldots)\gg) \\ &+ \rho_{B-2}(\ll(0,y_{-1},y_{-2},0,0,0,\ldots)\gg). \end{aligned} \tag{5.1}$$

So, to compute the first digit of the output, M only needs to check $3 \cdot 5 = 10$ possibilities.

This process continues for the subsequent digits, incorporating the carries r_i. So in the schema of our sample calculation, we would have

r_0	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}	r_{11}	r_{12}		
0	x_{-1}	x_{-2}	x_{-3}	x_{-4}	x_{-5}	x_{-6}	x_{-7}	x_{-8}	x_{-9}	x_{-10}	x_{-11}	x_{-12}	$\cdots$	
0	y_{-1}	y_{-2}	y_{-3}	y_{-4}	y_{-5}	y_{-6}	y_{-7}	y_{-8}	y_{-9}	y_{-10}	y_{-11}	y_{-12}	$\cdots$	+
z_0	z_{-1}	z_{-2}	z_{-3}	z_{-4}	z_{-5}	z_{-6}	z_{-7}	z_{-8}	z_{-9}	z_{-10}	z_{-11}	z_{-12}	$\cdots$	

For all subsequent digits z_{-i}, $i = 1,2,\ldots$ of the output, M determines numbers $z_i \in \Sigma$ and $r_i \in \{-2,-1,0,1,2\}$ such that

$$2r_{i-1} + x_{-i-2} + y_{-i-2} = 4z_{-i} + r_i. \tag{5.2}$$

Since $r_{i-1} \in \{-2,-1,0,1,2\}$, we see that $|2r_{i-1} + x_{-i-2} + y_{-i-2}| \leq 6$, so this is always possible. To determine z_{-i} and r_i we only need to check 15 possibilities again.

By induction on equations (5.1) and (5.2), we see that

$$\sum_{i \leq n+2} x_{-i} \cdot 2^{-i} + \sum_{i \leq n+2} y_i \cdot 2^{-i} = z_0 + \sum_{i \leq n} z_{-i} \cdot 2^{-i} + r_n \cdot 2^{-n-2} \tag{5.3}$$

for all $n \in \mathbb{N}$. Taking the limit for $n \to \infty$, we see that $\rho_{B-2}(\bar{x}) + \rho_{B-2}(\bar{y}) = \rho_{B-2}(\ll 1, z_0, z_{-1}, z_{-2}, \ldots \gg)$, hence M indeed computes addition.

To compute k digits of output, we thus need at most $15(k-1)+15 = 15k$ steps. Thus this machine operates in linear Time. Furthermore, to compute k digits of output, we only need to know $k+2$ digits of the input: $\text{LookAhead}_M((\bar{x},\bar{y}),k) \leq k+2$.

5.1 **Lemma (Complexity of local addition)** *Addition of two real numbers between 0 and 1 is computable, is linear in Time, and has LookAhead* $((\bar{x},\bar{y}),k) \leq k+2$.

Now we extend this to addition of all numbers, instead of only numbers between 0 and 1. Again, let two numbers x and y be given on two input tapes, with representatives $x = \rho_{B-2}(\ll(m,x_m,x_{m-1},\ldots)\gg)$ and $y = \rho_{B-2}(\ll(n,y_n,y_{n-1},\ldots)\gg)$. First we compare the two natural numbers m and n. Suppose, without loss of generality, that $m > n$. Then we pad y with $m-n$ zeroes, shift the 'decimal' point, and use our machine to add numbers between 0 and 1. That is, we add $x' = 2^{-m}x = \rho_{B-2}(\ll(0,x_m,x_{m-1},\ldots)\gg)$ and $y' = 2^{-m}y = \rho_{B-2}(\ll(0,0,0,\ldots,0,y_n,x_{n-1},\ldots)\gg)$. Obviously, we thus obtain $z' = x'+y' = 2^{-m}(x+y)$. So we only have to shift the point back m places to the left to get $z = x+y$.

Let us count the number of steps needed to perform all this shifting. The comparison of m and n takes at most $\max\{\log_2(m),\log_2(n)\}$ steps. The padding of y then takes at most $m-n$ steps. Then, our 'local addition machine' takes its linear number of steps. And finally the shifting of the point takes m steps. We also didn't need more LookAhead than m, and the LookAhead of our local addition machine. In total, we have proved the following theorem.

5.2 **Theorem (Complexity of addition)** *Addition is computable. Furthermore, if* $x,y \in \mathbb{R}$ *and* $k \in \mathbb{N}$, *then* $\text{Time}_{Add}((x,y),k) \leq c \cdot (\log_2(\max\{\lceil x\rceil,\lceil y\rceil\})+k)+d$ *for some constants* $c,d \in \mathbb{N}$, *and* $\text{LookAhead}_{Add}((x,y),k) \leq \log_2(\max\{\lceil x\rceil,\lceil y\rceil\})+k+2$.

So, the Time-complexity of real addition is 'linear in the integer parts of the summands' (Weihrauch, 2000, example 7.3.2).

Multiplication

If we handled multiplication in the same way as we analyzed addition, with the signed digit representation, we would soon have one big tangled mess on our hands. This why our definition of complexity is preferable over Weihrauch's. Using the Nested Interval representation, we show that real multiplication can be computed fast (i.e. in polynomial time, in this case quadratically), using the machine M of example 2.4.

We already know that multiplying two natural numbers m and n costs $\mathcal{O}(m \cdot n)$.[1] Since Euclid's algorithm is linear, we can thus deduce that multiplying two fractions $\frac{p}{q}$ and $\frac{r}{s}$ (with product $\frac{pr/\gcd(pr,qs)}{qs/\gcd(pr,qs)}$) also is of order $\mathcal{O}(\frac{p}{q} \cdot \frac{r}{s})$. In terms of input length: if $\frac{p}{q}$ takes m tape cells to store, and $\frac{r}{s}$ takes n, then multiplying them takes $\mathcal{O}(m \cdot n)$ steps.

Let two numbers $x, y \in \mathbb{R}$ be given on two input tapes, in Nested Interval representation, say $\bar{x} = ((p_0, q_0, c(0)), (p_1, q_1, c(1)), \ldots)$ and $\bar{y} = ((r_0, s_0, d(0)), (r_1, s_1, d(1)), \ldots)$.

> For such a low-level function as multiplication, that is intimately related to the representation, we need to look at actual tape cells, not the "$\mathbb{Q}$-cells" of the remark after definition 4.3. Assume therefore, *without loss of generality*, that $k \in \mathbb{N}$ is a tape cell that is just on the end of a $(p_i, q_i, c(i))$ and a $(r_i, s_i, d(i))$ triple. (The K from definition 4.3 could be considered a 'big K', whereas this k is small. So K counts the triples $(p, q, c(i))$, and k counts actual tape cells. In the limit for $k \to \infty$ or $K \to \infty$, it does not matter which k we look at.)

A precision measure for ρ_{NI} is $e((\bar{x}, \bar{y}), K) = \max\{c(K), d(K)\}$. To determine the Time-complexity, we need to count the number of steps K that M needs to output a prefix $\bar{z}$ such that $|\frac{pr}{qs} - \rho_{\mathrm{NI}}(\bar{z})| \leq \max\{c(K), d(K)\}$. Without loss of generality, we can assume that both $c(K) < 1$ and $d(K) < 1$, because $\lim_{K \to \infty} c(K) = 0$ and $\lim_{K \to \infty} d(K) = 0$, and we are only interested in the limit. So, surely, $\frac{p_K}{q_K} \cdot \frac{r_K}{s_K}$ suffices as z. The number of steps this takes, as we saw before, is $\mathcal{O}(K^2)$.

Afterwards, we need to make sure that the sequence of intervals we output is nested. Just like in step "$\rho_{\mathrm{CF}} \Rightarrow \rho_{\mathrm{NI}}$" of the proof of theorem 4.22, if we do this recursively, that takes $k-1$ maximums and minimums *of fractions*, which is of course $\mathcal{O}(k)$. We have found the following theorem.

5.3 **Theorem (Complexity of multiplication)** *Multiplication is computable. Furthermore, if $x, y \in \mathbb{R}$ and $k \in \mathbb{N}$, then $\mathit{Time}_{\mathit{Mult}}((x, y), k) = \mathcal{O}(k^2)$.*

Using the Nested Interval representation has enabled us to use the fact that we already know how to multiplicate fractions, and what Time-complexity that takes. This is an advantage over using the signed digit representation, because then we had to do that all over again, including all the details like carries, as we saw with addition[2].

The image on the cover is an illustration of the machine we just analyzed.

Maximum 5.2

We have already met computations of maximums and minimums of fractions before, but how about computing the maximum of two real numbers? Let us pick yet another representation to perform this analysis: the Dedekind Cut representation.

5.4 **Theorem (Complexity of real maximum)** *The function $\max : \mathbb{R}^2 \to \mathbb{R}$ is computable, with*
$\mathit{Time}_{\max}((x, y), k) = \mathcal{O}(k^2)$ and
$\mathit{LookAhead}_{\max}((x, y), k) \leq k$ for all $k \in \mathbb{N}$.

PROOF First, we construct a machine M that computes max, and then we determine its complexity.

Let two numbers $x = \rho_{\mathrm{DC}}(\ll(a_1, c(1), a_2, c(2), \ldots)\gg)$ and $y = \rho_{\mathrm{DC}}(\ll(b_1, d(1), b_2, d(2), \ldots)\gg)$ be given on two input tapes. M operates in stages $i = 1, 2, \ldots$. In stage i, M simply compares a_i and b_i. If

[1] This can of course be done faster, for example by a Schönhage-Strassen algorithm in $\mathcal{O}(n \cdot \log(n) \cdot \log(\log(n)))$.
[2] This aspect of our approach reminds a bit of the real-RAM-model.

$a_i \leq b_i$, then it outputs $b_i, d(i)$. If $a_i > b_i$, then it outputs $a_i, c(i)$. Clearly, this output is again in Dedekind Cut representation (notice that the modulus of convergence is still computable), and it is a representative of $\max\{x,y\}$. Hence M indeed computes max.

As to the LookAhead, we clearly see that to output the first $2K$ symbols (K times $\max\{a_i,b_i\}$, and K times the corresponding modulus of convergence), we only have to see the first $2K$ symbols of the input. Hence $\text{LookAhead}_{\max}((x,y),K) \leq K$ for all $K \in \mathbb{N}$.

As to the Time; to obtain the same precision as the input has after K symbols, we need to copy $2K$ symbols[3]. This takes $2K$ steps. However, to compare to fractions $\frac{p}{q}$ and $\frac{r}{s}$, we need to perform two multiplications: after all, we want to check $\frac{p}{q} < \frac{r}{s} \Leftrightarrow ps < qr$. Comparing two integers can well be linear in Time, but we cannot ignore the multiplications. Therefore, in total, $\text{Time}_{\max}((x,y),K) = \mathcal{O}(K^2)$. □

Trigonometric functions 5.3

By Taylor's theorem, we know that for all $n \in \mathbb{N}$, $\sin(x) = P_n(x) + R_n(x)$, where

$$P_n(x) = \sum_{i=0}^{n} \frac{(-1)^{2i+1}}{(2i+1)!} x^{2i+1},$$

and $|R_n(x)| \leq \frac{1}{(2n+3)!}$. This yields a straightforward method to compute the sine (this is how desk calculators do it).

In the previous subsections, we have constructed a machine M_{add} that operates in Time $\mathcal{O}(k)$, and machine M_{mult} that operates in Time $\mathcal{O}(k^2)$. Though we did not explicitly mention it, division can be done just as fast as multiplication (Weihrauch, 2000, theorem 7.3.12), so there is also a machine M_{div} that operates in Time $\mathcal{O}(k^2)$. From those, we construct a machine M that computes $P_n(x)$, and refine that for consecutive n. In the 'end', we will get a representative for $\sin(x)$ on the output tape. It does not even matter which representation ρ we use, as long as it is admissible.

What is the Time-complexity of this M? Well, if the input $\bar{x}$ has precision $e(\bar{x},k) = \varepsilon$, then we need to find the number of steps M takes to compute $P_n(\bar{x})$, where

$$|\sin(x) - \rho(P_n(\bar{x}))| \leq \varepsilon.$$

We overestimate:

$$\begin{aligned} |\sin(x) - \rho(P_n(\bar{x}))| \leq |R_n(x)| \leq \frac{1}{(2n+3)!} \leq \varepsilon \quad &\Leftarrow \quad (2n+3)^{2n+3} \geq (2n+3)! \geq \varepsilon^{-1} \\ &\Leftarrow \quad (2n+3)^2 \geq (2n+3)\log(2n+3) \geq -\log(\varepsilon) \\ &\Leftarrow \quad 2n+3 \geq -\log(\varepsilon) \\ &\Leftarrow \quad n \geq -\log(\varepsilon). \end{aligned}$$

So, M needs to perform four multiplications, an addition, and a division by a constant, at least $-\log(k)$ times. Using the machines M_{mult}, M_{add} and M_{div} above, we see that $\text{Time}_{\sin}(\bar{x},k) \leq k^2 \log(k)$.

5.5 **Theorem (Complexity of the sine)** *The sine is computable, and* $\text{Time}_{\sin}(\bar{x},k) \leq k^2\log(k)$.

The same holds for other trigonometric functions, like the cosine, and tangent, with an analogous Taylor expansion. In even more generality, we can state the following theorem.

5.6 **Theorem (Complexity of real-analytic functions)** *Let* $f : (\subseteq\mathbb{R}) \to \mathbb{R}$ *be* C^∞. *If* $f^{(k)}$ *is computable for all* $k \geq 1$, *then* f *itself is also computable.*

[3] At each stage, we pick the largest fraction of both inputs. Therefore, the precision can only get higher. If we were to compute a minimum, however, this approach would not be the most practical. It would be more elegant to use a representation that approaches from the top – like Nested Intervals.

The Time-complexity of a general C^∞ function, however, is more difficult to state. In the above example of the sine, all the derivatives were 'easy' functions. In general, we do not know how much Time it takes to compute them. We will see more about this when discussing integration.

Extracting roots 5.4

We are going to compute $x \mapsto \sqrt{x}$ with a different method than in example 2.5. This time, we apply the Newton-Raphson method to the function $f(p) = p^2 - x$. This technique says that

$$p_0 = x, \quad p_{n+1} = p_n - \frac{f(p_n)}{f'(p_n)} = p_n - \frac{p_n^2 - x}{2p_n}$$

is a good approximation to $\sqrt{x}$, with approximation error

$$|p_{n+1} - \sqrt{x}| \leq \mathcal{O}(|p_n^2|). \tag{5.4}$$

So, given x on an input tape, we copy $p_0 = x$ to a second tape, and consecutively compute $p_1, p_2, \ldots$. Every step in this computation takes two subtractions, two multiplications and one division. How many steps are needed? If $\rho(\bar{x})$ differs less than $e(\bar{x},k)$ from x, then $|p_{n+1} - M(\bar{x})| \leq \mathcal{O}(|p_n^2| + \sqrt{e(\bar{x},k)})$, by (5.4).

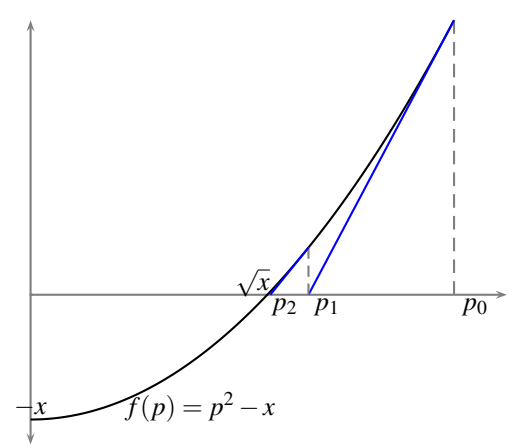


Figure 5.1: Illustration of the Newton-Raphson method to approximate $\sqrt{x}$.

So, n steps are needed, where n satisfies $\mathcal{O}(|p_n^2 + \sqrt{\varepsilon}|) = \mathcal{O}(\varepsilon)$, i.e. $\mathcal{O}(|p_n^2|) = \mathcal{O}(\varepsilon)$, where $\varepsilon = e(\bar{x},k) = \sqrt{p_0}$. So we can simply take $n = 1$, assuming, without loss of generality, that $p_1 < 1$. In other words: the kth approximant from the input is already good enough to compute the the kth approximant of the output. That computation thus takes two subtractions, two multiplications and one division. That makes a total of $2\mathcal{O}(k) + 2\mathcal{O}(k^2) + \mathcal{O}(k^2)$.

5.7 **Theorem (Complexity of the square root)** $Time_{\sqrt{\cdot}}(\bar{x},k) = \mathcal{O}(k^2)$.

More generally, this method yields a way to compute a pre-image $f^{-1}(x)$, given a computable $f : \mathbb{R} \to \mathbb{R}$ and computable real number x (Borwein and Borwein, 1987). Notice that we did not have to commit ourselves to any representation, since we already knew how (fast) to compute subtraction, multiplication and division.

Integration 5.5

Integration is an example that is essentially different than the approximative methods of numerical analysis that were the basis of all other examples in this chapter. It is also different in the sense that it is a function operator rather than a function. For a computable $f : [0,1] \to \mathbb{R}$, consider the function operator

$$F(f) = \int_0^1 f(x)dx$$

First, we give a method to compute F, and then we will consider complexity.

> Once we know that this F is computable, also $\mathrm{Int}(f,a,b) = \int_a^b f(x)dx$ is, and even $\mathrm{Prim}(f) = g$, where $g(y) = \int_a^y f(x)dx$ for a fixed computable $a \in \mathbb{R}$, is computable (Weihrauch, 2000, page 182).

The machine M we construct to compute F will take input in Rational Polygon representation, and yield output in Nested Interval representation. So, assume we are given a representative of f on the input tape; rational polygons p_n enclosing f arbitrarily narrowly between 0 and 1.

Well, we already know how to integrate these rational polygons: being piecewise linear, they have simple anti-derivatives on those pieces. Say that the rational polygon p_n is linear between the points $0 = x_0 \leq \ldots \leq x_m = 1$. Then we know that

$$I_n = \int_0^1 p_n(x)dx = \sum_{i=1}^{m} \frac{f(x_i) - f(x_{i-1})}{2}(x_i - x_{i-1})$$

Notice that this is computable via the Apply function operator (see lemma 2.24).

And as n increases, p_n gets closer to f, so I_n gets closer to the desired integral:

$$\begin{aligned} |F(f) - I_n| &= |\int_0^1 (f(x) - p_n(x))dx| \\ &\leq \int_0^1 |f(x) - p_n(x)|dx \\ &\leq \int_0^1 |2^{-n}|dx = 2^{-n}. \end{aligned}$$

Hence M indeed computes F.

5.8 **Lemma (Integration is computable)** *The real functional Int* $: \mathscr{F}([0,1]) \to \mathbb{R}, f \mapsto \int_0^1 f(x)dx$ *is computable.*

How much Time does this cost? Well, a precision measure for the Rational Polygon representation is $e(\overline{f},k) = 2^{-k}$. So we need to count the number of steps M needs to compute I_n, where n satisfies $|F(f) - I_n| \leq 2^{-k}$. In other words, where $n = k$. That is, we need to count the number of steps M needs to compute I_k.

For every prefix symbol of the input, we need to perform one division by 2 (which is quadratic in Time), $2m$ subtractions (linear in Time) *and* m *evaluations of* f. Analyzing the complexity of function evaluation (a complexity counterpart of lemma 2.24) does not give much results with our current rational polygon representation, mostly because we cannot easily state m in terms of k without ridiculously useless overestimations[4]. We could of course improve the representation, but that would take us outside the scope of this thesis. If we stick to this representation, we need some more information about f. For example, we can use the following lemma from (Müller, 1987).

5.9 **Lemma (Complexity of Taylor-coefficients)** *If* $f(x) = \sum_{i=0}^{\infty} a_i x^i$ *is computable in Time* $t : \mathbb{N} \to \mathbb{N}$,

- $t(m) \geq t(n)$ *whenever* $n \geq m$, *and*
- *there are constants* $N, c \in \mathbb{N}$ *such that* $t(N) > 0$ *and* $2t(n) \leq t(2n) \leq c \cdot t(n)$ *for all* $n \geq N$,

then the a_i *are uniformly computable in Time* $\mathscr{O}(k \cdot t(k))$. [5]

5.10 **Theorem (Complexity of integration)** *Let* $f : [0,1] \to \mathbb{R}$ *be real-analytic and computable in Time* $t : \mathbb{N} \to \mathbb{N}$. *If* t *is non-decreasing and satisfies*

$$\exists_{N,c \in \mathbb{N}} \forall_{n \geq N} \left[t(N) > 0 \quad \text{and} \quad 2t(n) \leq t(2n) \leq c \cdot t(n) \right],$$

then the primitive function $x \mapsto \int_0^x f(y)dy$ *is computable in Time* $\mathscr{O}(k^2 t(k))$.

PROOF (sketch) Suppose $f(x) = \sum_{i=0}^{\infty} a_i x^i$. According to the previous lemma, the a_i are then computable in Time $\mathscr{O}(k \cdot t(k))$. Then $\int_0^x f(y)dy = \sum_{i=0}^{\infty} a_i \int_0^x y^i dy = \sum_{i=0}^{\infty} \frac{a_i}{i+1} x^{i+1}$. Define $b_0 = 0$ and $b_{i+1} = \frac{a_i}{i+1}$. Because of the Time-bound on the a_i, and the fact that division is computable in Time $\mathscr{O}(k^2)$, b_i is computable in Time $\mathscr{O}(k^2 \cdot t(k))$. Since multiplication is of order k^2, and addition is linear in Time, we can thus compute $P_k(x) = \sum_{i=0}^{k} b_i x^i$ in Time $\mathscr{O}(k^2 \cdot t(k))$. A similar derivation as in theorem 5.5 now leads to the desired result. □

[4] We will see another hint at why this is so in theorem 7.12.

[5] A time bound t that satisfies the hypothesis of lemma 5.9 is called *regular*.

CHAPTER

6 Distributions

The theory of distributions arose around 1950, and became standard equipment in mathematical analysis. Distributions are a generalization of functions: the continuous functions can be regarded as a subset of them. Conversely, every distribution can be approximated with infinitely differentiable functions. Analytically, the calculus of distributions develops more smoothly: for example, the derivative of a distribution is again a distribution. One can compare this with the extension of $\mathbb{Z}$ to $\mathbb{Q}$, in which for every x, y with $y \neq 0$ the quotient $\frac{x}{y}$ exists.

But can the theory of computability and complexity also run on distributions? If so, does it run as smooth as with 'ordinary' functions?

What are distributions? 6.1

6.1 **Example** Consider the function $f : \mathbb{R} \to \mathbb{R} : x \mapsto |x|$. It is continuous on all of $\mathbb{R}$, and differentiable on $\mathbb{R} - \{0\}$. Though f is not differentiable at $x = 0$, it seems nevertheless natural to say that $f'(x) = \mathrm{sgn}(x)$, with $f'(0)$ undefined (nor really relevant).

On the other hand, we also want $f''(x) = 0$ on $\mathbb{R} - \{0\}$. But $f''(0)$ should not vanish, since then f' would be constant, thus f linear, which is incorrect!

This f' is an example of a problem where distributions offer a solution (Duistermaat, 1991).

Test functions

The function in the previous example had a special point where it is not differentiable: a singularity. Singularities of functions can be weakened by moving them to and fro, and then averaging with a weight function $\phi(y)$ that depends on the distance y over which the original function moved.

6.2 **Definition (Weight function)** A function $\phi : \mathbb{R} \to \mathbb{R}$ is called a weight function, if

- $\phi(x) \geq 0$ for all $x \in \mathbb{R}$,
- it is finitely supported: $\exists_{c>0}[|x| \geq c \Rightarrow \phi(x) = 0]$, and
- $\int_{-\infty}^{\infty} \phi(x)dx = 1$.

The procedure of moving and averaging can now be described as follows.

6.3 **Definition (Convolution)** Let f be a real function, and ϕ a weight function. The convolution of f and ϕ is defined as

$$(f \star \phi)(x) = \int_{-\infty}^{\infty} f(x-y)\phi(y)dy = \int_{-\infty}^{\infty} f(z)\phi(x-z)dz$$

Fixing $x = 0$ and replacing $\phi(y)$ by $\phi(-y)$, we get $\int_{-\infty}^{\infty} f(x)\phi(x)dx = \langle f, \phi \rangle$. The idea behind distributions is to view this as a function of all possible test functions ϕ (a weight function is a special kind of test function).

6.4 **Definition (Test function)** Let X be an open subset of $\mathbb{R}$. A test function in X is an infinitely differentiable function $\phi : X \to \mathbb{R}$ with compact support.

The (vector) space of all test functions in X is denoted with $C_0^\infty(X)$, or $\mathscr{D}(X)$.

By tinkering a bit, one can find test functions in abundance. For every $p \in \mathbb{R}$ and every open subset U of $\mathbb{R}$ containing p, there is a test function – even a weight function – whose support is a subset of U.

Distributions

6.5 **Definition (Distribution)** Let $X \subseteq \mathbb{R}$ be open. A distribution in X is a linear functional $T : \mathscr{D}(X) \to \mathbb{R}$, which is continuous in the sense that

$$\forall_{\phi_n, \phi \in \mathscr{D}(X)} [\lim_{n\to\infty} \phi_n = \phi \text{ (pointwise)} \Rightarrow \lim_{n\to\infty} T(\phi_n) = T(\phi) \text{ (pointwise)}\,].^{1}$$

The space of all distributions in X is denoted by $\mathscr{D}'(X)$ (since it is the linear dual of $\mathscr{D}(X)$).

Every continuous function can be seen as a distribution:

6.6 **Definition (Regular distribution)** Let $f : \mathbb{R} \to \mathbb{R}$ be continuous. The regular distribution T_f of f is

$$T_f(\phi) = \int_{-\infty}^{\infty} \phi(x) f(x) dx$$

(Notice that this integral is well-defined because ϕ has compact support, which yields uniform continuity of ϕf, and thus continuity of T_f. Notice also, that T_f is automatically linear because integration is: $T_{f+g}(\phi) = T_f(\phi) + T_g(\phi)$ and $T_{\lambda f}(\phi) = \lambda T_f(\phi)$. Hence T_f is really a distribution.)

6.7 **Example (The Heaviside function)** A well-known example of a regular distribution is the Heaviside function T_H, where $H : \mathbb{R} \to \mathbb{R}$ is the characteristic function of the positive x-axis, $H = \chi_{(0,\infty)}$.

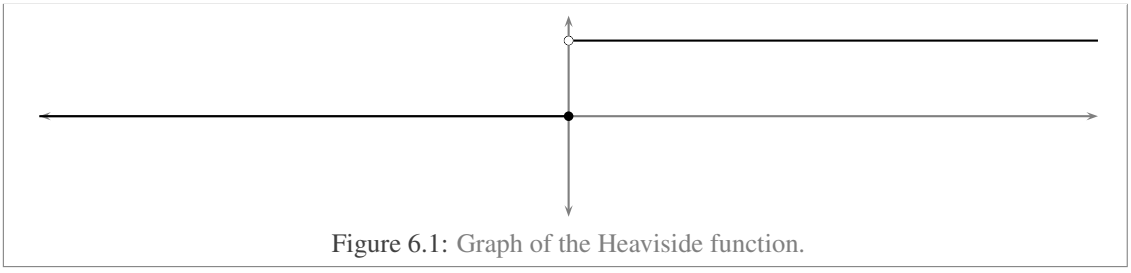


Figure 6.1: Graph of the Heaviside function.

But there are other distributions, too, that do not originate from functions at all. These are called singular distributions. The most famous one is without question the Dirac δ-'function', which could be viewed as the derivative of the Heaviside 'function' in the spirit of example 6.1.

6.8 **Example (A singular distribution: the Dirac δ-'function')** Define a linear functional $\delta : \mathscr{D}(\mathbb{R}) \to \mathbb{R}$ by $\delta(\phi) = \phi(0)$. Then δ meets the criteria to be a distribution, but there is no continuous function f such that $\delta = T_f$. When the role of the 'evaluation point' 0 is played by $x \in \mathbb{R}$, this distribution is also called δ_x.

[1] This continuity is usually called 'weak convergence' in analysis.

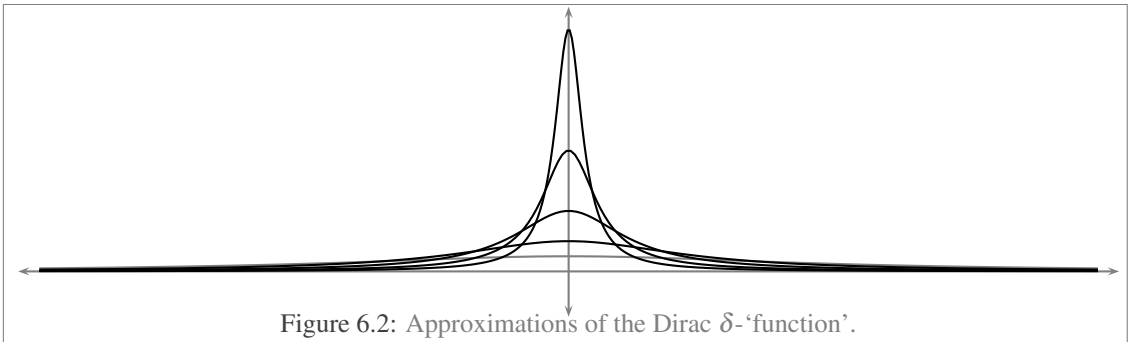


Figure 6.2: Approximations of the Dirac δ-'function'.

Properties of distributions

Suppose T_f is a regular distribution. Let $\varepsilon > 0$ and $a \in \mathbb{R}$. We define $\alpha : \mathbb{R} \to \mathbb{R}$ by

$$\alpha(x) = \begin{cases} e^{-\frac{1}{x}} & \text{if } x > 0 \\ 0 & \text{if } x \leq 0. \end{cases}$$

Then $\alpha \in C^\infty(\mathbb{R})$, has support $[0,\infty)$, and is strictly positive for $x > 0$. If we define $\beta_{a,\varepsilon} : \mathbb{R} \to \mathbb{R}$ by

$$\beta_{a,\varepsilon}(x) = \alpha(x-a+\varepsilon) \cdot \alpha(a+\varepsilon-x),$$

then $\beta_{a,\varepsilon} \in C^\infty(\mathbb{R})$ has support $[a-\varepsilon, a+\varepsilon]$, and moreover $\int_{-\infty}^{\infty} \beta_{a,\varepsilon}(x)dx > 0$. Finally, we define $\phi_{a,\varepsilon} : \mathbb{R} \to \mathbb{R}$ as

$$\phi_{a,\varepsilon}(x) = \frac{\beta_{a,\varepsilon}(x)}{\int_{-\infty}^{\infty} \beta_{a,\varepsilon}(x)dx}.$$

Then $\phi_{a,\varepsilon} \in \mathscr{D}(\mathbb{R})$ is a test function with

$$\int_{-\infty}^{\infty} \phi_{a,\varepsilon}(x)dx = 1.$$

Therefore

$$\lim_{\varepsilon \downarrow 0} T_f(\phi_{a,\varepsilon}) = \lim_{\varepsilon \downarrow 0} \int_{a-\varepsilon}^{a+\varepsilon} f(x)\phi_{a,\varepsilon}(x)dx = f(a).$$

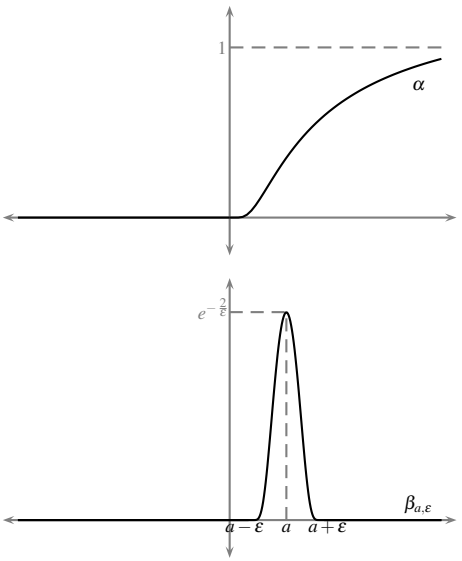


Figure 6.3: α and $\beta_{a,\varepsilon}$

Apparently, we can define a continuous function $f : X \to \mathbb{R}$ in two ways: as a function (by giving $f(a)$ for all $a \in \mathbb{R}$), or as a regular distribution (by giving $T_f(\phi)$ for all $\phi \in \mathscr{D}(X)$).

So distributions are truly an extension of continuous functions. For this reason distributions are also called *generalized functions*, and everything we usually do with functions can also be performed on distributions. For example, differentiation.

6.9 **Example (Differentiation of distributions)** Let $T \in \mathscr{D}'(X)$ be a distribution on X. We define its derivative T' by $T'(\phi) = -T(\phi')$ (also called ∂T, with $(\partial T)(\phi) = -T(\frac{\partial \phi}{\partial x})$).

By unravelling definitions, we see that $T' \in \mathscr{D}'(X)$.

Representations 6.2

Let us now contemplate how the theory of computability and complexity could develop on distributions. We will see that the only major problem is to represent test functions and distributions properly. And that is not all that difficult with the machinery we have built.

First, there are the test functions. These are very 'well-behaved' functions. A test function is infinitely differentiable, and moreover has compact support. So, on (something converging to) that compact support, we can apply our simple representation of (continuous) functions, definition 2.23, with rational polygons. However, we prefer to use the polynomial representation of definition 3.37, which is equally simple on compact sets.

6.10 **Remark ($\mathscr{D}(X)$ as a metric space)** Observe that just like in definition 3.11, $\mathscr{D}(X)$ can be made into a (complete) metric space, by equipping it with the metric

$$\delta_\infty(\phi,\theta) = \sum_{\ll i,j\gg=0}^{\infty} 2^{-\ll i,j\gg} \frac{|\phi-\theta|_{i,j}}{1+|\phi-\theta|_{i,j}}$$

where $|\psi|_{i,j} = \max_{m\le i} \|\psi^{(m)}|_j$, and $\|\psi\|_j = \sup\{|\psi(x)| \,:\, |x|\le j\}$ (Weihrauch and Zhong, 2003).

6.11 **Definition (Representation of $C^\infty(\mathbb{R})$)** We define a representation $\rho_{C^\infty(\mathbb{R})} : (\subseteq\{0,1\}^\omega) \twoheadrightarrow C^\infty(\mathbb{R})$ by

$$\rho_{C^\infty(\mathbb{R})}(\ll p_0,p_1,\ldots\gg^{-1}) = f \quad \text{iff} \quad \forall_j\big[\rho_{\mathrm{PO}}(\ll p_j\gg^{-1}) = f^{(j)}\big],$$

where p_j is a sequence $p_{j_1}, p_{j_2}, \ldots \in \mathscr{P}$ of polynomials with rational coefficients.

A representative of f is a sequence of rational polynomials approximating f and its derivatives.

6.12 **Definition (Representation of test functions)** Let $X\subseteq\mathbb{R}$ be a given open set. If we define $C^\infty_k(X)$ as $\{f\in C^\infty(X) \,:\, \mathrm{supp}(f)\subseteq[-k,k]\}$, then $C^\infty_0(X) = \bigcup_{k=1}^\infty C^\infty_k(X)$. Next, we define the representation $\rho_{C^\infty_k(X)}$ of $C^\infty_k(X)$ as the restriction of $\rho_{C^\infty(X)}$ to $C^\infty_k(X)$.

Finally, we define the representation $\rho_{\mathscr{D}(X)}$ of the test functions $\mathscr{D}(X)$ on X by

$$\rho_{\mathscr{D}(X)}(\ll k\gg^{-1}\circ w) = \rho_{C^\infty(X)}(w),$$

where $\mathrm{Dom}(\rho_{\mathscr{D}(X)}) = \{\ll k\gg^{-1}\circ w \,:\, k\in\mathbb{N}, w\in\Sigma^\omega, \rho_{C^\infty(X)}(w)\in C^\infty_k(X)\}$.

A representative of a test function ϕ consists of a bound of $\mathrm{supp}(\phi)$ and a C^∞-representative of ϕ, that is, a sequence of rational polynomials approximating ϕ and its derivatives.

The spaces $\mathscr{D}(X)$ of test functions on X and $\mathscr{D}'(X)$ of distributions on X are dual as vector spaces. That is, $\mathscr{D}'(X)$ consists (only) of all linear functions $\mathscr{D}(X)\to\mathbb{R}$.) So, once we have a representation of one of them, we can easily derive a representation for the other one, namely the dual representation. A linear functional is in particular continuous, so we can simply use a restricted representation of $C(X)$.

6.13 **Definition (Representation of distributions)** Let $X\subseteq\mathbb{R}$ be given. The representation $\rho_{\mathscr{D}'(\mathrm{X})} :\subseteq \{0,1\}^\omega \twoheadrightarrow \mathscr{D}'(X)$ of $\mathscr{D}'(X)$ is defined as the restriction of $[\rho_{\mathscr{D}(X)}\to\rho_{\mathrm{CS}}]$ to $\mathscr{D}'(X)$:

$$\rho_{\mathscr{D}'(\mathrm{X})} = [\rho_{\mathscr{D}(X)}\to\rho_{\mathrm{CS}}]\big|_{\mathscr{D}'(X)}.$$

Examples 6.3

6.14 **Example (Convolution)** Let $X \subseteq \mathbb{R}$. The function $\mathscr{D}(X) \times \mathscr{D}(X) \to \mathscr{D}(X)$, $(\phi, \theta) \mapsto \phi \star \theta$ is computable, in Time $\mathscr{O}(k^4)$

PROOF By definition of convolution, we need to construct a Turing machine that computes a representative of $x \mapsto \int_{-\infty}^{\infty} \phi(x-y)\theta(y)dy$, from input representatives of two test functions ϕ and θ.

When we have a representative of $(x,y) \mapsto \phi(x-y)\theta(y)$, we can conclude from lemma 5.8 that convolution is computable (in Time $\mathscr{O}(k^2 t(k))$, where t is a regular Time-complexity-bound of the representative of $(x,y) \mapsto \phi(x-y)\theta(y)$.) So this representative is what we aim for. (In section 5.5, we used the rational polygon representation of functions, but the idea persists for the polynomial representation of functions: we already know how to integrate a polynomial. The results of lemma 5.8 theorem 5.10 holds.)

From theorem 5.3 we know that multiplication of functions is almost as easy as multiplication of real numbers, in Time: $\mathscr{O}(k^2)$. Furthermore, a representative of $(x,y) \mapsto \phi(x-y)$ is nothing more than a reflection of a representative of ϕ itself. This is trivially computable, and we know from theorem 5.2 that an upper bound on its Time-complexity is $\mathscr{O}(k)$. And thus the computability of convolution is proven.

Since computing $\phi(z)$ or $\theta(z)$ up to precision K takes $\mathscr{O}(K)$ steps, we see that the total Time-complexity of this method to compute the convolution is $\mathscr{O}(k^4)$. □

> We have cheated a little bit in this proof, by not checking that the Time-bound on $(x,y) \mapsto \phi(x-y)\theta(y)$ is regular. This follows from the fact that both ϕ and θ are infinitely differentiable and have compact support, but is rather difficult to prove. We refer to (Müller, 1986) and (Müller, 1987).

6.15 **Corollary (Regular distributions)** *Let $X \subseteq \mathbb{R}$. The functional $\mathscr{F}(X) \to \mathscr{D}'(X)$, $f \mapsto T_f$ is computable.*

6.16 **Example (Dirac-distribution)** The Dirac-distribution δ is computable.

PROOF This is very easy: we need to build a Turing machine, that computes $\phi(0)$ upon input of a representative of a test function $\phi \in \mathscr{D}(\mathbb{R})$. So, actually, computing the Dirac-distribution is nothing more than computing the evaluation function, which has already been covered in lemma 2.24. Though we used the rational-polygon representation of functions in that lemma, the same argument applies to the polynomial representation of functions used here. □

6.17 **Example (Dirac-distribution inverse)** The function $\mathbb{R} \to \mathscr{D}'(\mathbb{R})$ defined by $x \mapsto \delta_x$ even has a computable inverse.

PROOF Again we need to construct a Turing machine that, given a representative of the distribution δ_x, computes x. Let $\beta : \mathbb{R} \to \mathbb{R}$ be the function defined by $\beta(x) = \begin{cases} \exp(\frac{-1}{1-x^2}) & \text{if } |x| < 1 \\ 0 & \text{if } |x| \geq 1 \end{cases}$.
Then β is infinitely differentiable on all of $\mathbb{R}$, and hence is in $\mathscr{D}(\mathbb{R})$. Also $\beta(x) \neq 0$ iff $|x| < 1$.

Furthermore, define $\delta : (\subseteq \{0,1\}^\omega) \to C^\infty(\mathbb{R})$ by

$$(\delta(\ll n,q \gg^{-1}))(x) = \beta(2^n(x-q)),$$

where $n \in \mathbb{N}$ and $q \in \mathbb{Q}$. Then $(\delta(\ll n,q \gg^{-1}))(x) \neq 0$ if and only if $|x-q| < 2^{-n}$.

Since $(T,\phi) \mapsto T(\phi) = \langle T, \phi \rangle$ is computable by lemma 2.24, there is a Turing machine M' that on input $t \in \mathrm{Dom}(\rho_{\mathscr{D}'(\mathbb{R})}), n \in \mathbb{N}, q \in \mathbb{Q}$ halts if and only if $\langle \rho_{\mathscr{D}'(\mathbb{R})}(t), \delta(\ll n,q \gg^{-1}) \rangle \neq 0$.

If $\delta_x = \rho_{\mathscr{D}'(\mathbb{R})}(t)$, then M' halts if and only if $|x-q| < 2^{-n}$. Therefore, we can construct a machine M that on input $t \in \rho^{-1}_{\mathscr{D}'(\mathbb{R})}(\delta_x)$ and for each $n \in \mathbb{N}$ finds some $q_n \in \mathbb{Q}$ such that $|x-q_n| < 2^{-n}$, and outputs $(q_n - 2^{-n}, q_n + 2^{-n})$ consecutively. Since this output is a (ρ_{NI}-)representative of x, M computes $\delta_x \mapsto x$. □

The previous examples are in contrast with the next one. The moral seems to be that regular distributions need not be so well-behaved.

6.18 **Example (Regular distributions inverse)** Let $X \subseteq \mathbb{R}$. The inverse of the function $\mathscr{F}(X) \to \mathscr{D}'(X)$ defined by $f \mapsto T_f$ is not even continuous. Moreover, there is a non-computable f such that T_f *is* computable!
PROOF For the first claim, let $f_n(x) = \sin(nx)$ and $f(x) = 0$. Then

$$lim_{n\to\infty}\langle T_{f_n}, \phi\rangle = lim_{n\to\infty}\int_{-\infty}^{\infty} \sin(nx)\phi(x)dx = 0 = \langle T_f, \phi\rangle$$

hence T_{f_n} weakly converges to T_f. But f_n does not converge to f!
For the second part of the statement, we refer to (Myhill, 1971). □

The fact that the representations of test functions and distributions are extensions in terms of earlier representations also means that properties of those earlier representations lift to distributions. In particular, arithmetic properties carry over almost effortlessly to distributions.

6.19 **Lemma (Arithmetic on distributions)** *Let $X \subseteq \mathbb{R}$.*
(a) *Translation $\mathscr{D}'(X) \times \mathbb{R} \to \mathscr{D}'(\mathbb{R})$ is computable.*
(b) *Addition of distributions $\mathscr{D}'(X) \times \mathscr{D}'(X) \to \mathscr{D}'(X)$ is computable.*
(c) *Scalar-multiplication $\mathscr{D}'(X) \times \mathbb{R} \to \mathscr{D}'(\mathbb{R})$ is computable.*
(d) *Reflection $\mathscr{D}'(X) \to \mathscr{D}'(X)$ is computable.*
PROOF We only prove (a) since the other parts are analogous; all we do is extend some operation H on $\mathscr{D}(\mathbb{R})$ to some operation H' on $\mathscr{D}'(\mathbb{R})$ by $H'(T_f) = T_{H(f)}$.
So, let a representative of $x \in \mathbb{R}$ and a representative of a distribution $T \in \mathscr{D}(\mathbb{R})$ be given on the input tapes. We are required to construct a Turing machine that computes a representative of the translated distribution T_x with $T_x(\phi) = T(\phi) + x$ for all $\phi \in \mathscr{D}(\mathbb{R})$.
Well, that is easy: we already know that the evaluation of a distribution on a test function is computable, and that addition of two real numbers is. Hence translation of distributions is computable.
Part (b) is concerned with the function $(S,T) \mapsto (S+T)$, where $(S+T)(\phi) = S(\phi) + T(\phi)$. Part (c) is all about the function $(T,\lambda) \mapsto \lambda T$, where $(\lambda T)(\phi) = \lambda T(\phi)$. Finally, part (d) deals with $T \mapsto -T$, where $(-T)(\phi) = -T(\phi)$. So they are all equally simple as (a), since we know the computability of the involved operations on $\mathbb{R}$. □

Because of our polynomial representation, even distribution is computable. In essence, this too carries over 'from below'. The only difference is that this time, the property carries over from polynomials, not from their coefficients: just as we already knew how to add to real numbers, we know how to differentiate polynomials.

6.20 **Lemma (Differentiation of distributions is computable)** *Let $X \subseteq \mathbb{R}$. The function $\mathscr{D}'(X) \to \mathscr{D}'(X)$, $T \mapsto T'$ is computable. Hence also the function $\mathscr{D}'(X) \times \mathbb{N} \to \mathscr{D}'(X)$, $(T,j) \mapsto T^{(j)}$ is computable.*

Complexity 6.4

The theory of complexity we developed in chapter 4 works fine on distributions, just as it did on functions; the definitions work exactly the same. All we had to do was pick a suitable representation. What about the relation between functions and distributions? We stumble on the same problem as in section 5.5, so we cannot just state the following theorem for all $f \in \mathscr{F}(X)$. However, if we had a result similar to theorem 5.10 for other functions than real-analytic ones, we would immediately also have the following lemma for those functions (except maybe with a different Time-bound).

6.21 **Theorem (Computing a real-analytic regular distribution is as fast as computing its base function)** *Let $X \subseteq \mathbb{R}$, let $f : X \to \mathbb{R}$ be real-analytic, and let $t : \mathbb{N} \to \mathbb{N}$ be regular.*
(a) *If f is computable with Time bound t, then $T_f \in \mathscr{D}'(X)$ is computable in Time $\mathcal{O}(k^2 t(k))$.*
(b) *If T_f is computable with Time bound t, then T_f is computable in Time $\mathcal{O}(k^2 t(k))$.*

PROOF The computability claim of part (a) essentially comes down to the fact that function evaluation is computable (lemma 2.24), that multiplication is computable (theorem 5.3), and that integration is computable (lemma 5.8). Given a real-analytic function $f = \rho_{C^\infty(\mathbb{R})}(w)$ and a test function $\phi = \rho_{\mathscr{D}(X)}(\ll K \gg^{-1} \circ u)$, the machine computes $T_f(\phi) = \int_{-\infty}^{\infty} \phi(x) f(x) dx = \int_{-K}^{K} \phi(x) f(x) dx$

Part (b) is similar. Given a representative of T_f and of $a \in \mathbb{R}$, the algorithm to compute $f(a)$ is to compute $T_f(\phi_{a,2^{-k}})$, where as before

$$\phi_{a,\varepsilon}(x) = \begin{cases} 0 & \text{if } x \leq a-\varepsilon \text{ or } x \geq a+\varepsilon \\ \frac{1}{\varepsilon^2}(x-a+\varepsilon) & \text{if } a-\varepsilon \leq x \leq a \\ \frac{1}{\varepsilon^2}(a-x+\varepsilon) & \text{if } a \leq x \leq a+\varepsilon \end{cases}$$

Both complexity bounds follow directly from theorem 5.10. and the proof of corollary 6.15. □

If we denote by $T_C \subseteq \mathscr{D}'(X)$ the set $\{T_f \ : \ f \in C\}$ for $C \subseteq \mathscr{F}(X)$, we can formulate this main theorem as follows:

6.22 **Corollary (Real-analytic functions and regular distributions are equally fast (up to a polynomial))**

$$C \subseteq [t]_{\text{Time}} \ \Leftrightarrow \ T_C \subseteq [t]_{\text{Time}},$$

for $C \subseteq \mathscr{F}(X) \cap C^\infty(X)$ and regular $t : \mathbb{N} \to \mathbb{N}$ of order at least polynomial.

So we can conclude that complexity classes are robust under 'generalization of functions' and 'degrading of regular distributions' (up to a polynomial transformation). Corollary 6.22 is an affirmative answer to our research question for this chapter: the theory of computational complexity works as smoothly on distributions as it does on real functions.

In hindsight, this is not all too astonishing: all we needed was to represent test functions (which are real functions) and distributions (which are real function operators) well, and that is easy because our theory was developed precisely to work well for real functions. Most of the pleasant properties of distributions that emerged from there are nothing more than a furtherance of the same properties of real functions, for which the theory is literally tailor-made.

Finding a suitable representation is not always this easy, though. A good representation cannot be too weak, because otherwise few functions will have tractable complexity bounds. Representations that are too weak can even render functions uncomputable that are intuitively a first requirement for computability. But a good representation should also not be too strong, since then its practical use would be impaired.

CHAPTER

7 Ko

Before Weihrauch developed the Type-2 Theory of Effectivity (TTE), as the model we used in chapter 2 is called, there were other definitions of computability. We have already mentioned the μ-recursiveness approach of (Blum et al., 1998), and Turing's original idea (Turing, 1936).

In this chapter, we discuss another definition by (Ko, 1991), based on oracles. This variation is also apparent in (Müller, 1986), and traits of it have inspired our generalized definition of complexity in chapter 4. The theory is essentially the same, but the different point of view poses interesting questions still.

Oracles 7.1

In chapter 2, we have used genuine Turing machines, and used the infinitude of the input tapes to handle real numbers. Ko opts to keep as much as possible finite, and alters the machine instead. Therefore, the notion of an oracle function is adopted. The intuition is to provide the machine with a different method to get arbitrarily close approximations to the input than infinite input. Other than that, an Oracle machine is just an ordinary Turing machine.

7.1 **Definition (Oracle Machine)** An Oracle Machine M is an ordinary Turing Machine, expanded with an extra tape T and two extra states q and a. If the machine enters state q with $n \in \mathbb{N}$ on tape T, then T is (instantly) filled with something, call it $\varphi(n)$. After this, M enters the a state and 'resumes' its 'normal' computation. This function φ is called an oracle function.

The computation of the Oracle Machine M on input s with oracle φ is written as $M^{\varphi}(s)$. Notice that φ is *not* included in M, but rather something that needs to be plugged into M to actually make M compute, but can still vary.

(Ko, 1991) leaves it intentionally unclear how the oracle function φ is provided to the Turing machine M. Our convention will be that φ is given on a separate read-only input tape. (This is possible, because φ is a function $\mathbb{N} \to \mathbb{N}$, and hence can be coded as an infinite sequence $((1,\varphi(1)),(2,\varphi(2)),\ldots)$, or even $(\varphi(1),\varphi(2),\ldots)$.) So the infinite tape of definition definition 2.1 still appears – be it covert.

The following definition resembles definition 3.5: both are all about 'the Cauchy property'.

7.2 **Definition (Cauchy function)** Let $x \in \mathbb{R}$ be a real number. We call a function $\varphi_x : \mathbb{N} \to \mathbb{Q}$ a Cauchy function for x, if $|\varphi_x(n) - x| \leq 2^{-n}$.

We have defined the Oracle machine, and saw that there is an intentional hole in the definition, intended for the oracle function. The Cauchy functions we just defined are exactly what we are going to fill the gap with. Here is the Ko-analogue of definition 2.3.

7.3 **Definition (Ko-computable real function)** Let $f : (\subseteq\mathbb{R}) \to \mathbb{R}$ be a real function. We call f Ko-computable if and only if there is an Oracle Machine M such that for any $x \in \mathrm{Dom}(f)$, for any Cauchy function φ_x for x, and for any input $n \in \mathbb{N}$, the output $M^{\varphi_x}(n) \in \mathbb{Q}$ is a fraction such that $|M^{\varphi_x}(n) - f(x)| \leq 2^{-n}$.

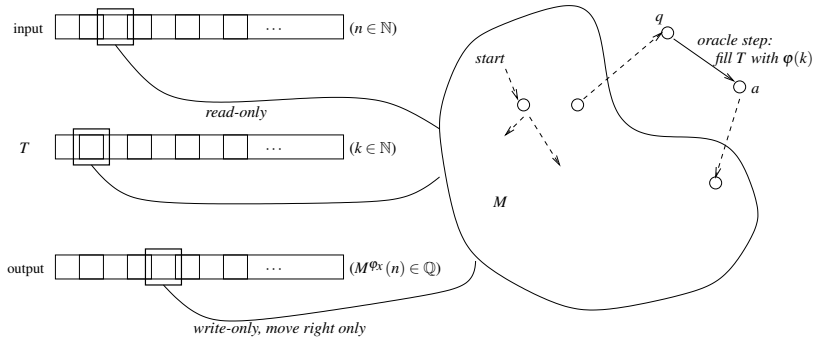


Figure 7.1: Illustration of an Oracle Machine

7.4 **Example (The identity is Ko-computable)** This seems trivial, and in fact it is. But still it is good to see an example 'in action', since Ko's definition does not feel all that intuitive. We will 'mystify the oracle', that is, we will not consider how exactly the oracle function is given, nor how T is filled exactly in the oracle step.

PROOF We will examine the following algorithm to compute the identity function (actually, we examine the Oracle Machine M induced by this algorithm):

1. Copy the contents of the input tape to tape T.
2. Enter state q (query the oracle).
3. Copy the contents of tape T to the output tape.
4. Halt.

Initially, tape T is blank, as well as the output tape, whereas the input tape contains some $n \in \mathbb{N}$. After step 1, T also contains n, so when the oracle mechanism works its magic in step 2, $\varphi_x(n)$ appears on tape T and we resume in state a. After step 3, the output also becomes $\varphi_x(n)$. So $M^{\varphi_x}(n) = \varphi_x(n) \in \mathbb{Q}$. Since φ_x is assumed to be a Cauchy function for x, we see that the described Oracle machine fits the criteria to compute id:

$$|M^{\varphi_x}(n) - \mathrm{id}(x)| = |\varphi_x(n) - x| \leq 2^{-n}$$

□

What's new? 7.2

Let us first prove there is no essential difference between definition 2.3 and definition 7.3. The only difference is the way of in- and output. This gives us two views on the same notion, which we investigate further in this chapter.

7.5 **Theorem (Computable and Ko-computable are equivalent)** *For every $f : (\subseteq\mathbb{R}) \to \mathbb{R}$ we have*

$$f \text{ is computable} \iff f \text{ is Ko-computable}$$

PROOF For the left-hand-side of the assertion, we use the Nested Interval representation, and pretend without loss of generality that one pair $[p_i, q_i]$ can be stored in one tape cell, as before.

($\Rightarrow$) Assume f is computable. Then, by definition, there is a Turing Machine M as in definition 2.1, and an $N \in \mathbb{N}$ such that in N steps, M computes the first k output symbols. Such an N exists for every k, so there is a function $p : \mathbb{N} \to \mathbb{N}$ that models this behavior $p(k) = N$. Note that p is automatically (ordinary, $\mathbb{N} \to \mathbb{N}$) computable, because of bounded minimalization: we can simply simulate a run of M and count steps until M writes the k-th output symbol.

We transform the Turing Machine M into an Oracle Machine M' that computes f as follows. The first thing M' does on input n, is to write $p(n)$ onto tape T, and query the oracle. Then we let the ordinary Turing Machine M work as if T were its input tape, for $p(n)$ steps. Since in $p(n)$ steps M can never read more than $p(n)$ symbols of input, the output up to precision $p(n)$ will be the same as on genuine (infinite) input. So M' (Ko-)computes f since M does.

($\Leftarrow$) On the other hand, assume f is Ko-computable. This time, we are given an Oracle Machine M' for f, and must construct a Turing Machine M. We will now heavily use the 'corrective' nature of our representation.

There is a Turing machine O for a Cauchy function, that maps infinite input $\bar{x} \in \mathrm{Dom}(\rho_{\mathrm{NI}})$ and $n \in \mathbb{N}$ to a $q(n) \in \mathbb{Q}$ with $|\rho_{\mathrm{NI}}(\bar{x}) - q(n)| \leq 2^{-n}$. We replace the oracle step in M' with O: M' is a Turing machine that works on an infinite input tape containing $\bar{x} \in \mathrm{Dom}(()\rho_{\mathrm{NI}})$, and a finite input tape containing $n \in \mathbb{N}$, to produce an output $y_n \in \mathbb{Q}$ such that $|y_n - f(\rho_{\mathrm{NI}}(\bar{x}))| \leq 2^{-n}$.

There is also a Turing machine that maps a fraction y_n to the pair $(y_n - 2^{-n}, y_n + 2^{-n})$. Now M will consecutively run M' on input 1, 2, 3, ..., to obtain $([y_1 - 2^{-1}, y_1 + 2^{-1}], [y_2 - 2^{-2}, y_2 + 2^{-2}], \ldots)$. After run $i+1$, M will output $[y_{i+1} - 2^{-(i+1)}, y_{i+1} + 2^{-(i+1)}] \cap [y_i - 2^{-i}, y_i + 2^{-i}]$. And thus we have constructed a Turing Machine M computing f.

□

Function operators

Our definitions of chapter 2 worked just as well for real function operators as for real functions. All you have to do is pick the right representation. This is an important aspect of the theory, for in recursion theory, higher type construction is a major fundamental construction.

When it comes to real function operators, Ko stays true to his definition 7.3. Consequently, he chooses to alter the machine model, instead of representing functions. By giving the input function through the same mysterious black box input mechanism as with real functions $\mathbb{R} \to \mathbb{R}$, the problem is hidden in the oracles.

7.6 **Definition (Type-3 Cauchy function)** Let $X \subseteq \mathbb{R}$ be bounded, $f \in \mathscr{F}(X)$, and $x \in \mathrm{Dom}(f)$. We call a function $\varphi_{f,x} : \mathbb{N} \to \mathbb{Q}$ a type-3 oracle function for f and x, if $|\varphi_{f,x}(n) - f(x)| \leq 2^{-n}$.

7.7 **Definition (Ko-Computable real function operator)** Let $F : (\subseteq \mathscr{F}(X)) \to \mathscr{F}(Y)$ be a real function operator. We call F Ko-computable if and only if there is an Oracle Machine M that, for any $f \in \mathrm{Dom}(F)$, for any $x \in \mathrm{Dom}(f)$, any type-3 Cauchy function $\varphi_{f,x}$ for f and x, and for any input $n \in \mathbb{N}$, the output $M^{\varphi_{f,x}}(n) \in \mathbb{Q}$ is a rational number such that $|M^{\varphi_{f,x}}(n) - F(f)(x)| \leq 2^{-n}$.

Complexity 7.3

Using Oracle Machines, the definitions of complexity are subtly different. The main difference from the Time definition on ordinary Turing machines is that the entire oracle step costs only *one* unit of time. Of course, this is not a big surprise, if you recall that that is exactly why the oracle exists; it is the raison d'être of oracles in general.

7.8 **Definition (Ko's Time function)** Let $f : (\subseteq \mathbb{R}) \to \mathbb{R}$ be Ko-computable. Then there is an Oracle Machine M that computes f. For $x \in \mathrm{Dom}(f)$, a Cauchy function φ_x for x, and precision $k \in \mathbb{N}$, we define Ko-Time$_M(\varphi_x, k)$ as the number of steps before M halts. Here, we count the oracle step as one.

7.9 **Definition (Ko's LookAhead function)** Let $f : (\subseteq \mathbb{R}) \to \mathbb{R}$ be Ko-computable by Oracle Machine M. For a Cauchy function φ_x for $x \in \mathrm{Dom}(f)$, and precision $k \in \mathbb{N}$, we define Ko-LookAhead$_M(\varphi_x, k)$ as the largest m for which $\varphi_x(m)$ is called upon during the computation.

Analogous to theorem 7.5, functions turn out to have the same complexity as Ko-complexity, up to a polynomial factor.

7.10 **Theorem (Complexity and Ko-complexity are equivalent up to a polynomial)** *A function $f : (\subseteq\mathbb{R}) \to \mathbb{R}$ is computable in polynomial Time if and only if it is Ko-computable in polynomial Ko-Time.*
PROOF

($\Rightarrow$) Suppose that f is computable in polynomial Time. That means there is a Turing Machine M that computes f in the representation ρ_{B-2}, with $k \mapsto \mathrm{Time}_M(\bar{x},k)$ a polynomial for all $\bar{x} \in \rho_{NI}^{-1}(\mathrm{Dom}(f))$. A precision measure for the signed digit representation is $e(\bar{x},k) = 2^{-k}$. So, this means that M takes $p(k)$ steps to compute output of the same precision as the input is on prefix k, where p is a polynomial. That is, M takes $p(k)$ steps to compute output of absolute precision 2^{-k}. In other words, M takes $p(k)$ steps to compute k digits of output. Notice that $p : \mathbb{N} \to \mathbb{N}$ itself is computable, since it is a polynomial.

We construct an Oracle Machine M' to Ko-compute f in the same fashion as in the first part of the proof of theorem 7.5. When determining the Time-complexity of this M', we must count the number of steps it takes to compute p, and add the number of steps it takes to compute f. By induction on theorem 5.2 and theorem 5.3, it is easily shown that it takes a polynomial number of steps to compute p. (Notice that this holds for Oracle Machines just as well; we just do not use the extra non-Turing Machine options). And we already knows it takes p steps to compute f. In total, the computation thus took a polynomial amount of steps.

($\Leftarrow$) On the other hand, suppose that f is Ko-computable in polynomial Time. Let M be the Turing Machine of the second part of the proof of theorem 7.5. A similar, easy, analysis as above learns that the Time-complexity of M is polynomial, since that of M' was.

□

NP and Ko 7.4

Normally, NP is defined as the set of all functions that are computable by a *non-deterministic* Turing machine in polynomial time, and P as the set of all functions computable by a *deterministic* Turing machine in polynomial time (Papadimitriou, 1994).

The major open question in complexity theory is, of course, whether P equals NP. Now, (Ko, 1991) proves striking theorems about this. Perhaps the most interesting one is the following.

7.11 **Theorem (Ko)** *Differentiation is computable $\Leftrightarrow P = NP$.*

This is remarkable, because we already saw in example 6.20 that differentiation is computable on distributions. Granted, that example deals with differentiation of a very special class of real functions (the infinitely differentiable ones with compact support), but still it suggests that differentiation (of 'all' real functions) might not be so difficult[1]. So what about theorem 7.11?

The first question addresses the definitions. That differentiation is computable means that there is a type-3 Oracle Machine that computes the (linear) function operator $d : f \mapsto f'$. But what is the domain of this differential operator d? In the definitions, we could input any computable function f, and f could have any domain $X \subseteq \mathbb{R}$: $f \in \mathscr{F}(X)$. Sure, computable functions are continuous, but continuity does not imply differentiability: the absolute-value function is a counterexample. So, d has a very precarious job: firstly it needs to decide the domain of its input f, and secondly, if a point is in f's domain, it needs to decide whether f is differentiable at that point. Only then can it commence the hard work of computing the derivative at that point. This d is perhaps not what you would expect behind the word 'differentiation'. But still, this gives us an equivalent formulation for the question $P = NP$.

[1] In fact, in the previous chapter we have used a very strong representation. But (Weihrauch, 2000, page 184–185) also proves that differentiation is computable *in another particular non-standard representation* than the one we used in the previous chapter.

But on the righthand-side in the theorem, something else seems fishy. Because *these P and NP are not the ones of the major open question*, the classes of functions $\mathbb{N} \to \mathbb{N}$ that are computable in polynomial time by a deterministic and non-deterministic Turing machine, respectively. These P and NP are but the classes of functions that are, or are not, respectively, computable in polynomial time *in Ko's model*! One dissimilarity, for example, again pops up when we ask about domains. In Ko's model, only functions with a computable domain are considered. Sure, this is more constructive, but in the classical P and NP, *all* functions participate.

Also, lemma 5.8 taught that integration is computable (and thus Ko-computable, by theorem 7.5), even in a very straightforward manner. The following theorem from (Ko, 1991) indicates why we had such trouble finding Time-complexity bounds for integration of a generic real integrable function in section 5.5.5, and had to resort to further assumptions on the function to integrate.

7.12 **Theorem (Ko)** $FP = \#P \Leftrightarrow$ *Integration, i.e.* $\mathrm{Prim}(x) = \int_0^x f(y)dy$, *is computable in polynomial time for every* f *that is computable in polynomial time.*

FP is the class of functions that are computable in polynomial Time, and $\#P$ is the class of functions that enumerate the number of accepting computations of polynomial-Time nondeterministic Turing machines, i.e. the number of solutions to an NP-problem. The open question $FP = \#P$ is of the same family as the question $P = NP$.

It is interesting to note that in traditional numerical analysis, integration is hard, whereas differentiation is easy (Burden and Faires, 2001). Apparently, this gives no guarantees at all about their respective levels of difficulty in a computability setting.

CHAPTER

8 Epilogue

Computing science is the science of computing, and we usually mean computing by physical machines. Since this last phrase has no known precise meaning, every formal reasoning about computing must be based on a model of computation. A widely accepted model of computation is the Turing machine. This model is discrete: it only considers functions on the natural numbers. However, a lot of computations in modern life use real numbers. So, if we want to show that one specific such computation is correct, which is one example of a formal reasoning, we will either need to generalize the Turing model, or we need another model of computation altogether. Such a model of computation is not 'true' or 'false' but can merely be more or less realistic or powerful. Since the Turing model is very realistic and powerful, the preferable option is to incorporate real computations in the Turing model.

In chapter 2, we developed a model of computation after (Weihrauch, 2000) that is suitable for real computation, and is based on genuine Turing machines. Turing machines force us to interpret the input and output of a computation, and in chapter 3 we have seen some of these representations.

Complexity naturally comes next: once it is known that a certain function is actually computable, how long does it take to compute it? We have given a definition of complexity in chapter 4 that allows any representation that was suitable for computability in the first place. Though it does not mean that we can analyze the complexity of more functions, this is an advantage over the complexity in (Weihrauch, 2000), in which one specific representation must be used for analysis. Basically, our complexity definition allows us to let go of low-level details like arithmetic very soon because of this. Therefore, the model of computation and complexity almost starts to look like another model of computation, the real-RAM model, which is highly powerful, but very unrealistic.

These merits have been exemplified in chapter 5, where we proved that real number arithmetic, maximum, trigonometric functions, and extracting roots and integration of real functions are all computable in our model, and moreover have low complexity. Furthermore, we have tested our model by applying it to distributions, a generalization of the notion of function, in chapter 6.

As another way to 'test' our model, we also looked at p-adic numbers, which are *the* other completion of $\mathbb{Q}$ over $\mathbb{R}$ (Gouvêa, 1993; Koblitz, 1977). It turns out that our model runs perfectly well on p-adic numbers, just as well as on real numbers. However, this was too trivial to include in this thesis.

Finally, we saw in chapter 7 that this theory incorporates another model of computation and complexity, that of (Ko, 1991). That model also looked promising, as it is very powerful. A downside is that though it is based on a variation of the Turing machine, Ko's model appears to be a lot less realistic than the model we described. That is, until it was proven equivalent, of course.

We can conclude that this approach is a sound theory of real computation.

Final remarks 8.1

We have looked almost exclusively at real functions in this thesis. But the theory works for functions on any set of continuum-cardinality or less. It also works for any limit space (Schröder, 2001), not just metric or topological ones. But, as explained in chapter 1, I chose for simplicity, and thus suppressed this itch to generalize. Hopefully, this thesis is a lot more readable as a result.

Another example of this itch arose in chapter 3. We could have formalized when a representation is 'good', with some axiom system. That would have rid us of the need to prove every new representation admissible.

This thesis has now been over a year in the making, five months of which I was abroad, studying other matters, and only contemplated 'Complexity in the Real world' once in a while. Chapter 4 took more time than all the others combined, by far. The definitions in that chapter might look simple and logical in hindsight, at least they do to me, but you have to consider the situation in the right way first.

The area of research has steadily grown since (Ko, 1991) and (Weihrauch, 2000). When the plan of attack for this thesis was devised, there had to be a selection in the subjects to be used. Highly interesting works like (Weihrauch, 2003a) and (Weihrauch, 2003b) have therefore unfortunately not been considered.

I am very pleased that I got to work on such fascinating material for my master's thesis, and I would be more than willing to continue research in this, or a related, direction. This thesis leaves more than enough threads for further investigation.

Future work 8.2

Information

Our definition of complexity hinges on 'information'. We have established an equilibrium by counting steps until 'as much information' has been output as is present in a prefix of the input. In our case, 'information' is understood to be 'precision'. After all, what is a representation but a method of approximation, and what other information does an approximation carry than ever more precision?

When we let go of this prejudice, we can understand 'information' to be anything. In particular, building from the model of complexity in this thesis, the door is wide open towards the Markov-approach, which understands 'information' to be a Turing program (which is finite), whereas we have used a graph (which is countably infinite) in section 3.11. Perchance this line of reasoning leads in the direction of something like (Staiger, 2002), without shrinking the universe to computable real numbers only.

Higher order functions

We have not looked at higher order functions in very much detail. For example: what is the relationship between the (Time-)complexity of a function and its rational polygon representation (i.e. is there a complexity-analogue of theorem 2.26)? Or even: what is the complexity of function evaluation (a complexity analogue of lemma 2.24)? It seems these questions are connected to the previous suggestion for further research. Most likely, any representation of functions will have to be extended with a computability demand like in our representations of $\mathbb{R}$.

Surreal numbers

The theory has been 'stress-tested' on distributions and on p-adic numbers, since these somewhat stretch the notions of function and real number, respectively. Another way to test the theory is to consider Conway's surreal numbers (Mamane, 2003). It looks like these are somewhat more incompatible with the theory of this thesis, because of their different constructive nature.

Other models of computation

We have only considered Turing machines so far, other models of computation were dismissed early on, because the Turing machine is so standard in formal reasonings about practical computations. However, other models of computation, like μ-recursiveness and λ-calculus, are more geared towards other things than computations, like theorem proving. It might be interesting to extend these models to real numbers nonetheless. However, the notion of complexity will become a problem. What is there to count in a λ-term that has any practical 'meaning', like Time?

Another way to build from this thesis is to extend the model in a formal way to semantics of higher (though still simple) programming languages.

Acknowledgement 8.3

First and foremost, I thank my supervisors Herman Geuvers and Dick van Leijenhorst. Both have the quality to make you want to make things precise and thus lead you straight to the bottleneck in your reasoning. And of course, my gratitude is with Ker-I Ko and Klaus Weihrauch for their books on the astonishing theory, to prof. Weihrauch also for sending me his recent article on distributions. I also thank Dan Gordon for sending me his (mechanically typeset) article on recursive functionals. Bas Spitters gave a lot of pointers, and Milad Niqui offered a helping hand on continued fractions; without them I would have been lost a whole lot longer.

Furthermore, my friends supported me greatly, especially roommate Ron van Kesteren, who frequently answered some stupid question of mine, or, if he couldn't, led me back to the path I had strayed from. And of course my girlfriend Lotte Hollands for her moral support and for endlessly enduring me thinking out loud to her. And how can I omit my parents, without whose support I would not have been where I am now?

This thesis has been typeset with LaTeX, and that experience has been so pleasant that I would like to thank the authors and maintainers of this software, especially of the great package pstricks.

Finally, my gratitude is with those who ever taught me. Those brave explorers who struck the first match in some corner of my dark brain. In particular, Henk Barendregt, Arnoud van Rooij and Wim Veldman were the first to interest me in the subjects that eventually led to this thesis.

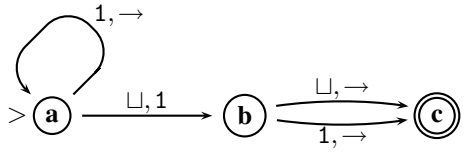
APPENDIX

A The classical case

Computability: Turing machines A.1

In the 1930's, Alan Turing analyzed what it means 'to compute' something, to break it down in 'atomic actions' and 'ways to combine these' – and he has succeeded at that. His answer: that a function is computable means that there is a Turing Machine that computes it.

Intuitively, a Turing machine is a diagram of states and transitions between them. And there is a tape with a head somewhere over it. When the machine is in a certain state, it looks at the symbol on the tape under the head, transits to another state and performs a write with the tape head. The best way to explain it is probably through an example.



Above, you see a Turing machine. It has states $\{a, b, c\}$. Of these, a is the start state; the state the machine is in when the computation begins. State c is the only halting state: upon reaching state c, the computation stops. Every transition has a label consisting of two parts. Together, they dictate what to do, depending on what the head reads on the tape. The first part says when to take this transition, and the second part says what to do with the tape. Imagine we put three 1's on the tape, and leave the rest blank (␣):

Initially, the tape head is on the first cell, and the machine is in the start state a. On the cell under the tape head is the symbol 1, so we take the transition labeled $1, \rightarrow$. The new state is state a, and the tape looks like this after this step:

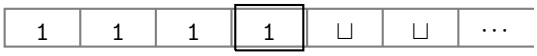
(state: a)

And so the computation continues, step by step:

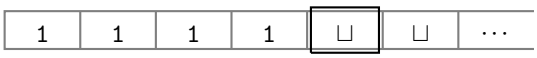
(state: a)

(state: a)

(state: b)



(state: c)



And then the computation is finished. Upon input 3, the machine has computed the output 4. In general, when we input n, the machine will output $n+1$. Hence we see that the function $f(n) = n+1$ is computable.

Normally, we encode input and output in *binary* notation, rather than *unary*. That means that our *alphabet* usually is $\{0, 1, \sqcup\}$. Next to changing the alphabet, there are a lot of other possible variations. There are Spartan ones as minimalistic as possible, like the example we just saw. There are fancy ones with multiple tapes. Or tapes that are infinite to the left as well as to the right. Or even random access memory. The real power of the Turing machine is that all of this does not matter: all kinds of Turing machines have the same expressive power. Any two kinds can simulate one another fastly.

We can even encode a Turing machine itself on a tape. After all, it is nothing but a (big) table of states and transitions. The grand theorem is that there is a *Universal Turing Machine*, that reads such a Turing program and an input, and from that computes the same value that the Turing program would. These Universal Turing Machines are basically our computers.

The famous Church-Turing thesis states that any function $\mathbb{N} \to \mathbb{N}$ that is *effectively* computable (by humans, i.e. computable *at all*) is Turing-computable. The thesis was first stated in 1935, and any attempts since to broaden the class of computable functions by improving upon the Turing machine (or otherwise) have failed. So it is hardly defendable not to believe in the Church-Turing thesis. Nevertheless, it is a philosophical idea, and therefore cannot be proved formally.

Of course, the discussion up to now is not very formal, and there are lots of details we skipped over. For example: there has to be some mechanism to prevent the tape head from falling off the tape to the left. And the machine has to be deterministic: in any state and at any time, there can be only one possible transition. But that does not really matter: any computing scientist knows how to go about Turing machines. And if all this is new to you, you have no need for such formal definitions at all. Moreover, as we argued, all the details are not that important. However, if you still want them, we refer to (Lewis and Papadimitriou, 1998), in which the kind of Turing machine we use is described in full detail.

Complexity A.2

Once the question of which functions $\mathbb{N} \to \mathbb{N}$ are computable was more or less solved, the natural follow-up was to ask how 'hard' a computable function is to compute. In other words, we want to measure certain things about the computation of a function.

We study a class K of algorithms, taking input and yielding output in some alphabet Σ. An example for K is the class of all Turing Machines. This is the K we use in this thesis, but one could look at other choices for K.

Furthermore, we have

- A notion of *input length* $|\cdot| : \Sigma^* \to \mathbb{N}$. It measures how many tape cells the input takes.
- A *resource measure*, or complexity measure, $\mu : K \times \Sigma^* \to \mathbb{N}$. Examples are the number of steps a Turing machine M takes before terminating upon given input w, or the number of tape cells M has used before terminating.
- A *resource function* is the thing we measure to. Basically they are just functions $T : \mathbb{N} \to \mathbb{N}$, such that $T(n)$ bounds the resource measure on input of length n.

Let $M \in K$ be a Turing machine computing a function f, and let $T : \mathbb{N} \to \mathbb{N}$ be a resource function. Assume that when M terminates on input $u \in \Sigma^*$ of length $|u| = n$, then it does so 'within resources' $T(n)$. In this way, T, $|\cdot|$, μ (and K) induce a collection $[T]_\mu$ of functions that are computable 'within resources' T – actually, this collection is not one of functions but one of algorithms. This collection, usually abbreviated to $[T]$ is called a *complexity class*.

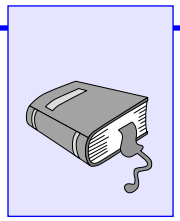
From this point on, the classical literature deals with complexity classes. Basically, given an algorithm, one asks in which complexity class(es) it is. Specifically, it is interesting which algorithms are computable fastly – that is, within polynomial resources – and which ones are not – within exponential resources. The

former class is usually called P, for *p*olynomial, whereas the latter is called NP, for *n*on-deterministic *p*olynomial.

> The latter name actually comes from Turing machines in which more transitions are possible and magically choose the right transition to stay within polynomial resources. But this amounts to the same as normal, deterministic, Turing machines that operate within exponential resources (Papadimitriou, 1994).

The major open problem is whether $P = NP$.

Once we have defined notions similar to input length, resource measure and resource function, and defined a complexity class, the rest comes naturally; it is automatically defined, and all the theorems of classical complexity apply, as long as those extended notions adhere to the same rules. The gist of it is that if one is to build a theory of complexity on a generalized model of computability, all one has to do is to give the right notions of resource measures.



Bibliography

Bailey, D., Borwein, P., and Plouffe, S. (1997). On the rapid computation of various polylogarithmic constants. *Mathematics of Computation*, 66(218):903–913.

Beeler, M., Gosper, R., and Schroeppel, R. (Feb, 29, 1972). Hakmem. *MIT AI Memo 239*.

Bishop, E. and Bridges, D. S. (1985). *Constructive Analysis*, volume 279 of *Grundlehren der Mathematischen Wissenschaften*. Springer, Berlin.

Blum, L., Cucker, F., Shub, M., and Smale, S. (1998). *Complexity and Real Computation*. Springer-Verlag.

Borwein, J. M. and Borwein, P. B. (1987). *Pi and the AGM : a study in analytic number theory and computational complexity*. Wiley Interscience.

Burden, R. L. and Faires, J. D. (2001). *Numerical Analysis*. Brooks/Cole. Wadsworth Group, seventh edition.

de Berg, M., Schwarzkopf, O., van Kreveld, M., and Overmars, M. (2000). *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition.

Dedekind, R. (1963). *Essays on the Theory of Numbers*. New York: Dover Publications.

Di Gianantonio, P. (1991). A golden ratio notation for the real numbers.

Duistermaat, J. J. (1991). Distributies. Utrecht University. Corrected version january 2000.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: a guide to the theory of NP-completeness*. Freeman, New York.

Gordon, D. and Shamir, E. (1983). Computation of recursive functionals using minimal initial segments. *Theoretical Computer Science*, 23:305–315.

Gouvêa, F. Q. (1993). *p-adic Numbers: An Introduction*. Universitext. Springer-Verlag.

Grzegorczyk, A. (1955). Computable functionals. *Fundamenta Mathematicae*, pages 168–202.

Kelley, J. L. (1955). *General Topology*. Princeton University Press.

Ko, K. (1991). *Complexity Theory of Real Functions*. Progress in Theoretical Computer Science. Birkhaüser.

Koblitz, N. (1977). *p-adic Numbers, p-adic Analysis, and Zeta-Functions*. Graduate Texts in Mathematics. Springer-Verlag.

Lewis, H. R. and Papadimitriou, C. H. (1998). *Elements of the Theory of Computation*. Prentice-Hall, second edition.

Mamane, L. E. (2003). Surreal numbers in coq. Master's thesis, Technische Universiteit Eindhoven.

Ménissier-Morain, V. (1994). *Arithmétique exacte, conception, algorithmique et performances d'une implémentation informatique en précision arbitraire*. PhD thesis, Université Paris 7.

Muller, J.-M. (1989). Arithmétique des ordinateurs. *Etudes et recherches en informatique*.

Müller, N. T. (1986). Subpolynomial complexity classes of real functions and real numbers. In Kott, L., editor, *Proceedings of the 13th International Colloquium on Automata, Languages, and Programming*, volume 226 of *Lecture Notes in Computer Science*, pages 284–293, Berlin. Springer.

Müller, N. T. (1987). Uniform computational complexity of Taylor series. In Ottmann, T., editor, *Proceedings of the 14th International Colloquium on Automata, Languages, and Programming*, volume 267 of *Lecture Notes in Computer Science*, pages 435–444, Berlin. Springer.

Myhill, J. (1971). A recursive function defined on a compact interval and having a continuous derivative that is not recursive. *Michigan Mathematical Journal*, pages 97–98.

Normann, D. (2002). Exact real number computations relative to hereditarily total functionals. *Theoretical Computer Science*, 284:437–453.

Olds, C. (1963). *Continued fractions*. Random House.

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

Potts, P. (1998). *Exact Real Arithmetic using Mobius Transformations*. PhD thesis, University of London, Imperial College of Science, Technology and Medicine, Department of Computing.

Schröder, M. (2001). Admissible representations of limit spaces. In Blanck, J., Brattka, V., and Hertling, P., editors, *Computability and Complexity in Analysis*, volume 2064 of *Lecture Notes in Computer Science*, pages 273–295, Berlin. Springer. 4th International Workshop, CCA 2000, Swansea, UK, September 2000.

Staiger, L. (2002). The kolmogorov complexity of real numbers. *Theoretical Computer Science*, 284:455–466.

Stoelinga, M. (1997). Exact representations of and computability on real numbers. Master's thesis no. 404, under supervision of dr. Erik Barendsen, University of Nijmegen.

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265.

van Rooij, A. (1986). *Analyse voor Beginners*. Epsilon Uitgaven, third edition.

Veldman, W. (1987). Berekenbaar en bewijsbaar (recursietheorie). Nijmegen University. Lecture notes.

Weihrauch, K. (2000). *Computable Analysis, an Introduction*. Texts in Theoretical Computer Science. Springer-Verlag.

Weihrauch, K. (2003a). Computational complexity on computable metric spaces. *Mathematical Logic Quarterly*, 49(1):3–21.

Weihrauch, K. (2003b). Continuity in computable analysis (abstract). In Brattka, V., Schröder, M., Weihrauch, K., and Zhong, N., editors, *Computability and Complexity in Analysis*, volume 302 of *Informatik Berichte*, pages 99–100. FernUniversität in Hagen. International Conference, CCA 2003, Cincinnati, USA, August 28–30, 2003.

Weihrauch, K. and Zheng, X. (1999). Effectiveness of the global modulus of continuity on metric spaces. *Theoretical Computer Science*, pages 439–450.

Weihrauch, K. and Zhong, N. (2003). Computability theory of generalized functions. *Journal of the ACM*, pages 469–505.

Wilf, H. S. (1986). *Algorithms and Complexity*. Prentice-Hall.

Index