Computational Complexity of Euclidean Sets: Hyperbolic Julia Sets are Poly-Time Computable

by

Mark Braverman

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Computational Complexity of Euclidean Sets: Hyperbolic Julia Sets are Poly-Time Computable

Mark Braverman

Master of Science

Graduate Department of Computer Science

University of Toronto

2004

We investigate different definitions of the computability and complexity of sets in $\mathbb{R}^k$, and establish new connections between these definitions. This allows us to connect the computability of real functions and real sets in a new way. We show that equivalence of some of the definitions corresponds to equivalence between famous complexity classes. The model we use is mostly consistent with [Wei00].

We apply the concepts developed to show that hyperbolic Julia sets are polynomial time computable. This result is a significant generalization of the result in [RW03], where polynomial time computability has been shown for a restricted type of hyperbolic Julia sets.

Acknowledgements

First of all, I would like to thank my graduate supervisor, Stephen Cook. Our weekly meetings not only allowed me to complete this thesis, but also gave me a much broader and deeper understanding of the entire field of theoretical computer science. Working with him has made my learning process a pleasant one.

I would like to thank Michael Yampolsky from the Department of Mathematics. His guidance has allowed me to study the complex dynamical background required to deal with Julia sets. Without his support parts of this thesis would have never been written.

I would like to thank my fellow graduate students. It was a pleasure to talk to them both about research and other topics. I have learned a lot from them. The theory students seminar was particularly useful.

I would like to thank my parents, Elena and Leonid, for all their support and for encouraging me in the pursuit of my studies.

I would like to thank my girlfriend, Anna Malts, for her support and for helping me in editing this thesis.

Last but not least, I would like to thank the University of Toronto and NSERC for supporing me financially during my studies.

Contents

Chapter 1

Introduction

The questions of computability and complexity in the standard discrete setting have been extensively studied during the past 70 years in the framework of the computability and computational complexity theory. The main objects to be computed in this setting are languages $L \subset \{0,1\}^*$ and functions $f : \{0,1\}^* \to \{0,1\}^*$. Every basic object we operate with in this setting is just a finite string $w \in \{0,1\}^*$ which is presented as a *finite* sequence of bits. The standard model of computation in this setting is the Turing Machine, which is computationally equivalent to the ordinary RAM computer.

In this work we investigate questions concerning the computability and complexity of subsets of the Euclidean space $\mathbb{R}^k$. Some work has to be done to formalize computability in this setting because the basic objects are now points in the uncountable set $\mathbb{R}^k$, which cannot be encoded as finite strings and stored on an ordinary computer. Different models for this setting has been proposed and studied. Works on the subject include [BCSS], [BW99], [CK95], [Ko91], [PR89], [TW98], [Tur36] and [Wei00]. Most of these focus on the computability of real functions, functionals (functions on functions) and operators, with set computability being a by-product of the more general discussion.

In contrast, we mainly focus on the computability and complexity of sets with function computability as a background. We present five different definitions of real sets

computability and show that they are all equivalent. These definitions are consistent with [Wei00], [BW99] and one of the definitions in [CK95]. Intuitively, these definitions are concerned with the possibility of "drawing" the set with an arbitrarily high precision provided as a parameter. It should be noted that our approach is very different from the Blum, Cucker, Shub and Smale approach presented in [BCSS]. We believe that the definition we chose better achieves the goal of understanding the true hardness of drawing a set.

It is natural to define the computational complexity $T(n)$ of a set as the time needed to generate a 2^{-n}-precise image of it. Here, however, the five definitions are no longer equivalent, because their concept of "generating a 2^{-n}-precise image" is different. Moreover, it can be shown that some of them are unconditionally nonequivalent, while others are equivalent if and only if $P = NP$, which is unlikely.

The main real set complexity definition we choose is the one that measures the cost of "zooming" into the set on a fixed-resolution display. This notion is easily made mathematically precise. The same definition has been discussed in [Wei00] and was used in [RW03] when discussing the complexity of some Julia sets. We establish some new connections between this and other set complexity definitions.

We then study the complexity of hyperbolic Julia sets. Julia sets are an example of a very simple iterated process on the complex plane $\mathbb{C}$ which yields very complicated fractal sets in the limit. These sets are not only extremely interesting mathematical objects, but also have amazing graphical images, which makes drawing them an even more exciting problem. [Mil00] is a good source of further information on Julia sets.

Hyperbolic Julia sets are a particularly nice type of Julia sets with a relatively well understood structure. We show that all hyperbolic Julia sets are poly-time computable under the definition mentioned above. This result is a direct generalization of [RW03], where poly-time computability has been shown for a set of hyperbolic Julia sets of degree 2. We then argue that the same result cannot be extended far beyond the hyperbolic

case.

We conclude with some open problems and ideas for future research.

1.1 The Thesis Structure

In section 2.1 we begin with some standard definitions of the computability and complexity of real functions. In section 2.2 we present five definitions of set computability. We show that they are all equivalent, give some examples of computable sets and provide a brief comparison with the BCSS approach from [BCSS]. We conclude the chapter with section 2.3, where we provide a new connection between the computability of a real function and the computability of its graph as a set.

In chapter 3 we extend our discussion from computability to the computational complexity of real sets. As mentioned above the definitions are no longer equivalent in the complexity setting. In section 3.1 we present the set complexity and poly-time computability definitions, and in section 3.2 we establish the relation between the different definitions of poly-time computability (which are no longer equivalent).

In chapter 4 we prove that hyperbolic Julia sets are poly-time computable under the definition established in chapter 3. Sections 4.1–4.3 provide the general background on some complex dynamics. We refer the interested reader to [Mil00] for more information. Sections 4.4–4.6 build up towards a uniform poly-time algorithm for computing hyperbolic Julia sets. In section 4.7 we argue that the result cannot be extended much beyond the hyperbolic case.

In chapter 5 we discuss some open problems and directions for future work. We offer some ideas for future research in general real computability theory (section 5.1), computability in complex dynamics (sections 5.2–5.4) and general computability of physical/dynamical systems (sections 5.5–5.6).

Chapter 2

Computability of Real Sets

2.1 Computability and Complexity of Real Functions

The questions of computability and complexity of real functions is of tremendous importance both from the theoretical and practical point of view. The majority of scientific computations rely on our ability to perform computations over the reals both correctly and efficiently. The computations can vary from performing simple arithmetical operations to solving systems of complex partial differential equations. These problems are usually addressed by the field of numerical analysis. Similarly to the traditional complexity theory in the case of discrete algorithms, one would like to have a framework which addresses the complexity of the *computational problems* in the continuous world. Addressing these issues is the primary goal of the real function computability and complexity field.

Numerous scientific papers and several books have been written on the subject, see [BCSS], [Ko91], [PR89], [TW98] and [Wei00], for example. In contrast to the discrete case, where the vast majority of researchers accepts the Turing Machine as the model of computation, there is no such consensus in the continuous case. For example, the definitions for computability of real functions in [BCSS] and [Ko91] are far from being

equivalent.

In the model of [BCSS] we allow the machines to store real numbers with infinite precision and to perform basic arithmetic operations on them in one unit of time, but also require the output to be the precise value of the function f. In the model of [Ko91], on the other hand, we can only perform bit operations as an ordinary Turing Machine, but our goal is also much more modest: given a good rational approximation of the input x, to compute a good rational approximation of $f(x)$. The latter model is much better for describing the real-life scientific computation processes. These computations are usually performed using rational numbers of some precision (e.g. 32 or 64 bits), and output an approximate answer with some estimate of the computational error. This will be the model of our choice in the present work. This model is essentially equivalent to the models discussed in [PR89] and [Wei00].

2.1.1 Computable Real Numbers

There are countably many Turing Machines and uncountably many real numbers. Thus we cannot expect to be able to represent any real number x using some TM M. This makes the notion of the *computability* for real numbers interesting: a real number is computable if and only if it can be approximated arbitrarily well by a TM. We present here one of the common equivalent definitions for the computability of real numbers, see [Ko91], [Ko98] and [Wei00] for more details.

We are using *dyadic* approximations to represent numbers. The set of the dyadic numbers is defined as follows.

Definition 2.1.1 *The set of the dyadic numbers $\mathbb{D}$ is defined by*

$$\mathbb{D} = \left\{ \frac{p}{2^r} \ : \ p \in \mathbb{Z}, \ \textit{and } r \in \mathbb{N} \right\}$$

Note that dyadic numbers always have a finite binary presentation. All the results in this work would be the same with the set $\mathbb{Q}$ of rationals instead of $\mathbb{D}$, but the proofs and

the presentation are generally nicer with $\mathbb{D}$.

We can now define a computable real number.

Definition 2.1.2 *A real number $x \in \mathbb{R}$ is computable if and only if there exists a Turing Machine $M_x(n)$, such that on an input n $M_x(n)$ outputs a dyadic number $d \in \mathbb{D}$ such that $|x - d| < 2^{-n}$.*

The definition above is very robust, and is equivalent to other definitions for computable numbers. To illustrate this we prove that the definition is equivalent to another definition, which is essentially the definition originally given by Turing in [Tur36].

Theorem 2.1.3 *For a real number x the following are equivalent.*

1. *x is computable as per definition 2.1.2,*
2. *there exists a Turing Machine M_x^2 that outputs the binary expansion of x,*
3. *for any d there exists a Turing Machine M_x^d that outputs the expansion of x in base d (in particular, one can take $d = 10$).*

Proof: We will only prove (1) $\Leftrightarrow$ (2), the proof that (1) $\Leftrightarrow$ (3) is very similar.

(1) $\Rightarrow$ (2) There are two cases:

Case 1: x has a finite binary presentation. In this case M_x^2 just prints the finite binary presentation of x and then gets into an infinite loop of printing zeros.

Case 2: x doesn't have a finite binary presentation. For simplicity we can assume that $x \in [0, 1]$ (the integer part of x does not affect its computability properties). We need to give an algorithm of outputting the i-th binary digit of x. Suppose we have already found the first $i - 1$ digits of x. Denote them by $0.d_1d_2 \dots d_{i-1}$. We need to determine d_i. Denote $q = 0.d_1d_2 \dots d_{i-1}1$. x does not have a finite binary presentation, hence $x \neq q$. So there is an n such that $|x-q| > 2 \cdot 2^{-n}$, and by using M_x to query x with increasingly high precision we can eventually determine (after reaching the precision of 2^{-n}) whether $x > q$ or $x < q$. If $x < q$ output $b_i = 0$ and if $x > q$ output $b_i = 1$.

Note that if we run this algorithm with $x = q$, it will never terminate, hence the condition $x \neq q$ is essential here.

(2) $\Rightarrow$ (1) Denote the first n binary digits M_x^2 outputs by $q_n = 0.d_1 d_2 \dots d_n$. Then $q \in \mathbb{D}$ and $|x - q| < 2^{-n}$, hence q_n is a good output for $M_x(n)$. ■

It follows from definition 2.1.2 that there can only be countably many computable real numbers, since there are only countably many Turing Machines. There are uncountably many real numbers, and hence "most" real numbers are not computable. In particular, the Lebesgue measure of the set of computable numbers is zero.

On the other hand, many "commonly used" real numbers are computable. All the rational numbers are obviously computable. All the algebraic numbers are computable, since we can apply the Newton-Raphson method to obtain an arbitrarily good approximations of the roots of a given polynomial. Many commonly used transcendental real numbers are also computable. For example e is computable, since e can be written as $e = \sum_{n=0}^{\infty} \frac{1}{n!}$, and we can obtain arbitrarily good dyadic approximations of e using this series (which we, in fact, do). π is also computable, one (not the most efficient) way to compute π is using the series $\pi = 4\ tan^{-1}(1) = 4 \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$.

To obtain an example of an uncomputable number, consider a non-recursive predicate $A : \mathbb{N} \to \{0, 1\}$. Then according to part (2) of theorem 2.1.3, the number $x_A = \sum_{n=1}^{\infty} \frac{A(n)}{2^n}$, cannot be computable, since the binary presentation of x_A is $x_A = 0.A(1)A(2)A(3)\dots$.

One can define the *time complexity* of real numbers along the lines of definition 2.1.2. Here, and later in this work, the notation $M(n)$ stands for a Turing Machine M with n given as a parameter on the input tape. In a slight abuse of notation, we interchange the Turing Machine M with the *function* $M(n)$ computed by it.

Definition 2.1.4 *A real number $x \in \mathbb{R}$ is said to be computable in time $T(n)$ if and only if there exists a Turing Machine $M_x(n)$, such that on an input n $M_x(n)$ runs in time $T(n)$ outputs a dyadic number $d \in \mathbb{D}$ such that $|x - d| < 2^{-n}$. In particular, x is said to be* polynomial time *computable if it is computable in time $p(n)$ for some polynomial n.*

It is also obvious how to define the *space complexity* of a number x.

It should be noted that the time complexity version of theorem 2.1.3 does not hold in general. In other words, there is a number x which is polynomial time computable under the definition (1) of theorem 2.1.3, but not under definition (2) of the same theorem.

2.1.2 Computability and Complexity of Real Functions

The basic idea behind any function computability definition is that given an input x, we want to output the value $f(x)$ of the function f on the input x. In the case of real functions, the space of the parameters ($\mathbb{R}^k$) is uncountable, hence we cannot represent the inputs by finite binary strings. Instead, we will be using the notion of an *oracle machine* to define the computability of real functions (see [Ko91] for example).

Definition 2.1.5 *Let $\phi : \{0,1\}^* \to \{0,1\}^*$ be an arbitrary function. An oracle Turing Machine M^ϕ with oracle ϕ is a Turing Machine, which in addition to the ordinary working tape is equipped with a* query tape *and two states: the query state and the answer state. When the machine enters the query state it replaces the string s currently on the query tape by $\phi(s)$, moves the head to the first cell of the query tape and puts the machine in the answer state.*

For time complexity purposes the operation of querying ϕ requires one time unit.

The definition extends very naturally to a machine $M^{\phi_1\phi_2\cdots\phi_k}$ with k oracles. We use oracles to represent real numbers. Formally,

Definition 2.1.6 *We say that the oracle $\phi : \mathbb{N} \to \mathbb{D}$ represents the real number x if for every $n \in \mathbb{N}$, $|x - \phi(n)| < 2^{-n}$. In other words, ϕ provides an arbitrarily good dyadic approximation of x.*

Note that for a computable number x it is possible to substitute the oracle $\phi(n)$ for x by a machine $M(n)$ computing x.

We give the definition for computability of real functions based on the definition from [Ko91] (see also [Ko98]).

Definition 2.1.7 *A function $f : S \to \mathbb{R}$, where $S \subset \mathbb{R}$ is computable if there exists an oracle Turing Machine $M^{\phi}(n)$ such that whenever ϕ represents a number $x \in S$, $M^{\phi}(n)$ outputs a dyadic $y_n \in \mathbb{D}$ such that $|f(x) - y_n| < 2^{-n}$.*

The definition extends naturally to $f : S \to \mathbb{R}$, where $S \subset \mathbb{R}^k$ for an arbitrary k. In this case we require the existence of an oracle machine $M^{\phi_1\phi_2\cdots\phi_k}(n)$ such that whenever ϕ_i represents x_i for $i = 1, 2, \ldots, k$ and $(x_1, x_2, \ldots, x_k) \in S$, $M^{\phi_1\phi_2\cdots\phi_k}(n)$ outputs a dyadic $y_n \in \mathbb{D}$ such that $|f(x_1, x_2, \ldots, x_k) - y_n| < 2^{-n}$.

As in the case of computable numbers, there are only countably many computable functions, and hence "most" functions are not computable. In particular the constant function $f(x) = a$ for all $x \in \mathbb{R}$ is computable if and only if the real number a is computable, hence even this simple kind of functions is not computable for all but countably many a's.

On the other hand, just as in the case of numbers most "common" functions are computable on a suitable domain. Examples of computable functions are: polynomials and rational functions on an appropriate domain, trigonometric, exponential functions etc. For example, the function e^x is computable on $\mathbb{R}$ using the Maclaurin series expansion $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$, which converges on the whole $\mathbb{R}$. It is also easy to see that a composition of two computable functions is computable.

One of the most important properties of computable functions is that they are continuous (cf. [Ko98], Prop. 2.5; [Wei00], Thm. 4.3.1). We will prove this fact here to illustrate the concept of computability for functions.

Theorem 2.1.8 *Let $f : S \to \mathbb{R}$ be a computable function on $S \subset \mathbb{R}^k$, then f is continuous on S.*

Proof: We will prove the theorem for $k = 1$. The proof for a general k works exactly in the same way.

Denote the machine computing f on S by $M^\phi(n)$. Let $x \in S$ be any number in S. Suppose $\varepsilon > 0$ is given. We would like to find a $\delta > 0$ such that $|f(x) - f(y)| < \varepsilon$ whenever $|x - y| < \delta$. Let n be such that $2 \cdot 2^{-n} < \varepsilon$.

Suppose that the binary presentation of x is $l + 0.d_1d_2\ldots$ for some $l \in \mathbb{Z}$. Let ϕ be an oracle for x which on input m outputs a dyadic number $l + 0.d_1d_2\ldots d_md_{m+1} \in \mathbb{D}$ so that $|x - d| < 2^{-m}$, as required by the definition of an oracle. Then the machine M^ϕ on input n will output a number y_n such that $|f(x) - y_n| < 2^{-n}$. Suppose that the largest number with which M queries ϕ is r. Denote $z = l + 0.d_1d_2\ldots d_rd_{r+1}$. Then for any w such that $|w - z| < 2^{-r}$ the answers of ϕ for x are also valid answers for w (because $M^\phi(n)$ queries x only with precision up to 2^{-r}). Hence $M^\phi(n)$ must also work for any such w. So $|f(w) - y_n| < 2^{-n}$ for any such $w \in S$.

Set $\delta = 2^{-(r+1)}$. If $|x - y| < \delta$ for some $y \in S$, then $|y - z| \leq |y - x| + |x - z| < \delta + 2^{-(r+1)} = 2^{-(r+1)} + 2^{-(r+1)} = 2^{-r}$, hence $|f(y) - y_n| < 2^{-n}$, and $|f(x) - f(y)| \leq |f(x) - y_n| + |f(y) - y_n| < 2^{-n} + 2^{-n} = 2 \cdot 2^{-n} < \varepsilon$, which completes the proof. ■

Theorem 2.1.8 implies that some very simple discontinuous functions are not computable. For example, the step function

$$\chi_{[0,\infty)}(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases} \tag{2.1}$$

is not computable on $\mathbb{R}$. The intuitive reason is, that on input $x = 0$ querying x with arbitrarily high precision will not help us to determine the value of the function. On the other hand $\chi_{[0,\infty)}$ is easily seen to be computable on $\mathbb{R}\backslash\{0\}$. We will return to this example later in this work.

We define the complexity of a real function f along the lines of definition 2.1.7.

Definition 2.1.9 *A function $f : S \to \mathbb{R}$, where $S \subset \mathbb{R}$ is computable in time $T(n)$ if there exists an oracle Turing Machine $M^\phi(n)$ such that whenever ϕ represents a number*

$x \in S$, $M^{\phi}(n)$ terminates after no more than $T(n)$ steps and outputs a dyadic $y_n \in \mathbb{D}$ such that $|f(x) - y_n| < 2^{-n}$.

We say that f is computable in polynomial time if there is a polynomial $p(n)$ such that f is computable in time $p(n)$.

Definition 2.1.9 extends naturally to domains of higher dimension and to the space complexity of a function.

Note that the step function (2.1) is not computable in poly-time on $\mathbb{R}\backslash\{0\}$ since it could take arbitrarily long to determine whether the input x is positive or negative. On the other hand, this function is computable in constant time on $\mathbb{R}\backslash(-\varepsilon, \varepsilon)$ for every $\varepsilon > 0$, since we only have to query the input with fixed precision ε to determine the exact value of the function.

Most "common" functions mentioned above are poly-time computable. On the other hand, basic operations on poly-time computable functions might not yield a poly-time computable function. The properties of basic operations on poly-time computable functions and their complexity have been studied in numerous papers. Many results in this direction are presented in [Ko91] and [Ko98]. We will present only two of them to give the reader a general idea on the type of these results.

The first result deals with the complexity of the *maximization* of a poly-time computable function. The result says that performing maximization in poly-time is as hard as solving NP-complete problems. ([Ko98], Cor. 3.8; see also [Fri84]).

Theorem 2.1.10 *The following are equivalent:*

1. *P=NP*
2. *For each poly-time computable $f : [0,1]^2 \to \mathbb{R}$, the function $g(x) = \max\{f(x,y) : 0 \leq y \leq 1\}$ is poly-time computable.*
3. *For each poly-time computable $f : [0,1] \to \mathbb{R}$, the function $h(x) = \max\{f(y) : 0 \leq y \leq x\}$ is poly-time computable.*

4. *For each poly-time computable* $f : [0,1] \to \mathbb{R}$ *that is infinitely differentiable (i.e.* $f \in C^\infty[0,1]$*), the function* $k(x) = \max\{f(y) : 0 \leq y \leq x\}$ *is poly-time computable.*

The second result deals with the complexity of the *integrals* of poly-time computable functions. It says that the complexity of an integral of a poly-time computable function is equivalent to the complexity of #P. ([Ko98], Cor. 3.22; see also [Fri84] and [Ko86]).

Theorem 2.1.11 *The following are equivalent:*

1. *FP= #P.*

2. *For each poly-time computable* $f : [0,1] \to \mathbb{R}$*, the function* $g(x) = \int_0^x f(t)dt$ *is poly-time computable.*

We will be using the notions of real functions computability introduced above in our discussion on the computability of real sets in the next section.

2.2 Computability of Real Sets

In this section we will present several equivalent definition of the computability of real sets. We will mainly concentrate on questions of computability of compact subsets of $\mathbb{R}^k$. A subset of $\mathbb{R}^k$ is *compact* if and only if it is *closed* and *bounded*. We denote the set of all compact subsets of $\mathbb{R}^k$ by $\mathbb{K}^*_k$, we omit the k and just write $\mathbb{K}^*$ whenever the dimension is obvious from the context.

The computability of compact sets has been discussed in [CK95], [BW99], [RW03] and especially in [Wei00]. The definitions presented here are equivalent to the corresponding definitions in these works. We will present different definitions, arising from the mathematical and the computer graphics perspective, and then show that all of them are equivalent.

We classify the definitions into two main groups: *global computability* is concerned with "computing" the entire set at once, while *local computability* addresses the possibility

of "computing" small local portions of the set. We will show that all these definitions are equivalent, but the distinction will come useful when discussing the complexity of the sets.

2.2.1 Global Computability of Compact Sets

For the rest of the section we fix C to be a compact subset of $\mathbb{R}^k$. For the global computability purposes we want to know whether a Turing Machine can "compute" the entire set C with an arbitrarily good precision. The main question is what do we mean by computing the entire set C.

For the first definition we use the computer graphics intuition. Suppose that the dimension is $k = 2$. To compute, or to "draw", C we would like to output an image which is a union U_n of some simple objects, say filled circles which will be the picture of C. Here n is the precision parameter, controlling the "quality" of the picture. There are two requirements from U_n:

1. U_n covers C, i.e. $C \subset U_n$, and
2. U_n does not contain points which are far from C, in other words for every $u \in U_n$ there is a $c \in C$ such that $|u - c| < 2^{-n}$.

Another way to formalize the second condition is using the following notation.

Definition 2.2.1 *We fix the following notations:*

- *For a point $x \in \mathbb{R}^k$ we denote by $B(x, r)$ the* open *ball of radius r around x:* $B(x, r) = \{y : |x - y| < r\}$.
- *For a point $x \in \mathbb{R}^k$ we denote by $\overline{B(x, r)}$ the* closed *ball of radius r around x:* $\overline{B(x, r)} = \{y : |x - y| \leq r\}$.
- *For a set $S \subset \mathbb{R}^k$ we denote by $B(S, r)$ the open r-neighborhood of S:* $B(S, r) = \{y : \exists x \in S \text{ such that } |x - y| < r\} = \bigcup_{x \in S} B(x, r)$.

With these notations the second condition above becomes $U_n \subset B(C, 2^{-n})$.

To formalize the definition of the computable real sets we need to decide what kind of circles do we allow to use in constructing U_n. We want the Turing Machine computing C to output U_n. A Turing Machine can only operate with finite binary strings, hence it is reasonable to make U_n a finite union of filled circles with dyadic centers and dyadic radii. We denote this class of sets by $\mathcal{C}_k$:

$$\mathcal{C}_k = \left\{ \bigcup_{i=1}^{l} \overline{B(x_i, r_i)} \subset \mathbb{R}^k \quad : \quad l \in \mathbb{N}, x_i \in \mathbb{D}^k, r_i \in \mathbb{D} \right\} \tag{2.2}$$

we omit the k and just write $\mathcal{C}$ when the dimension is clear from the context. There is a very natural encoding of sets from $\mathcal{C}$ using binary strings. U_n which is the "picture" of C discussed above is from $\mathcal{C}_2$. We can now summarize the discussion in the following short definition.

Definition 2.2.2 *A compact set $C \subset \mathbb{R}^k$ is said to be* **globally computable** *if there exists a Turing Machine $M(n)$ which on input n outputs an encoding of a set $U_n \in \mathcal{C}_k$ such that $C \subset U_n \subset B(C, 2^{-n})$. In this case we say that $M(n)$* globally computes *C.*

Note that for $k = 2$ this definition corresponds to the intuitive discussion in the beginning of the section. It is easy to see that the definition would be completely equivalent if we used the rationals $\mathbb{Q}$ instead of the dyadic numbers $\mathbb{D}$ in the definition of $\mathcal{C}_k$ in (2.2). Figure 2.1 illustrates definition 2.2.2 in dimension 2. The disks on the figure give a picture of the set C.

To illustrate the concept of global set computability we prove the following simple lemma.

Lemma 2.2.3 *For any $c \in \mathbb{R}$, the singleton $C = \{c\}$ is computable if and only if the number c is computable as per definition 2.1.2.*

Proof: C is computable $\Rightarrow$ c is computable. Let $M(n)$ be a machine computing the set C. We would like to use it to approximate the number c with an arbitrarily good precision.

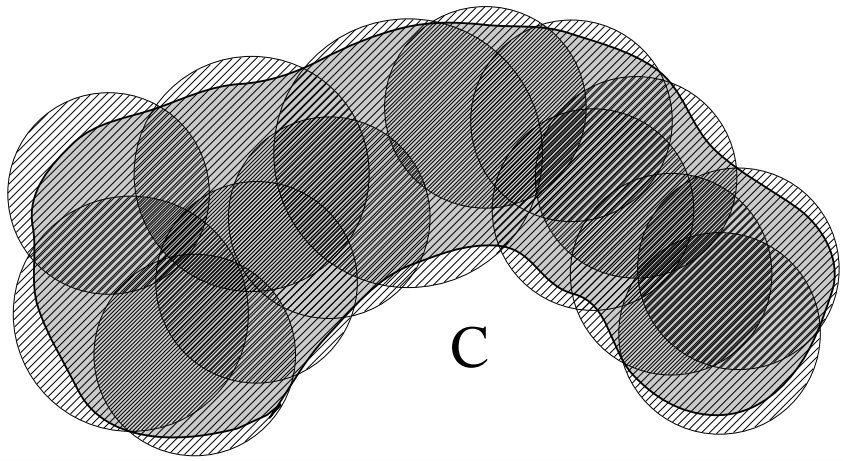

Figure 2.1: An illustration of definition 2.2.2: the disks give a picture of the set C.

On an input n we first run $M(n)$ and obtain a set U_n satisfying definition 2.2.2 for C. Let $x_1 \in \mathbb{D}$ be the center of the first ball in U_n. Then $x_1 \in U_n \subset B(C, 2^{-n}) = B(c, 2^{-n})$. Hence $|x_1 - c| < 2^{-n}$, and we can output x_1 as a 2^{-n} dyadic approximation of c. This shows that the number c is computable.

c is computable $\Rightarrow$ C is computable. Let $M(n)$ be a machine computing the number c. We would like to use it to compute the set C with precision 2^{-n} as per definition 2.2.2.

On an input n we first run $M(n+1)$ to obtain $y = y_{n+1} \in \mathbb{D}$ such that $|c-y| < 2^{-(n+1)}$. We output $U_n = B(y, 2^{-(n+1)}) \in \mathcal{C}$.

We need to show that U_n satisfies the conditions of definition 2.2.2. $|c - y| < 2^{-(n+1)}$, and hence $c \in B(y, 2^{-(n+1)}) = U_n$. On the other hand, $y \in B(c, 2^{-(n+1)})$ and hence $U_n = B(y, 2^{-(n+1)}) \subset B(c, 2^{-(n+1)} + 2^{-(n+1)}) = B(c, 2^{-n}) = B(C, 2^{-n})$, which completes the proof. ■

The second global definition for computability we give is very similar to the first one but is motivated differently. In the case of real numbers in definition 2.1.2 on input n we were outputting a dyadic 2^{-n}-approximation of x in the *Euclidean metric*, which is the natural metric on $\mathbb{R}$. In the case of real sets, on input n we would like to output a set V_n

from $\mathcal{C}$, which is a 2^{-n}-approximation of C in the *Hausdorff metric*, which is the natural metric on $\mathbb{K}^*$ – the set of the compact sets.

We start by introducing the definition of the Hausdorff metric.

Definition 2.2.4 *Let $S, T \in \mathbb{K}_k^*$ be two compact subsets of $\mathbb{R}^k$. Then the* Hausdorff distance *d_H between S and T is defined by*

$$d_H(S,T) = \inf\{r \ : \ S \subset B(T,r) \ \ \textit{and} \ \ T \subset B(S,r)\} \tag{2.3}$$

In particular, note that if $S = \{s\}$ and $T = \{t\}$ are singletons, then $d_H(S,T) = |s-t|$ – the ordinary Euclidean distance between s and t.

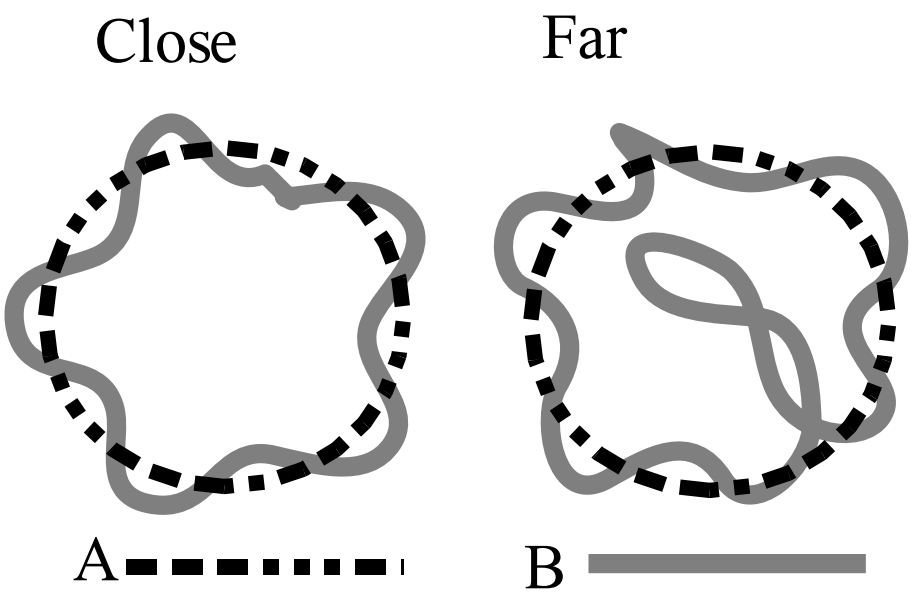

Figure 2.2: An illustration of the Hausdorff metric

On figure 2.2 we see an illustration of the Hausdorff metric. On the left picture the set A is contained in a small neighborhood of the set B and vice versa and the sets are close to each other in the Hausdorff metric. On the right picture, we need to take a large neighborhood of A to cover B, and the sets are far in the Hausdorff metric.

We can now say that a set C is *Hausdorff approximable*, if we can approximate C arbitrarily well by set from $\mathcal{C}$ is the metric d_H. Formally,

Definition 2.2.5 *We say that a compact set $C \subset \mathbb{R}^k$ is* **Hausdorff approximable**, *if there exists a Turing Machine $M(n)$ which on input n outputs an encoding of a set $V_n \in \mathcal{C}_k$ such that $d_H(C, V_n) < 2^{-n}$.*

We would like to prove that the two definitions are, in fact, equivalent.

Theorem 2.2.6 *Definitions 2.2.2 and 2.2.5 are equivalent. That is, a compact set C is globally computable if and only if it is Hausdorff approximable.*

Proof: C is globally computable $\Rightarrow$ C is Hausdorff approximable. This direction is trivial. Suppose that $M(n)$ globally computes C, and denote the output of $M(n)$ by $U_n \in \mathcal{C}$. $C \subset U_n \subset B(C, 2^{-n})$, and hence by definition 2.2.4, $d_H(C, U_n) < 2^{-n}$ and U_n is a 2^{-n}-approximation of C in the Hausdorff metric.

C is Hausdorff approximable $\Rightarrow$ C is globally computable . Suppose that $M(n)$ Hausdorff approximates C. We run $M(n+1)$ to obtain a $2^{-(n+1)}$-Hausdorff approximation $V = V_{n+1} \in \mathcal{C}$ of C. Write $V = \bigcup_{i=1}^{l} B(x_i, r_i)$. Then $B(V, 2^{-(n+1)}) = \bigcup_{i=1}^{l} B(x_i, r_i + 2^{-(n+1)}) \in \mathcal{C}$. Denote $U_n = B(V, 2^{-(n+1)})$. $d_H(C, V) < 2^{-(n+1)}$, hence $V \subset B(C, 2^{-(n+1)})$ and $C \subset B(V, 2^{-(n+1)}) = U_n$. We have $U_n = B(V, 2^{-(n+1)}) \subset B(B(C, 2^{-(n+1)}), 2^{-(n+1)}) = B(C, 2^{-n})$, hence $C \subset U_n \subset B(C, 2^{-n})$, and U_n is a valid 2^{-n}-approximation for globally computing C, which completes the proof. ■

Both definitions 2.2.2 and 2.2.5 address the issue of the *global computability* of C, i.e. computing the entire set C. In practice, however, we are often interested only in drawing a small portion of C. The difference becomes especially substantial when discussing the computational complexity of C. In the next section we will develop tools to address local computability of C.

2.2.2 Local Computability of Compact Sets

The difference between local and global computability is similar to the difference between deciding a language L and listing all the words of length n in L. Note that unlike the

discrete case we cannot expect an oracle Turing Machine to decide whether x is in the set C or not. If such a machine existed, it would have been computing the characteristic function χ_C. The function χ_C is not continuous unless $C = \emptyset$ or $C = \mathbb{R}^k$, and hence it cannot be computable by theorem 2.1.8.

As in the global case, we will not require "sharp" answers, but rather allow some "gray area", for which the answer can be either way. More precisely, for a parameter n we try to draw one pixel of the image of C on a 2^{-n}-grid. We would like to draw a pixel if its center is 2^{-n}-close to C and not draw it if its center is $2 \cdot 2^{-n}$-far from C. To obtain a good global picture of C it is enough to consider all the pixels with centers on the 2^{-n} grid. Formally, we define:

Definition 2.2.7 *We say that the compact set $C \subset \mathbb{R}^k$ is* **locally computable**, *if there is a Turing Machine $M(d, n)$, which computes a function from the family*

$$f(d,n) = \begin{cases} 1 & \text{if } B(d, 2^{-n}) \cap C \neq \emptyset \\ 0 & \text{if } B(d, 2 \cdot 2^{-n}) \cap C = \emptyset \\ 0 \text{ or } 1 & \text{otherwise} \end{cases} \tag{2.4}$$

here $d \in \mathbb{D}^k$ is a dyadic point, and n is a natural number.

This definition is very similar to the definitions in [BW99], [Wei00] and [RW03]. Informally, we see that making one evaluation of f amounts to deciding whether to put one pixel on the image of C or not. On figure 2.3 we see some sample values of the function f.

Our first goal is to show the equivalence of the local and global computability.

Theorem 2.2.8 *Definitions 2.2.2 and 2.2.7 are equivalent. That is, a set is globally computable if and only if it is locally computable. If C is computable according to either of these definitions, we just say that C is* computable.

Proof: C is globally computable $\Rightarrow$ C is locally computable.

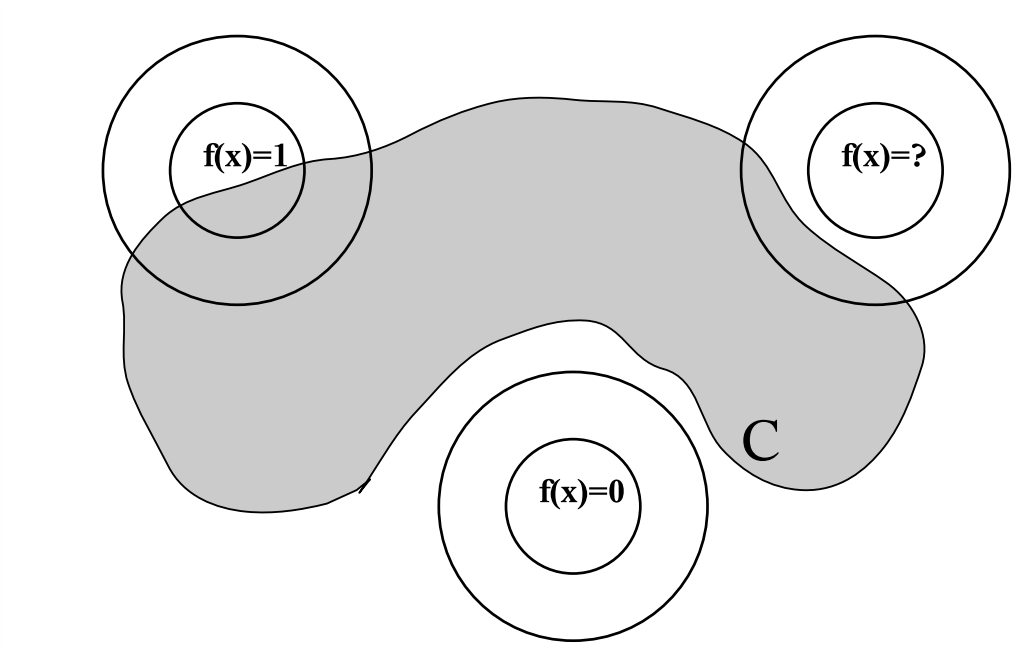

Figure 2.3: Sample values of f. The radius of the inner circle is 2^{-n}.

To compute a function f from the family (2.4) on input (d, n) we first compute the $2^{-(n+1)}$-picture $U_{n+1} \in \mathcal{C}_k$ of C using the machine provided by the definition of the global computability. We can then easily check, using simple operations on dyadic numbers whether $B(d, 2^{-n}) \cap U_{n+1}$ is empty or not. If it is empty, output 0, otherwise, output 1.

$C \subset U_{n+1}$, so if $B(d, 2^{-n}) \cap C \neq \emptyset$, then $B(d, 2^{-n}) \cap U_{n+1} \neq \emptyset$, and we will output 1 in this case.

On the other hand, $U_{n+1} \subset B(C, 2^{-(n+1)})$. Assuming $B(d, 2 \cdot 2^{-n}) \cap C = \emptyset$, we have $d(d, C) \geq 2 \cdot 2^{-n}$, and $d(d, U_{n+1}) \geq 2 \cdot 2^{-n} - 2^{-(n+1)} > 2^{-n}$. Hence $B(d, 2^{-n}) \cap U_{n+1} = \emptyset$, and we will output 0 in this case.

C is locally computable $\Rightarrow$ C is globally computable.

C is bounded, hence there is an $M > 0$ such that $C \subset [-M, M]^k$. Denote by G_n the 2^{-n} gridpoints in $[-(M+1), M+1]^k$. Given and $f(d, n)$ as in (2.4) computing C locally,

we give the global picture U_n as follows

$$U_n = \bigcup \left\{ B(d, 2^{-(n+2)}) \ : \ d \in G_{n+c} \text{ and } f(d, n+2) = 1 \right\}. \tag{2.5}$$

Here c is a small constant such that $2^{c-2} > \sqrt{k}$. Obviously $U_n \in \mathcal{C}_k$. We need to see that $C \subset U_n \subset B(C, 2^{-n})$.

For any $x \in C$, there is a gridpoint $d \in G_{n+c}$ such that $|d - x| < 2^{-(n+2)}$. For this point $f(d, n+2)$ outputs 1, and so $x \subset B(d, 2^{-(n+2)}) \subset U_n$. Hence $C \subset U_n$.

On the other hand, for any $y \in U_n$ there is a $d \in G_{n+c}$ such that $y \in B(d, 2^{-(n+2)})$ and $f(d, n+2) = 1$. So $d(d, C) < 2 \cdot 2^{-(n+2)} = 2^{-(n+1)}$ and $d(y, C) \leq |y - d| + d(d, C) < 2^{-(n+2)} + 2^{-(n+1)} < 2^{-n}$. Hence $U_n \subset B(C, 2^{-n})$, which completes the proof. ■

Formula (2.5) gives a simple algorithm for generating a picture of a locally computable set: test the gridpoints, and output the balls centered at gridpoints where f is 1. On figure 2.4 we see an example of such computation.

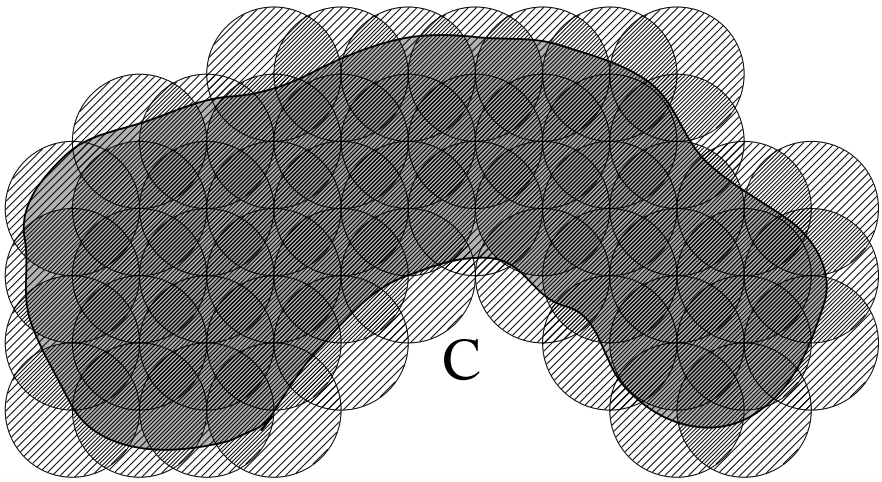

Figure 2.4: An illustration of the approximation of C obtained according to (2.5)

Another local computability definition has to do with the *distance function* d_C from the set C. The distance function d_C is defined as follows:

$$d_C(x) = \inf_{y \in C} |x - y| = \min_{y \in C} |x - y|,$$

the last equality holds by the compactness of C. We have $d_C(x) = 0$ for all $x \in C$. Note that unlike the characteristic function χ_C, the distance function is always continuous, and so it is reasonable to expect it to be computable. In fact, the function $d_C(x)$ is computable if and only if the set C is locally computable, or just computable (cf. [BW99], [Wei00]).

On figure 2.5 we see the graph of the distance function for a one dimensional set C.

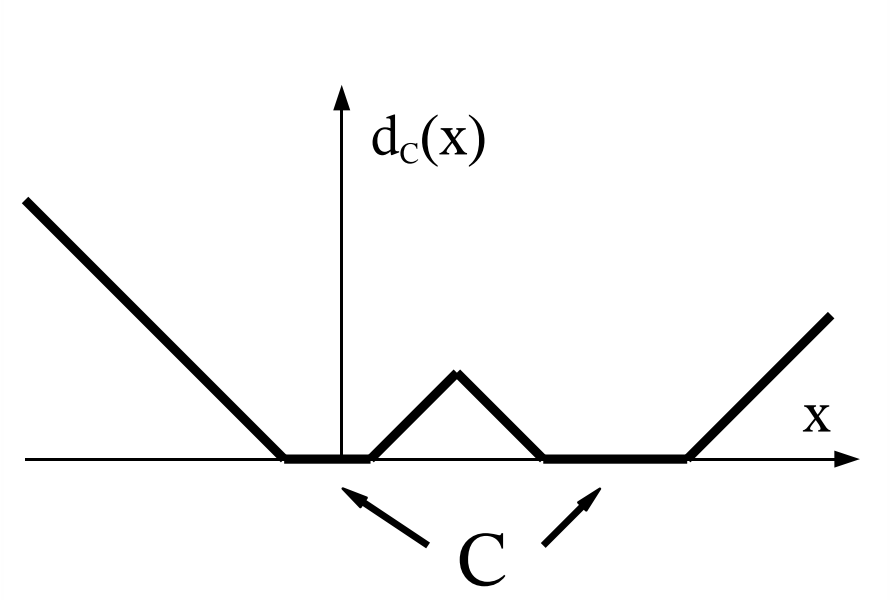

Figure 2.5: An example of the distance function d_C

Theorem 2.2.9 *A set C is computable if and only if its* distance function d_C *is computable.*

Proof: We prove that C is globally computable $\Rightarrow$ d_C is computable $\Rightarrow$ C is locally computable. By theorem 2.2.8, this implies that d_C is computable $\Leftrightarrow$ C is computable.

C is globally computable $\Rightarrow$ d_C is computable.

We want to give an oracle Turing Machine M^ϕ which computes d_C according to definition 2.1.7. To compute $M^\phi(n)$ first use ϕ to query x within an error of $2^{-(n+2)}$. Denote the obtained dyadic approximation by d, we have $|x - d| < 2^{-(n+2)}$. Second, use the machine which globally computes C to obtain a $2^{-(n+2)}$-approximation $U = U_{n+2} \in \mathcal{C}$

as in definition 2.2.2. d is a dyadic number, and U is presented in terms of balls with dyadic centers and radii, so we can approximate the distance $d(d, U) = d_U(d)$ within an error of $2^{-(n+1)}$. The only operation we would have to do in this computation is approximating a square root of a rational number (while computing distances), and this can be efficiently done using Newton's method. We obtain a dyadic number a such that $|a - d_U(d)| < 2^{-(n+1)}$. We claim that $|a - d_C(x)| < 2^{-n}$, and so a is a valid answer for $M^\phi(n)$.

We have $C \subset U \subset d(C, 2^{-(n+2)})$. Hence $d_C(d) - 2^{-(n+2)} \leq d_U(d) \leq d_C(d)$, and $|d_C(d) - d_U(d)| < 2^{-(n+2)}$. We also have $|x - d| < 2^{-(n+2)}$, and hence $|d_C(x) - d_C(d)| < 2^{-(n+2)}$. We summarize

$$|d_C(x)-a| \leq |d_C(x)-d_C(d)|+|d_C(d)-d_U(d)|+|d_U(d)-a| < 2^{-(n+2)}+2^{-(n+2)}+2^{-(n+1)} = 2^{-n}.$$

Which completes the proof of this part.

d_C is computable $\Rightarrow$ C is locally computable.

This part is easier to prove. Assume that $M^\phi(n)$ computes the distance function d_C. We would like to compute $f(d, n)$ from the family (2.4). On an input (d, n), first run $M^\phi(n + 1)$ with ϕ being (the trivial) oracle for the dyadic number d. This gives us a $2^{-(n+1)}$-approximation a of $d_C(d)$. If $a < 3 \cdot 2^{-(n+1)}$ (so d is close to C) output 1, otherwise output 0. We claim that our output satisfies the conditions of (2.4) from the definition of local computability.

Suppose that $B(d, 2^{-n}) \cap C \neq \emptyset$, then $d_C(d) < 2^{-n}$, and $a < d_C(d) + 2^{-(n+1)} < 2^{-n} + 2^{-(n+1)} = 3 \cdot 2^{-(n+1)}$. Hence we output 1 in this case.

If, on the other hand, $B(d, 2 \cdot 2^{-n}) \cap C = \emptyset$, then $d_C(d) \geq 2 \cdot 2^{-n}$, and $a > d_C(d) - 2^{-(n+1)} \geq 2 \cdot 2^{-n} - 2^{-(n+1)} = 3 \cdot 2^{-(n+1)}$. So out output in this case is 0, which completes the proof. ■

In the next section we will see yet another equivalent definition of set computability. It is sometimes very useful when proving the computability of actual sets (the hyperbolic

Julia sets, for example). The equivalence proof is more involved in this case than in the proofs seen above.

2.2.3 Ko Computability of Compact Sets

The notion of set computability we will describe in this section has been introduced by A. Chou and K. Ko in [CK95] under the name of *strong P-recognizability* (see also [Ko98]). We will call it *Ko computability* to avoid confusion due to the fact that, as we will see later, in the complexity setting this definition is *weaker* than the standard definition.

Ko computability is defined in terms of oracle machines. On an oracle ϕ representing a real number x our goal is to output 1 if x is in C, and 0 if x is 2^{-n}-far from C. Once again, the area $0 < d(C, x) \leq 2^{-n}$ is a "gray area", where either answer is acceptable. Formally, we define:

Definition 2.2.10 *We say that a set C is* **Ko computable***, if there exists an oracle Turing Machine $M^{\phi}(n)$, such that if ϕ represents a real number x, then on an input n the output of $M^{\phi}(n)$ is*

$$M^{\phi}(n) = \begin{cases} 1 & \text{if } x \in C \\ 0 & \text{if } B(x, 2^{-n}) \cap C = \emptyset \\ 0 \text{ or } 1 & \text{otherwise} \end{cases} \tag{2.6}$$

Note that $M^{\phi}(n)$ corresponds a 0 or a 1 to any real number x. What $M^{\phi}(n)$ computes cannot be a function, because it is not continuous (unless C is trivial). Indeed, what $M^{\phi}(n)$ computes is not a function, because the values of $M^{\phi}(n)$ in the "gray area" depend not just on the *value* of the input x, but also on the *particular oracle* ϕ representing x.

For example, let $C = \{0\}$. Then the following algorithm for $M^{\phi}(n)$ Ko computes C:

1. Query $d = \phi(n+1)$, so that $|x - d| < 2^{-(n+1)}$,
2. If $|d| < 2^{-(n+1)}$ output 1, otherwise output 0.

It is easy to see that the output of the algorithm above satisfies (2.6). Consider it running on the input $x = 2^{-(n+1)}$. If x is represented by an oracle such that $\phi(n+1) < 2^{-(n+1)}$, the algorithm returns 1, and if x is represented by an oracle such that $\phi(n+1) \geq 2^{-(n+1)}$, the algorithm returns 0. Hence we see that in this case $M^{\phi}(n)$ does not compute a well defined function at $x = 2^{-(n+1)}$.

We would like to show that Ko computability is equivalent to the set computability as per definition 2.2.7. To the best of our knowledge, this result has not appeared in the literature.

Theorem 2.2.11 *Definitions 2.2.7 and 2.2.10 are equivalent. That is, a compact set C is Ko computable if and only if it is locally computable.*

Proof:

C is locally computable $\Rightarrow$ C is Ko computable.

This is the easy direction in the proof. Suppose we have a Turing Machine computing an $f(d,n)$ from the family (2.4). In order to Ko compute C, we first query $d = \phi(n+2)$, and then return $f(d, n+2)$. We need to show that (2.6) is satisfied.

If $x \in C$, then $|x-d| < 2^{-(n+2)}$ implies that $x \in B(d, 2^{-(n+2)}) \cap C$, so $B(d, 2^{-(n+2)}) \cap C \neq \emptyset$, and by (2.4) $f(d, n+2)$ returns 1.

If $B(x, 2^{-n}) \cap C = \emptyset$, then $|x-d| < 2^{-(n+2)}$ implies that $B(d, 2^{-n} - 2^{-(n+2)}) \cap C = \emptyset$. So $B(d, 2 \cdot 2^{-(n+2)}) \cap C \subset B(d, 2^{-n} - 2^{-(n+2)}) \cap C = \emptyset$, and by (2.4) $f(d, n+2)$ returns 0, which completes the proof.

C is Ko computable $\Rightarrow$ C is locally computable.

This direction is much more involved, and will require some preparation. For convenience purposes assume that $C \subset \left[\frac{1}{4}, \frac{3}{4}\right]$, and hence in (2.4) we only need to consider $d \in [0,1]$. The proof extends trivially to bigger intervals (C is bounded, because it is compact). It is also easy to extend the proof to dimensions $k > 1$.

We construct an infinite tree T. In every vertex of T we write a dyadic number. The numbers on level l are dyadics of the form $m \cdot 2^{-l}$. The root, which is on level 1 is

labeled by $\frac{1}{2} = 0.1$ (all the numbers in this section are in binary notation). Each vertex v on a level l has 3 children. If the label of v is $m \cdot 2^{-l}$ then the labels of its children are $m \cdot 2^{-l} - 2^{-(l+1)}$, $m \cdot 2^{-l}$ and $m \cdot 2^{-l} + 2^{-(l+1)}$, or in other words $(2m-1) \cdot 2^{-(l+1)}$, $2m \cdot 2^{-(l+1)}$ and $(2m+1) \cdot 2^{-(l+1)}$. On figure 2.6 we see the first three levels of the tree (cf [Wei00], section 7.2, signed digit representation).

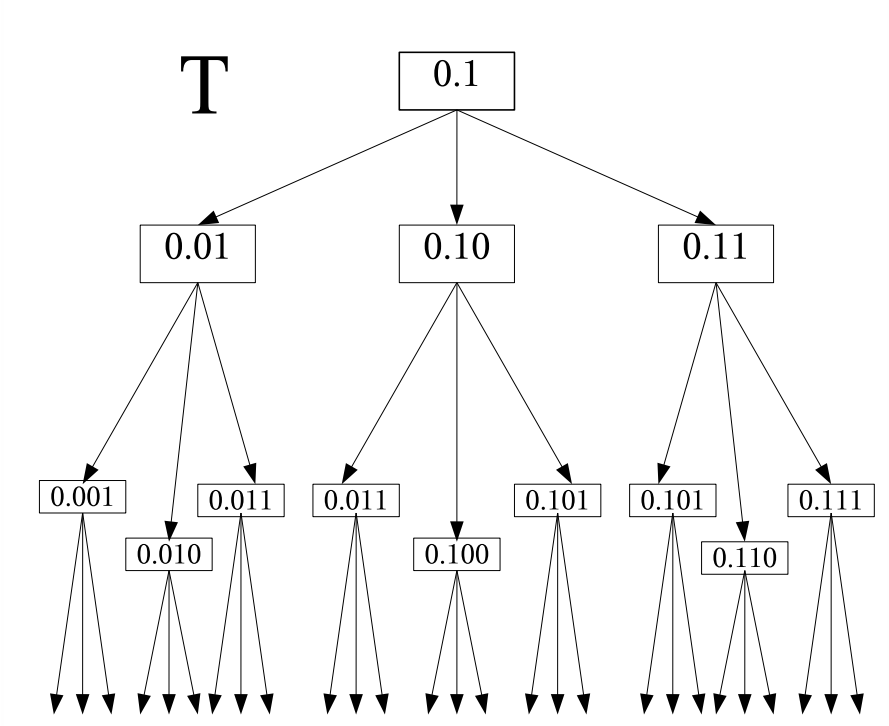

Figure 2.6: The first three levels of the tree T

It is easy to see that numbers on every path p in the tree converge to a real number $x_p \in [0,1]$. Conversely, for every $x \in [0,1]$ there is a path p such that $x_p = x$ (e.g. choose p to be the prefixes of the binary expansion of x). Moreover, if we denote by $p(n)$ the label of the n-th node on p, then $|x - p(n)| \leq 2^{-n} < 2^{-(n-1)}$. Hence $\phi(n) = p(n+1)$ is a valid oracle for x_p.

We will now describe how to compute $f(d,n)$ as in (2.4). On an input (d,n) find two nodes v_1 and v_2 on level n such that $label(v_1) \leq d \leq label(v_2)$ and $|label(v_1) - d| < 2^{-n}$, $|label(v_2) - d| < 2^{-n}$ (if d is an integer multiple of 2^{-n}, we can choose $v_1 = v_2$). Denote

the paths from the root to v_1 and v_2 by p_1 and p_2, respectively. Denote the machine Ko computing C by $M^{\phi}(n)$. We simulate the computation of $M^{\phi}(n)$ on the subtrees with roots v_1 and v_2 as follows.

Consider the simulation with root v_1 (the simulation with root v_2 is done in the same way). For every oracle query $\phi(m)$ with $m < n$, we return the value of $p_1(m+1)$ (which is a valid output for the oracle). Otherwise we consider all the possible descendants of v_1 on level $m+1$, and create a separate computation for each of them (thus, we create 3^{m-n+1} computations). Consider one of the copies and denote the path leading to the selected vertex on level $m+1$ by p (p extends p_1). If we are now asked about $\phi(r)$ for some $r < m+1$, we return the value of $p(r+1)$, and otherwise we again consider all possible descendants of $p(m+1)$ on level $r+1$, and split the computation into 3^{r-m} computations. We continue this process until all computations terminate.

If any one of the computations (either starting from v_1 or from v_2) returns 1, we return 1. Otherwise return 0. We need to show two things:

1. The algorithm terminates.

2. It gives answers that satisfy (2.4).

We will be using König's lemma.

König's Lemma: If a every vertex in a tree has a finite degree, then the tree is infinite if and only if it has an infinite branch.

Suppose that the computation does not terminate. We can view the entire computation as a tree where the nodes are the subcomputations described above and a computation C_1 is the parent of the 3^s computations it launches. If the entire computation does not terminate, then there are two possibilities: either one of the computations C' does not terminate without calling to subcomputations, or the tree of all the computations to be performed is an infinite tree.

In the first case denote the path in T leading to C' by p'. Then p' corresponds to a dyadic number d', and also gives an oracle ϕ' for d'. Note that C' is reached and executed by simulating $M^{\phi'}(n)$. Hence $M^{\phi'}(n)$ does not terminate in this case, contradiction.

In the second case, by König's lemma, there must be an infinite branch in the computations tree. Denote the branch by $C_1, C_2, C_3, \ldots$. That is, C_1 calls C_2, C_2 calls C_3 etc. Note that each C_i works with a path p_i of T and p_{i+1} strictly extends p_i for each i, hence the infinite sequence of C_i corresponds to an infinite path p in T. The path converges to a real number $x \in [0,1]$, and p gives rise to an oracle ϕ for x. By the construction, the sequence of $C_1, C_2, C_3, \ldots$ simulates the computation of $M^{\phi}(n)$. Hence $M^{\phi}(n)$ does not terminate, contradiction. This shows that the algorithm terminates.

We now have to show the correctness of the algorithm.

Case 1: $B(d, 2^{-n}) \cap C \neq \emptyset$**.** In this case, either v_1 or v_2 has a descending path p in T which converges to an $x \in C$. Consider the oracle ϕ corresponding to this path. One of the computation paths of the algorithm will be a simulation of $M^{\phi}(n)$, and hence will have to output 1.

Case 2: $B(d, 3 \cdot 2^{-n}) \cap C = \emptyset$**.** In this case all points corresponding to descendants of v_1 and v_2 are at distance at most $2 \cdot 2^{-n}$ from d, and hence are at least 2^{-n}-far from C. Hence any computation corresponding to simulating M^{ϕ} along any of the paths must output 0.

Note that we are only able to compute a function satisfying a condition weaker than (2.4). Namely, we compute a function f such that

$$f(d,n) = \begin{cases} 1 & \text{if } B(d, 2^{-n}) \cap C \neq \emptyset \\ 0 & \text{if } B(d, 3 \cdot 2^{-n}) \cap C = \emptyset \\ 0 \text{ or } 1 & \text{otherwise} \end{cases}$$

It is very easy to use f to compute a function that satisfies (2.4). Take

$$g(d,n) = f(d - 2^{-(n+1)}, n+1) \vee f(d, n+1) \vee f(d + 2^{-(n+1)}, n+1)$$

If $|d-c| < 2^{-n}$ for some $c \in C$, then either $|d-c+2^{-(n+1)}| < 2^{-(n+1)}$, or $|d-c| < 2^{-(n+1)}$, or $|d-c-2^{-(n+1)}| < 2^{-(n+1)}$. So one of the f's will return 1. On the other hand, if $B(d, 2 \cdot 2^{-n}) \cap C = \emptyset$, then $B(d-2^{-(n+1)}, 3 \cdot 2^{-(n+1)}) \cap C = \emptyset$, $B(d, 3 \cdot 2^{-(n+1)}) \cap C = \emptyset$ and $B(d+2^{-(n+1)}, 3 \cdot 2^{-(n+1)}) \cap C = \emptyset$. Hence g returns 0 in this case. This completes the proof. ■

2.2.4 Computability of Compact Sets: Summary

We have presented five definitions of compact sets computability, and have shown that they are all equivalent. We summarize the results in the following theorem:

Theorem 2.2.12 *For a compact set $C \subset \mathbb{R}^k$ the following definitions are equivalent. If any of them holds, we say that the set C is* computable*:*

1. *C is globally computable (definition 2.2.2),*
2. *C is Hausdorff approximable (definition 2.2.5),*
3. *C is locally computable (definition 2.2.7),*
4. *the distance function $d_C(x)$ is computable,*
5. *C is Ko computable (definition 2.2.10).*

2.2.5 Examples of Computable Sets

Most "usual" sets are computable. Theorem 2.2.12 gives us tools to prove computability of particular sets using the most convenient definition of set computability. Hausdorff approximability and Ko computability are usually the easiest ones to demonstrate.

One of the richest families of computable sets is the family of sets arising from computable real functions. In section 2.3 we will show that a graph of a computable function is a computable set. In particular, the graphs of the "standard" functions such as polynomials, rational functions (on a proper domain), trigonometric and exponential functions

are all computable. This is not surprising since, in fact, we are able to generate good images of these graphs.

This result can be extended to the filled graph of f, which is the area between the graph and the parameters plane (the x axis in the two dimensional case).

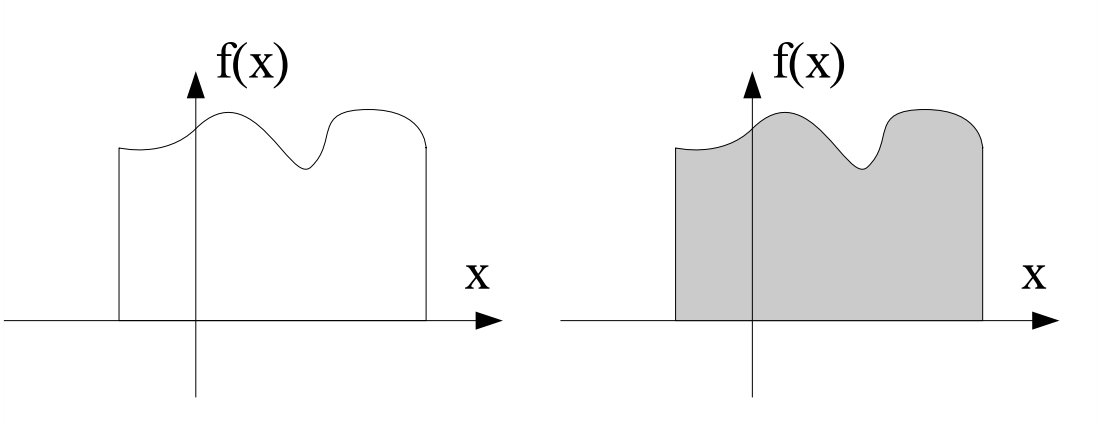

Figure 2.7: If the function f is computable on some closed interval, then its graph (left) and its filled graph (right) are computable.

By taking finite unions and intersections of these graphs and filled graphs we can obtain quite complicated computable sets.

Another interesting family of computable sets are the self-similar fractal images. The most famous set in this family is probably the Cantor set C. To define the Cantor set let $C_0 = [0, 1]$ be the $[0, 1]$ interval. Let C_1 be the set obtained from C_0 by removing its middle third: $C_1 = C_0 \backslash (\frac{1}{3}, \frac{2}{3}) = [0, \frac{1}{3}] \cup [\frac{2}{3}, 1]$. We then remove the middle thirds from each of the two intervals of C_1 to obtain $C_2 = [0, \frac{1}{9}] \cup [\frac{2}{9}, \frac{1}{3}] \cup [\frac{2}{3}, \frac{7}{9}] \cup [\frac{8}{9}, 1]$. Continue this process to obtain a chain of closed sets $C_0 \supset C_1 \supset C_2 \supset \ldots$. Define $C = \cap_{i=0}^{\infty} C_i$. See figure 2.8 for a graphical illustration of the construction.

The Cantor set has a fractal structure because each of its halves is similar to the entire set C with a factor of $\frac{1}{3}$. C has an irrational Hausdorff dimension of $\log_3 2$, which is smaller than 1 but bigger than 0.

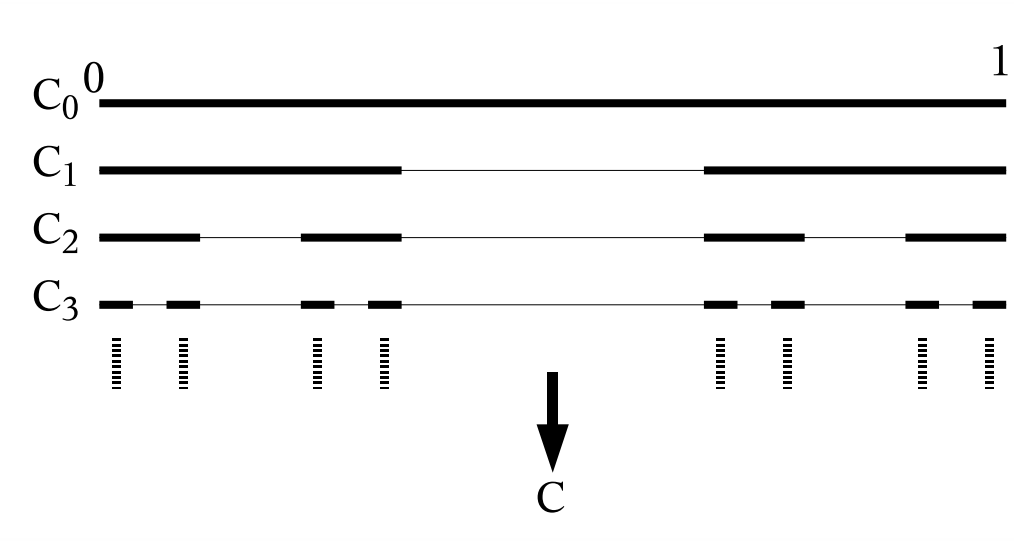

Figure 2.8: The construction of the Cantor set C.

We establish that C is computable by showing it is approximable in the Hausdorff metric. C_i is a 3^{-i}-good approximation of C in the Hausdorff metric, and it is very easy to approximate C_i in the Hausdorff metric, since C_i is just a union of 2^i simple intervals. Thus C is easily computable.

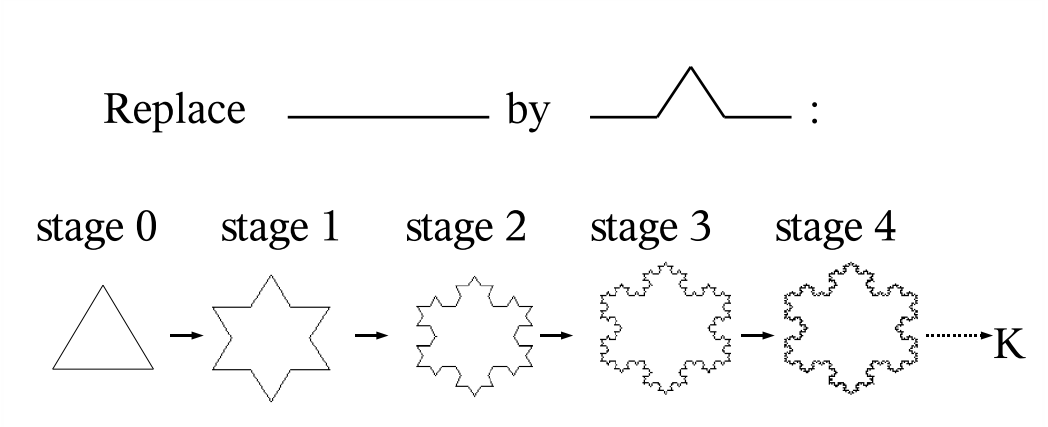

Figure 2.9: The construction of Koch snowflake K.

Another famous computable fractal is the *Koch snowflake* K. The Koch snowflake is obtained from an equilateral triangle by continuously replacing each side of length

l by four sides of length $\frac{l}{3}$, as seen on figure 2.9. K is a set of Hausdorff dimension $\log_3 4$, which is less than 2 and more than 1. K is the union of three self-similar sets (corresponding to the sides of the original equilateral triangle).

As in the case of the Cantor set, the i-th stage of the construction is a $2^{-\Theta(i)}$-approximation of K, and it is easy to compute the i-th stage which is a simple union of line segments. This shows that K is Hausdorff approxiamble, and hence computable.

Another family of computable fractals, the hyperbolic Julia sets. Questions surrounding their computability will be discussed in chapter 4.

2.2.6 A Comparison With the BCSS Approach

The approach taken in the present work is very different from the real computation approach developed by Blum, Cucker, Shub and Smale and presented in [BCSS].

In the BCSS approach storage of reals with an arbitrary precision is allowed. In exchange, we are required to make sharp decisions similarly to the discrete case. That is, to compute a set A we need to be able to answer the question of whether $x \in A$ for all $x \in \mathbb{R}^k$.

Note that we cannot expect to make sharp decisions in our model. We cannot answer this question even the simple set $A = \{0\}$, because no matter how good an approximation of x we have, we can never be sure whether $x = 0$, or x is just very close to 0. The set $\{0\}$ is trivially computable in our model (where we only require approximations of the set).

It is seen in [BCSS] that sets computable in the BCSS model can only have an integer Hausdorff dimension. In particular the Cantor set and the Koch snowflake, having Hausdorff dimensions of $\log_3 2$ and $\log_3 4$ respectively, are not computable in this model. These sets are computable in our model, as seen in the previous section. Most Julia sets are not computable in the BCSS model, even though all the hyperbolic Julia sets are computable in our model as seen in chapter 4.

The two models are incomparable. We have seen examples of computable sets which are not computable in the BCSS modes. In the opposite direction, it is easy to see that any singleton $C = \{c\}$ is computable in the BCSS model, but it has been shown in lemma 2.2.3 that a singleton $\{c\}$ is computable in our model if and only if c is a computable number as per definition 2.1.2.

The fact that we are actually able to generate good images of the Cantor set, the Koch snowflake and the hyperbolic Julia sets suggests that the definition we are using is good for describing the hardness of "drawing" a set. The BCSS approach makes a strong connection between real computation on the exact-arithmetic Turing Machines and algebraic geometry, but it sometimes classifies easy-to-draw sets as hard because of their high complexity from the algebraic point of view.

2.3 Connecting Computable Functions with Computable Sets

In the discrete case there are tight connections between the computability of functions and the decidability of sets. We would like to establish similar connections in the case of real functions and real sets.

2.3.1 The Continuous Functions Case

In the discrete case we know that a function $f : \{0,1\}^* \to \{0,1\}$ which is defined on all inputs is computable if and only if its graph $G_f = \{(x, f(x)) \ : \ x \in \{0,1\}^*\}$ is decidable. A similar connection holds in the case of real functions if the function f is *continuous*:

Theorem 2.3.1 *Let $S \subset \mathbb{R}^k$ be a closed rectangle with computable coordinates. A continuous function $f : S \to \mathbb{R}$ is computable if and only if its graph $G_f = \{(x, f(x)) \ : \ x \in S\} \subset \mathbb{R}^{k+1}$ is a computable set.*

Proof: For simplicity assume that $k = 1$ and $S = [0, 1]$. The proof works in a similar fashion in the more general case. Note that in this proof we make nontrivial use of the fact that Ko computability is equivalent to the other types of computability.

f is computable $\Rightarrow$ G_f is computable.

We show how to Ko compute G_f assuming f is computable. Denote by $M^{\phi}(n)$ the machine which computes f. On an input (x, y) given by a pair of oracles (ϕ, ψ) we need to determine whether (x, y) is on G_f, or is it 2^{-n}-far from the graph.

We start by running $M^{\phi}(n+1)$ to obtain a d such that $|d - f(x)| < 2^{-(n+1)}$. If $|d - y| < 2^{-(n+1)}$, return 1, otherwise return 0.

Suppose $(x, y) \in G_f$, then $y = f(x)$, so $|d - y| = |d - f(x)| < 2^{-(n+1)}$, and the algorithm returns 1 in this case.

If (x, y) is 2^{-n}-far from G_f, then in particular $|f(x) - y| \geq 2^{-n}$, hence $|d - y| \geq |f(x) - y| - |d - f(x)| > 2^{-n} - 2^{-(n+1)} = 2^{-(n+1)}$, and the algorithm returns 0 in this case.

G_f is computable $\Rightarrow$ f is computable.

G_f is computable, in particular it is globally computable. Hence there is a machine $M(n)$ which outputs a $U_n \in \mathcal{C}$ such that $G_f \subset U_n \subset B(G_f, 2^{-n})$. We set $k = 1$, and repeat the following operations, increasing k by one each time:

1. Query for $d = \phi(k)$ such that $|d - x| < 2^{-k}$,

2. Compute $U_k = M(k)$,

3. If the set Y_k of y such that there is a $z \in (d - 2^{-k}, d + 2^{-k})$ with $(z, y) \in U_k$ is bounded within some interval $(s - 2^{-n}, s + 2^{-n})$, return s.

Note that all the operations performed are operations with dyadic numbers which can be easily executed. We need to show that this algorithm terminates and outputs a 2^{-n}-approximation of $f(x)$ upon termination.

Suppose that the algorithm terminates. Note that $x \in (d - 2^{-k}, d + 2^{-k})$ and $(x, f(x)) \in G_f \subset U_n$, hence $f(x)$ is among the y's taken into account when terminating, and so $f(x) \in (s - 2^{-n}, s + 2^{-n})$. We obtain $|s - f(x)| < 2^{-n}$, so s is a valid output.

To show that the algorithm terminates, we will show that the set of y's discussed above must shrink to a point. That is, for any $\varepsilon > 0$, there is a k such that the set Y_k has radius less than ε. f is computable, hence it is continuous at x, so there is a $\delta > 0$ such that $|f(z) - f(x)| < \varepsilon/2$ whenever $|x - z| < \delta$. Choose k so that $2^{-k} < \min\{\delta/3, \varepsilon/2\}$. We claim that for any $y \in Y_k$, $|f(x) - y| < \varepsilon$.

Let $y \in Y_k$. This means that there is a $z \in (d-2^{-k}, d+2^{-k}) \subset (x-2\cdot 2^{-k}, x+2\cdot 2^{-k})$ such that $(z, y) \in U_k \subset B(G_f, 2^{-k})$. Hence there must be a w such that $|(z, y) - (w, f(w))| < 2^{-k}$. In particular, $|z - w| < 2^{-k}$ and $|y - f(w)| < 2^{-k}$. Hence $|w - x| \leq |z - x| + |w - z| < 2 \cdot 2^{-k} + 2^{-k} = 3 \cdot 2^{-k} < \delta$. So $|f(w) - f(x)| < \varepsilon/2$, and $|f(x) - y| \leq |f(x) - f(w)| + |f(w) - y| < \varepsilon/2 + 2^{-k} < \varepsilon$, which completes the proof. ■

2.3.2 The Non-Continuous Functions Case

The tight connection between computability of a function and its graph suggests a way to extend the definition of computable functions beyond the continuous case.

The computability of the graph G_f of a function f could be possibly used as a criterion in defining a richer class of computable functions. This leaves room for further research in this direction. More will be discussed in section 5.1.

Chapter 3

Computational Complexity of Real Sets

In chapter 2 we have seen five equivalent definitions for the computability of real sets. For each of them it is not hard to associate the notions of time and space complexity. We will first define the complexity notions associated with the different definitions, and decide on the "right" notion of complexity describing the "true" complexity of a set. We will then show that the local complexity notions are all different unless $P = NP$.

3.1 Defining the Complexity of Sets

3.1.1 The Global Complexity

The definition of global complexity follows naturally from the definition 2.2.2 of global computability.

Definition 3.1.1 *Let $T : \mathbb{N} \to \mathbb{N}$ be a time function. A compact set $C \subset \mathbb{R}^k$ is said to be* **globally computable** *in time $T(n)$ if there exists a Turing Machine $M(n)$ running in time $T(n)$ which on input n outputs an encoding of a set $U_n \in \mathcal{C}_k$ such that $C \subset U_n \subset B(C, 2^{-n})$.*

Computability in bounded space $S(n)$ is defined in a similar way.

We would like to have a notion of "efficiently computable" sets, similarly to the notion of poly-time languages in the discrete case.

Letting C be efficiently computable if we can make $T(n) < p(n)$ for some polynomial p seems too restrictive. U_n is supposed to be a 2^{-n}-approximate picture, hence it is reasonable to expect it to contain many balls of size $\sim 2^{-n}$, and if C is a reasonably large set it would take exponentially many such balls just to cover it. As a specific example consider the segment $[0,1] \times \{0\} \subset \mathbb{R}^2$. It is easy to see that any ball in U_n can have a radius of at most 2^{-n}, and hence covers at most a $2 \cdot 2^{-n}$ section of the segment. Hence any U_n contains at least 2^{n-1} balls – exponentially many. So $T(n)$ for computing this simple set would be exponential in n. This suggests that a globally efficiently computable set should be computable in time polynomial in 2^n rather than in n.

Definition 3.1.2 *We say that a set C is* **globally poly-time computable** *if it is globally computable in time $2^{O(n)}$.*

It is trivial to give definitions similar to 3.1.1 and 3.1.2 for Hausdorff approximability instead of global computability.

Definition 3.1.3 *Let $T : \mathbb{N} \to \mathbb{N}$ be a time function. A compact set $C \subset \mathbb{R}^k$ is said to be* **Hausdorff approximable** *in time $T(n)$ if there exists a Turing Machine $M(n)$ running in time $T(n)$ which on input n outputs an encoding of a set $V_n \in \mathcal{C}_k$ such that $d_H(C, V_n) < 2^{-n}$.*

Definition 3.1.4 *We say that a set C is* **poly-time Hausdorff approximable** *if it is Hausdorff approximable in time $2^{O(n)}$.*

As in theorem 2.2.6, one can see that global poly-time computability is equivalent to poly-time Hausdorff approximability.

Theorem 3.1.5 *Definitions 3.1.2 and 3.1.4 are equivalent, that is, a set C is globally poly-time computable if and only if it is poly-time Hausdorff approximable.*

3.1.2 The Local Complexity

We can now define the complexity analogues of the local computability definitions. We will see that for many applications local complexity is the "right" measure on the complexity of the set. We start by presenting the time-complexity analogues of definitions 2.2.7, 2.2.10 and the distance function computability.

Definition 3.1.6 *Let $T : \mathbb{N} \to \mathbb{N}$ be a time function. We say that the compact set $C \subset \mathbb{R}^k$ is* **locally computable** *in time $T(n)$, if there is a Turing Machine $M(d, n)$ running in time at most $T(n + |d|)$, which computes a function from the family given by (2.4).*

We say that C is **locally poly-time computable**, *or just* **poly-time computable**, *if it is computable in time $p(n)$, for some polynomial p.*

This is the definition introduced in [BW99] and [Wei00] and used in [RW03]. We will see later some intuition which illustrates why this is the "right" definition which measures the computational complexity of the set C.

Definition 3.1.7 *Let $T : \mathbb{N} \to \mathbb{N}$ be a time function. We say that the compact set $C \subset \mathbb{R}^k$ is* **Ko computable** *in time $T(n)$, if there is an oracle Turing Machine $M^\phi(n)$ running in time at most $T(n)$, which computes a function from the family given by (2.6). Here we charge one time unit for each query to the oracle, but our ability to query the oracle is limited by the fact that it takes n time units to read the answer of $\phi(n)$.*

We say that C is **Ko poly-time computable**, *or just* **Ko P-computable**, *if it is computable in time $p(n)$, for some polynomial p.*

Definition 3.1.7 for poly-time computability for sets was introduced by Chou and Ko in [CK95] under the name of *strong P-recognizability*. We give it a different name here since, in fact, this definition is *weaker* than the other definitions for poly-time computability of sets presented here.

The third poly-time computability notion corresponds to the computability of the distance function d_C.

Definition 3.1.8 *We say that the compact set $C \subset \mathbb{R}^k$ is* **distance computable** *in time $T(n)$ if the function $d_C(x)$ is computable in time $T(n)$ as per definition 2.1.9.*

We say that C is **distance poly-time computable** *or* **distance P-computable** *if it is distance computable in time $p(n)$ for some polynomial $p(n)$. In other words, C is distance poly-time computable if the function $d_C(x)$ is a poly-time computable function.*

It is very easy to give the analogies of definitions 3.1.6, 3.1.7 and 3.1.8 for space complexity instead of time complexity.

3.1.3 The Intuition Behind the Local Complexity

Our goal in discussing the real set computability and complexity is to determine which sets can be computed and how fast they can be computed. Computing the set C in our setting corresponds to generating a 'good' image of C. In dimension 2 a set is computable if and only if we can display arbitrarily precise images C.

Computing one value of $f(d, n)$ in (2.4) corresponds to deciding whether to draw one pixel of size $\sim 2^{-n}$ with center d in the image of C or not. As seen in the proof of theorem 2.2.8, it would take $\Theta(2^{2n})$ calculations of $f(\bullet, n)$ to generate a 2^{-n}-two-dimensional approximation of C and display it. Suppose that the complexity of f is $T(n)$, then it would take $O(2^{2n} \cdot T(n))$ time to generate such an image, and the product's growth is most likely dominated by the 2^{2n} term. Such an image would be of an exponential size, and completely impractical to display.

Usually, when we display a strong zoom-in into the image, all we can display is a very small portion of the set. Suppose we have a display with resolution 1000×1000 pixels. In this case it is only meaningful to try to compute 10^6 pixels of the set and display them. Hence the complexity of displaying such a portion with a zoom-in of 2^n is $O(10^6 \cdot T(n))$.

This product's growth is dominated by $T(n)$. If $T(n)$ grows slowly, say polynomially, then it should be easy to draw such a zoom-in, while if $T(n)$ grows fast, it isn't easy.

This shows that the local complexity is the true complexity of generating zoom-ins into the set, and the poly-time computable sets as per definition 3.1.6 are the easy-to-draw and easy-to-zoom-in sets.

Having established the importance of the local set complexity one could ask whether the analogue of theorem 2.2.12 holds here. In particular are definitions 3.1.6, 3.1.7 and 3.1.8 for poly-time sets equivalent? The answer is negative in general unless $P = NP$, as will be seen in the next section.

3.2 Comparing the Local Complexity Definitions

Our goal is to compare definitions 3.1.6, 3.1.7 and 3.1.8 for poly-time computable sets. We will see that under the assumption that $P \neq NP$ distance P-computability is strictly stronger than poly-time computability, which is strictly stronger than Ko P-computability.

We start by comparing distance P-computability to poly-time computability.

3.2.1 Distance P-Computability vs Poly-Time Computability

In this section we will prove that distance P-computability is stronger than poly-time computability, and it is strictly stronger if $P \neq NP$. Formally, we prove the following theorem.

Theorem 3.2.1 *Let $C \subset \mathbb{R}^k$ be a compact set.*

1. *If C is distance P-computable, then C is poly-time computable,*
2. *for $k = 1$ the converse holds: if C is poly-time computable, then it is distance P-computable,*

3. *for $k > 1$ the converse is equivalent to $P = NP$: "if C is poly-time computable, then it is distance P-computable in general" $\Leftrightarrow$ "$P = NP$", which is most unlikely. Hence in this case distance P-computability is strictly stronger than poly-time computability.*

We see that there is a substantial difference between the case when $k = 1$ and the case when $k > 1$. Intuitively, in the one-dimensional case for any x there are only two "candidates" which could be the closest point to x in C. We can then apply a binary-search-like technique to localize them in two small intervals and choose the closer one. On the other hand, in higher dimensions, there are exponentially many directions and it is NP-hard to choose the one which leads to the closest point to x in C.

Proof: **Part 1:** This follows immediately from the reduction in the proof of the "d_C is computable $\Rightarrow$ C is locally computable" direction of theorem 2.2.9. The reduction is trivially poly-time.

Part 2: Suppose that $k = 1$, that is $C \subset \mathbb{R}$ is a one-dimensional compact set, and suppose it is poly-time computable, we want to show that its distance function $d_C(x)$ is poly-time computable. By the assumption we have a Turing Machine M(d,n) running in time polynomial in $|d| + n$ and computing a function $f(d, n)$ from the family (2.4). C is compact, so for simplicity we can assume that $C \subset [0, 1]$ and $C \neq \emptyset$. We have an oracle ϕ for x, and we need to compute $d_C(x)$ within an error of 2^{-n} in time polynomial in n.

We first run $\phi(n+1)$ to obtain a $2^{-(n+1)}$-approximation d of x with $|d| \leq n+2$. Without loss of generality we can assume that $x \in (-0.5, 1.5)$ (the construction generalizes trivially for other values of x). Denote the points in C which are closest to d by $l < d$ and $r > d$. If $d \in C$ then $l = r = d$ (such points must exist by the compactness of C). We have $d_C(d) = \min(r - d, d - l)$.

We will estimate $d_C(d)$ by computing a sequence of numbers $a_0, a_1, \ldots$ such that $|d_C(d) - a_i| < 2^{-i}$, and $a_i = \frac{k_i}{2^{n+2}}$ for some integer k_i. To compute a_0 run $M(d, 1)$. If $M(d, 1) = 0$ set $a_0 = 1$, otherwise set $a_0 = 0$. It is easy to see from the definition

of the function computed by M that if $a_0 = 0$, then $d_C(d) < 1$ and if $a_0 = 1$, then $1/2 \leq d_C(d) \leq 3/2$, and in either case $|d_C(d) - a_0| < 1$.

Now we have a_i and we would like to compute a_{i+1} satisfying the condition. Consider $S_i = (d - a_i - 2^{-i}, d - a_i + 2^{-i}) \cup (d + a_i - 2^{-i}, d + a_i + 2^{-i})$. Then S_i is either a union of two intervals or a single interval of total length $\leq 4 \cdot 2^{-i}$. Moreover, the point closest to d in C is contained in S_i. Cover S_i with balls of radius $2^{-(i+2)}$ and centers which are integer multiples of $2^{-(i+2)}$. This can be done with at most 18 balls. For each ball of this form with center c_j run $M(c_j, i+2)$. Let c_k be a c_j with $M(c_j, i+2) = 1$ which is closest to d. Take $a_{i+1} = |d - c_k|$. Obviously a_{i+1} is of the right form. We claim that $|d_C(d) - a_{i+1}| < 2^{i+1}$.

Let $p \in S_i$ be the point in C which is closest to d ($p = d$ if $d \in C$). Let c_j be the center we have considered which is closest to p. Then $|c_j - p| < 2^{-(i+2)}$, hence (recall that $p \in C$) $M(c_j, i+2) = 1$, so the c_k chosen is at least as close to d as c_j, so $a_{i+1} = |d - c_k| \leq |d - c_j| \leq |d - p| + |p - c_j| < |d - p| + 2^{-(i+2)} = d_C(d) + 2^{-(i+2)}$.

On the other hand, $M(c_k, i+2) = 1$, hence $d_C(c_k) < 2^{-(i+1)}$, and $d_C(d) \leq |d - c_k| + d_C(c_k) < a_{i+1} + 2^{-(i+1)}$. So $-2^{-(i+2)} < d_C(d) - a_{i+1} < 2^{-(i+1)}$, and $|d_C(d) - a_{i+1}| < 2^{-(i+1)}$.

Returning a_i will give the desired answer. Note that we make $18n$ calls to the polynomial machine M with parameter of length at most $n + 2$, and so the algorithm described above is polynomial.

Part 3: We will prove this part for dimension $k = 2$. It is easy to see that the proof generalizes to higher dimensions. There are two directions in the proof, we start with the harder direction.

"if C is poly-time computable, then it is distance P-computable" $\Rightarrow$ "$P = NP$"

We construct a poly-time computable set C such that distance P-computing C would allow us to efficiently solve SAT.

We subdivide the square $[0,1] \times [0,1]$ into infinitely many squares as on figure 3.1. Note that there are 2^i squares with side length 2^{-i} for $i \geq 1$ to a total area of $\sum_{i=1}^{\infty} 2^i \cdot (2^{-i})^2 =$

$\sum_{i=1}^{\infty} 2^{-i} = 1$. We give the squares a natural numbering and it is easy to recover the square b_n from its index n. The side length of b_n is roughly $\frac{1}{n}$.

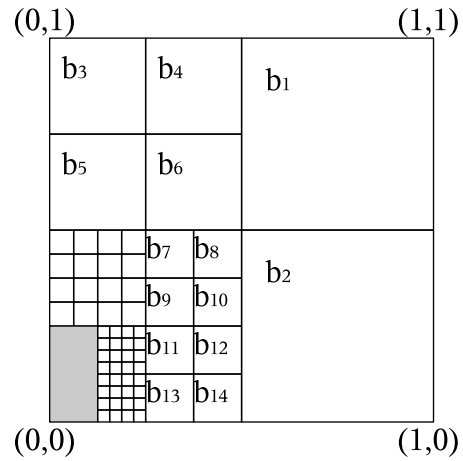

Figure 3.1: The subdivision of the $[0,1] \times [0,1]$ square.

Consider some standard encoding $e : \phi_n \mapsto n$ which encodes boolean formulas with numbers so that the length of ϕ_n is within a linear factor of the length of n. We can now describe the set C. For each n consider the square b_n. Denote by 2^{-i} the side length of b_n, and its center by c_n. i is roughly $\log n$. We include the circle with center c_n and radius $2^{-(i+2)} = b_n/4$ in the set C. Also, if n is an encoding of some formula ϕ with $k \leq i$ variables (the number of variables cannot exceed the length of the encoding) we include additional points in the set as follows.

Consider the circle S with center at c_i and radius $2^{-(i+2)} - 2^{-(i+k+2)}$ (S is inside the circle we have drawn before, and is very close to it). Subdivide S into 2^k equal parts corresponding in a natural way to the truth assignments for ϕ_n. Denote the partition by $S = S_1 \cup S_2 \cup \ldots \cup S_{2^k}$. For a truth assignment x for ϕ_n corresponding to some S_l we include the middle third of S_l in C if and only if $\phi_n(x) = 1$. In particular, if ϕ_n is unsatisfiable, then there is no part of S included in C. On figure 3.2 we see an illustration of the construction above for $\phi_n(x, y, z) = (x \oplus y) \wedge z$. To make the set closed we also

add the point $\{(0,0)\}$ into the set.

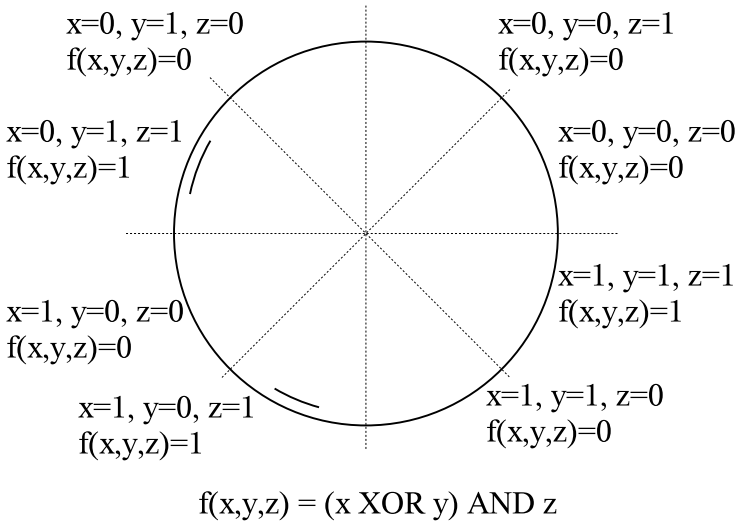

Figure 3.2: The circles corresponding to $f(x,y,z) = (x \oplus y) \wedge z$.

We first observe that C is poly-time computable. We do not present the formal proof because it would be a very technical computation which does not add to the understanding of the construction. Intuitively, if we try to draw a low resolution pixel, we don't need to distinguish between the inner and the outer circle, while if we try to draw a high resolution pixel, we will only have to worry about at most one or two segments of the inner circle corresponding to one or two truth assignments.

Given a dyadic point d and an integer n we would like to output 1 if $B(d, 2^{-n}) \cap C \neq \emptyset$ and 0 if $B(d, 2 \cdot 2^{-n}) \cap C = \emptyset$. We first check the circles to which d is closest. If we find more than 4 circles which are 2^{-n}-close to d, we can output 1. Otherwise, if d is outside all the circles, we can easily check whether to output 1 or 0 without worrying at all about the inner circles. Suppose that d is inside some circle S_n of radius $2^{-(i+2)}$ corresponding to the formula ϕ_n with k variables. Denote its inner circle by S. If 2^{-n} is bigger than twice the distance $2^{-(i+k+2)}$ between the circles, then we can compute $f(d,n)$ without worrying about the inner circle, and if 2^{-n} is smaller than twice this distance,

then the size of $B(d, 2^{-n})$ is comparable to the size of the segments on the inner circles corresponding to the truth assignments to ϕ_n, and to compute $f(d, n)$ we would have to check at most two such assignments.

Now suppose that C is distance P-computable. That is, the distance function $d_C(x)$ is poly-time computable. We will show how to solve SAT in poly-time. Suppose we are given a boolean formula ϕ. By the encoding e we have selected $\phi = \phi_n$ for some n such that $\log n = O(|\phi|)$. We can then find the square b_n with a dyadic center c_n, and side length 2^{-i}, $i = \Theta(\log n)$. Let $k \leq |\phi|$ be the number of variables in ϕ. If ϕ is satisfiable, the inner circle is not empty in C, and $d_C(c_n) = 2^{-(i+2)} - 2^{-(i+k+2)}$, and if ϕ is unsatisfiable then the inner circle is empty, and $d_C(c_n) = 2^{-(i+2)}$. So all we have to do to decide the satisfiability of ϕ is to compute $d_C(c_n)$ within a precision $2^{-(i+k+3)}$. This can be done in time polynomial in $(i + k + 3)$, which is also polynomial in $|\phi|$.

"$P = NP$" $\Rightarrow$ "if C is poly-time computable, then it is distance P-computable"

Suppose we want to compute $d_C(x)$ with precision 2^{-n}. We can query the oracle for x for a dyadic d with $n + 2$ digits so that $|d - x| < 2^{-(n+1)}$. All we have to do now is to compute $d_C(d)$ with precision $2^{-(n+1)}$.

To do this we can use part 2. Consider the set $C_0 \subset \mathbb{R}$ defined as

$$C_0 = \{\pm l \ : \ l = d(d, x) \ \text{ for some } x \in C\}.$$

Obviously $d_C(d) = d_{C_0}(0)$. All we have to do is to show that C_0 is a poly-time computable set. On an input (a, n) we need to output $f(a, n) = 1$ if $B(a, 2^{-n}) \cap C_0 \neq \emptyset$ and $f(a, n) = 0$ if $B(a, 2 \cdot 2^{-n}) \cap C_0 = \emptyset$. Let $M(c, n)$ be the machine computing the set C. Consider

$$f(a, n) = \exists b \in \mathbb{D}^k_{n+k+4} \ : \ (a - 2^{-n} < |d - b| < a + 2^{-n}) \wedge (M(b, n + 1) = 1). \qquad (3.1)$$

Suppose $B(a, 2^{-n}) \cap C_0 \neq \emptyset$, then there is an $x \in C$ with $a - 2^{-n} < |d - x| < a + 2^{-n}$, and it is not hard to see that we can find a b as in (3.1).

On the other hand, if $B(a, 2 \cdot 2^{-n}) \cap C_0 = \emptyset$, then all points b as in (3.1) are at least 2^{-n}-far from C, and hence $M(b, n + 1) = 0$ for such b's.

To finalize the proof we observe that f is in poly-time by the assumption that $P = NP$. ■

Our next goal is to compare poly-time computability to Ko P-computability.

3.2.2 Poly-Time Computability vs Ko P-Computability

In this section we will show that poly-time computability implies Ko P-computability, and the converse holds if and only if $P = NP$, we will also show a much weaker version of the converse, showing that Ko P-computability implies exponential-time computability. Unlike the results in the previous section, the results here hold for any dimension $k \geq 1$.

Theorem 3.2.2 *Let $C \subset \mathbb{R}^k$ be a compact set.*

1. *If C is poly-time computable, then C is Ko P-computable,*

2. *the converse is equivalent to $P = NP$: "if C is Ko P-computable, then it is poly-time computable in general" $\Leftrightarrow$ "$P = NP$", which is most unlikely. Hence poly-time computability is strictly stronger than Ko P-computability,*

3. *a weaker version of the converse holds: if C is Ko P-computable, then it is exponential time computable. Moreover, if the machine which Ko P-computes C reads at most $p(n)$ bits of the input, then C is computable in time $n^{O(1)}2^{O(p(n)+n)} = 2^{O(p(n)+n)}$.*

Proof: **Part 1:** This follows immediately from the proof of the "C is locally computable $\Rightarrow$ C is Ko computable" direction of theorem 2.2.11. The reduction is obviously poly-time.

Part 2: "if C is Ko P-computable, then it is poly-time computable" $\Rightarrow$ "$P = NP$"

We will prove this for dimension $k = 1$. The proof generalizes trivially to higher dimensions.

We construct a one dimensional Ko P-computable set such that if C is poly-time computable then $P = NP$. We subdivide the interval $[0, 1]$ into infinitely many closed disjoint intervals $K_1, K_2, \ldots$ such that the length of K_i is $\Theta(1/i^2)$. To be concrete, we take $K_i = [1/(i+1), 1/i]$, as seen on figure 3.3. Each interval corresponds to an encoding of a propositional formula or to nothing.

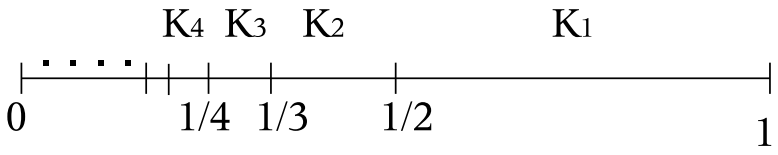

Figure 3.3: The subdivision of the $[0, 1]$ interval

If K_i corresponds to nothing we just include the middle fifth of K_i. Otherwise, if K_i corresponds to a formula ϕ with r variables, we subdivide the middle fifth of K_i into 2^r subintervals, where each subinterval corresponds to a truth assignment τ for ϕ. Note that $r < \log i$, because i is the encoding of ϕ. We include the interval if and only if $\phi^\tau = TRUE$. We also include the point $\{0\}$ in the set. Denote the obtained set by C.

Our first claim is that C is Ko P-computable. Given an oracle ϕ representing a number x, a machine $M^\phi(n)$ which Ko computes C works as follows. If $x < 2^{-n}$ then output 1, this would be correct, since $0 \in C$. Otherwise we are sure that $x > 2^{-n-1}$ and so $x \in K_i$ for $i < 2^{3n}$. Hence by querying $O(n)$ bits of x we can find the two subintervals of some K_i's in which x might lye. From here it is a matter of decoding and making two substitutions before we can output a valid answer for $M^\phi(n)$. Everything is done in polynomial time, and hence C is Ko P-computable.

Now we show that if C is poly-time computable, then $P = NP$. Given a propositional formula ϕ of length $n = |\phi|$ we look at its encoding i. The length of the encoding is $\Theta(n)$. Hence the length of the interval K_i is $2^{-\Theta(n)}$. Thus, we can place one question of the

form $f(d,l)$, where d is close to the middle of K_i and

1. $\overline{B(d,2^{-l})}$ covers the middle fifth of K_i,
2. $\overline{B(d,2\cdot 2^{-l})} \subset K_i$,
3. $l = O(n)$.

$f(d,l)$ returns 1 if and only if ϕ is satisfiable. By the assumption f runs in time polynomial in l, and hence polynomial in n. So $SAT \in P$, and hence $P = NP$, which completes the proof.

"$P = NP$" $\Rightarrow$ "if C is Ko P-computable, then it is poly-time computable"

This part of the proof gives a nondeterministic poly-time algorithm to compute a Ko P-computable set. The proof is carried out for dimension 1, but same arguments hold for an arbitrary dimension k. Suppose that $P = NP$ and that C is Ko P-computable, we want to prove that C is poly-time computable. The proof can be viewed a special modification of the "C is Ko computable $\Rightarrow$ C is locally computable" direction proof in theorem 2.2.11. Given the numbers $d \in \mathbb{D}_n$ and l we want to compute the value of $f(d,l)$ as in the definition of computability in time polynomial in $n = |d|$ and l. Suppose that the machine $M^{\phi}(n)$ Ko computing C as in definition 2.2.10 runs in polynomial time $p(n)$. Then M can only query x within a precision of $2^{-p(n)}$, since it can only read $p(n)$ bits of the answer from the oracle ϕ. This claim is formalized as follows.

Lemma 3.2.3 *If x is a dyadic number in $\mathbb{D}_{p(n)}$ and $|y-x| \leq 2^{-p(n)-1}$, then there is an oracle ψ representing y and an oracle ϕ' representing x such that $M^{\phi'}(n) = M^{\psi}(n)$.*

Proof: Let ψ work as follows. On input r, ψ outputs $\psi(r)$ such that $|y-\psi(r)| < 2^{-r-1}$ and $\psi(r) \in (x - 2^{-p(n)-1}, x + 2^{-p(n)-1})$ with $\psi(r) \in (x - 2^{-p(n)-1}, x)$ if $y < x$ and $\psi(r) \in [x, x + 2^{-p(n)-1})$ otherwise. Of course, the oracle ψ satisfies the requirement from an oracle as per definition 2.1.6. On input r the oracle ϕ' for x works as follows. It outputs

$\phi'(r)$ such that $|\phi'(r)-x| < 2^{-r}$ and that $\phi'(r) \in [x, x+2^{-p(n)-1})$ if $\psi(r) \in [x, x+2^{-p(n)-1})$, and $\phi'(r) \in (x-2^{-p(n)-1}, x)$ if $\psi(r) \in (x-2^{-p(n)-1}, x)$.

Note that both ϕ' and ψ are valid oracles for x and y respectively, and that they always agree on the first $p(n)+1$ digits, and so the $p(n)$-time bounded computation of M with these two oracles gives the same answer. This completes the proof of the lemma. ■

Suppose that for some $x \in \mathbb{D}_{p(n)}$, $[x-2^{-p(n)-1}, x+2^{-p(n)-1}] \cap C \neq \emptyset$, we define the following two oracles for x:

1. ϕ' outputs precisely x, denote this oracle ϕ_x^+,

2. ϕ' outputs x with infinitely many 1's at the end, denote this oracle ϕ_x^-.

Suppose $y \in [x-2^{-p(n)-1}, x+2^{-p(n)-1}] \cap C$ then by the lemma there is an oracle ψ for y and an oracle ϕ' for x such that $M^{\phi'}(n) = M^{\psi}(n)$. A closer look at the proof shows that we can choose $\phi' = \phi_x^+$ or $\phi' = \phi_x^-$. By the definition of Ko P-computability, $M^{\psi}(n) = 1$, and hence either $M^{\phi_x^+}(n) = 1$ or $M^{\phi_x^-}(n) = 1$. Note that both $M^{\phi_x^+}(n)$ and $M^{\phi_x^-}(n)$ can be simulated by a non-oracle machines $M^+(x,n)$ and $M^-(x,n)$ which just simulate the oracle by writing the appropriate presentation of x. Both M^+ and M^- run in time polynomial in n and $|x|$. We summarize as follows, for $x \in \mathbb{D}_{p(n)}$,

$$[x-2^{-p(n)-1}, x+2^{-p(n)-1}] \cap C \neq \emptyset \Rightarrow (M^+(x,n)=1) \vee (M^-(x,n)=1), \tag{3.2}$$

where M^+ and M^- run in time polynomial in n provided that $p(n)$ is a polynomial.

In the opposite direction, it follows from the definition of Ko P-computability that

$$[x-2^{-n}, x+2^{-n}] \cap C = \emptyset \Rightarrow (M^+(x,n)=0) \wedge (M^-(x,n)=0). \tag{3.3}$$

Given a pair (d,l) as in the definition of computability, we first choose n such that $n, p(n) > l+2$. We claim that the function

$$f(d,l) = \exists x \in \mathbb{D}_{p(n)} \cap \overline{B(d, 2^{-l}+2^{-p(n)})} : (M^+(x,n)=1) \vee (M^-(x,n)=1) \tag{3.4}$$

outputs the correct answer for computing C as required by the definition. First of all, since we have assumed that $P = NP$ and obviously $f(d,l) \in NP$, $f(d,l)$ as defined above is poly time computable.

We now need to prove that f outputs the correct answer.

(i) Suppose $\overline{B(d,2^{-l})} \cap C \neq \emptyset$. Let $y \in \overline{B(d,2^{-l})} \cap C$, then there is an $x \in \mathbb{D}_{p(n)}$ with $|y-x| \leq 2^{-p(n)-1}$. We have

$$|x-d| \leq |x-y| + |y-d| \leq 2^{-p(n)-1} + 2^{-l} < 2^{-l} + 2^{-p(n)},$$

hence $x \in \mathbb{D}_{p(n)} \cap \overline{B(d,2^{-l}+2^{-p(n)})}$. We also have $y \in [x-2^{-p(n)-1}, x+2^{-p(n)-1}] \cap C$, and so by (3.2), $(M^+(x,n)=1) \vee (M^-(x,n)=1)$ holds, and $f(d,l)=1$ in this case.

(ii) Suppose $\overline{B(d,2\cdot 2^{-l})} \cap C = \emptyset$. Then for any $x \in \mathbb{D}_{p(n)} \cap \overline{B(d,2^{-l}+2^{-p(n)})}$, if $y \in [x-2^{-n}, x+2^{-n}]$, then

$$|y-d| \leq |x-d| + |x-y| \leq 2^{-l} + 2^{-p(n)} + 2^{-n} < 2^{-l} + 2^{-l-1} + 2^{-l-1} = 2\cdot 2^{-l},$$

hence $y \notin S$. We have obtained $[x-2^{-n}, x+2^{-n}] \cap C = \emptyset$, and by (3.3), $(M^+(x,n) = 1) \vee (M^-(x,n)=1)$ does not hold for any x. So $f(d,l)=0$ in this case, which completes the proof.

Part 3: This part is obtained immediately by applying brute force to evaluate $f(d,l)$ according to (3.4), yielding an exponential algorithm for computing C. ■

3.2.3 Comparing Local and Global Poly-Time Computability

The comparison between the local notions of poly-time computability and the global ones is not completely fair. The definitions of local P-computability requires some local information to be computed with precision 2^{-n} in time polynomial in n, while the global poly-time computability allows exponential time to compute the entire set. In particular, if we use exponentially many local computations to draw the entire set C (an this is often the case), we can use exponentially much time for each local computation, because

$2^{O(n)} \times 2^{O(n)} = 2^{O(n)}$. This suggests that the notion of global poly-time computability should be weaker than the notion of local poly-time computability.

In fact, the the class of locally poly-time computable sets is analogous to the complexity class P, while the class of the globally poly-time computable sets is analogous to $E = DTIME(2^{O(n)})$ (not to be confused with $EXP = DTIME(2^{n^{O(1)}})$). It is known that $P \subset E$ and the inclusion is strict, so local poly-time computability is strictly weaker than global poly-time computability.

Theorem 3.2.4 *Local poly-time computability is strictly stronger than global poly-time computability. That is, if a set is locally poly-time computable then it is globally poly-time computable. On the other hand, there is a set A which is globally poly-time computable but not locally poly-time computable.*

Proof: The first statement follows immediately from the the construction in the "C is locally computable $\Rightarrow$ C is globally computable" direction of theorem 2.2.8.

For the second statement we use the partition of the $[0,1]$ interval from the proof of theorem 3.2.2. Recall that we have $K_i = [1/(i+1), 1/i]$ for $i = 1, 2, 3, \ldots$. Let $f : \mathbb{N} \to \{0,1\}$ be a function in $E \backslash P$ (here the complexity of computing $f(n)$ is measured in the *size* $\log n$ of the input). Let

$$A_f = \{0\} \cup \bigcup_{f(i)=1} K_i.$$

We claim that A_f is globally but not locally poly-time computable.

To see that A_f is globally poly-time computable, observe that to obtain a 2^{-n}-image of A_f we only need to worry about $K_1, K_2, \ldots, K_{2^n}$, hence by making 2^n evaluations of f on inputs of size at most n we can generate the image. All this takes $2^{O(n)}$ time because $f \in E$.

Assume that A_f is locally poly-time computable. Then by running the machine computing A_f on a point close to the middle of K_i with precision parameter $\sim \log i^2$ to

obtain the value of $f(i)$. So $f(i)$ is computable in time $poly(\log(i^2)) = poly(\log i)$, and $f \in P$. Contradiction. ■

Continuing the same analogy, the weaker property of Ko P-computability corresponds to the class NP. It can be seen that the relation between Ko P-computability and the global poly-time computability partially corresponds to the relation between the classes NP and E. As of now, it is unknown whether $NP \subset E$.

Theorem 3.2.5 *1. Ko P-computability implies global poly-time computability if and only if $NP \subset E$,*

2. Global poly-time computability does not imply Ko P-computability.

Proof: We will only give sketches of the proofs.

Part 1. Ko P-computability implies global poly-time computability $\Rightarrow NP \subset E$.

Note that $SAT \in E$ (which is obviously true) does not imply that $NP \subset E$ (which is unknown). Instead, we observe that the construction in the proof of theorem 3.2.2 can be applied with any $L \in NP$ instead of SAT. We would obtain a set S_L which is Ko P-computable, and globally poly-time computing it would allow us to decide L in $2^{O(n)}$ time.

$NP \subset E \Rightarrow$ Ko P-computability implies global poly-time computability.

Assume $NP \subset E$. By the constriction in the proof of theorem 3.2.2 we can use (3.4) to locally compute a Ko P-computable set S in $2^{O(n)}$ time. We then use $2^{O(n)}$ applications of the local computation to poly-time globally compute S.

Part 2. Consider the set A_f from the proof of the previous theorem 3.2.4. It is globally poly-time computable, but Ko P-computing it would involve evaluating $f(n)$ in polynomial time, which is impossible. ■

Chapter 4

Complexity of Hyperbolic Julia Sets

4.1 Introduction

Julia sets are some of the best known illustrations of a highly complicated chaotic system generated by a very simple mathematical process. These sets have been deeply studied in the framework of complex dynamics during the last century. Julia sets are not only an intriguing mathematical object, but also a major source of amazing images. Many computer programs have been written to generate these images. Algorithms for computing Julia sets have been presented and discussed in [Sau87], [Pick98] and [Mil00] (Appendix H), for example.

It appears, however, that none of the algorithms and their implementations cope well with zooming in. With the computer using fixed-precision numbers, rounding errors affect the computation badly when we try to zoom in. These programs also seem to work poorly near some "hard" polynomials, for example, with $p(z) = z^2 + 1/4 + \varepsilon$, $0 < \varepsilon \ll 1$. We will return to this example in section 4.7.

We give the first polynomial bound on the complexity of an arbitrary hyperbolic Julia set. The class of hyperbolic polynomials is very rich. For example, in the case $p(z) = z^2 + c$, $p(z)$ is hyperbolic for all c's outside and for all known c's in the interior

of the Mandelbrot set (but not on the boundary). It is a major open problem whether $z^2 + c$ is hyperbolic for every c in the interior of the Mandelbrot set (see [McMl94] for more information).

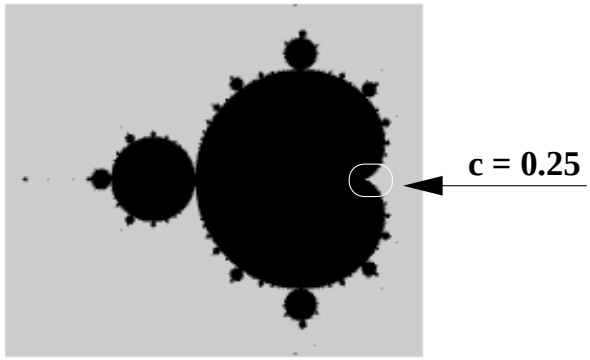

Figure 4.1: The Mandelbrot set with the point $c = 1/4$ highlighted.

Although the hyperbolic Julia sets were shown to be recursive in [Zh98], complexity bounds were proved only in a restricted case in [RW03]. In accordance with the discussion in section 3.1 we show that hyperbolic Julia sets are locally poly-time computable as per definition 3.1.6. This result is a generalization of the result in [RW03], where it has been shown that a special type of hyperbolic Julia sets of the form J_{z^2+c} for $|c| < 1/4$ are poly-time computable.

The algorithm that we present is not uniform in $p(z)$. That is, the Turing Machine computing $J_{p(z)}$ depends on $p(z)$. We will show in section 4.7, however, that no uniform Turing Machine computing $J_{p(z)}$ exists. Our algorithm can be modified to be uniform in the *hyperbolic* $p(z)$'s, such a modification will remain polynomial in the precision of the computation, but might be arbitrarily hard in the polynomial $p(z)$. Explicitly, we obtain a bound of $K(p) \cdot nM(n)$, where $K(p)$ is a coefficient depending on the polynomial, 2^{-n} is the required precision and $M(n)$ is the complexity of multiplying two n-bit numbers.

4.2 Julia Sets and Hyperbolic Julia Sets

We will give one of the equivalent definitions of the hyperbolic Julia set. More detailed information, as well as proofs and further references can be found in [Mil00] and [McMl94]. [Mil00] gives a particularly good exposition of the hyperbolic Julia sets.

For the rest of the chapter we fix our polynomial to be $p(z)$. Note that $p(z)$ is a polynomial with *complex* coefficients. Let $p^k(z)$ denote the k-th iteration of $p(z)$, i.e. $p^1(z) = p(z)$, $p^2(z) = p(p(z))$ and in general $p^{k+1}(z) = p(p^k(z))$. By a convention, $p^0(z) = z$. We define the **orbit** of z as the sequence $(z, p(z), p^2(z), \ldots)$.

A point z is called **periodic** if $p^k(z) = z$ for some $k \geq 1$. The minimal such k is called the **period** of z. A periodic point z with period k and its (finite in this case) orbit $(z, p(z), \ldots, p^{k-1}(z))$ are said to be **attracting** if $|(p^k)'(z)| < 1$ and **repelling** if $|(p^k)'(z)| > 1$. If we iterate a point in a small neighborhood of an attracting periodic point, we will approach the attracting orbit, while if we iterate a point in a small neighborhood of a repelling periodic point, we will escape the neighborhood.

In the simple case of $k = 1$, a periodic orbit of length one is just a **fixed point** z of p so that $p(z) = z$. In this case z is an attracting fixed point if and only if $|p'(z)| < 1$. If z is an attracting fixed point, then the orbit of any point x in some small neighborhood of z will be approaching z exponentially fast.

We say that a point c is a **critical** point of $p(z)$ if $p'(c) = 0$. We are now ready to state one of the equivalent definitions of a hyperbolic polynomial.

Definition 4.2.1 *A polynomial $p(z)$ of degree ≥ 2 is said to be* hyperbolic *if every critical point c of $p(z)$ converges to an attracting periodic orbit of $p(z)$ or to ∞, or is part of a periodic orbit itself. In the latter case we say that the orbit of c is* superattracting.

Here we include ∞ as a special case to simplify matters, but in fact, by considering the Riemann sphere instead of the complex plane, we can regard ∞ as a superattracting periodic point of $p(z)$, since we have $p(\infty) = \infty$ and $p'(\infty) = 0$.

We can now give a simple definition of the Julia set in the hyperbolic case. See [Mil00] for a proof that in the hyperbolic case this definition is equivalent to the general definition of the Julia set.

Definition 4.2.2 *The Julia set J_p of a hyperbolic polynomial $p(z)$ is the set of all points w, such that the orbit of w does not converge to an attracting periodic orbit of $p(z)$ or to ∞. The complement of the Julia set is denoted $K_p = J_p^c$ and is called the Fatou set of the polynomial $p(z)$. These Julia sets are called* hyperbolic Julia sets.

We summarize the most important facts about hyperbolic Julia sets we will be using in the following lemma. See [Mil00] for details and proofs.

Lemma 4.2.3 *For a hyperbolic polynomial $p(z)$ the following facts hold:*

1. *The interior of J_p is empty.*

2. *$J_p = p(J_p) = p^{-1}(J_p)$.*

3. *$p(z)$ has at most $deg(p) - 1$ attracting periodic orbits (regarding an orbit as a set).*

The definition itself gives a very naive "algorithm" for computing J_p. Namely, set a threshold T. To determine whether a point w is in J_p compute the first T elements of the orbit of w, $p(w), p^2(w), \ldots, p^T(w)$. If the orbit gets *close* to one of the attracting orbits, say that $w \notin J_p$, otherwise say that $w \in J_p$. In fact, many of the computer programs that draw Julia sets use this method. The problem, of course, is how to choose a good T and how to define "close". If T is not chosen properly, we might reject w's which are very close to J_p or accept w's which are far away from J_p. We will have to develop more theory in order to choose T which makes the method above work properly. The tool which we will use to control the distance between w and J_p is one of the fundamental tools in complex dynamics, called the Poincaré metric.

4.3 The Poincaré Metric

The Poincaré metric, known also as the hyperbolic metric, is a metric which naturally arises on hyperbolic Riemann surfaces. It is beyond the scope of this work to discuss the metric in full generality, so we will restrict our attention to subsets of the complex plane $\mathbb{C}$. See [Mil00] for a more comprehensive exposition. It is known that any connected open subset $S \subset \mathbb{C}$ of the complex plane which omits at least 2 points is a hyperbolic Riemann surface and has a unique (up to a multiplication by a constant) Poincaré metric d_S. We call these subsets of $\mathbb{C}$ **hyperbolic sets**.

To define the Poincaré metric, we first need to describe another fundamental mathematical concept – the notion of a covering map. While covering maps are defined in many different topological settings, we will define it for our case: the hyperbolic subsets of $\mathbb{C}$. A map $f : X \to Y$ between two hyperbolic subsets of $\mathbb{C}$ is said to be a **covering map**, if for each $y \in Y$ there is a neighborhood $N(y)$ of y such that for each connected component N' of $f^{-1}(N(y))$, the map $f : N' \to N(y)$ is a conformal (locally shape preserving) isomorphism.

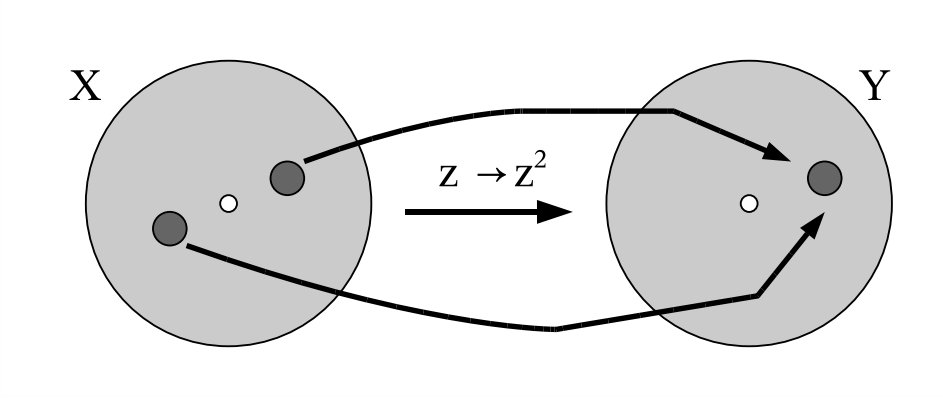

Figure 4.2: The map $z \mapsto z^2$ is a two-fold covering map on the punctured unit disk.

In general, covering maps allow us to analyze the structure of Y using the structure of X. The following theorem illustrates the relevance of covering maps in our case.

Theorem 4.3.1 *Let $D = \{z : |z| < 1\}$ be the open unit disk and let $S \subset \mathbb{C}$ be a hyperbolic set. Then there is a covering map $q : D \to S$.*

We first define the Poincaré metric on the unit disk, and then use the covering map from theorem 4.3.1 to give the Poincaré metric on a general hyperbolic set S. The property defining the Poincaré metric on the unit disk is its invariance under conformal automorphisms of the disk D as seen in the following theorem.

Theorem 4.3.2 *There exists one and, up to a multiplication by a positive constant, only one Riemannian metric on the disk D which is invariant under every conformal automorphism of D.*

Explicitly, we can choose the metric to be given by $p_D(z) = 2/(1 - |z|^2)$ so that the length of an arc γ in the Poincaré metric is given by

$$l_D(\gamma) = \int_\gamma \frac{2|d\gamma(z)|}{1 - |z|^2}.$$

Note that the metric tends to ∞ as z approaches the boundary of the disk.

We can now define the Poincaré metric on an arbitrary hyperbolic subset S of $\mathbb{C}$ as the metric induced from the metric on D by the covering map. If we denote a covering map by $q : D \to S$, then we define the metric $p_S(z) = p_D(w)/|q'(w)|$, where $w \in q^{-1}(z)$. It follows from the definition of p_D that p_S does not depend on the choice of q and w. The map q preserves lengths of paths: for an arbitrary path γ in D,

$$l_S(q(\gamma)) = \int_\gamma p_S(q(z))|q'(z)||d\gamma(z)| = \int_\gamma (p_D(z)/|q'(z)|) \cdot |q'(z)||d\gamma(z)| =$$

$$\int_\gamma p_D(z)|d\gamma(z)| = l_D(\gamma).$$

In fact, the Poincaré length is preserved not only by q, but by any covering map. This fact is stated in the following theorem, known as Pick's theorem, which will be applied several times in our arguments below. See [Mil00] for a proof.

Theorem 4.3.3 (Theorem of Pick) *Let S and T be two hyperbolic subsets of $\mathbb{C}$. If $f : S \to T$ is a holomorphic map, then exactly one of the following three statements is valid:*

1. *f is a conformal isomorphism from S onto T, and maps S with its Poincaré metric isometrically to T with its Poincaré metric.*

2. *f is a covering map but is not one-to-one. In this case, it is locally but not globally a Poincaré isometry. Every smooth path $P : [0, 1] \to S$ of arclength l in S maps to a smooth path $f \circ P$ of the same length l in T.*

3. *In all other cases, f is a strict contraction with respect to the Poincaré metrics on the image and preimage.*

We now have the basic complex-analytic background required for the construction, and we are ready to prove that the hyperbolic Julia sets are poly-time computable. We will first show that hyperbolic Julia sets are Ko P-computable. By theorem 3.2.2 this implies that these sets are computable in exponential time. We then use a technique similar to the technique used in [RW03] to obtain a poly-time algorithm from the exponential one.

4.4 Hyperbolic Julia Sets are Ko P-Computable

As noted above, we will present an algorithm that uses some nonuniform information, i.e. information which depends on the polynomial $p(z)$ but not on the precision parameter n. The polynomial $p(z)$ itself is given to the algorithm as an oracle that outputs its coefficients with any required precision. Denote the polynomial $p(z) = c_m z^m + c_{m-1} z^{m-1} + \ldots + c_1 z + c_0$. We can query each c_i with precision 2^{-r} with time cost r. As per the definition of Ko P-computability, the input x to the algorithm is also given as an oracle. We want to decide whether $x \in J_p$.

We will now list the nonuniform information used by the algorithm. This information can be computed from the initial data (i.e. the coefficients of $p(z)$) as will be noted later, see theorem 4.6.1. We still list it as nonuniform to spare overcomplicated technical details from the reader.

4.4.1 Nonuniform constants information

The key nonuniform information we need is information about the attracting orbits of $p(z)$. It is known from lemma 4.2.3 that $p(z)$ has at most $m-1$ attracting orbits. We assume that we are given the number o of the orbits, the periods $l_1, l_2, \ldots, l_o$ of the orbits, and a "good" approximation of the orbits.

Denote the orbits by

$$X = ((x_{11}, x_{12}, \ldots, x_{1l_1}), (x_{21}, x_{22}, \ldots, x_{2l_2}), \ldots, (x_{o1}, x_{o2}, \ldots, x_{ol_o})).$$

We want a dyadic approximation

$$A = ((a_{11}, a_{12}, \ldots, a_{1l_1}), (a_{21}, a_{22}, \ldots, a_{2l_2}), \ldots, (a_{o1}, a_{o2}, \ldots, a_{ol_o}))$$

of X with $|a_{ij} - x_{ij}| < \varepsilon_o$, where ε_o is some fixed positive number which does not depend on the required precision, and will be specified later.

For each attracting orbit o_i we would like to have a measure on the convergence to o_i near its points. We formulate it as follows. For each x_{ij} we associate a radius $r_{ij} > 0$ and to each orbit we associate a number $s_i < 1$ such that

1. If $|x - x_{ij}| < r_{ij}$ then x converges to the i-th orbit.

2. For each i, j, $p(B(x_{ij}, r_{ij})) \subset B(x_{i,j+1}, s_i \cdot r_{i,j+1})$, where $j+1$ is taken *mod* l_i.

Intuitively, s_i gives a numerical measure on how fast the i-th orbit attracts points. Let $r_a = \min r_{ij}$ be a unified radius of convergence, so that if $|x - x_{ij}| < r_a$, then x converges to the i-th orbit.

We would also like to have a radius of "attraction" to ∞, namely R and R' such that:

1. If $x > R$ then the orbit of x diverges to ∞,

2. $R' > R$, and

3. $|p(x)| > R'$ whenever $|x| > R$.

• Denote $\tilde{d}_l = \min(R' - R, \min_{i,j}(r_{ij} - s_i \cdot r_{ij}))$ and choose ε_o above to be $\tilde{d}_l/4$.

As one might guess from the definition of the hyperbolic Julia sets, critical points play a key role in the dynamics of the Julia set. We will need some information about the critical points. Since the critical points are just the solutions of $p'(z) = 0$, there are $m-1$ critical points (counted with multiplicity). Denote the critical points $Y = (y_1, \ldots, y_{m-1})$. We do not need to know the critical points themselves, but rather some information about their convergence to the attracting orbits. From the definition of $p(z)$ being hyperbolic, we know that each y_i converges to one of the attracting orbits, and so for each y_i there is a neighborhood of y_i which converges to the same orbit. We would like to have a unified radius r_c and a number q such that if $|y - y_i| < r_c$ for some i, then there are j and k such that

$$|p^q(y) - x_{jk}| < r_a \quad \text{or} \quad |p^q(y)| > R'.$$

There are two more useful bounds that we will be using. They are very easy to compute from $p(z)$. We will need two bounds on the derivative $|p'(z)|$:

• A *lower* bound $d > 0$ such that $|p'(z)| > d$ whenever $|z - y_i| > \frac{r_c}{2}$ for all i.

• An *upper* bound $D \geq 1$ such that $|p'(p(y))| < D$ and $|p'(y)| < D/2$ whenever $|y| < R'$.

4.4.2 The Main Construction

We are now ready to present the main construction. Define

$$\tilde{U} = B(0, R') \backslash \left(\bigcup_{i=1}^{o} \bigcup_{j=1}^{l_i} \overline{B(x_{ij}, s_i \cdot r_{ij})} \right).$$

$\tilde{U}$ can be thought of as a large disk with holes punctured near the attracting orbits. Any point outside $\tilde{U}$ converges either to an attracting orbit or to ∞, so $J_p \subset \tilde{U}$. Denote $\tilde{V} = p^{-1}(\tilde{U})$. It is not hard to see that $\tilde{V}$ is contained in $\tilde{U}$, and in fact is bounded away from the boundary of $\tilde{U}$:

Lemma 4.4.1

$$\tilde{V} \subseteq B(0,R) \backslash \left(\bigcup_{i=1}^{o} \bigcup_{j=1}^{l_i} \overline{B(x_{ij}, r_{ij})} \right) \subset \tilde{U}, \quad \textit{and} \quad d(\tilde{U}^c, \tilde{V}) \geq \tilde{d}_l.$$

Below is a graphical illustration of $\tilde{U}$ and $\tilde{V}$.

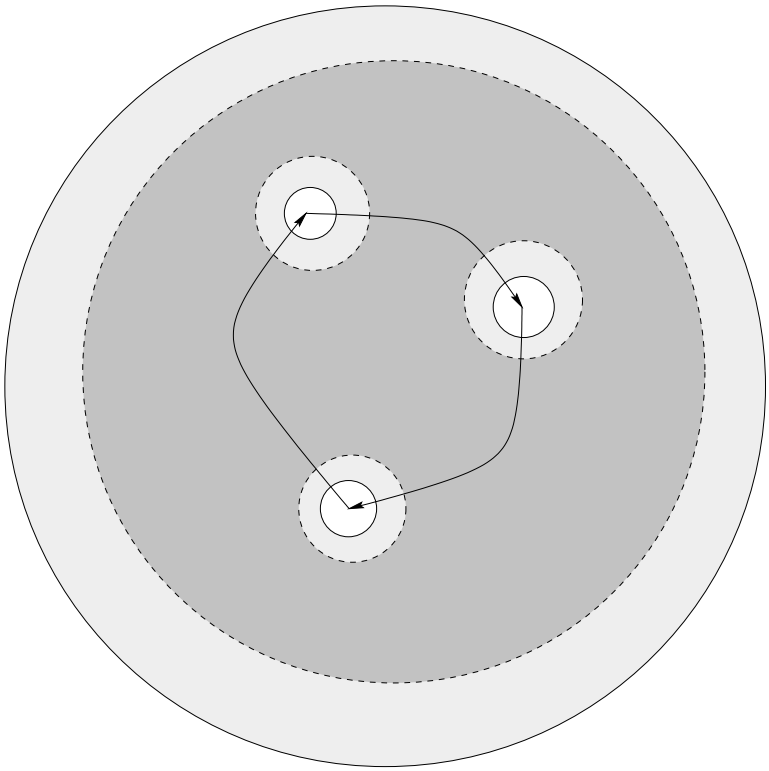

Figure 4.3: An illustration of $\tilde{U}$ and $\tilde{V}$. $\tilde{U}$ shown in light gray, $\tilde{V} \subset \tilde{U}$ is shown in dark gray.

Denote $U = p^{-q-1}(\tilde{U})$, $V = p^{-q-1}(\tilde{V})$. Then by the definition of r_c and q, $B(y_i, r_c) \cap U = \emptyset$ for any critical point y_i. Lemma 4.2.3 implies that $J_p = p^{-q-2}(J_p) \subset p^{-q-2}(\tilde{U}) = V$.

As a corollary of lemma 4.4.1, we conclude that $V \subset U$, and furthermore we can compute a lower bound d_l on the distance between V and U^c.

Lemma 4.4.2

$$d(U^c, V) \geq d_l = \frac{\tilde{d}_l}{D^{q+1}}.$$

Proof: The lemma follows immediately from lemma 4.4.1 and from the fact that D is an upper bound on the derivative $|p'(z)|$ on $\tilde{U}$. ■

We will be repeatedly using the following simple lemma.

Lemma 4.4.3 *Suppose $S \subset T$ are two open subsets of $\mathbb{C}$, which are hyperbolic viewed as Riemann surfaces. Let p_S and p_T denote the Poincaré metrics of S and T, respectively. Then*

$$p_T(z) < p_S(z)$$

for each $z \in S$.

Proof: We apply the theorem of Pick (theorem 4.3.3). Let

$$\iota : S \hookrightarrow T$$

be the inclusion map. Then, by Pick's theorem, ι decreases all non-zero distances. Hence $p_T(z) < p_S(z)$ for all $z \in S$. ■

U and V are obviously hyperbolic sets, and hence the Poincaré metric is defined on them. Denote the weight function of the Poincaré metric on U by p_U and the weight function of the Poincaré metric on V by p_V. Denote the Poincaré metrics themselves by d_U and d_V, respectively. We have the following simple lemma bounding the metric p_U. This lemma is a simple corollary of lemma 4.4.3

Lemma 4.4.4 *$p_U(z) > \frac{2}{R'}$ for all $z \in U$, and $p_U(z) < \frac{2}{d_l}$ for all $z \in V$.*

Proof: We know that $U \subset B(0, R')$, hence $p_U > p_{B(0,R')}$ on V. But $p_{B(0,R')}$ is given by

$$p_{B(0,R')}(z) = \frac{2}{R' - \frac{|z|^2}{R'}} \geq \frac{2}{R'}.$$

Hence $p_U(z) > p_{B(0,R')}(z) \geq \frac{2}{R'}$.

Let z be an arbitrary point in V. Then $B(z, d_l) \subset U$ by the definition of d_l in claim 4.4.2 and so $p_U < p_{B(z,d_l)}$. But $p_{B(z,d_l)}$ is given by

$$p_{B(z,d_l)}(x) = \frac{2}{d_l - \frac{|x-z|^2}{d_l}}.$$

Hence $p_U(z) < p_{B(z,d_l)}(z) = \frac{2}{d_l}$, for all $z \in V$. ■

We will need the following lemma in our construction.

Lemma 4.4.5 *$p_V(z) \geq c \cdot p_U(z) > p_U(z)$ for all $z \in V$, where $c > 1$ is some constant explicitly computable from d_l and R'.*

It is not hard to prove lemma 4.4.5 by applying a compactness argument. We insist on giving a constructive proof and computing the constant c. We will need the ability to compute c when making the construction uniform (see section 4.6).

Proving lemma 4.4.5 is quite technically involved, and will require some preparation. We start with an auxiliary construction. Let G be a two dimensional grid in $\mathbb{C}$ with step length $d_l/4$ in both directions (i.e. $G = \frac{d_l}{4} \cdot \mathbb{Z} + \frac{d_l}{4} \cdot \mathbb{Z}i$). We define a set W such that $V \subset W \subset U$ as follows.

$$W = \bigcup_{x \in G, B(x,d_l/4) \cap V \neq \emptyset} B(x, d_l/4).$$

In other words, we throw the ball $B(x, d_l/4)$ into W if and only if its intersection with V is not empty. The definition of W has something in common with the definition of a $d_l/4$-picture of V.

It is easy to see that $V \subset W \subset B(V, d_l/2)$. It follows that

Lemma 4.4.6 *$B(W, d_l/2) \subset U$.*

Proof: $B(W, d_l/2) \subset B(V, d_l/2+d_l/2) = B(V, d_l) \subset U$, where the last inclusion follows from lemma 4.4.2. ■

One can prove the following lemma exactly as we have proved lemma 4.4.4, using W instead of V and $d_l/2$ instead of d_l.

Lemma 4.4.7 *$p_U(z) < \frac{2}{d_l/2} = \frac{4}{d_l}$ for all $z \in W$.*

We now prove a bound on the Euclidean radius of W.

Lemma 4.4.8 *For every x, y in the same connected component of W there is a path γ connecting x to y of Euclidean length $< \frac{32R'^2}{d_l}$.*

Proof: The set V is contained in $B(0, R')$, hence there are less than $\frac{2R'}{d_l/4} \times \frac{2R'}{d_l/4} = \frac{64R'^2}{d_l^2}$ balls of radius $d_l/4$ in the definition of W. We can choose a path γ from x to y such that γ visits each ball in the definition of W at most once. Moreover, we can make γ a broken line such that at most a $diam(B(z, d_l/4)) = d_l/2$ portion of γ is charged to each ball $B(z, d_l/4)$ in the definition of W. Hence the length of such a path γ would be $< \frac{64R'}{d_l^2} \cdot \frac{d_l}{2} = \frac{32R'^2}{d_l}$. ■

Lemma 4.4.8 allows us to give a bound on the Poincaré d_U-diameter of V.

Lemma 4.4.9 *For x, y in the same connected component of V, $d_U(x, y) < M_W = \frac{128R'^2}{d_l^2}$.*

Proof: Let x, y be in the same connected component of V. $V \subset W$ implies that x, y are also in the same connected component of W. Hence by lemma 4.4.8 there is a path γ of Euclidean length $< \frac{32R'^2}{d_l}$. We can use lemma 4.4.7 to bound the Poincaré length $l_U(\gamma)$ in the Poincaré metric of U:

$$d_U(x, y) \leq l_U(\gamma) = \int_\gamma p_U(z)|d\gamma(z)| \leq \int_\gamma \frac{4}{d_l}|d\gamma(z)| \leq length(\gamma) \cdot \frac{4}{d_l} < \frac{32R'^2}{d_l} \cdot \frac{4}{d_l} = \frac{128R'^2}{d_l^2}.$$

■

We are now ready to prove lemma 4.4.5.

Proof: (of lemma 4.4.5). First of all, as we have previously mentioned, such a c must exist by a compactness arguments. We know that $p_V(z) > p_U(z)$ for all $z \in V$ and $p_V(z)/p_U(z) \to \infty$ as z tends to the boundary of V. Hence $c = \min_V p_V(z)/p_U(z)$ must exist and we must have $c > 1$. Computing c is not essential for proving that J_p is computable for each *fixed* polynomial $p(z)$, since we might as well have given c as part of the nonuniform input. It is essential, however, to compute a specific value of c in terms of more basic (and computable in the hyperbolic case) information about $p(z)$ in order to get a *uniform* poly-time computability result for hyperbolic Julia sets (theorem 4.6.1). We will give a lower bound > 1 on c in terms of d_l and R'. Our goal is to set a computable lower bound on c for theoretical purposes. We believe that the bound can be improved using more involved analysis.

We have $V \subset W \subset U$ and hence $p_V(z) > p_W(z)$ for all $z \in V$. Hence it is enough to find a constant $c > 1$ such that $p_W(z) > c \cdot p_U(z)$ for all $z \in W$. Fix a specific point $w \in W$. Let $W_0 \subset U_0$ be the connected components of W and U respectively which contain w. Then by definition $p_W(w) = p_{W_0}(w)$ and $p_U(w) = p_{U_0}(w)$. From lemma 4.4.6 we conclude that $B(W_0, d_l/2) \subset U_0$.

As it has been already mentioned in theorem 4.3.1, there are covering maps $q_W : D_W \to W_0$ and $q_U : D_U \to U_0$ where $D_W = D_U = \{z : |z| < 1\}$ are just two open unit disks. Let $\tilde{w}$ be a preimage of w under q_W, that is $q_W(\tilde{w}) = w$.

Consider the inclusion map $\iota : W_0 \hookrightarrow U_0$. From the map lifting property of the covering map q_U, there is a map $\tilde{\iota} : D_W \to D_U$ such that the diagram

$$\begin{array}{ccc} D_W & \xrightarrow{\tilde{\iota}} & D_U \\ q_W \downarrow & & \downarrow q_U \\ W_0 & \xhookrightarrow{\iota} & U_0 \end{array} \tag{4.1}$$

commutes. (see [Jo02] pp. 7-15 for more information about map lifting).

Let $x \in U_0 \backslash W_0$ be a point such that $d_U(x, w) < 2M_W$. Such a point must exist since there is a point y on the boundary ∂W_0 with $d_U(w, y) \leq M_W$ by a trivial modification

of lemma 4.4.9 to use W instead of V (the same proof is still valid), and a point x in U_0 outside W_0 with $d_U(x,y) < M_W$. Hence $d_U(x,w) \leq d_U(x,y) + d_U(y,w) < M_W + M_W = 2M_W$. Let γ be a path in U_0 from w to x such that the Poincaré length $l_U(\gamma) < 2M_W$. By the commutativity of diagram (4.1) we know that $q_U(\tilde{\iota}(\tilde{w})) = \iota(w) = w$, hence by the path lifting property there is a path $\tilde{\gamma}$ in D_U which is a lift of γ such that $\tilde{\gamma}(0) = \tilde{\iota}(\tilde{w})$. Denote $\tilde{x} = \tilde{\gamma}(1) \in D_U$. Then we have $q_U(\tilde{x}) = x$, i.e. $\tilde{x}$ is a lift of x. Observe that q_U is a covering map, and hence by Pick's theorem (theorem 4.3.3) it preserves Poincaré lengths of paths. Hence $d_{D_U}(\tilde{x}, \tilde{\iota}(\tilde{w})) \leq l_{D_U}(\tilde{\gamma}) = l_U(\gamma) < 2M_W$.

It is known that for every point z in the unit disk, there is an automorphism of the unit disk which takes z to the center of the disk. (See [Mil00], for example). Denote by $\phi : D_U \to D_U$ an automorphism that takes $\tilde{x}$ to 0. So $\phi(\tilde{x}) = 0$ and $\phi^{-1}(0) = \tilde{x}$. Consider the following commutative diagram

$$\begin{array}{ccccc} D_W & \xrightarrow{\phi\circ\tilde{\iota}} & D'_U & \xrightarrow{\phi^{-1}} & D_U \\ q_W \downarrow & & & & \downarrow q_U \\ W_0 & & \xhookrightarrow{\iota} & & U_0 \end{array} \tag{4.2}$$

Where D'_U is just a copy of the unit disk D_U. $\phi^{-1} : D'_U \to D_U$ is an automorphism, so by Pick's theorem it is a Poincaré isometry. Hence

$$d_{D'_U}(0, \phi \circ \tilde{\iota}(\tilde{w})) = d_{D_U}(\phi^{-1}(0), \phi^{-1} \circ \phi \circ \tilde{\iota}(\tilde{w})) = d_{D_U}(\tilde{x}, \tilde{\iota}(\tilde{w})) < 2M_W.$$

We know that the Poincaré metric on the unit disk is given by $p_{D'_U}(z) = 2/(1-|z|^2)$, hence

$$\int_0^{|\phi\circ\tilde{\iota}(\tilde{w})|} \frac{2|dz|}{(1-|z|^2)} < 2M_W.$$

So we obtain

$$2M_W > \int_0^{|\phi\circ\tilde{\iota}(\tilde{w})|} \frac{2dt}{1-t^2} = \int_0^{|\phi\circ\tilde{\iota}(\tilde{w})|} \left(\frac{dt}{1+t} + \frac{dt}{1-t}\right) > \int_0^{|\phi\circ\tilde{\iota}(\tilde{w})|} \frac{dt}{1-t}$$
$$= -\log(1-t)\Big|_0^{|\phi\circ\tilde{\iota}(\tilde{w})|} = -\log(1-|\phi\circ\tilde{\iota}(\tilde{w})|).$$

Hence $|\phi \circ \tilde{\iota}(\tilde{w})| < 1 - e^{-2M_W}$.

We will now show that $0 \notin \phi \circ \tilde{\iota}(D_W)$. Suppose, on the contrary, that there is a $\tilde{z} \in D_W$ with $\phi \circ \tilde{\iota}(\tilde{z}) = 0$. Denote $z = q_W(\tilde{z}) \in W_0$. Then $\iota(z) = q_U(\tilde{\iota}(\tilde{z})) = q_U(\phi^{-1} \circ \phi \circ \tilde{\iota}(\tilde{z})) = q_U(\phi^{-1}(0)) = q_U(\tilde{x}) = x$, which contradicts our assumption that $x \notin W_0$. This shows that the map $\phi \circ \tilde{\iota} : D_W \to D'_U$ can be factored through the *punctured unit disk* $PD = \{z : 0 < |z| < 1\}$. In other words, the diagram

$$\begin{array}{ccccccc} D_W & \xrightarrow{\phi\circ\tilde{\iota}} & PD & \xhookrightarrow{\iota'} & D'_U & \xrightarrow{\phi^{-1}} & D_U \\ q_W \downarrow & & & & & & \downarrow q_U \\ W_0 & & & \xhookrightarrow{\iota} & & & U_0 \end{array} \tag{4.3}$$

where ι' is an inclusion map of the punctured unit disk into the simple unit disk, commutes.

We can now analyze the ratio $p_W(w)/p_U(w)$, which is the goal of the proof. Pick's theorem says that covering maps locally preserve the Poincaré metric. Hence we have $p_W(w) = p_{D_W}(\tilde{w})/|q'_W(\tilde{w})|$ and $p_U(w) = p_{D_U}(\tilde{\iota}(\tilde{w}))/|q'_U(\tilde{\iota}(\tilde{w}))|$. The functions $\phi \circ \tilde{\iota} : D_W \to PD$ and $\phi^{-1} : D'_U \to D_U$ do not increase the Poincaré metric, hence $p_{PD}(\phi \circ \tilde{\iota}(\tilde{w})) \cdot |(\phi \circ \tilde{\iota})'(\tilde{w})| \le p_{D_W}(\tilde{w})$ and $p_{D_U}(\tilde{\iota}(\tilde{w})) \cdot |(\phi^{-1})'(\phi \circ \tilde{\iota}(\tilde{w}))| \le p_{D'_U}(\phi \circ \tilde{\iota}(\tilde{w}))$. Finally, by the commutativity of diagram (4.3), we have $|q'_W(\tilde{w})| = |(\phi \circ \tilde{\iota})'(\tilde{w})| \cdot |(\phi^{-1})'(\phi \circ \tilde{\iota}(\tilde{w}))| \cdot |q'_U(\tilde{\iota}(\tilde{w}))|$. Using all these facts together we obtain

$$\frac{p_W(w)}{p_U(w)} = \frac{p_{D_W}(\tilde{w})/|q'_W(\tilde{w})|}{p_{D_U}(\tilde{\iota}(\tilde{w}))/|q'_U(\tilde{\iota}(\tilde{w}))|} = \frac{p_{D_W}(\tilde{w}) \cdot |q'_U(\tilde{\iota}(\tilde{w}))|}{p_{D_U}(\tilde{\iota}(\tilde{w})) \cdot |q'_W(\tilde{w})|} \ge$$

$$\frac{p_{PD}(\phi \circ \tilde{\iota}(\tilde{w})) \cdot |(\phi \circ \tilde{\iota})'(\tilde{w})| \cdot |q'_U(\tilde{\iota}(\tilde{w}))|}{(p_{D'_U}(\phi \circ \tilde{\iota}(\tilde{w}))/|(\phi^{-1})'(\phi \circ \tilde{\iota}(\tilde{w}))|) \cdot |q'_W(\tilde{w})|} =$$

$$\frac{p_{PD}(\phi \circ \tilde{\iota}(\tilde{w})) \cdot |(\phi \circ \tilde{\iota})'(\tilde{w})| \cdot |(\phi^{-1})'(\phi \circ \tilde{\iota}(\tilde{w}))| \cdot |q'_U(\tilde{\iota}(\tilde{w}))|}{p_{D'_U}(\phi \circ \tilde{\iota}(\tilde{w})) \cdot |(\phi \circ \tilde{\iota})'(\tilde{w})| \cdot |(\phi^{-1})'(\phi \circ \tilde{\iota}(\tilde{w}))| \cdot |q'_U(\tilde{\iota}(\tilde{w}))|} = \frac{p_{PD}(\phi \circ \tilde{\iota}(\tilde{w}))}{p_{D'_U}(\phi \circ \tilde{\iota}(\tilde{w}))}.$$

There are explicit formulas for $p_{D'_U}$ and p_{PD}. $D_{U'}$ is just the unit disk, and we are familiar with its Poincaré metric. PD is the punctured unit disk, its Poincaré metric is given by (see [Mil00], p. 19) $p_{PD}(z) = 1/(|z| \cdot |\log|z||)$. Denote $r = |\phi \circ \tilde{\iota}(\tilde{w})|$. We know that $r < 1 - e^{-2M_W}$. Then

$$\frac{p_W(w)}{p_U(w)} \ge \frac{p_{PD}(\phi \circ \tilde{\iota}(\tilde{w}))}{p_{D'_U}(\phi \circ \tilde{\iota}(\tilde{w}))} = \frac{1/(r|\log r|)}{2/(1-r^2)} = \frac{1-r^2}{2r|\log r|} = -\frac{1-r^2}{2r\log r}.$$

Denote $f : [0,1] \to R$ by $f(r) = -\frac{1-r^2}{2r\log r}$. Then f is differentiable on the interval $(0,1)$ and

$$f'(r) = -\frac{-2r(2r\log r) - (2+2\log r)(1-r^2)}{(2r\log r)^2} = -\frac{2r^2 - 2\log r - 2 - 2r^2\log r}{(2r\log r)^2}.$$

By the simple fact that follows from the Taylor expansion of $\log r$ around 1: $\log r < (r-1) - (r-1)^2/2$ for all $0 < r < 1$,

$$2r^2 - 2\log r - 2 - 2r^2\log r = 2r^2 - 2 - (2+2r^2)\log r > 2r^2 - 2 - (2+2r^2)((r-1) - (r-1)^2/2) =$$

$$2r^2 - 2 - 2r + 2 - 2r^3 + 2r^2 + (1+r^2)(r-1)^2 = -2r^3 + 4r^2 - 2r + 1 - 2r + 2r^2 - 2r^3 + r^4 =$$

$$r^4 - 4r^3 + 6r^2 - 4r + 1 = (r-1)^4 > 0.$$

Hence $f'(r) < 0$ for all $r \in (0,1)$, and the function f is decreasing on this interval. Thus $r < 1 - e^{-2M_W}$ implies that $f(r) > f(1 - e^{-2M_W})$. So

$$\frac{p_W(w)}{p_U(w)} \geq \frac{p_{PD}(\phi \circ \tilde{\imath}(\tilde{w}))}{p_{D'_U}(\phi \circ \tilde{\imath}(\tilde{w}))} = f(r) > f(1 - e^{-2M_W}) =$$

$$-\frac{1 - (1 - e^{-2M_W})^2}{2(1 - e^{-2M_W})\log(1 - e^{-2M_W})}. \tag{4.4}$$

Choose

$$c = f(1 - e^{-2M_W}) = -\frac{1 - (1 - e^{-2M_W})^2}{2(1 - e^{-2M_W})\log(1 - e^{-2M_W})} = 1.$$

Since c does not depend on w, (4.4) is satisfied for all $w \in W_0$. Hence if $v \in W_0 \cap V$, then $\frac{p_V(v)}{p_U(v)} > \frac{p_W(v)}{p_U(v)} > c$. All we have to see now is that $c > 1$. We apply L'Hospital's rule to obtain

$$\lim_{r \to 1^-} f(r) = \lim_{r \to 1^-} -\frac{1 - r^2}{2r\log r} = \lim_{r \to 1^-} -\frac{-2r}{2\log r + 2} = 1.$$

f is decreasing on $(0,1)$ and approaches 1 as $r \to 1$, hence $f(r) > 1$ for all $r \in (0,1)$ in particular $c = f(1 - e^{-2M_W}) > 1$. In fact, one can see that $c = 1 + \Theta((e^{-2M_W})^2)$. This completes the proof. ■

Observe that by our construction V does not contain critical points of p, hence the map $p : V \to U$ is an m-fold covering map. Thus by Pick's theorem (theorem 4.3.3) it is a local Poincaré isometry. Formally, we obtain

Lemma 4.4.10 *$p_V(x) = |p'(z)| \cdot p_U(p(x))$ for all $x \in V$.*

We are now ready to prove the following lemma which is the key to our construction.

Lemma 4.4.11 *Let $d_U(J_p, z)$ denote the distance between the point z and the Julia set J_p in the Poincaré metric d_U. Then $d_U(J_p, z)$ is well defined for all $z \in U$ and $d_U(J_p, p(x)) \geq c \cdot d_U(J_p, x)$ for all $x \in V$, where $c > 1$ is a binary constant computable from the initial data.*

Proof: First of all, we need to show that $d_U(J_p, z)$ is finite. In other words, we need to show that for each connected component U_0 of U, $J_p \cap U_0 \neq \emptyset$. We prove by induction on k that for each connected component Y_0 of $Y^k = p^{-k}(\tilde{U})$, $Y_0 \cap J_p \neq 0$. In particular, applying the result with $k = q + 1$ will imply that $J_p \cap U_0 \neq \emptyset$, since U_0 is a connected component of $U = p^{-(q+1)}(\tilde{U})$.

For the base of the induction it is obvious that $Y_0 = \tilde{U}$ is connected and $J_p \cap Y_0 \neq \emptyset$. For the step fix $k \geq 0$. Assume that the claim holds for k, we want to prove it for $k + 1$. Consider the map $p : Y^{k+1} \to Y^k$. p is a polynomial, so it is an open map. In other words, p maps open sets to open sets. It is not hard to see from here that by topological observations p must map connected components of Y^{k+1} *onto* connected components of Y^k. Let Y_0 be a connected component of Y^{k+1}. Then $p(Y^{k+1})$ is a connected component of Y^k, and so by the induction assumption there is a $j' \in J_p \cap p(Y^{k+1})$. Hence there is a $j \in Y^{k+1}$ with $p(j) = j' \in J_p$ hence by lemma 4.2.3, $j \in J_p$, which completes the proof that $d_U(J_p, z)$ is finite, and so it is well defined.

Fix an arbitrary $x \in V$. Let $c > 1$ be the computable constant we have found in the proof of lemma 4.4.5. Denote $d_U(J_p, p(x)) = l \geq 0$. We will show that $d_U(J_p, x) \leq (l+\varepsilon)/c$ for all $\varepsilon > 0$, and so $d_U(J_p, x) \leq l/c = d_U(J_p, p(x))/c$.

By the definition of the distance $d_U(J_p, p(x))$, for any $\varepsilon > 0$ there is a path γ in U such that $\gamma(0) = p(x)$, $\gamma(1) \in J_p$ and the length of γ in the Poincaré metric of U, $l_U(\gamma) \leq l + \varepsilon$. $p : V \to U$ is a covering map, so by the path lifting property, we can lift γ to a path

$\tilde{\gamma}$ in V, such that $\gamma = p(\tilde{\gamma})$ and $\tilde{\gamma}(0) = x$. In particular, we have $p(\tilde{\gamma}(1)) = \gamma(1) \in J_p$, hence by lemma 4.2.3, $\tilde{\gamma}(1) \in J_p$, and so $d_U(J_p, x) \leq l_U(\tilde{\gamma})$. By lemma 4.4.10 we know that $p : V \to U$ is a local Poincaré isometry. In particular, it preserves path lengths. So $l_V(\tilde{\gamma}) = l_U(\gamma)$. We apply lemma 4.4.5 to obtain

$$d_U(J_p, x) \leq l_U(\tilde{\gamma}) = \int_0^1 p_U(\tilde{\gamma}(t))|\tilde{\gamma}'(t)|dt \leq \frac{1}{c} \cdot \int_0^1 p_V(\tilde{\gamma}(t))|\tilde{\gamma}'(t)|dt =$$

$$\frac{l_V(\tilde{\gamma})}{c} = \frac{l_U(\gamma)}{c} \leq \frac{l+\varepsilon}{c}.$$

Hence $d_U(J_p, x) \leq l/c = d_U(J_p, p(x))/c$, which completes the proof. ■

4.4.3 The Algorithm

Lemma 4.4.11 gives us a tool to estimate the speed at which a point $x \notin J_p$ runs away from J_p in the Poincaré metric. If initially the Euclidean distance $d(J_p, x) > \varepsilon$, then by lemma 4.4.4, $d_U(J_p, x) > \frac{2\varepsilon}{R'}$, and assuming that the orbit of x stays in V in s steps we have by lemma 4.4.11 that $d_U(J_p, p^s(x)) > \frac{2\varepsilon}{R'} \cdot c^s$. But then by lemma 4.4.9 we have $s \leq \log_c \frac{M_W}{2\varepsilon/R'}$. If $\varepsilon = 2^{-n}$ the estimate on s is linear in n. This allows us to obtain a poly time algorithm which Ko P-computes J_p:

Algorithm 1

Input: The non-uniform input as described above,

the input $x, c_0, c_1, \ldots, c_m$ given on an oracle tape and m, n.

Output: 0 if $d(J_p, x) > 2^{-n}$, 1 if $x \in J_p$, 0 or 1 otherwise, as required in (2.6).

1. Compute c described above.
2. Compute a natural number $N = O(n)$ such that

   $N \geq 2 + \log_c (M_W \cdot R' \cdot 2^{n-1}) + q$.

3. Compute $p^N(x)$ within an error of $\frac{\tilde{d}_l}{4}$.

   3.1. If $p^N(x) \notin \tilde{U}$, output 0,

   3.2. If $p^N(x) \in \tilde{V}$, output 1,

   3.3. Otherwise, output 0 or 1.

Note that we know the attracting orbits with precision good enough to unambiguously decide if one of the possibilities 3.1 or 3.2 holds. By the discussion above $d(J_p, x) > 2^{-n}$ implies that $p^N(x) \notin \tilde{U}$, and so the algorithm outputs 0 in this case. If, on the other hand, $x \in J_p$, then by lemma 4.2.3 $p^N(x) \in J_p \subset \tilde{V}$, and the algorithm would output 1. This shows the correctness of the algorithm.

Observe that $N = O(n)$, so we perform linearly many operations on x, hence we require linearly many bits of x in order to achieve the required (fixed) precision level at the end of the computation. Hence by theorem 3.2.2 we know that J_p is computable in time $poly(n) \cdot 2^{O(n)} = 2^{O(n)}$. Algorithm 1 *does not* compute J_p in our definition, since it might reject points $x \notin J_p$ which are very close to J_p (closer than 2^{-n-1}). In the next section we will be referring to the *exponential time* algorithm computing J_p in our sense as algorithm 1.

4.5 J_p is Poly-Time Computable

We are now ready to prove the main result of the paper, namely the poly-time computability of the Julia set J_p. We use the result from the previous section combined with a technique very similar to the technique used in [RW03] to pass from an exponential to a polynomial time algorithm. The goal is to estimate the distance from x to J_p both from above and below.

4.5.1 Estimating the Distance from J_p

The following lemma is our main tool for giving estimates on the distance from J_p.

Lemma 4.5.1 *There are positive constants $\alpha, \beta > 0$ computable from the initial data such that for any $z \in B(0, R')$ and $0 < \varepsilon < \alpha$ satisfying $d(J_p, z) \leq \varepsilon$, we have*

$$(|p'(z)| - \beta\varepsilon)d(J_p, z) \leq d(J_p, p(z)) \leq (|p'(z)| + \beta\varepsilon)d(J_p, z).$$

Proof: Once again, in the proof of this lemma, we do not try to achieve the best possible parameter values. A more involved analysis would probably yield better numbers.

Let $\alpha \leq \frac{r_c d}{4D}$. Then $d(J_p, z) < \alpha$ implies that $|p'(z)| > d$. Recall that we have denoted the polynomial $p(z) = c_m z^m + c_{m-1} z^{m-1} + \ldots + c_1 z + c_0$. Let

$$\beta > \frac{D^2}{d^2} \sum_{i=2}^{m} \left(\left(\frac{D\alpha}{d} \right)^{i-2} \cdot \sum_{k=0}^{m-i} \frac{(i+k)!}{i!k!} |c_{i+k}| R'^k \right).$$

This is obviously an easily computable number.

Let $j \in J_p$ be the element of J_p which is closest to z, i.e. $|j - z| = d(J_p, z) \leq \varepsilon$. J_p is compact, so such an element must exist. Then $p(j) \in J_p$ by lemma 4.2.3. Expanding to Taylor series around z (which has finitely many elements in this case), we obtain

$$d(J_p, p(z)) \leq |p(j) - p(z)| = \left| \sum_{i=0}^{m} \frac{p^{(i)}(z)(j-z)^i}{i!} - p(z) \right| = \left| \sum_{i=1}^{m} \frac{p^{(i)}(z)(j-z)^i}{i!} \right| \leq |p'(z)||j-z| +$$

$$\left| \sum_{i=2}^{m} \frac{p^{(i)}(z)(j-z)^i}{i!} \right| = |p'(z)||j-z| + |j-z|^2 \left| \sum_{i=2}^{m} \left((j-z)^{i-2} \sum_{k=0}^{m-i} \frac{(i+k)!}{i!k!} c_{i+k} z^k \right) \right| \leq$$

$$|p'(z)||j-z| + \varepsilon|j-z| \sum_{i=2}^{m} \left(\left(\frac{D\alpha}{d} \right)^{i-2} \cdot \sum_{k=0}^{m-i} \frac{(i+k)!}{i!k!} |c_{i+k}| R'^k \right) < (|p'(z)| + \beta\varepsilon) d(J_p, z).$$

In the opposite direction, let $l \in J_p$ be the point of J_p closest to $p(z)$, i.e. $d(J_p, p(z)) = |l - p(z)|$. $|p'(z)|$ is bounded from above by D, so it is not hard to see that p cannot expand distances by more than a factor of D, and so $|l - p(z)| = d(J_p, p(z)) \leq D \cdot d(J_p, z) \leq D\varepsilon < D\alpha \leq d \cdot \frac{r_c}{4}$. Let γ denote the straight line segment connecting $p(z)$ to l, that is $\gamma(0) = p(z)$,$\gamma(1) = l$ and $length(\gamma) = |l - p(z)| < d \cdot \frac{r_c}{4}$. Then we can lift γ to $\tilde{\gamma}$ So that

$\tilde{\gamma}(0) = z$ and $p(\tilde{\gamma}) = \gamma$. This can be done starting from $\tilde{\gamma}(0) = z$ as long as $\tilde{\gamma}$ does not hit any singularities. We will show that $\tilde{\gamma}$ cannot get $\frac{r_c}{2}$-close to any singularity so in particular $\tilde{\gamma}$ never passes through a singular point and $|p'(x)| > d$ for all x on $\tilde{\gamma}$.

Suppose $\tilde{\gamma}' : [0, t] \to \mathbb{C}$ for $0 \leq t \leq 1$ is a partial lift of γ until the first time it gets $\frac{r_c}{2}$-close to a singularity y_i, so that $\tilde{\gamma}'(0) = z$ and $|\tilde{\gamma}'(t) - y_i| = \frac{r_c}{2}$. This implies that $d(J_p, \tilde{\gamma}'(t)) \geq d(J_p, y_i) - \frac{r_c}{2} > r_c - r_c/2 = r_c/2$. Also $d(J_p, z) < \alpha < r_c/4$ and so $length(\tilde{\gamma}') \geq |z - \tilde{\gamma}'(t)| > \frac{r_c}{4}$. Note that $\tilde{\gamma}'$ does not pass close to any singularities, and so the contraction factor on it is at least d. $p(\tilde{\gamma}')$ is a portion of γ, and so $length(\gamma) \geq length(p(\tilde{\gamma}')) \geq d \cdot length(\tilde{\gamma}') > d \cdot \frac{r_c}{4}$, which is a contradiction.

Let $\tilde{\gamma}$ be a lift of γ as discussed before. Denote $\tilde{l} = \tilde{\gamma}(1)$. Then $p(\tilde{l}) = l$, and by lemma 4.2.3, $\tilde{l} \in J_p$. We have $d(J_p, z) \leq |\tilde{l} - z| \leq length(\tilde{\gamma}) \leq length(\gamma)/d = d(J_p, p(z))/d \leq D \cdot d(J_p, z)/d < D \cdot \alpha/d$. Just as before, we expand to Taylor series around z to obtain

$$d(J_p, p(z)) = |p(\tilde{l}) - p(z)| = \left| \sum_{i=0}^{m} \frac{p^{(i)}(z)(\tilde{l} - z)^i}{i!} - p(z) \right| = \left| \sum_{i=1}^{m} \frac{p^{(i)}(z)(\tilde{l} - z)^i}{i!} \right| \geq |p'(z)||\tilde{l} - z| -$$

$$\left| \sum_{i=2}^{m} \frac{p^{(i)}(z)(\tilde{l} - z)^i}{i!} \right| = |p'(z)||\tilde{l} - z| - |\tilde{l} - z|^2 \left| \sum_{i=2}^{m} \left((\tilde{l} - z)^{i-2} \sum_{k=0}^{m-i} \frac{(i+k)!}{i!k!} c_{i+k} z^k \right) \right| \geq$$

$$|p'(z)||\tilde{l} - z| - \frac{D \cdot d(J_p, z)}{d} \frac{D \cdot d(J_p, z)}{d} \sum_{i=2}^{m} \left(\left(\frac{D\alpha}{d} \right)^{i-2} \cdot \sum_{k=0}^{m-i} \frac{(i+k)!}{i!k!} |c_{i+k}| R'^k \right) \geq$$

$$|p'(z)| d(J_p, z) - \frac{\varepsilon D^2 \cdot d(J_p, z)}{d^2} \sum_{i=2}^{m} \left(\left(\frac{D\alpha}{d} \right)^{i-2} \cdot \sum_{k=0}^{m-i} \frac{(i+k)!}{i!k!} |c_{i+k}| R'^k \right) > (|p'(z)| - \beta\varepsilon) d(J_p, z),$$

which completes the proof. ■

4.5.2 The Algorithm

Lemma 4.5.1 says that the change in the distance from z to J_p is locally controlled by the expansion factor $|p'(z)|$ up to some small relative error. More precisely, if we choose $\alpha < r_c/2$, then we know that $|p'(z)| > d$, and so we can bound the *relative* error: $d(J_p, p(z)) \leq (|p'(z)| + \beta\varepsilon) d(J_p, z) = (1 + \frac{\beta\varepsilon}{|p'(z)|}) |p'(z)| d(J_p, z) \leq (1 + \frac{\beta\varepsilon}{d}) |p'(z)| d(J_p, z)$. Similarly, $d(J_p, p(z)) \geq (1 - \frac{\beta\varepsilon}{d}) |p'(z)| d(J_p, z)$. We apply the lemma to obtain an algorithm

which computes J_p in poly-time. We first give a slightly weaker version of the algorithm. Assume for this algorithm that $\alpha < \tilde{d}_l$ (which is a constant).

Algorithm 2

Input: The non-uniform input as described above,

the input $c_0, c_1, \ldots, c_m$ given on an oracle tape, m, n and

a binary x of length $O(n)$ are given as inputs.

Output: 0 if $d(J_p, x) > 2^{-n}D$, 1 if $d(J_p, x) < 2^{-n}/32$, either 0 or 1 otherwise.

1. Compute c, α and β described above.
2. Compute a natural number $N = O(n)$ such that

   $N \geq 2 + \log_c (M_W \cdot R' \cdot 2^{n-1}) + q$.
3. Compute $p^N(x)$ within an error of $\frac{\tilde{d}_l}{4}$.

   3.1. If $p^N(x) \in \tilde{V}$, output 1,

   3.2. Otherwise, goto 4.
4. Choose $\varepsilon \in 1/O(n)$ a power of 2 such that $\varepsilon < \alpha$, $(1 + \frac{\beta\varepsilon}{d})^{N+1} < 2$,

   and $(1 - \frac{\beta\varepsilon}{d})^{N+1} > 1/2$.

   4.1. If $\varepsilon \leq 2^{-n}$ decide the question of the algorithm using **Algorithm 1**.
5. Compute the smallest $M \leq N + 1$ such that $2^n\varepsilon/D < \Pi_{i=0}^{M-1}|p'(p^i(x))| < 2^n\varepsilon$,

   if no such M exists, output 0.
6. Apply **Algorithm 1** to check if $d(J_p, p^i(x)) < \varepsilon/16$ or $d(J_p, p^i(x)) > \varepsilon/2$, for

   each $0 \leq i \leq M$, to do this it is enough to compute $p^i(x)$ with precision $\varepsilon/16$,
7. Output 0, if **Algorithm 1** outputs $d(J_p, p^i(x)) > \varepsilon/2$ in one of the cases,

   and 1 otherwise.

Note that the algorithm is not quite a computation of J_p as per definition 3.1.6 of set complexity, because the ratio between the radius of acceptance and the radius of rejection is $32D$ rather than 2. This can be fixed by covering the ball $B(x, 2^{-n})$ with $< (32D)^2$ small balls with centers on the $32D \times 32D$ grid, and running Algorithm 2 on each of the grid points. Hence we can compute J_p within a constant factor from the complexity of

Algorithm 2. In the next section we will prove the correctness and analyze the running time of Algorithm 2.

4.5.3 Analysis of Algorithm 2

We will present a series of short lemmas about the algorithm, which will show its correctness.

Lemma 4.5.2 *If $d(J_p, x) > 2^{-n}D$, then the algorithm does not exit in line 3.*

Proof: Exactly as in the analysis of algorithm 1. Even under a weaker assumption of $d(J_p, x) > 2^{-n}$ we know that the orbit of x must exit $\tilde{U}$ in at most N steps, and so the algorithm will not exit in line 3 in this case. ■

Lemma 4.5.3 *ε as described in line 4 exists.*

Proof: We have $\left(1 + \frac{\beta}{d} \cdot \frac{d}{\beta(N+1)}\right)^{N+1} \approx e$ and $\left(1 - \frac{\beta}{d} \cdot \frac{d}{\beta(N+1)}\right)^{N+1} \approx \frac{1}{e}$. So we can choose $\varepsilon = O\left(\frac{d}{\beta(N+1)}\right) = 1/O(n)$ which satisfies the condition of line 4. ■

Lemma 4.5.4 *If $d(J_p, x) < 2^{-n}/32$ and the algorithm reaches line 5, then there is an M as described in line 5 (and the algorithm finds it).*

Proof: We know that the orbit of x exits $\tilde{U}$ in no more than $N + 1$ steps, otherwise the algorithm would have stopped in line 3.

We have assumed that $\varepsilon < \alpha < \tilde{d}_l$, so $d(J_p, p^{N+1}(x)) \geq d(J_p, \tilde{U}^c) > d(\tilde{V}, \tilde{U}^c) \geq \tilde{d}_l > \varepsilon$. On the other hand, by the condition in line 4.1, $d(J_p, x) < 2^{-n}/32 < \varepsilon/32$. Let $0 < T \leq N + 1$ be the smallest natural number such that $d(J_p, p^T(x)) > \varepsilon$. Then $p^{T-1}(x)$ must be in $\tilde{U}$ (because any point of the orbit outside $\tilde{U}$ satisfies the condition). We also have $d(J_p, p^i(x)) \leq \varepsilon$ for all $i = 0, 1, \ldots, T - 1$, hence by the bound on the relative error we have from lemma 4.5.1 (see the discussion after the statement of the lemma), we have

$$\varepsilon < d(J_p, p^T(x)) \leq \left(1 + \frac{\beta\varepsilon}{d}\right)^T \Pi_{i=0}^{T-1} |p'(p^i(x))| d(J_p, x) < 2\Pi_{i=0}^{T-1} |p'(p^i(x))| (2^{-n}/32).$$

Hence $\Pi_{i=0}^{T-1}|p'(p^i(x))| > 2^n\varepsilon$. $p^i(x) \in \tilde{U}$ for all $i < T$ we have $|p'(p^i(x))| < D/2$ for all such i. We also have $\Pi_{i=0}^{0}|p'(p^i(x))| = 1 < 2^n\varepsilon$. So we can pick a smallest $1 \leq M < T$ such that $\Pi_{i=0}^{M-1}|p'(p^i(x))|$ falls in the interval $(2^n\varepsilon/D, 2^n\varepsilon)$, which completes the proof. ■

The last two lemmas show that the algorithm does the right thing if it reaches lines $6-7$.

Lemma 4.5.5 *If $d(J_p, x) < 2^{-n}/32$ and the algorithm reaches line 6, then it outputs 1.*

Proof: We prove that the algorithm does not reject for any $0 \leq i \leq M$ by induction on i. It suffices to show that $d(J_p, p^i(x)) < \varepsilon/16$ for all $0 \leq i \leq M$.

$d(J_p, x) < 2^{-n}/32 < \varepsilon/32 < \varepsilon/16$, so the statement holds for $i = 0$. Suppose that the statement holds for some $0, 1, 2, \ldots, i-1 \leq M-1$, we want to show that it holds for i. By the minimality of M in line 5 we know that $\Pi_{j=0}^{i-1}|p'(p^j(x))| < 2^n\varepsilon$. By the induction hypothesis $d(J_p, p^j(x)) < \varepsilon/16 < \varepsilon$ for all $j < i$. Hence by the estimate on the relative error which follows from lemma 4.5.1 and we have

$$d(J_p, p^i(x)) \leq \left(1 + \frac{\beta\varepsilon}{d}\right)^i \left(\Pi_{j=0}^{i-1}|p'(p^j(x))|\right) d(J_p, x) < 2 \cdot 2^n\varepsilon \cdot 2^{-n}/32 = \varepsilon/16.$$

So the algorithm outputs 1 in this case. ■

Lemma 4.5.6 *If $d(J_p, x) > 2^{-n}D$ and the algorithm reaches line 6, then it outputs 0.*

Proof: Assume, on the contrary, that the algorithm outputs 1. This means that $d(J_p, p^i(x)) \leq \varepsilon/2 < \varepsilon$ for all $i = 0, 1, \ldots, M$. Hence we can apply the lower bound on the relative error that follows from lemma 4.5.1 to obtain

$$d(J_p, p^M(x)) \geq \left(1 - \frac{\beta\varepsilon}{d}\right)^M \left(\Pi_{j=0}^{M-1}|p'(p^j(x))|\right) d(J_p, x) > \frac{1}{2}(2^n\varepsilon/D)2^{-n}D = \varepsilon/2,$$

contradiction. ■

We now discuss the time complexity of the algorithm. Denote the time complexity of Algorithm 1 by $S_1(n)$ and the time complexity of Algorithm 2 by $S_2(n)$.

Lemma 4.5.7 *The time complexity of Algorithm 2 is* $O(nM(n) + nS_1(\log n + O(1)))$*, where* $M(n)$ *is the time complexity of multiplying two binary* n*-digit numbers.*

Proof: We analyze the complexity of the algorithm line by line.

Line 1 — Requires constant $O(1)$ time.

Line 2 — By doubling N each time can be easily done in $O(M(n)\log n)$ steps.

Line 3 — Requires computations with precision $O(n)$ bits, hence the total complexity of this line is $O(nM(n))$.

Line 4 — Requires $O(n)$ steps if we want to write ε down. Observe that $\varepsilon = 1/O(n)$ and so $\log(1/\varepsilon) = \log n + O(1)$.

Line 4.1 — Requires $O(S_1(\log \frac{1}{\varepsilon})) = O(S_1(\log n + O(1)))$ to finish the running of the algorithm.

Line 5 — Requires $O(nM(n))$ time.

Line 6 — We need $O(nM(n))$ time to compute the orbit and then we use $O(n)$ applications of algorithm 1 with parameter $\log(1/O(\varepsilon))$ which require $O(nS_1(\log n + O(1)))$ time.

Line 7 — Requires $O(n)$ time.

By adding up all the complexities we see that the time complexity of the algorithm is dominated by the time complexity of line 6, which is $O(nM(n) + nS_1(\log n + O(1)))$. ■

From the discussion following Algorithm 1 we know that $S_1(n) = 2^{O(n)}$, hence $S_2(n) = O(nM(n) + n2^{O(\log n)}) = poly(n)$.

4.5.4 Improving Algorithm 2

We can now improve Algorithm 2, by using the poly-time Algorithm 2 in steps 4.1, 6 and 7 instead of the exponential Algorithm 1:

Algorithm 3

Runs exactly as **Algorithm 2**, except that it uses **Algorithm 2** instead of **Algorithm 1** in lines 4.1, 6 and 7.

The time complexity of Algorithm 3 is $O(nM(n) + nS_2(\log n + O(1)))$, where $S_2(k)$ is the complexity of running Algorithm 2 with input k. We have $S_2(k) = poly(k)$, hence Algorithm 3 runs in time $S_3(n) = O(nM(n) + n \cdot poly(\log n)) = O(nM(n))$.

Using Schönhage-Strassen algorithm for fast multiplication (see [Knu97] p. 311 and [SS71]), we can bound $M(n) \leq O(n \log n \log\log n)$. We conclude,

Theorem 4.5.8 *For every fixed hyperbolic polynomial $p(z)$, the Julia set J_p is poly-time computable in time $O(nM(n)) \leq O(n^2 \log n \log\log n)$.*

The result above shows that every *fixed* hyperbolic Julia set is poly-time computable. Our next goal is to show that they are also *uniformly* computable with the polynomial $p(z)$ being given as a parameter, with no hardwired information about it.

4.6 Uniformizing the Construction

The construction in the previous sections can indeed be uniformized over all *hyperbolic* polynomials. As seen in the following theorem.

Theorem 4.6.1 *The Julia set J_p, where the coefficients of a* **hyperbolic** *$p(z)$ are given as oracles, can be locally computed with precision 2^{-n} in time $O(nM(n))$, where the constant factor in the $O(\bullet)$ depends on $p(z)$ but not on n. In other words the time complexity of locally computing J_p is bounded by $K(p) \cdot nM(n)$ for some $K(p)$. Here, again, $M(n) \leq O(n \log n \log\log n)$ is the complexity of multiplying two n-bit numbers.*

Our goal in this section is to prove theorem 4.6.1. It should be noticed that as seen from the construction, the constant $K(p)$ could be very big. For example it will grow exponentially fast in $\frac{1}{\varepsilon}$ for $p(z) = z^2 + 1/4 + \varepsilon$.

All we have to do in order to prove theorem 4.6.1 is to show that the nonuniform information given in section 4.4.1 can extracted directly from the coefficients of $p(z)$ given as oracles. At this point we are not concerned with the complexity of computing this information.

It is not hard to see that using standard numerical analysis techniques one can approximate the critical points $y_1, y_2, \ldots, y_{m-1}$ with an arbitrarily good precision. For every natural l we can find all the periodic orbits of period l by solving the polynomial equation $p^l(z) - z = 0$. Then by approximations we can check if each of the orbits is an attracting orbit, and if it is, we can find some contraction factor > 1 near the orbit, and output the numbers x_{ij}, r_{ij} and s_i as required in the nonuniform information. This means that if we will keep looking for all the attracting orbits we will *eventually* find all of them.

Once we have found *all* the attracting orbits it is very easy to compute the rest of the nonuniform information (i.e. R, R', r_c, q, $\tilde{d}_l$, d and D) using standard numerical analysis techniques. The problem, of course, is how do we know at which point have we found *all* the attracting orbits. The number of there orbits is not fixed a-priori, so the algorithm described above will never terminate. To deal with this problem, we use the following theorem about attracting orbits. See [Mil00], Theorem 8.6 for a discussion and a proof.

Theorem 4.6.2 *If p is a polynomial of degree ≥ 2, then the immediate basin of every attracting periodic orbit contains at least one critical point.*

In particular, for every attracting periodic orbit o of p there is a critical point y_i such that the orbit of y_i converges to o. On the other hand, we have assumed that p is hyperbolic, hence every critical point converges to an attracting periodic orbit. So for each i the algorithm above will eventually find the attracting periodic orbit to which y_i converges. Once we have found the attracting periodic orbits $O = \{o_1, o_2, \ldots, o_s\}$ such that y_i converges to one of the orbits in O or to ∞ for $i = 1, 2, \ldots, m-1$, we know that by theorem 4.6.2 there can be no other attracting periodic orbits and we can stop looking

for them at this point. We summarize the process of looking for attracting periodic orbits as follows:

Keep looking for attracting periodic orbits $o_1, \ldots, o_s$ until each y_i converges to one of these orbits or to ∞.

We will omit the details on how to implement this procedure. This completes the proof of theorem 4.6.1.

4.7 Can the Results be Improved?

It is a natural question of whether the result of theorem 4.6.1 can be extended beyond the hyperbolic Julia sets. In particular, is there a uniform algorithm which computes the Julia set J_p from the coefficients of p for an *arbitrary* polynomial p.

Note that now we are no longer trying to compute a single set J_p but rather a *set-valued function* $J : p(z) \mapsto J_p$. In other words, given a good approximation of the coefficient of $p(z)$ we hope to be able to compute J_p with a given precision. We have seen that one of the equivalent definitions of set computability is Hausdorff approxiambility. That is, given a precision parameter n and the coefficients of $p(z)$ as oracles, we would like to give a 2^{-n}-Hausdorff approximation of J_p.

We restrict our attention to quadratic polynomials of the form $p(z) = z^2 + c$ for $c \in \mathbb{C}$. In this case J is a set valued function $J : c \mapsto J_{z^2+c}$. Denote the set of the compact subsets of $\mathbb{C}$ by K^*, and view K^* as a metric space with the Hausdorff metric on it. Then uniformly computing J_{z^2+c} for all c is equivalent to computing the function $J : c \mapsto J_{z^2+c}$ in the sense of definition 2.1.7 with K^* replacing $\mathbb{R}^k$ as the target space. For these functions the analogue of theorem 2.1.8 holds. We omit the proof here.

Lemma 4.7.1 *If a function $f : \mathbb{R}^m \to K^*$, where K^* is the set of compact sets in $\mathbb{C}$, is computable by an oracle machine M^ϕ, then f is continuous in the Hausdorff metric.*

Hence if $J : c \mapsto J_{z^2+c}$ is computable, it must be continuous in the Hausdorff metric.

On the other hand, we have the following result which follows from [Dou94], section 11, theorem 11.3:

Lemma 4.7.2 *The function $J : c \mapsto J_{z^2+c}$ is not continuous at $c = \frac{1}{4}$ in the Hausdorff metric.*

Lemmas 4.7.1 and 4.7.2 imply together that

Theorem 4.7.3 *No oracle machine M^ϕ, where ϕ represents the number c, can compute the Julia set J_{z^2+c}.*

In fact, most of the programs written to draw Julia sets perform poorly near the discontinuity c=0.25, for polynomials of the form $p(z) = z^2 + 0.25 + \varepsilon$ for small positive values of ε.

This theorem does not contradict theorem 4.6.1, because theorem 4.6.1 only works for hyperbolic polynomials, while the polynomial $p(z) = z^2 + 0.25$ around which the discontinuity occurs in lemma 4.7.2 is parabolic and not hyperbolic. Note that the point $c = 0.25$ lies on the boundary of the Mandelbrot set (see figure 4.1).

In particular we have shown that hyperbolic and parabolic Julia sets cannot be uniformly computed by the same oracle machine.

Chapter 5

Directions of Future Work

In this chapter we give several directions in which the present work can be extended.

5.1 Extending the Definition of Computable Functions

As it was pointed out in section 2.1, all computable functions must be continuous. Even the simplest step function $\chi_{[0,\infty)}$ is not computable under the present definition.

The general purpose of the computability theory is to classify problems into "easy" (computable) and "hard" (noncomputable). Keeping this purpose in mind one could argue that the simple function mentioned above is misclassified by the current computability definition.

An alternative definition of computability can be proposed based on the connection established in theorem 2.3.1 between computable functions and computable sets. Theorem 2.3.1 says that a *continuous* function is computable if and only if its graph is computable. This suggests extending the definition of computability to discontinuous functions by saying that a function is computable if and only if its graph is computable. The step function $\chi_{[0,\infty)}$ would obviously be computable under this definition because its

graph is just a union of two rays.

Formally, if $\mathcal{C}$ is a class of functions extending the class of continuous functions we can define.

Definition 5.1.1 (tentative) *Let $\mathcal{C}$ be a class of functions from some compact rectangle $S \subset \mathbb{R}^k$ to $\mathbb{R}$ extending the class of the continuous functions. We say that a function $f \in \mathcal{C}$ is computable if and only if the closure of its graph $G_f = \{(x, f(x)) \ : \ x \in S\}$ is computable as a subset of $\mathbb{R}^{k+1}$.*

Note that in the case where $\mathcal{C}$ is just the class of the continuous functions, definition 5.1.1 is equivalent to the standard definition 2.1.7 of functions computability.

One of the difficulties is to select the right class $\mathcal{C}$ of functions. We want $\mathcal{C}$ to be rich enough to include simple discontinuous functions such as the step function mentioned above. On the other hand, we don't want $\mathcal{C}$ to be too rich. Suppose we had chosen $\mathcal{C}$ to be the class of *all* functions. Consider an arbitrary $A \subset [0,1]$ such that $\overline{A} = [0,1]$ and $\overline{A^c} = [0,1]$. Let $f_A : [0,1] \to [0,1]$ be defined by

$$f_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases}$$

Then the closure of the graph of f_A is a union of two line segments, so it is computable according to definition 5.1.1 *regardless* of our choice of the set A. This doesn't seem right.

The best choice for the set $\mathcal{C}$ is yet to be investigated. One possible choice is the class of functions with bounded variation, or some extension of this class.

Questions regarding the computability of the basic operations, such as composition, maximum, integration etc. on computable functions under the new definitions are also to be investigated.

5.2 Computability and Complexity of Julia Sets of Other Types

In the present work we have resolved the questions regarding the complexity of hyperbolic Julia sets. The questions regarding the computability and complexity of other classes of Julia sets remain open.

We believe that *parabolic* Julia sets are computable, but it is unclear whether they are poly-time computable.

The complexity of other classes of Julia sets, in particular sets with Siegel discs and Cremer points, is yet to be investigated. It appears, however, that some Julia sets of these types are noncomputable (see next section). More information about the different types of Julia sets can be found in [Mil00].

5.3 Noncomputable Julia Sets

Theorem 4.7.3 has established that there is no Turing Machine *universally* computing the Julia set J_p as a function of the polynomial p. Whether there is a *particular* polynomial $p(z)$ such that no machine can compute J_p given the coefficients of p as oracles is a much more difficult problem.

A diagonalization-like technique can be employed to nonconstructively demonstrate the existence of such a polynomial of the form $p(z) = z^2 + c$. The fundamental idea is still to employ theorem 4.7.1 along with the structure of the discontinuities of $J : c \mapsto J_{z^2+c}$. The work on noncomputable Julia sets is joint with Michael Yampolsky from the University of Toronto.

5.4 Computability and Complexity of Mandelbrot's Set

The Mandelbrot set is probably the most famous fractal object arising in complex dynamics. For a complex number $c \in \mathbb{C}$ consider the sequence $0, c, c^2 + c, (c^2 + c)^2 + c, \ldots$ obtained from 0 by iterating the map $z \mapsto z^2 + c$. The Mandelbrot set M is the set of points for which this sequence does not diverge to ∞.

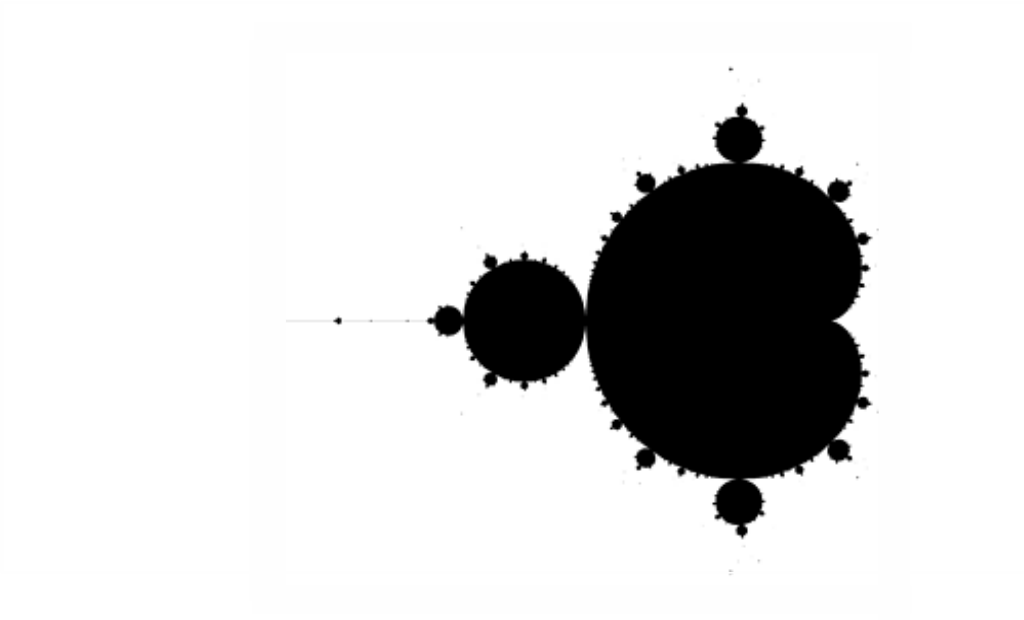

Figure 5.1: The Mandelbrot set.

The Mandelbrot set can be viewed as a set of indexes c for Julia sets of the form J_{z^2+c}. The points outside the set and all the known points inside the set correspond to hyperbolic Julia sets. It is a major open problem in complex dynamics whether *all* the points in the interior of the Mandelbrot set correspond to hyperbolic Julia sets (see [McMl94] for more details). The conjecture is also known as the question of whether the hyperbolic components are dense in the Mandelbrot set.

The question of the computability of the Mandelbrot set is open. Assuming the

density of the hyperbolic components it can be shown that it is computable. On the other hand, it is hard to imagine a proof of the computability of M without assuming the hyperbolic components density or a similar conjecture.

Even assuming the hyperbolic components density conjecture, the question of the computational complexity of M is still wide open. It would be interesting to get efficient algorithms for computing M even in the Ko computability sense.

5.5 Computability in Other Dynamical Systems

Dynamical systems occur everywhere in the real world. Almost every physical process which is complex enough has a dynamical system with a very complicated behavior attached to it. This makes the study of dynamical systems important in general, and in the particular setting of the computability and complexity theory.

The Julia sets are some of the best studied dynamical systems. This allowed us to answer several key questions regarding their computability and complexity. The situation is more complicated and less studied for other dynamical systems. Examples of such systems range from the Newtonian N body problem, through population dynamics to chemical reactions. Answering basic computability questions as well as developing algorithms is of both practical and theoretical importance.

5.6 Church's Thesis

It has been suggested (see [Yao02] for example) that any computationally hard-to-simulate physical system can be used to challenge Church's thesis and the extended Church's thesis.

The Church-Turing thesis, or just Church's thesis, is the belief that if a function can be computed by any conceivable physical device, then it can be computed by a Turing Machine. The Extended Church-Turing thesis asserts also that a Turing Machine is as

efficient as any physical device. Formally, if a problem can be solved in time $T(n)$ using some physical device, then it can be solved in time $(T(n))^k$ for some k on a Turing Machine.

The extended Church-Turing thesis has been recently challenged by the possibility of constructing a quantum computer which would factor integers in polynomial time.

The noncomputability results for Julia sets (see section 5.3) suggest that dynamical systems can contain a high level of noncomputability. Some of the dynamical systems arrive from physical systems, which gives us the hard-to-simulate physical system we need to challenge the Church-Turing thesis. The problem is that we need some level of robustness in the physical system in order to use it for computation. We need this robustness to compensate for our inability to provide the input with infinite accuracy.

An interesting direction is to try to provide an *observable parameter* for some dynamical system which is

1. Noncomputable or hard to compute, and

2. is sufficiently robust (not too sensitive to minor changes of the initial conditions).

Such a system, if it exists, might even have some practical computational applications.

Bibliography

[BCSS] L. Blum, F. Cucker, M. Shub, S. Smale, Complexity and Real Computation, Springer, New York, 1998.

[BW99] V. Brattka, K. Weihrauch, Computability of Subsets of Euclidean Space I: Closed and Compact Subsets, Theoretical Computer Science, **219**, pp. 65-93, 1999.

[CK95] A. Chou, K. Ko, Computational complexity of two-dimensional regions, SIAM J. Comput. **24**, pp. 923-947, 1995.

[Dou94] A. Douady, Does a Julia set depend continuously on the polynomial? Proc. Symposia in Applied Math.: Complex Dynamical Systems: The Mathematics Behind the Mandelbrot Set and Julia Sets, vol **49**, 1994, ed R. Devaney (Providence, RI: American Mathematical Society) pp. 91-138.

[Fri84] H. Friedman, On the Computational Complexity of Maximization and Integration, Advances in Math. **53**, pp. 80-98, 1984.

[Jo02] J. Jost, Compact Riemann Surfaces, Second edition, Springer, 2002.

[Ko86] K.Ko, Approximation to Measurable Functions and its Relation to Probabilistic Computation, Annals of Pure and Applied Logic, **30**, 173-200, 1986.

[Ko91] K. Ko, Complexity Theory of Real Functions, Birkhäuser, Boston, 1991.

[Ko98] K. Ko, Polynomial-time computability in analysis, in "Handbook of Recursive Mathematics," Volume **2**, Recursive Algebra, Analysis and Combinatorics, Yu. L. Ershov et al. (Editors), 1998, pp. 1271-1317.

[Knu97] D. Knuth, The Art of Computing Programming, v. 2: Seminumerical Algorithms, 3rd ed., Addison-Wesley, 1997.

[McMl94] C. McMullen, Complex Dynamics and Renormalization, Princeton University Press, Princeton, New Jersey, 1994.

[Mil00] J. Milnor, Dynamics in One Complex Variable - Introductory Lectures, second edition, Vieweg, 2000.

[Pick98] C. A. Pickover (ed.), Chaos and Fractals – Computer Graphical Journey, Ten Year Compilation of Advanced Research. Elsevier, 1998.

[PR89] M. B. Pour-El, J. I. Richards, Computability in Analysis and Physics, Springer-Verlag, 1989.

[RW03] R. Rettinger, K. Weihrauch, The Computational Complexity of Some Julia Sets, in STOC'03, June 9-11, 2003, San Diego, California, USA.

[Sau87] D. Saupe, Efficient Computation of Julia Sets and Their Fractal Dimension. Physica D, **28**, pp. 358–370, 1987.

[SS71] A. Schönhage, V. Strassen, Schnelle Multiplikation grosser Zahlen, Computing 7: pp. 281–292, 1971.

[TW98] J. F. Traub, A. G. Werschultz, Complexity and Information, Cambridge University Press, 1998.

[Tur36] A. M. Turing, On Computable Numbers, With an Application to the Entscheidungsproblem. In *Proceedings, London Mathematical Society*, 1936, pp. 230-265.

[Wei00] K. Weihrauch, Computable Analysis, Springer, Berlin, 2000.

[Yao02] A. Yao, Classical Physics and the Church-Turing Thesis, Electronic Colloquium on Computational Complexity, Report No. **62**, 2002.

[Zh98] N. Zhong, Recursively enumerable subsets of R^q in two computable models: Blum-Schub-Smale machine and Turing machine. Theoretical Computer Science, **197**, pp. 79-94, 1998.