# On the Complexity of Real Functions

Mark Braverman[1]
Department of Computer Science
University of Toronto
mbraverm@cs.toronto.edu

## Abstract

*We establish a new connection between the two most common traditions in the theory of real computation, the Blum-Shub-Smale model and the Computable Analysis approach. We then use the connection to develop a notion of computability and complexity of functions over the reals that can be viewed as an extension of both models. We argue that this notion is very natural when one tries to determine just how "difficult" a certain function is for a very rich class of functions.*

## 1 Introduction

Questions of computability and complexity over the reals are of extreme importance for understanding the connections between nature and computations. Assessing the possibilities of scientific computing in simulating and predicting natural processes depends on an agreed upon and well studied notion of "real computation". Addressing issues of physics and the Church-Turing Thesis also requires a strong notion of computability over continuous spaces [23].

In the discrete setting, where the objects are from $\{0,1\}^*$, the concepts of computability and complexity are very well studied. There are different approaches that have been proved to yield equivalent definitions of computability and (almost) equivalent definitions of complexity. From the Formal Logic point of view we have the notion of recursiveness, from Computational Complexity we have the notion of a Turing Machine, and modeling the usual computer yields the notion of abstract RAM. All these converge to the same well accepted concept of *computability*.

In the continuous setting, where the objects are numbers in $\mathbb{R}$, the situation is far less clear. Real Computation has been studied since Turing's original 1936 paper [21], where he introduced the Turing Machine. In that paper he dealt with the computability notion of a single real number. Today, we still do not have a unified and widely accepted notion of real computability and complexity.

Consider the two simplest objects over $\mathbb{R}^n$: real sets and real functions. Several different approaches to the computability of these objects have been proposed over the last seven decades. Unlike the discrete case, most approaches deal with computability of *functions* before the decidability of *sets*. In the current paper we consider the two approaches that are most common today: the tradition of Computable Analysis – the "bit model", and the Blum-Shub-Smale approach. These two approaches have been developed to capture different aspects of real computation, and in general are not equivalent.

The bit-model is very natural as a model for scientific computing when continuous functions on compact domains are involved. In particular, the "calculator" functions, such as $\sqrt{x}$, $\sin x$ and $\log x$ are bit-computable, at least on some properly chosen domains. None of these functions are computable in the BSS model, because computability in the BSS model requires the function to have a very special algebraic structure (essentially being piecewise-rational). This makes the BSS notion inapplicable to scientific computing unless some modifications are made. The BSS model mimics some aspects of the actual way numerical analysts think about problems, and in fact [4] contains many strong results on solving fundamental numerical problems in algebra, such as finding roots of a polynomial. The model's disadvantage is the difficulty in interpreting negative results. The function $e^x$ and its graph are not BSS computable, and yet we can easily compute the exponential function and plot its graph. Modifications that can be made to the model to deal with these problems have been discussed in [20]. To our knowledge, these modifications have not been formalized.

On the other hand, there are very natural and simple functions that are BSS computable but not bit-computable. The simplest step function $\chi_{[0,\infty)}$ is an example.

In the set setting, the models also often give fundamentally different answers to questions of computability. Consider, for example, the Koch snowflake $K$, the Julia set

[1]Research is partially supported by an NSERC postgraduate scholarship

$J = J_{z^2+1/4}$, and the Mandelbrot set $M$. In the BSS model none of these sets are computable (see [4]). In the bit model, $K$ is very easy to draw and is computable, $J$ is computable, although not as easily as $K$ [3], and the question of whether $M$ is computable is open and depends on some deep conjectures from complex dynamics [14]. For most "common" sets BSS is a more restrictive model, but this is not always the case, as demonstrated in section 3.2.

We propose three simple natural modifications to the BSS model. Firstly, we allow the machines to use only computable constants. This amounts to working on a smaller field of the computable reals. Secondly, we allow the machines to make some errors. This has been done informally before by analyzing the "error+condition number" of a problem in the BSS model. Thirdly, we require an *a priori* estimate on the running time given as a function of the error parameter $\varepsilon$. In section 3.2 we present an example showing the importance of this condition. We show that under these modifications the BSS model becomes equivalent to the bit-model for sets. This involves simulating an infinite-precision machine with a finite precision one. It can only be done symbolically using the algebraic structure of the machine's constants and $\mathbb{Q}$, followed by a general quantifier elimination over $\mathbb{R}$.

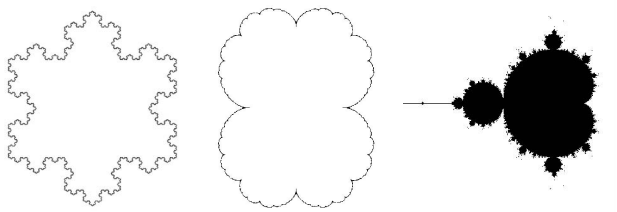

**Figure 1. The sets $K$ (left), $J$ (center), and $M$ (right)**

The primary goals of this paper are:

- to show that under the reasonable modifications mentioned above, the two models become computationally equivalent for *sets*;
- to propose a new notion of computability for *functions* which extends both models, and takes advantage of some positive features in both; and
- based on the above, to propose the "right" notion of computational complexity for discontinuous functions, extending naturally from the continuous case.

This gives us the "right" notion of complexity for functions for which no such notion previously existed. For example, the 2-valued function $\sqrt{\ } : \mathbb{C} \to \mathbb{C}$, as well as its single-valued branches defined on subsets of $\mathbb{C}$.

The paper is organized as follows. In Section 2 we discuss the bit model, and present some old and some new results. In Section 3 we discuss the BSS model and possible modifications to it. In Section 4 we propose a new computability and complexity definition for some discontinuous and multi-valued functions.

## 2 The Bit Model

### 2.1 The Model of Computation

The computability of functions in the bit model as we know it today was first proposed by Grzegorczyk [13] and Lacombe [16]. It has since been developed and generalized. More recent references on the subject include [15], [18], and [22].

The basic model of computation here is an oracle Turing Machine. Denote by $\mathbb{D} = \{\frac{k}{2^\ell} \ : \ k \in \mathbb{N}, \ \ \ell \in \mathbb{N}\}$ the set of the dyadic numbers. These are the rationals that have a finite binary expansion. An oracle for a number $x$ is a function $\phi : \mathbb{N} \to \mathbb{D}$ such that $|\phi(n) - x| < 2^{-n}$. The oracle terminology is just a natural way to separate the complexity of computing $x$ from the complexity of computing *on* $x$ as a parameter. For most purposes one can think of the oracle $\phi$ for $x$ as an infinite tape containing the binary expansion of $x$.

Consider a function $f : \mathbb{R} \to \mathbb{R}$. In plain language, a program $M$ computing $f$ would output a good approximation of $f(x)$, provided approximations of the input $x$. More formally, the oracle machine $M^\phi(n)$ outputs a $2^{-n}$-approximation of $f(x)$ for any valid oracle $\phi$ for $x$. The definition extends naturally to a function $f : \mathbb{R}^k \to \mathbb{R}^\ell$. Here $M = M^{\phi_1,\phi_2,\ldots,\phi_k}(n)$ is allowed to query each of the $k$ parameters with an arbitrarily good precision and is required to output the $\ell$ values of $f$ with precision $2^{-n}$.

### 2.2 Basic Properties and Examples

One of the main properties of computable functions is that they are continuous. For a computable $f : \mathbb{R} \to \mathbb{R}$, the machine $M_f^\phi(n)$ can query the input $x$ via $\phi$ with only finite level of precision before it outputs an approximation $q$ of $f(x)$. This means that $q$ should be a good approximation of $f(x')$ for all $x'$ in some small neighborhood of $x$. Thus $f(x')$ should be close to $f(x)$ whenever $x'$ is close enough to $x$. A formal proof can be found in most standard references, e.g. in [22].

The *computable* $\Rightarrow$ *continuous* property implies that even the simplest discontinuous function, the *step function*

$$s_0(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases} \qquad (1)$$

is not computable under this definition. This is in contrary to the intuition that $s_0$ must be a "simple" function. One could also argue that some physical systems, e.g. quantum energy levels, are best described using step functions and other simple discontinuous functions. One of the goals of the present work is to develop notions which deal with this problem.

We recall the classical definition of computable real numbers, introduced by Turing in [21]. Informally, this definition says that a real number $x$ is computable if we can compute arbitrarily good approximations of $x$.

**Definition 1** *A number $x \in \mathbb{R}$ is computable if and only if a representation $\phi$ of $x$ as described above can be computed.*

A constant function $f(x) = a$ is computable if and only if the number $a$ is computable. Most "standard" continuous functions are computable in this model. For instance, the exponential function $f(x) = e^x$ is computable on $\mathbb{R}$ using the Taylor series expansion of $e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$. It is not hard to estimate the number of terms and the precision of $x$ we need to consider in order to get a $2^{-n}$-approximation of $e^x$.

Denote by $\mathcal{C}$ the set of computable real numbers. Then $\mathcal{C}$ is closed under applications of any computable functions. In particular $\mathcal{C}$ is a field. Using the convergence of the Newton method for approximating roots of polynomials one sees that $\mathcal{C} + i\mathcal{C} \subset \mathbb{C}$ is an algebraically closed field.

The time complexity $T_f(n)$ for a function $f$ is the worst-case time complexity for computing a $2^{-n}$-approximation of $f(x)$ given an oracle $\phi$ for $x$. We charge $m$ time units for querying $\phi(m)$. Note that this definition is completely in the classical setting, and all the usual complexity function classes are defined here.

For continuous functions, the complexity notion backs our intuitive perception of "hard" vs "easy" functions. For example, all the continuous "calculator" functions are computable in time not exceeding $O(n^3)$.

### 2.3 Computability and Complexity of Real Sets

Definitions of effective subsets of $\mathbb{R}^n$ based on the concept of computability have been proposed as early as the mid 50's [17]. We refer the reader to [10] and [22] for a more detailed discussion. By "computing" a set $S$, we mean generating increasingly precise "images" of $S$. We restrict our attention to bounded subsets of $\mathbb{R}^n$.

Consider the two-dimensional case, which is closely related to computer graphics. Intuitively, in this case, the set $S$ is computable if we can draw arbitrarily good "zoom-ins" into it. One can view a $2^{-n}$-precise image of $S$ on a screen as a collection of radius-$2^{-n}$ pixels such that the following two conditions are fulfilled:

1. If a pixel contains a point from $S$, then it is colored black. This ensured that the entire set appears on the screen.
2. If a pixel is far (say $2^{-n}$-far) from $S$, then it is colored white. This ensures that the picture is a faithful image of $S$.

We can take the pixels to be balls of radius $2^{-n}$ with a dyadic center $d \in \mathbb{D}^n$. Formally,

**Definition 2** *We say that a bounded set $S$ in $\mathbb{R}^n$ is bit-computable, if there is a machine $M(d, n)$ computing a function from the family*

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

On Fig. 2 we see some sample values of the function $f$. It should be noted that the definition remains the same if we take square pixels instead of the round ones. It is also unchanged if we replace the ratio between the inner and the outer radius with some $\alpha > 1$ instead of 2.

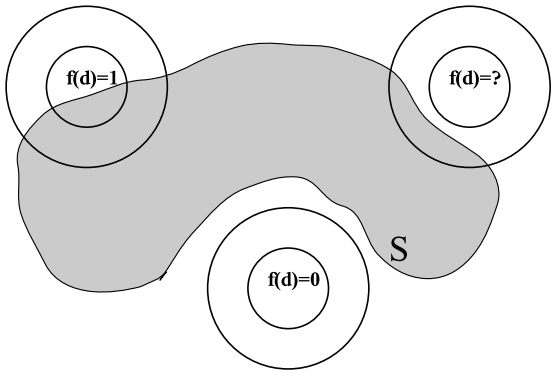


**Figure 2. Sample values of $f$. The radius of the inner circle is $2^{-n}$.**

The time complexity $T_S(n)$ is defined as the *worst-case* running time of a machine $M(d, n)$ computing a function from the family (2). Low time complexity means that it is easy to draw deep zoom-ins of the set $S$. We say that $S$ is poly-time computable if $T_S(n) < p(n)$ for some polynomial $p$.

To see why this is the "right" definition, consider the two-dimensional case. Suppose we are trying to draw $S$ on a computer screen which has a $1000 \times 1000$ pixel resolution. A $2^{-n}$-resolution picture of $S$ has $O(2^{2n})$ pixels of size $2^{-n}$, and thus would take time $O(T_S(n) \cdot 2^{2n})$ to compute. This quantity is exponential in $n$, even if $T_S(n)$

is bounded by a polynomial. But we are drawing $S$ on a finite-resolution display, and we will only need to draw $1000 \cdot 1000 = 10^6$ pixels. Thus the running time would be $O(10^6 \cdot T_S(n)) = O(T_S(n))$. This running time is polynomial in $n$ if and only if $T_S(n)$ is polynomial. Hence $T_S(n)$ reflects the 'true' cost of zooming into $S$.

So far, one might have been left with the impression that the computability definition above is not very robust. In fact, on the contrary, it is equivalent to several other reasonable definitions. For example, $S$ is computable if and only if the distance function, $d_S(x) = \inf_{y \in S} |x - y|$, is computable.

We now present a more subtle equivalent definition. This definition was first introduced by Chow and Ko in [12] under the name of *strong recognizability*. It wasn't known at the time that this definition is, in fact, equivalent to the usual bit-computability definition (Definition 2), as we will show below.

The idea of the definition is to relax the conditions of Definition 2. We are given a point $x$ as an oracle to $M^\phi(n)$, and we must output 1 if $x \in S$ and 0 if $x$ is $2^{-n}$-far from $S$. Formally (renaming the concept to avoid confusion),

**Definition 3** *[12] We say that a set $S$ is* weakly *computable if there is an oracle Turing Machine $M^\phi(n)$ such that if $\phi$ represents a real number $x$, then the output of $M^\phi(n)$ is*

$$M^\phi(n) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } B(x, 2^{-n}) \cap S = \emptyset \\ 0 \text{ or } 1 & \text{otherwise} \end{cases} \quad (3)$$

This definition is obviously weaker than the standard one, because we can output whatever answer we want if $x \notin S$ is very close to $S$. On the other hand, if we try to use a "weak computer" for $S$ to draw $S$ by running it on some grid, we might be left with an empty picture. This occurs especially in the cases where $S$ is a curve that contains none of the grid points.

We can prove that, despite the apparent weakness of the weak definition, it is equivalent to the standard bit computability. The construction uses the fact that the weak computation must terminate on every input $x$ regardless of the valid oracle $\phi$ for $x$. Due to space constraints, we only give a sketch of the proof here. The complete proof can be found in [11].

**Theorem 4** *A set is bit-computable if and only if it is weakly computable.*

**Proof Outline:** Obviously, the "hard" direction is to show that every weakly computable set is, in fact, bit-computable. To simplify matters, suppose that the set $S$ is a one-dimensional set. We assume that $S$ is weakly computable, and want to show that it is bit-computable.

We are given a point $d$ in $\mathbb{D}$, and $n > 0$, and want to return 1 if $(d - 2^{-n}, d + 2^{-n}) \cap S \neq \emptyset$ and 0 if $(d - 2 \cdot 2^{-n}, d + 2 \cdot 2^{-n}) \cap S = \emptyset$.

We consider the infinite tree $T$ of all the oracles for all the points in $(d - 2^{-n}, d + 2^{-n})$. On Fig. 3, the first three levels of $T$ are presented for $d = \frac{1}{2}$ and $n = 1$ (all the numbers are written in binary). Each infinite path in $T$ represents a real number in the interval we are interested in. For each real $x$ in the interval, there is a path converging to it. In fact, there is usually more than one such path.

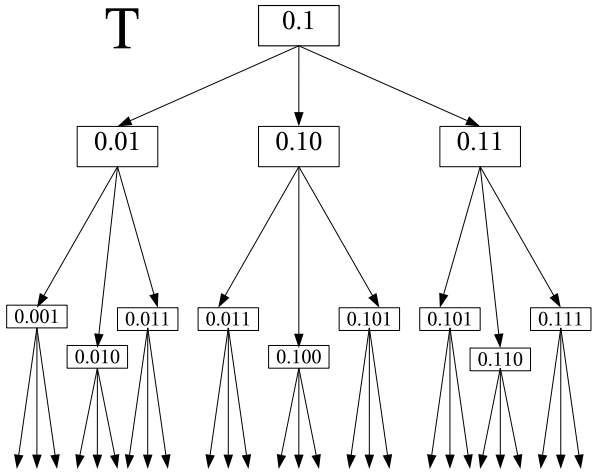


**Figure 3. The first three levels of the tree $T$**

We simulate the run of the weak machine $M^\phi(n)$. The goal is to make the simulation for all the real points in the interval $(d - 2^{-n}, d + 2^{-n})$ simultaneously.

If the machine asks for $x$ with precision $2^{-m}$, $m < n$, we respond with $d$ as the approximation. This is a valid oracle value for any $x$ in the interval.

If $m \geq n$ we consider all the possible descendants of $d$ on the level with $m + 1$-bit long numbers, and create a separate computation for each of them (thus creating $3^{m-n+1}$ computations). Consider one of the copies and denote the path leading to the selected vertex on level $m + 1$ by $p$. If we are now asked about $\phi(r)$ for some $r < m + 1$, we return the value of $p(r + 1)$ consistent with the current path. 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 either of the computations returns 1, we return 1 for the bit-computation; otherwise we return 0. We first show that the computation always terminates.

Suppose that the computation does not terminate. The entire computation can be viewed as a tree where the nodes

are the subcomputations described above and a computation $C_i$ 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 $C'$ fails to terminate without calling to subcomputations, or the tree of all the computations to be performed is an infinite tree.

In the first case the points represented by the oracle leading to $C'$ would cause $M^\phi(n)$ to run forever. 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 $\phi$ 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. Note that for the proof to work we need the fact that every Cauchy sequence in $\mathbb{D}$ converges to a limit in our domain $\mathbb{R}$.

For the correctness we see that if there is an $x \in S$ in the interval, then there is an oracle $\phi$ for $x$ which corresponds to an infinite path in $T$. The computation branch corresponding to this path will return 1, and the algorithm will output 1. If, on the other hand, $x$ is far from $S$, then every branch in the computation corresponds to a point which is at least $2^{-n}$-far from $S$, and all of them will return 0, and the algorithm outputs 0 in this case. ∎

A version of the following theorem has been first proved in [24], (see also [9]). Theorem 4 allows us to give a simple proof of it by showing that the graph $\Gamma_f$ is *weakly* computable. The simplified proof can be found in [11].

**Theorem 5** *Suppose $D$ is a computable closed and bounded set, and $f$ is continuous on $D$. Then $f$ is computable if and only if its graph $\Gamma_f = \{(x, f(x)) : x \in D\}$ is computable.*

# 3 The BSS Model and its Modifications

## 3.1 The Model

The BSS model is quite different from the bit model of computability. Like the bit model, it also extends the standard Turing machines to deal with the continuous reality. In the bit model the extension is through *application* of the standard machine to continuous problems using oracles and naming systems for continuous objects (cf. [22]). The BSS approach extends the *model* of computation itself. We present here an informal description of the model, which is equivalent to a formal one, but is simpler to comprehend for a reader who is new to the subject.

The BSS model in general is formulated for computation over an arbitrary ring or field $R$ (for our purposes one can take $R = \mathbb{R}$ or $R = \mathbb{C}$). The machines in this model are allowed to store an entire element of $R$ in one memory cell. The operations the machine is allowed to perform on numbers are **(i)** the ring operations ($+$, $-$, $\cdot$, and $\div$ if $R$ is a field); and **(ii)** branching conditioned on exact comparisons between numbers ($=$ and $<$, $\leq$, if $R$ is ordered). Initially, the program is allowed to have only some finite number of constants from $R$. A machine *computes* a function $f : R^k \to R^\ell$ on a domain $D \subset R^k$, if on an input $x \in D$, it outputs $f(x) \in R^\ell$. A machine *decides* a set $S \subset R^n$ if it computes the characteristic function $\chi_S(x)$ on $\mathbb{R}^n$. If one takes $R = \mathbb{Z}_2$ (or any other finite ring), the BSS model becomes equivalent to the standard (discrete) Turing Machine.

The BSS machines are closely related to the corresponding arithmetic circuits over $R$, just as ordinary Turing Machines correspond to boolean circuits.

## 3.2 Examples of BSS Computable and Uncomputable Sets

A big family of examples of BSS computable sets are the semi-algebraic sets. In particular, any singleton $S = \{c\}$, any line segment and any ball in $\mathbb{R}^n$ is BSS computable. Note that $\{c\}$ as well as the constant function $f(x) = c$ are computable, *regardless* of whether the number $c$ can be actually approximated or not. Unlike the bit model, in the BSS model the step function $s_0(x)$ is easily computable in constant time.

On the other hand, as we have mentioned in the introduction, simple functions and sets that lack the specific algebraic structure are not BSS computable. Examples of those include the functions $\sqrt{x}$, $\sin x$ and $\log x$, and sets such as Koch's snowflake $K$ and the graph of the exponential function $e^x$ (cf. [8]). This is caused by the advantage given to the algebraic operations ($+$, $-$, $\cdot$ and $\div$) in the BSS model.

We will now present a somewhat more subtle example of a BSS computable set which is not bit-computable. It will be useful later in the discussion. First, let $R(x, y)$ be a computable (binary) predicate such that the predicate $H(x) = \exists y\ R(x, y)$ is uncomputable and if $H(x)$ holds, then the $y$ satisfying $R(x, y)$ is unique. One can take $x$ to be the encoding of a Turing machine, and $R(x, y)$ = "the machine encoded by $x$ halts after exactly $y$ steps". Then $R(x, y)$ is computable by a simple simulation, while $H(x)$ is the halting problem, which is undecidable.

We construct the following closed set $C_0 \subset [0,1] \times [0,1]$. Denote $I_i = [\frac{1}{i+1}, \frac{1}{i}]$ for $i = 1, 2, \ldots$. Then $[0,1] =$

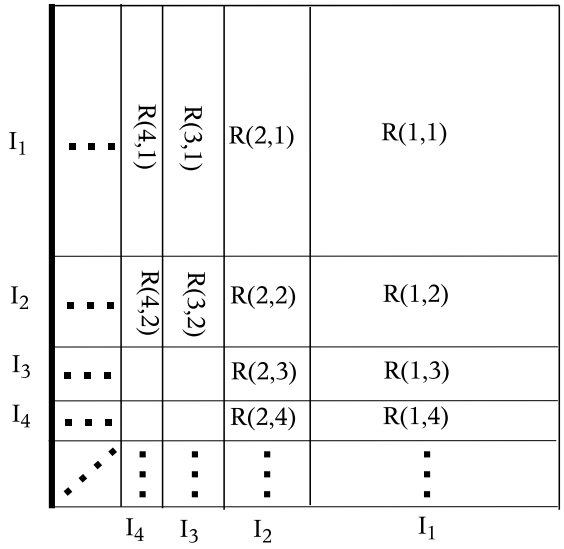


**Figure 4. A BSS computable set $C_0$ one cannot draw.**

$\cup_{i=1}^{\infty} I_i \cup \{0\}$. Define

$$C_0 = (\{0\} \times [0,1]) \bigcup \left( \bigcup_{R(i,j)=1} I_i \times I_j \right).$$

It is not hard to see that $C_0$ is closed. There are no accumulation points on $(0,1] \times \{0\}$ because for each value of $i$ there is at most one value of $j$ such that $R(i,j) = 1$. See Fig. 4 for a schematic construction of $C_0$.

$C_0$ is BSS computable. We first check whether a given point $(x,y)$ is on one of the axes. If it isn't, we can localize the rectangle $I_{i,j}$ in which $(x,y)$ lies, and output $R(i,j)$.

Observe that one cannot draw a good image of the set $C_0$. In fact, in order to decide whether to put a pixel in a small neighborhood of the point $\left(\frac{2i+1}{i(i+1)}, 0\right)$, the middle of the interval $I_i$ on the $x$-axis, one needs to essentially compute $H(i)$, which is impossible.

## 3.3 Possible Modifications to the BSS Model

In this section we discuss possible modifications to the BSS models which address some issues raised in the previous section. Modifications to the model have been proposed in [6, 7] leading to a notion of "feasible real RAM" which is essentially equivalent to the bit-computability. The main idea there was, that exact comparisons are not possible on real-life devices, and should not be permitted in the model. We argue that this is not always the case. From the practical standpoint, some physical and other natural systems are best described with discontinuous functions (e.g. "switch on/off"). From the theoretical point of view, this restriction bars us from classifying discontinuous and multi-valued functions. The simple step function, and a function solving the halting problem fall into the same "uncomputable" category.

### 3.3.1 Uncomputable Constants.

The first concern to address is the use of uncomputable numbers. It is unreasonable to say that a function $f(x) = a$, where $a$ is a constant encoding the halting problem is computable. The simple solution is to restrict the BSS machines to use only computable constants.

Recall that the computable numbers, which we denote by $\mathcal{C}$, are the numbers that can be approximated arbitrarily well on a computer. $\mathcal{C}$ is a countable real closed field, and that $\mathcal{C} + i\mathcal{C} \subset \mathbb{C}$ is algebraically closed. Thus, it makes sense to discuss BSS machines over $\mathcal{C}$ rather than on the entire $\mathbb{R}$. To emphasize the field $\mathcal{C}$ we are working with, we will denote this model by $BSS_{\mathcal{C}}$. It now follows easily that simple geometric objects, such as singletons, line segments and balls are $BSS_{\mathcal{C}}$-computable if and only if they are bit-computable.

### 3.3.2 Computation Errors

. The next possible modification addresses the problem of the BSS uncomputability of functions such as $e^x$. The BSS model is in part based on the fact that real-life computers usually use the four arithmetic operations as a base to performing real computations. $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$ can be viewed as an infinite-degree polynomial, and is *approximated* arbitrarily well with the finite degree polynomials $p_n(x) = \sum_{i=0}^{n} \frac{x^i}{i!}$. In fact, real-life programs never compute $e^x$, but only $p_n(x)$ with some suitably chosen $n$. This can be done using only the four arithmetic operations.

We further modify $BSS_{\mathcal{C}}$ by allowing the machines to err within a given precision $\varepsilon$. We denote this model by $BSS_{\mathcal{C}}^{\varepsilon}$. In this model, a BSS machine $M(x,\varepsilon)$ is said to compute $f(x)$, if on an input $(x,\varepsilon)$, $\varepsilon > 0$, it outputs $f(x)$ with an error of at most $\varepsilon$, and using only computable constants. Note that the simple step function $s_0(x)$, as well as any other $BSS_{\mathcal{C}}$ computable functions, is computable in $BSS_{\mathcal{C}}^{\varepsilon}$.

One can now define the $BSS_{\mathcal{C}}^{\varepsilon}$ computability of sets in a natural way. A bounded set $S$ is $BSS_{\mathcal{C}}^{\varepsilon}$-computable, if there is a BSS machine $M(x,\varepsilon)$ which uses only computable constants, and on input $(x,\varepsilon)$ it outputs 1 if $x \in S$ and 0 if $d(x,S) > \varepsilon$. With this definition, the graph of $e^x$ becomes computable. The sets $K$ and $J$ mentioned in the introduction also become computable.

It should be noted that if we drop the requirement of using only computable numbers, all the bounded sets are easily seen to be computable. It is not hard to encode all the

"pictures" of any bounded set $S$ into one (possibly uncomputable) number $c$. In other words, all the sets are computable in $BSS^{\varepsilon}$.

#### 3.3.3 Unbounded Computation Branches.

The third modification addresses the problem highlighted by the example $C_0$ on Fig. 4. The problem is the excessive power the BSS model gets from the possibility of having arbitrarily long computation paths (as it happened in the example above).

In the case of "simple" computations, such as $e^x$, or its graph, we can easily estimate the number of steps the machine would have to perform as a function of $\varepsilon$. We include this condition as an additional restriction on the $BSS^{\varepsilon}_{\mathcal{C}}$ machines.

We say that a function (or a set) is $BSS^{\varepsilon,b}_{\mathcal{C}}$ computable, if it is $BSS^{\varepsilon}_{\mathcal{C}}$ computable by a machine $M$, and the running time of $M$ can be bounded by $\tau\left(\left\lfloor\frac{1}{\varepsilon}\right\rfloor\right)$ for some integer computable function $\tau : \mathbb{N} \to \mathbb{N}$. We say that $\tau(2^n)$ is the time complexity of the set.

Under this restriction the set $C_0$ on Fig. 4 is not computable, while the function $e^x$ on $[0, 1]$, the graph of $e^x$ and the step function $s_0(x)$ are computable. Other "calculator" functions, such as $\sqrt{x}$, $\sin x$ and $\log x$ also become computable. As before, if we allow the use of uncomputable constants, all the sets become computable.

The restriction guarantees that a computation with any finite precision has a branching tree of a computable finite size. Note that the machine computing $C_0$ has an infinite branching tree regardless of the precision: the time it takes to localize $x > 0$ into an interval $[1/(n + 1), 1/n]$ grows with $n$, regardless of the precision we are computing with.

We summarize the modifications to the BSS model in the following diagram:

| | | | | |
|---|---|---|---|---|
| $BSS_{\mathcal{C}}$ | $\subset$ | $BSS^{\varepsilon}_{\mathcal{C}}$ | $\supset$ | $BSS^{\varepsilon,b}_{\mathcal{C}}$ |
| ($C_0$ comp., $e^x$ not comp.) | | ($C_0$ comp., $e^x$ comp.) | | ($C_0$ not comp., $e^x$ comp.) |
| $\cap$ | | $\cap$ | | $\cap$ |
| $BSS$ | $\subset$ | $BSS^{\varepsilon}$ | $\supset$ | $BSS^{\varepsilon,b}$ |
| ($C_0$ comp., $e^x$ not comp.) | | (everything is comp.) | | (everything is comp.) |

We will now show that $BSS^{\varepsilon,b}_{\mathcal{C}}$-computability is equivalent to bit computability for bounded *sets*. Note that they are still different for *functions*, since the step function $s_0(x)$ is $BSS^{\varepsilon,b}_{\mathcal{C}}$-computable, but not bit-computable. In Section 4 we will connect $BSS^{\varepsilon,b}_{\mathcal{C}}$ function computability to bit computability.

### 3.4 Computability of sets in $BSS^{\varepsilon,b}_{\mathcal{C}}$

We prove the following theorem. Due to space constraints we only outline most of the proof here, leaving some details out. The complete proof can be found in [11].

**Theorem 6** *Let $S \subset \mathbb{R}^k$ be a bounded set. Then $S$ is $BSS^{\varepsilon,b}_{\mathcal{C}}$-computable if and only if it is bit-computable.*

**Proof Outline:** *$S$ is bit-computable $\Rightarrow$ $S$ is $BSS^{\varepsilon,b}_{\mathcal{C}}$-computable.* This is the easier direction. A BSS machine over any $R$ is more general than a regular Turing Machine, and it can simulate the bit machine computing $S$.

*$S$ is $BSS^{\varepsilon,b}_{\mathcal{C}}$-computable $\Rightarrow$ $S$ is bit-computable.* This is a more involved direction. The reduction we will give is not uniform in $S$. It cannot be uniform due to the fact that $BSS^{\varepsilon,b}_{\mathcal{C}}$ uses arbitrary computable constants. Even the simplest questions, such as "$a = b$?", are not decidable for arbitrary computable reals $a$ and $b$ presented by Turing machines computing them. Denote the $BSS^{\varepsilon,b}_{\mathcal{C}}$ machine computing $S$ by $M(x, \varepsilon)$.

**The nonuniform information needed.** Suppose that the BSS machine $M$ uses $\ell$ constants $a_1, a_2, \ldots, a_\ell \in \mathbb{R}$. We would need the following algebraic information about $a_1, \ldots, a_\ell$, in addition to the Turing machines approximating them:

1. The algebraic degree $d_i$ of $a_i$ over the field $\mathbb{Q}(a_1, \ldots, a_{i-1})$, and
2. if this algebraic degree is finite (i.e. $a_i$ is algebraic over $\mathbb{Q}(a_1, \ldots, a_{i-1})$), the minimal polynomial $p_i(x) \in \mathbb{Q}(a_1, \ldots, a_{i-1})[x]$ of degree $d_i$ with leading coefficient 1, such that $p_i(a_i) = 0$. $p_i$ is presented symbolically, with non leading coefficients given as rational functions with non-zero denominators.

The next two lemmas show that the nonuniform information above suffices to decide any polynomial relation.

**Lemma 7** *Provided the nonuniform information as above, for any symbolic polynomial $p(x_1, x_2, \ldots, x_\ell) \in \mathbb{Q}[x_1, x_2, \ldots, x_\ell]$ we can check whether $p(a_1, a_2, \ldots, a_\ell) = 0$.*

**Proof Outline:** By induction on $\ell$. If $a_\ell$ is transcendental over $\mathbb{Q}(a_1, a_2, \ldots, a_{\ell-1})$, the equation becomes a set of $deg_{a_\ell} p$ equations in $a_1$, $a_2$, ..., $a_{\ell-1}$, which we solve by the induction hypothesis.

If $a_\ell$ is algebraic over $\mathbb{Q}(a_1, a_2, \ldots, a_{\ell-1})$, we use the non-uniform information to reduce the degree of $p$ in $a_\ell$ below $d_\ell$. Then the equation becomes a set of at most $d_\ell$ equations in $a_1$, $a_2$, ..., $a_{\ell-1}$. ■

**Lemma 8** *Provided the nonuniform information as above, for any symbolic polynomial $p(x_1, \ldots, x_\ell) \in \mathbb{Q}[x_1, \ldots, x_\ell]$ we can check whether $p(a_1, a_2, \ldots, a_\ell) > 0$.*

**Proof:** By Lemma 7 we can first check whether $p(a_1, a_2, \ldots, a_\ell) = 0$. If yes, we output 'no'. Otherwise, using increasingly good approximations, we will eventually be able to tell whether $p(a_1, a_2, \ldots, a_\ell) > 0$ or $p(a_1, a_2, \ldots, a_\ell) < 0$. ∎

Given a dyadic $d \in \mathbb{D}^k$ and $n \in \mathbb{N}$, we would like to compute $f(d, n)$ as in (2). In other words, we would like to check whether $d$ is $2^{-n}$-close or $2 \cdot 2^{-n}$-far from $S$.

For the rest of the proof set $\varepsilon = 2^{-n}$. We know there is a computable bound on the running time of $M(x, \varepsilon)$ in terms of $\varepsilon$. We compute this bound $B = B(\varepsilon)$. This means that $M(\bullet, \varepsilon)$ can have at most $2^B$ different computation paths. Each potential path has an output (either 0 or 1), and a set of rational constraints on the input $x = (x_1, \ldots, x_k)$ and the constants $a_1, \ldots, a_\ell$ that ensure that this path is followed. If the constraints are not satisfiable by any $(x_1, \ldots, x_k)$, it means that the path is never actually followed. The rational constraints can be rewritten as polynomial ones.

Choose some computation path $\gamma$ on which $M(\bullet, \varepsilon)$ outputs 1. We denote the polynomial constraints to be satisfied in order to follow $\gamma$ by $C_\gamma(x_1, \ldots, x_k, a_1, \ldots, a_\ell)$. We are interested whether there is an $x \in B(d, 2^{-n}) \cap S$. In particular, we would like to know whether there is such an $x$ that is accepted by the path $\gamma$. This is stated by the following quantified formula

$$f_\gamma(a_1, \ldots, a_\ell) = \exists x_1, \ldots, x_k \ \ ((x_1 - d_1)^2 + \ldots$$

$$+(x_k - d_k)^2 < 2^{-2n}) \wedge C_\gamma(x_1, \ldots, x_k, a_1, \ldots, a_\ell).$$

Using a quantifier elimination algorithm, we can convert $f_\gamma(a_1, \ldots, a_\ell)$ into a quantifier-free formula $g_\gamma(a_1, \ldots, a_\ell)$ which has the same truth value. We then can use Lemmas 7 and 8 to decide whether $g_\gamma(a_1, \ldots, a_\ell)$ is true or false (which is the same as deciding $f_\gamma(a_1, \ldots, a_\ell)$). $f_\gamma$ has only some constant number of existential quantifiers with no alternations, hence the complexity of the quantifier elimination can be reduced to be exponential in $k$ (and polynomial in the other parameters). See [2] and [19] for the algorithms and their analysis.

As an answer, we output the following

$$f(d, n) = \bigvee_{\gamma \text{ is a 1-valued path of } M(\bullet, \varepsilon)} f_\gamma(a_1, \ldots, a_\ell). \tag{4}$$

As there are at most $2^B$ such paths $\gamma$, the computation will involve computing $f_\gamma$ at most $2^B$ times.

Now, if there is an $x$ in $B(d, 2^{-n}) \cap S$, it will satisfy $f_{\gamma_x}$ in (4) for the corresponding path $\gamma_x$ of $M(x, \varepsilon)$. If there is no $x$ in $B(d, 2 \cdot 2^{-n}) \cap S$, (4) can never be satisfied. ∎

*Remark:* From the complexity point of view, even the simplest simulation of BSS machines using Turing machines appears to be quite costly. In the case where the machine is an arithmetic straight line program a discussion on the complexity of such a simulation can be found in [1].

# 4 Complexity of Real Functions

In this section we propose a new definition for the computability and complexity of real functions, and establish its connections with bit and BSS computability.

## 4.1 Computability of Real Functions

The main idea arises from the equivalence between the function computability and the set computability in case of a continuous function, which was established in Theorem 5. We have a good notion of bit-computability for sets which coincides with $BSS_{\mathcal{C}}^{\varepsilon,b}$-computability. We use it to define a computability notion for functions:

**Definition 9** *We say that a bounded real function $f$ on a bounded domain $D$ is* graph computable*, if its graph $\Gamma_f = \{(x, f(x)) : x \in D\}$ is computable as a set.*

By Theorem 5, Definition 9 coincides with the bit computability definition in the case of continuous functions on closed domains. In some sense, graph computability extending bit-computability for functions, is similar to the notion of Lebesgue integral extending the Riemann integral.

$BSS_{\mathcal{C}}^{\varepsilon,b}$-computable functions on "nice" domains have $BSS_{\mathcal{C}}^{\varepsilon,b}$-computable graphs. Here a "nice" domain is bounded semi-algebraic and $BSS_{\mathcal{C}}$-computable, e.g. $D = [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_k, b_k]$ with computable endpoints. It follows from Theorem 6 that these functions are graph computable. So graph-computability extends the modified BSS function computability on "nice" domains.

## 4.2 Complexity of Real Functions

By now we have established that graph computability is a useful notion extending both bit computability and a natural modified version of the BSS computability. Our next goal is to give a reasonable definition for *graph complexity* of real functions. Should we take the complexity of computing $\Gamma_f$? Or the complexity of weakly computing it? It turns out that neither one extends the bit-complexity of $f$ in the continuous case. In fact, we have the following theorem.

**Theorem 10** *Let $f : [0, 1]^k \to [0, 1]^\ell$ be a continuous function, with a polynomial modulus of continuity $\mu(n)$ ($|f(x) - f(y)| < 2^{-n}$ whenever $|x - y| < 2^{-\mu(n)}$). Denote the following properties:*

$(a)$ *The graph $\Gamma_f$ is poly-time computable as a set.*

$(b)$ *$f$ is a poly-time computable function.*

$(c)$ *The graph $\Gamma_f$ is weakly poly-time computable as a set.*

*Then we have the following:* **(i)** $(a) \Rightarrow (c)$ *and* $(b) \Rightarrow (c)$*;* **(ii)** $(b) \Rightarrow (a)$ *implies $P = NP$; and* **(iii)** $(a) \Rightarrow (b)$ *(and also $(c) \Rightarrow (b)$), implies that integer factoring and other one-way functions can be done in polynomial time.*

We omit the proof of Theorem 10 here. It can be found in [11]. Intuitively, when we compute $y = f(x)$ according to $(b)$, we have access to an oracle for $x$, and need to output an approximation of $y$. This corresponds to a computation of $\Gamma_f$ which is weak in $x$ and strong in $y$. $(a)$ is stronger because it requires $\Gamma_f$ to be strongly computable in both coordinates, and $(c)$ is weaker because it allows $\Gamma_f$ to be weakly computable in both coordinates. It is somewhat surprising that for some technical reasons $(a)$ *does not* imply $(b)$ in general.

What we need is a complexity notion that extends part $(b)$ in Theorem 10 to all graph-computable functions, i.e. one that is weak in $x$ and strong in $y$.

Recalling Definition 3 of weak computability we see that $x$ has to be given to the machine by an oracle. The computation should be strong in $y$, so we should be able to strongly compute a vertical cross-section $A_m$ of the graph $\Gamma_f$. By analogy with Definition 3, we must **(i)** always include $f(x)$ in $A_m$, and **(ii)** $A_m$ must only contain images of points in $B(x, 2^{-m})$. Thus (viewing $f(x)$ as a set which may contain more than one point), we have

$$f(x) \subset A_m \subset f\left(B(x, 2^{-m})\right). \tag{5}$$

The computation should be strong in $y$. Thus we should compute the set $A_m$ in the sense of Definition 2. Given a dyadic point $d$ in the range space and a precision parameter $n$, we must output an answer consistent with (5). If $d$ is $2^{-n}$-close to some point in $f(x)$, we must output 1. If $d$ is $2 \cdot 2^{-n}$-far from *all* points in $f(B(x, 2^{-m}))$, we must output 0. Otherwise, we are in the gray area.

Formally, we arrive at the following definition.

**Definition 11** *We say that a bounded multi-valued function $f : D \subset \mathbb{R}^k \to \mathbb{R}^\ell$ for some bounded computable $D$ is* **graph-computable in time** *$T(n, m)$ if there is an oracle Turing machine $M^\phi(m, d, n)$ which, given an oracle $\phi$ for $x \in D$, computes a function from the family*

$$M^\phi(m, d, n) = \tag{6}$$

$$\begin{cases} 1, & \text{if } |d - y| < 2^{-n} \text{ for some } y \in f(x) \\ 0, & \text{if } |d - y| \geq 2 \cdot 2^{-n} \text{ for all } y \in f\left(B(x, 2^{-m})\right) \\ 0 \text{ or } 1 & \text{otherwise} \end{cases}$$

*and the computation time is bounded by $T(m, n)$.*

**Examples:** The step function $s_0$ defined in (1) is now computable in linear time. For an input $x$ output $A_m = \{0\}$ if $x < -2^{-(m+1)}$, output $A_m = \{1\}$ if $x > 2^{-(m+1)}$, and $A_m = \{0, 1\}$ if $-2^{-m} < x < 2^{-m}$. Note that the overlaps between the different possibilities ensure that we can always follow at least one of them.

More generally, consider the complexity of $\chi_A$ for some set $A$. On an input $x$ we need to output one of the three possible sets: $\{0\}$, $\{1\}$ or $\{0, 1\}$. We must include 1 if $x \in A$, and must exclude it if $x$ is $2^{-m}$-far from $A$. Similarly, we must include 0 if $x \notin A$, and must exclude it if $x$ is $2^{-m}$-far from $A^c$. We see that the complexity of $\chi_A$ in this case is roughly equal to the sum of the *weak* complexities of $A$ and $A^c$.

The two-valued function $\sqrt{\ } : \mathbb{C} \to \mathbb{C}$ is computable efficiently. On an input $x$, $A_m$ is a $2^{-m}$ approximation of $\{\sqrt{x}, -\sqrt{x}\}$. Note that this set converges to the set $\{0\}$ as $x \to 0$. Reasonable branches of $\sqrt{\ }$, such as the standard one ($r \cdot e^{i\theta} \mapsto \sqrt{r} \cdot e^{i\theta/2}$, $0 \leq \theta < 2\pi$), is also efficiently computable under Definition 11.

To be considered the "right" complexity notion, Definition 11 has to be consistent with the previous definitions of computability and complexity. First of all, a function is computable according to it if and only if it is graph computable. The proof uses Theorem 4 on the equivalence of weak and regular bit-computability. It can be found in [11].

**Theorem 12** *A function $f : D \subset \mathbb{R}^k \to \mathbb{R}^\ell$ for some closed and bounded computable $D$ is computable as per Definition 11 if and only if its graph is computable (that is, it is computable as per Definition 9).*

Next, we see that this definition extends standard function complexity in the continuous case. In particular, if $f$ has a reasonably small modulus of continuity, it is bit-poly-time computable if and only if it is graph-poly-time computable.

**Theorem 13** *Let $f : D \to \mathbb{R}^k$ be a continuous function, where $D \subset \mathbb{R}^\ell$ is bounded. Then the following holds:*

1. *If $f$ is computable in time $T(n)$ according to the standard bit complexity definition, then it is graph-computable in time $T(n + 2) + O(n)$.*
2. *If $f$ is graph-computable in time $S(m, n)$, and the modulus of continuity for $f$ is a computable $\mu = \mu(n)$ (so that $|f(x) - f(y)| < 2^{-n}$ whenever $|x - y| < 2^{-\mu(n)}$), then $f$ is computable in time $O(n \cdot S(\mu(n + 2), n + 2))$ according to the standard bit complexity definition.*

The harder part here is the second claim. It is true, because for $m = \mu(n + 2)$,

$$f(x) \in A_m \subset f(B(x, 2^{-\mu(n+2)})) \subset B(f(x), 2^{-(n+2)}).$$

Thus by approximating the set $A_m$, we can approximate $f(x)$.

Theorems 12 and 13 show that graph complexity is the natural complexity notion for graph computable functions. It also extends the natural complexity definitions for other extensions of the continuous functions. For example, the piecewise continuous functions with finitely many computable discontinuity points.

## 5 Closing Remarks

In Section 3 it has been shown that under some quite natural modifications, the BSS model and the bit model are equivalent for *sets*. While these modifications make sense from the point of view of actual computations, some of the merits of the BSS model are inevitably lost.

There are interesting complexity questions that can only be formulated in the original model. For example, the question of $P_{\mathbb{R}}$ vs. $NP_{\mathbb{R}}$, and other questions regarding complexity classes over $\mathbb{R}$ and $\mathbb{C}$. Another example is the question whether linear programming can be solved in polynomial time using only exact arithmetic and branching on $<$ and $=$. Nonetheless, as we have seen above, modifications are necessary when one considers the practical applications of the model to computability.

In the *functions* case matters are more complicated. The traditional recursive analysis approach seems to work quite well in the case when the underlying functions are continuous. In Section 4 the standard notion of function complexity has been generalized to a richer class which includes some discontinuous functions. This class might be even too rich in some cases. The exact class of functions $\mathcal{F}$ to which the notion should be applied depends on the specific applications of this notion.

Suggestions for a "second-generation BSS machine" included incorporating errors and condition numbers into the model [20]. It is possible that with the modifications both approaches would converge to a common notion of real-function computability, and maybe even complexity.

# Acknowledgments

I would like to thank my graduate supervisor, prof. Stephen Cook, for his insights and support during the preparation of this paper and for the many hours we spent discussing Real Computation.

I would like to thank prof. Toniann Pitassi, for her advice during the preparation of this paper.

## References

[1] E. Allender, P. Bürgisser, J. Kjeldgaard-Pedersen, and P. B. Miltersen. On the complexity of numerical analysis. ECCC report TR05-037, 2005.

[2] S. Basu, R. Pollack, and M. F. Roy. *Algorithms in Real Algebraic Geometry*. Springer-Verlag, 2003.

[3] I. Binder, M. Braverman, and M. Yampolsky. Filled julia sets with empty interior are computable. e-print, Oct. 2004. math.DS/0410580.

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

[5] L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bulletin of the Amer. Math. Soc.*, 21:1–46, 1989.

[6] P. Boldi and S. Vigna. $\delta$-uniform BSS machines. *J. of Complexity*, 14:234–256, 1998.

[7] V. Brattka. Feasible real random access machines. *J. of Complexity*, 14:490–526, 1998.

[8] V. Brattka. The emperor's new recursiveness: The epigraph of the exponential function in two models of computability. In *Masami Ito and Teruo Imaoka, editors, Words, Languages & Combinatorics III*, pages 63–72, 2003.

[9] V. Brattka. Plottable real number functions. In *Marc Daumas and et al., editors, RNC'5 Real Numbers and Computers*, pages 13–30. INRIA, September 2003.

[10] V. Brattka and K. Weihrauch. Computability of subsets of euclidean space I: Closed and compact subsets. *Theoretical Computer Science*, 219:65–93, 1999.

[11] M. Braverman. On the complexity of real functions. e-print, Feb. 2005. cs.CC/0502066.

[12] A. Chou and K. Ko. Computational complexity of two-dimensional regions. *SIAM J. Comput.*, 24:923–947, 1995.

[13] A. Grzegorczyk. Computable functionals. *Fund. Math.*, 42:168–202, 1955.

[14] P. Hertling. Is the Mandelbrot set computable? *Math. Log. Quart.*, 51(1):5–18, 2005.

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

[16] D. Lacombe. Classes récursivement fermés et fonctions majorantes. *C. R. Acad. Sci. Paris*, 240:716–718, 1955.

[17] D. Lacombe. Les ensembles récursivement ouverts ou fermés, et leurs applications à l'analyse récursive. *C. R. Acad. Sci. Paris*, 246:28–31, 1958.

[18] M. B. Pour-El and R. J. I. *Computability in Analysis and Physics*. Springer-Verlag, Berlin, 1989.

[19] J. Renegar. On the computational complexity and geometry of the first order theory of the reals. *J. of Symb. Comp.*, 13:255–352, 1992.

[20] S. Smale. Complexity theory and numerical analysis. *Acta Numerica*, 6:523–551, 1997.

[21] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings, London Mathematical Society*, pages 230–265, 1936.

[22] K. Weihrauch. *Computable Analysis*. Springer-Verlag, Berlin, 2000.

[23] A. C. Yao. Classical physics and the church-turing thesis. ECCC report TR02-062, 2002.

[24] Q. Zhou. Computable real-valued functions on recursive open and closed subspaces of $\mathbb{R}^q$. *Math. Log. Quart.*, 42:379–409, 1996.