Polynomial differential equations compute all real computable functions

Olivier Bournez\textsuperscript{a,b,*}, Manuel L. Campagnolo\textsuperscript{c,d},
Daniel S. Graça\textsuperscript{e,d}, Emmanuel Hainry\textsuperscript{f,b}

\textsuperscript{a}Inria Lorraine, France
\textsuperscript{b}LORIA (UMR 7503 CNRS-INPL-INRIA-Nancy2-UHP), Campus scientifique,
BP 239, 54506 Vandœuvre-Lès-Nancy, France
\textsuperscript{c}DM/ISA, Universidade Técnica de Lisboa, 1349-017 Lisboa, Portugal
\textsuperscript{d}CLC, DM/IST, Universidade Técnica de Lisboa, 1049-001 Lisboa, Portugal
\textsuperscript{e}DM/FCT, Universidade do Algarve, C. Gambelas, 8005-139 Faro, Portugal
\textsuperscript{f}Institut National Polytechnique de Lorraine, France

Abstract

In the last decade, the field of analog computation has experienced renewed interest. In particular, there have been several attempts to understand which relations exist between the many models of analog computation. Unfortunately, most models are not equivalent.

It is known that Euler’s Gamma function is computable according to computable analysis, while it cannot be generated by Shannon’s General Purpose Analog Computer (GPAC). This example has often been used in order to argue that the GPAC is less powerful than digital computation.

However, as we will demonstrate, when computability with GPACs is considered in the framework of recursive analysis, we obtain two equivalent models of analog computation.

Using this approach, it has been shown recently that the Gamma function becomes computable by a GPAC. Here we extend this result by showing that, in an appropriate framework, the GPAC and computable analysis are actually equivalent from the computability point of view. Since GPACs are equivalent to systems of polynomial differential equations then we show that all real computable functions can be defined by such models.

Preprint submitted to Elsevier Science 7 March 2006
1 Introduction

According to the Church-Turing thesis, all “reasonable models” of digital computation, based on the intuitive notion of algorithm, are computationally equivalent to the Turing machine.

No similar result is known when considering analog computation. While many analog models have been studied including the BSS model [1], Moore’s $\mathbb{R}$-recursive functions [2], neural networks [3], or computable analysis [4–6], but none was able to affirm itself as “universal”. In part, this is due to the fact that few relations between them are known. Moreover some of the known results assert that these models are not equivalent, making the idea of a Church-Turing thesis for analog models an apparently unreachable goal. For example the BSS model allows discontinuous functions while only continuous functions can be computed in the framework of computable analysis [6].

Here, we will show that this goal may not be as far as those results suggest. Indeed, we will prove the equivalence of two models of analog computation that were previously considered nonequivalent: computable analysis and Shannon’s General Purpose Analog Computer (GPAC).

The GPAC was introduced in 1941 by Shannon [7] as a mathematical model of an analog device: the Differential Analyzer [8]. The Differential Analyzer was used from the 1930s to the early 60s to solve numerical problems. For example, differential equations were used to solve ballistics problems. These devices were first built with mechanical components and later evolved to electronic versions. A GPAC may be seen as a circuit built of interconnected black boxes, whose behavior is given by Figure 1, where inputs are functions of an independent variable called the time (in an electronic Differential Analyzer, inputs usually correspond to electronic voltages). These black boxes add or multiply two inputs, generate a constant, or solve a particular kind of Initial Value Problem defined with Ordinary Differential Equations (ODE for short).

While many of the usual real functions are known to be generated by a GPAC, a notable exception is the Gamma function $\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt$ [7]. If we have in mind that this function is known to be computable under the computable analysis framework [4], the previous result has long be interpreted as evidence

* Expanded version of an article to be published in the proceedings of conference Theory and Applications of Models of Computation (TAMC06), Lecture Notes in Computer Science.
* Corresponding author.

Email addresses: Olivier.Bournez@loria.fr (Olivier Bournez), mlc@math.isa.utl.pt (Manuel L. Campagnolo), dgraca@ualg.pt (Daniel S. Graça), Emmanuel.Hainry@loria.fr (Emmanuel Hainry).
Fig. 1. Different types of units used in a GPAC.

that the GPAC is a somewhat weaker model than computable analysis.

However, we believe that this limitation is due to the notion of GPAC-computability rather than the model itself.

The GPAC usually computes in “real time” - a very restrictive form of computation. But if we change this notion of computability to the kind of “converging computation” used in recursive analysis, then it has been shown recently that the $\Gamma$ function becomes computable [9]. Notice that this “converging computation” with GPACs corresponds to a particular class of $\mathbb{R}$-recursive functions [2,10,11]. As in [11] we only consider a Turing-computable subclass of $\mathbb{R}$-recursive functions, but here we restrict our focus to functions that can be defined as limits of solutions of polynomial differential equations.

In the present paper, we further strengthen this result and show that actually every computable function can be computed by a GPAC in the above sense. Reciprocally, we show that under some reasonable hypothesis, the converse is also true.

In other words, we prove that (non-real time) GPAC computability coincides with computability according to recursive analysis.

It is worth noting that it was shown in [12] that Turing machines can be simulated by GPACs. Since real computable functions are those computed by function-oracle Turing machines [5], this paper also shows that the result in [12] can be extended to such models.

The outline of this paper is as follows. In Section 2 we describe the GPAC and we recall that GPACs are equivalent to systems of polynomial ordinary differential equations. The constructions in the remainder of the paper will rely on such systems, which are explicitly continuous-time in nature. Then, we define a notion of “converging computation”, which we call GPAC-computability in opposition to the original “real-time” notion of GPAC-generability. We also recall the definition of computable real functions according to Computable

1 If not otherwise stated, the expression “computable function” is interpreted in the computable analysis sense.
Analysis. To conclude the preliminaries, we review how Turing machines can be simulated with ODEs. In Section 3 we prove the equivalence between computable real functions and $\theta_j$-GPAC-computable functions, which are computable by polynomial ODEs which can access a basic $C^{j-1}$ but non analytic function called $\theta_j$. Toward this end, we will have to show how to simulate function-oracle Turing machines with such systems. Finally, we will show in Section 4 how to remove $\theta_j$ from our constructions to obtain the main result of the paper.

2 Preliminaries

2.1 The GPAC

The GPAC was originally introduced by Shannon in [7], and further refined in [13–15,9]. The model basically consists of families of circuits built with the basic units presented in Figure 1, not all kinds of interconnections are allowed since this may lead to undesirable behavior (e.g. non-unique outputs. For further details, refer to [15]).

Shannon, in his original paper, already mentions that the GPAC generates polynomials, the exponential function, the usual trigonometric functions, their inverses. More generally, Shannon claims that all functions generated by a GPAC are differentially algebraic, i.e. they satisfy the condition of the following definition:

**Definition 1** The unary function $y$ is differentially algebraic (d.a.) on the interval $I$ if there exists a nonzero polynomial $p$ with real coefficients such that

$$p(t, y, y', \ldots, y^{(n)}) = 0, \quad \text{on } I. \quad (1)$$

As a corollary, and noting that the Gamma function $\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt$ is not d.a. [16], we get that

**Proposition 2** The Gamma function cannot be generated by a GPAC.

However, Shannon’s proof relating functions generated by GPACs with d.a. functions was incomplete (as pointed out and partially corrected in [13,14]). Actually, as pointed out in [15], the original GPAC model suffers from several robustness problems. However, for the more robust class of GPACs defined in [15], the following stronger property holds:

**Proposition 3** A scalar function $f : \mathbb{R} \to \mathbb{R}$ is generated by a GPAC iff it is
Fig. 2. Generating cos and sin via a GPAC: circuit version on the left and ODE version on the right. One has $y_1 = \cos$, $y_2 = \sin$, $y_3 = -\sin$.

A component of the solution of a system

$$y' = p(y, t), \quad (2)$$

where $p$ is a vector of polynomials. A function $f : \mathbb{R} \to \mathbb{R}^k$ is generated by a GPAC iff all of its components are.

From now on, we will mostly talk about GPACs as being systems of ODEs of the type (2).

For a concrete example of the previous proposition, see Figure 2. GPAC-generable functions (in the sense of [15]) are obviously d.a.. Another interesting consequence is the following (recall that solutions of analytic ODEs are always analytic – cf. [17]):

**Corollary 4** If $f$ is a function generated by a GPAC, then it is analytic.

As we have seen in Proposition 2, the Gamma function is not generated by a GPAC. However, it has been recently proved that it can be computed by a GPAC if we use the notion of GPAC computability presented in [9].

Notice that in Shannon’s original definition of the GPAC nothing is assumed about the constants and initial conditions of the ODE (2). In particular, there can be non-computable reals. This kind of GPAC can trivially lead to super-Turing computations. To avoid this, the model of [9] can actually be reinforced as follows:

**Definition 5** A function $f : [a, b] \to \mathbb{R}$ is GPAC-computable\(^2\) iff there exists some computable polynomial $p : \mathbb{R}^{n+1} \to \mathbb{R}^n$ and $n - 1$ computable values $\alpha_1, ..., \alpha_{n-1}$ such that:

1. $(y_1, ..., y_n)$ is the solution of ODE $y' = p(y, t)$ with initial condition $(\alpha_1, ..., \alpha_{n-1}, x)$ set at time $t_0 = 0$
2. There are $i, j \in \{1, ..., n\}$ such that $\lim_{t \to \infty} y_j(t) = 0$ and $|f(x) - y_i(t)| \leq y_j(t)$.\(^3\)

\(^2\) Note that in this paper, the term GPAC-computability refers to this particular notion. The expression “generated by a GPAC” corresponds to Shannon’s notion of computability.

\(^3\) We assume that $y(t)$ is defined for all $t \geq 0$. 

5
We remark that $\alpha_1, \ldots, \alpha_{n-1}$ are auxiliary parameters needed to compute $f$.

**Proposition 6 ([9])** The $\Gamma$ function is GPAC-computable.

In this paper, we show that this actually holds for all computable functions. Indeed, we prove that if a real function $f$ is computable, then it is GPAC-computable. Reciprocally, we prove that if $f$ is GPAC-computable, then it is computable. In other words, our main result is: For compact domains, GPAC-computable functions are precisely computable functions in the classical sense.

2.2 Computable Analysis

*Recursive analysis* or *computable analysis*, was introduced by Turing [18], Grzegorczyk [19], and Lacombe [20].

The idea underlying computable analysis is to extend the classical computability theory so that it might deal with real quantities. See [6] for an up-to-date monograph of computable analysis from the computability point of view, or [5] for a presentation from a complexity point of view.

Following Ko [5], let $\nu_Q : \mathbb{N}^3 \to \mathbb{Q}$ be the following representation of dyadic rational numbers by integers: $\nu_Q(p, q, r) \mapsto (-1)^p \frac{q}{2^r}$.

Given a sequence $(y_n, z_n)_{n \in \mathbb{N}}$, where $y_n, z_n \in \mathbb{N}$, we write $(y_n, z_n) \leadsto x$ to denote the following property: for all $n \in \mathbb{N}$, $|\nu_Q(y_n, z_n, n) - x| < 2^{-n}$.

**Definition 7 (computability)**

1. A point $\mathbf{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$ is said computable (denoted by $\mathbf{x} \in \text{Rec}(\mathbb{R})$) if for all $j \in \{1, \ldots, d\}$, there is a computable sequence $(y_n, z_n)_{n \in \mathbb{N}}$ of integers such that $(y_n, z_n) \leadsto x_j$.

2. A function $f : X \subseteq \mathbb{R}^d \to \mathbb{R}$, where $X$ is compact, is said computable (denoted by $f \in \text{Rec}(\mathbb{R})$) if there is some $d$-oracle Turing machine $M$ with the following property: if $\mathbf{x} = (x_1, \ldots, x_d) \in X$ and $(\alpha_n^d) \leadsto x_j$, where $\alpha_n^d \in \mathbb{N}^2$, then $M$ with oracles $(\alpha_n^d)_{n \in \mathbb{N}}$ computes a sequence $(\beta_n)_{n \in \mathbb{N}}$, where $\beta_n \in \mathbb{N}^2$, satisfying $(\beta_n) \leadsto f(\mathbf{x})$. A function $f : X \subseteq \mathbb{R}^d \to \mathbb{R}^k$, where $X$ is compact, is said computable if all its projections are.

The following result is taken from [5, Corollary 2.14]

**Proposition 8** A real function $f : [a, b] \to \mathbb{R}$ is computable iff there exist two recursive functions $m : \mathbb{N} \to \mathbb{N}$ and $\psi : \mathbb{N}^4 \to \mathbb{N}^3$ such that:

---

$^4$ A computable sequence of integers $(x_n)_{n \in \mathbb{N}}$ is a sequence such that $x_n = f(n)$ for all $n \in \mathbb{N}$ where $f : \mathbb{N} \to \mathbb{N}$ is recursive.
(1) $m$ is a modulus of continuity for $f$, i.e. for all $n \in \mathbb{N}$ and all $x, y \in [a, b]$, one has

$$|x - y| \leq 2^{-m(n)} \implies |f(x) - f(y)| \leq 2^{-n}$$

(2) For all $(i, j, k) \in \mathbb{N}^3$ such that $\nu_Q(i, j, k) \in [a, b]$ and all $n \in \mathbb{N}$,

$$|\nu_Q(\psi(i, j, k, n)) - f((-1)^{i/j^2})| \leq 2^{-n}.$$

2.3 Simulating TMs with ODEs

To prove the main result of this paper, we need to simulate a TM with differential equations and, in particular, we need to compute the iterates of a given function. This can be done with the techniques described in [21].

**Proposition 9** Let $f : \mathbb{N} \to \mathbb{N}$ be some function. Then it is possible to iterate $f$ with an ODE, i.e. there is some (non-analytic) $g : \mathbb{R}^3 \to \mathbb{R}^2$, such that for all $x_0 \in \mathbb{N}$, the solution of the system of ODEs $\frac{dy}{dt}(\overline{y}, \overline{y}) = g(\overline{y}, \overline{y}, t)$, with $\overline{y}(0) = y(0) = x_0$ satisfies $|y(m) - f^{|m|}(x_0)| < 1/4$ for all $m \in \mathbb{N}$.\(^5\)

**Proof.** Branicky’s construction from [21] involves non differentiable functions. To avoid this, we describe Branicky’s idea following the approach in [22, p. 37], where the functions can be arbitrarily smooth. Before presenting the whole procedure, we need some auxiliary functions. In particular, let $\theta_j : \mathbb{R} \to \mathbb{R}$, $j \in \mathbb{N} - \{0, 1\}$ be the function defined by

$$\theta_j(x) = 0 \text{ if } x < 0, \quad \theta_j(x) = x^j \text{ if } x \geq 0.$$

This function can be seen [23] as a $C^{j-1}$ version of Heaviside’s step function $\theta(x)$, where $\theta(x) = 1$ for $x \geq 0$ and $\theta(x) = 0$ for $x < 0$. Note that $\theta_j$ is differentiable but non analytic. Consider also the integer part function $r : \mathbb{R} \to \mathbb{R}$ defined by

$$r(0) = 0, \quad r'(x - 1/4) = c_j \theta_j(-\sin 2\pi x),$$

where $c_j = \left(\int_0^1 \theta_j(-\sin 2\pi x)dx\right)^{-1}$. The function $r$ has the following property: $r(x) = n$, whenever $x \in [n - 1/4, n + 1/4]$, for all integer $n$, as illustrated in Figure 3. Now consider the following system of ODEs

$$\overline{y}' = \lambda_j(\overline{f}(\overline{y}) - \overline{y})^3\theta_j(\sin 2\pi t)$$

$$\overline{y}' = \lambda_j(r(\overline{y}) - \overline{y})^3\theta_j(-\sin 2\pi t)$$

\(^5\) $f^{|k|}$ gives the $k$th iteration of $f$.\(7\)
where \( \tilde{f} : \mathbb{R} \to \mathbb{R} \) is an arbitrary extension to the reals of function \( f : \mathbb{N} \to \mathbb{N} \), \( \overline{y}(0) = \overline{y}(0) = x_0 \in \mathbb{N} \) and \( \lambda_j > 8c_j \). Notice that, if \( k \in \mathbb{N} \), then for \( t \in [k, k + \frac{1}{2}] \), \( y'(t) = 0 \), and for \( t \in [k - \frac{1}{2}, k] \), \( \overline{y}(t) = 0 \) (cf. Figure 4). According to the construction presented in the proof of Prop. 3.4.2 from [22], we can conclude that \( \left| f[k](x_0) - y(k) \right| < 1/4 \) for all \( k \in \mathbb{N} \). For better readability of this paper, we use the following convention: underlined and overlined variables will only be used in the context of Branicky’s construction and we assume that these variables have, respectively, a behavior similar to those of \( y \) and \( \overline{y} \).

Using the previous construction, it is not difficult to simulate the evolution of a Turing machine: For simplicity, and without loss of generality, we only consider Turing machines using 10 symbols and set the following coding. Let \( M \) be some one tape Turing machine, with \( m \) states and 10 symbols. Then to each state we associate a number in \( \{1, \ldots, m\} \) and to each symbol we associate a number in \( \{0, \ldots, 9\} \), assuming that the blank symbol corresponds to the 0. If

\[
...B B B a_{-k} a_{-k+1} \ldots a_{-1} a_0 a_1 \ldots a_n B B B...
\]
is the tape content of M, then it can be coded in the following two integers:

\[ y_1 = a_0 + a_1 10 + \ldots + a_n 10^n \quad y_2 = a_{-1} + a_{-2} 10 + \ldots + a_{-k} 10^{k-1}. \]  

(5)

The whole state of M is then given by its state \( s \), and two integers \( y_1 \) and \( y_2 \). The transition function of M corresponds therefore to a function \( f_M : \mathbb{N}^3 \to \mathbb{N}^3 \). Branicky’s trick then provides a system of ODEs that iterates function \( f_M \) and hence simulates the corresponding Turing machine.

More generally, if M has \( l \) tapes, then its transition function is defined over \( \mathbb{N}^{2l+1} \). In this paper we also take the following convention: if a TM computes a function \( h : \mathbb{N} \to \mathbb{N} \), we suppose that when it starts computing, the less significant digit is read by the head. Similarly, when the machine stops, we assume that the head is over the least significant digit.

Note that Branicky’s construction relies on non-analytic functions (e.g. \( \theta_j \)), which is problematic for us (cf. Corollary 4). However, in [12] it was shown how to implement Branicky’s construction for simulating Turing machines in GPACs. The idea is to approximate non-analytic functions with analytic ones, and to control the error committed along the entire simulation. The following result is an adaptation of Theorem 4 from [12].

**Proposition 10 ([12])** Suppose that \( f_M : \mathbb{N}^3 \to \mathbb{N}^3 \) is the transition function of a Turing machine M, under the encoding presented in Equation (5), \( x_0 \in \mathbb{N}^3 \) represents an initial configuration and \( \varepsilon \) is an arbitrary constant satisfying \( 0 < \varepsilon < 1/4 \). Then there is a polynomial ODE and some \( \alpha \in \mathbb{R}^n \)

\[ z' = g(z, t), \quad z(0) = (\tilde{x}_0, \alpha) \]

such that for all \( \tilde{x}_0 \in \mathbb{R}^3 \) satisfying \( \| \tilde{x}_0 - x_0 \|_\infty \leq \varepsilon \), one has \(^7\)

\[ \| z_1(t) - f_M^{[j]}(x_0) \|_\infty \leq 2\varepsilon. \]

for all \( j \in \mathbb{N} \) and for all \( t \in [j, j + 1/2] \).

Remark that the previous result relies on the encoding in Equation (5), which leaves some “room” around each integer. This allows to simulate the evolution of the Turing machine with a bounded error, which can be done with analytic functions. The drawback of that encoding is that it grows exponentially with the size of the input and it is therefore unbounded. However, this is believed to be necessary since it is conjectured that no analytic map on a compact,

---

6 Remark that, according to [12] (for similar results on other encodings cf. [24]), there is always a polynomial extension \( \tilde{f}_M : \mathbb{R}^{2l+1} \to \mathbb{R}^{2l+1} \) of \( f_M \).

7 For simplicity, we denote \( z \) by \((z_1, z_2)\), where \( z_1 \in \mathbb{R}^3 \) and \( z_2 \in \mathbb{R}^n \).
finite dimensional space can simulate a Turing machine through a reasonable encoding [25].

3 The result

The main result of this paper relates computable analysis with the GPAC, showing their equivalence in the framework described in the previous section.

**Theorem 11 (Main result)** A function \( f : [a, b] \to \mathbb{R} \) is computable iff it is GPAC-computable.

For easier readability, we split the proof in two sections. In this one we present two preliminary lemmas and then prove a weaker version of main theorem of the paper. Its proof will be completed in Section 4.

By Proposition 3, dynamical systems defined by an ODE of the form \( y' = p(t, y) \), where \( p \) is a vector of polynomials are equivalent to GPACs. For simplicity, we first suppose in the proofs that we have access to the function \( \theta_j \) and refer to the systems \( y' = p(t, y, \theta_j(y)) \) (6)

where \( \theta_j(y) \) means that \( \theta_j \) is applied componentwise to \( y \), as \( \theta_j \)-GPACs [9]. Similarly to Definition 5, we then define a notion of \( \theta_j \)-GPAC-computability. In Section 4, we will see how these functions \( \theta_j \)'s can be suppressed. To prove Theorem 11 we will need to simulate a cyclic sequence of TM computations, where the input is sequentially incremented when each computation finished. The following two lemmas show that this can be done with a \( \theta_j \)-GPAC. In particular, the first one shows how to implement a GPAC that tells when the computation of a Turing machine with input \( n \) has halted. Then the second lemma uses this result to perform sequential simulations of a given TM.

**Lemma 12** Let \( M \) be a Turing machine with input \( (k, n) \) which halts within time \( t(k, n) \), and \( m : \mathbb{N} \to \mathbb{N} \) a recursive function. Then, there exists a \( \theta_j \)-GPAC with a variable \( y_{\text{clock}} \) which takes value 1 an infinite countable number of times such that if \( t_n \) denotes the \( n \)th time \( y_{\text{clock}} = 1 \), then for all \( k_n \leq 2^{m(n)} \), \( t_{n+1} - t_n \geq t(k_n, n) \).

**Proof.** Consider the following non-halting Turing machine \( M_{\text{clock}} \): it has states denoted by \( 1, \ldots, d \). Given a value of \( n \), \( M_{\text{clock}} \) simulates \( M \) with input 0, \( n \). When \( M \) halts, \( M_{\text{clock}} \) simulates \( M \) with input 1, \( n \) and so on. This procedure is carried out until the first argument has value \( > 2^{m(n)} \). When this happens, \( M_{\text{clock}} \) enters in resetting state \( d \), and then restarts the whole procedure with second argument equal to \( n + 1 \).
By construction, note that if \( k_n \leq 2m(n) \), then M with input \((k_n,n)\) will never halt in time less than \( t_{n+1} - t_n \) where \( t_n \) denotes the \( n \)th time \( M_{\text{clock}} \) enters resetting state \( d \).

From Proposition 10, there is an ODE with the form (6) that simulates \( M_{\text{clock}} \). Notice that, from Branicky’s simulation (more precisely from the analytic implementation of Branicky’s simulation used in the proof of Proposition 10), there are two components of the solution of Equation (6) that represent the state of \( M_{\text{clock}} \), namely \( y_q_{\text{clock}} = (y_{q1_{\text{clock}}}, y_{q2_{\text{clock}}}) \). Then, when one of these variables reaches a value \( \geq d - 1/2 \), we know that the computation of \( M_{\text{clock}} \) for the current \( n \) is over. Furthermore, they have integer values at integer times. So, for any integer time \( t \) (more precisely, for every time interval \([t, t+1/2]\)), we have

\[
y_{\text{clock}} = \frac{\theta_j(y_{q_{\text{clock}}} - d + 0.5)}{0.5^j} = \begin{cases} 
1, & \text{if } M_{\text{clock}} \text{ is in state } d \text{ at time } t \\
0, & \text{otherwise.} 
\end{cases} 
\]  

Note that the function \( \theta_j \) is only applied to \( y_{q_{\text{clock}}} - d + 0.5 \). This will be important for the case in which we remove the \( \theta_j \) from the system.

To be able to describe GPAC constructions in a modular way, it helps to break down the system into several intermixed systems. For example, the variables of vector \( y \) of ODE \( y' = p(t, y) \) can be split into two blocks \( y_1, y_2 \). The whole system then rewrites into two sub-systems \( y'_1 = p_1(t, y_1, y_2) \), and \( y'_2 = p_2(t, y_1, y_2) \). This allows to describe each subsystem separately: we will consider that \( y_2 \) is an “external input” of the first, and \( y_1 \) is an “external input” of the second. By abuse of notation, we will still call such sub-systems GPACs.

**Lemma 13** Let \( M \) be a Turing machine with two inputs and two outputs, that halts on all inputs, and \( m : \mathbb{N} \rightarrow \mathbb{N} \) a recursive function. Let \( L \in \mathbb{N} - \{0\} \). If \((k_n)\) is a sequence of natural integers that satisfies \( k_n \leq L \times 2^m(n) \) and \( u_1(n) = k_n, u_2(n) = n \), then there is a \( \theta_j \)-GPAC

\[
y' = p(t, y, \theta_j(y), u_1, u_2),
\]  

with the following properties:

1. The \( \theta_j \)-GPAC simulates \( M \) with inputs \( u_1(n), u_2(n) \), starting with \( n = 0 \). When this simulation finishes, \( n \) is incremented and the simulation restarts with inputs \( u_1(n+1) \) and \( u_2(n+1) \), and so on.
2. Three variables of the \( \theta_j \)-GPAC act as a “memory”: they keep the value of the last \( n \) where the computation \( M(k_n, n) \) was carried out up to the end, and the corresponding two outputs.
Proof. Using the construction presented in [12], given $k, n \in \mathbb{N}$, we can build a GPAC $y' = p(t, y)$ that simulates robustly a Turing machine $M$ with inputs $k, n$. Here, we build a GPAC that cyclically simulates $M$ with inputs $k_n, n$, starting with $n = 0$. Once the computation for $k_n, n$ is finished, we restart a new computation with inputs $k_{n+1}, n + 1$. The main problems are: increase sequentially values of $n$; reset computations; and memorize the last successful computation of $M$.

First we focus on increasing the value of $n$, and resetting simulations of $M$ with inputs $k_{n+1}, n + 1$. One first problem is to be able to tell when this must happen. One simple solution would be to read the variable that codes the state of $M$, to detect that the simulation is over. This would be enough for this lemma. However, later, we will need to consider the case where $M$ has invalid (i.e. non-integers) inputs, hence we cannot always rely on the value of the state variable. Instead, we use the construction below which is more robust to errors and relies on the $\theta_j$-GPAC of the previous lemma: the simulations of $M$ should be resetted iff variable $y_{\text{clock}}$ value 1. This provides a solution which is guaranteed to be robust to any error on $k_n$, since the GPAC of previous lemma does not depend on $k_n$ at all.

We can now re-initialize $M$ with $k_{n+1}, n + 1$. Indeed, because $M$ can be simulated by equations of the type (4), we can adapt these equations to cyclically simulate $M$.

Let’s see in details how this can be done. If $(x_R, y_R)$ is the pair of variables that codes the right part of the first tape of $M$ and $f_{x_R}(x, y, q)$ gives the next value of $x_R$ ($x, y$ correspond to the variables simulating the Ist and 2nd tape of $M$ and $q$ correspond to the state of $M$) then $(x_R, y_R)$ can be given by the following type of equation:

\[ x_R' = \lambda_j(f_{x_R}(x, y, q) - x_R)^3\theta_j(\sin 2\pi t)(1 - y_{\text{clock}}) + \lambda_j(k_{n+1} - x_R)^3\theta_j(\sin 2\pi t)y_{\text{clock}} \]
\[ x_R' = \lambda_j(x_R - y_R)^3\theta_j(-\sin 2\pi t). \]

The first line of Equation (9) is the case where the computation isn’t over, and the second line defines the process of resetting (we need to input $k_{n+1}$ in the first tape). So the right part of the first tape has input $k_{n+1}$ and therefore $x_R$ should be $k_{n+1}$. In the 3rd line, we don’t have to use $y_{\text{clock}}$ because $x_R$ will follow the updated value of $x_R$. The constructions for the left part of the tapes and the state of $M$ are done similarly.
Now it only remains to show how we can memorize the result of the last successful computation of $M$. We can do this using an ODE of type (9)

$$y'_{mem} = \lambda_j (y_{mem} - y_{mem})^3 \theta_j (\sin 2\pi t) y_{clock} +$$

$$y'_{mem} = \lambda_j \sigma^d (y_{mem}) - y_{mem}^3 \theta_j (-\sin 2\pi t),$$

where $mem \in \mathbb{N}$ is the variable whose value needs to be memorized when $y_{clock}$ is 1.

Here, function $\sigma$ is the error-contracting function (already considered in [12]) defined by

$$\sigma(x) = x - 0.2 \sin(2\pi x).$$

It satisfies $\sigma(n) = n$ for all $n$ integer. This function contracts the error in the vicinity of integers as follows: let $n \in \mathbb{Z}$ and let $\varepsilon \in [0, 1/2)$. Then there is some contracting factor $\lambda_\varepsilon \in (0, 1)$ such that, for $\forall \delta \in [-\varepsilon, \varepsilon]$, $|\sigma(n + \delta) - n| < \lambda_\varepsilon \delta$.

The idea is similar to the one inherent to Equation (9). When $y_{clock} = 1$, we should store the value $mem$. However, due to use of inexact values, the error present in $y_{mem}$ increases during a one-time interval (the same happening to $y_{mem}$). Therefore, we just use the function $\sigma$ iterated $d$ times, for some fixed $d$, applied to the variables $y_{mem}, y_{mem}$ to compensate the deviation due to the error.

Next, we prove the “if” direction of Theorem 11 and the “only if” direction of a weaker version of Theorem 11.

### 3.1 Proof of the “if” direction

Let $f$ a GPAC-computable function. We want to show that $f$ is computable in the sense of computable analysis. By definition, we know that there is a polynomial ODE

$$y' = p(t, y)$$

$$y(0) = (\alpha, x)$$

which solution has two components $y_i : \mathbb{R}^2 \rightarrow \mathbb{R}$ and $y_j : \mathbb{R}^2 \rightarrow \mathbb{R}$ such that

$$|f(x) - y_i(x, t)| \leq y_j(x, t)$$

and

$$\lim_{t \rightarrow \infty} y_j(x, t) = 0$$

From standard error analysis of Euler’s algorithm, function $y_i$ and $y_j$ can be computed using Euler’s algorithm on ODE $y' = p(t, y)$ up to any given precision $2^{-n}$. However, in general $p$ does not satisfy a Lipschitz condition on
the real line, assumption needed to use this method. Instead one can use the approach of [26], where it is shown that $p$ is effectively Lipschitz on the real line and that the solution can be computed there.

3.2 Proof of the “only if” direction

**Lemma 14** If a function $f : [a, b] \to \mathbb{R}$ is computable then it is $\theta_j$-GPAC-computable.

**Proof.** By hypothesis, there is an oracle Turing machine such that for all oracles $(j(n), l(n))_{n \in \mathbb{N}} \sim x \in [a, b]$, the machine outputs a sequence $(z_n) \sim f(x)$, where $z_n \in \mathbb{N}^2$. From now on we suppose that $a > 0$ (the other case will be studied later). We can then assume that $j(n) = 0$ for any $n$ and, hence, the sign function $j$ is not needed. Using Proposition 8, it is not difficult to conclude that there are computable functions $m : \mathbb{N} \to \mathbb{N}$, $\text{abs}, \text{sgn} : \mathbb{N}^2 \to \mathbb{N}$ such that given $x \in [a, b]$ and non-negative integers $k, n$ satisfying $|k/2^m(n) - x| < 2^{-m(n)}$, one has

$$
|1 - 2 \times \text{sgn}(k, n)) \frac{\text{abs}(k, n)}{2^n} - f(x)| < \frac{1}{2^n}.
$$

(11)

Now, given some real $x$, we would like to design a $\theta_j$-GPAC that ideally would have the following behavior: given initial conditions $n = 1$ and $x$, it would:

1. Obtain from real $x$ and integer $n$, an integer $k$ satisfying $|k/2^m(n) - x| < 1/2^m(n)$,
2. Compute $\text{sgn}(k, n)$ and $\text{abs}(k, n)$;
3. When $\text{sgn}(k, n), \text{abs}(k, n)$ are obtained, compute

$$
(1 - 2 \times \text{sgn}(k, n)) \frac{\text{abs}(k, n)}{2^n}
$$

(12)

and memorize the result just till another cycle is completed;
4. Take $n = n + 1$ and restart the cycle.

With the lemmas 12 and 13, we showed how to simulate the Turing machine $M$, increment the value of $n$, and restart a new cycle with a GPAC. We still

---

8. In general, and following Prop. 8, one should use $(-1)^{\text{sgn}(k, n)}$ instead of the equivalent expression (for $\text{sgn}(k, n) \in \{0, 1\}$) $1 - 2 \times \text{sgn}(k, n)$. However, we prefer to use the last one since, in our simulation, $\text{sgn}(k, n)$ may take non-integer values.

9. We don’t have to worry with the encoding function in Equation (5) since $k, n, \text{sgn}(k, n), \text{abs}(k, n) \in \mathbb{N}$ and the encoding of any $x \in \mathbb{N}$ written on the tape is simply $x$. 

---
Fig. 5. Graphical representations of functions $r_i$ and $\omega_i$ ($i = 0, 1, 2$).

have to address the first step of the algorithm above: given some real $x$, and some integer $n$, we need to compute an integer $k$ satisfying

$$\left|2^{-m(n)}k - x\right| < 2^{-m(n)}$$

There is an obvious choice: take $k = \lfloor x2^{m(n)} \rfloor$. The problem is that the discrete function “integer part” $\lfloor \cdot \rfloor$ cannot be obtained by a GPAC (as a non-continuous and hence non-analytic function). Our solution is the following:

Recall that the function $r$, defined by Equation (3), has the following property:

$$r(x) = n, \text{ whenever } x \in [n - 1/4, n + 1/4], \text{ for all integer } n.$$  

Then we cover $\lfloor \cdot \rfloor$ over all of its domain by three functions $y_r_i(t) = r(t - 1/4 - i/3)$, for $i = 0, 1, 2$ and we define (see below) a set of detecting functions $\omega_i$ such that $\omega_i(t) \neq 0$ iff $y_r_i(t)$ is an integer and $t \notin 1/2\mathbb{Z} + i/3$ (cf Figure 5). Hence we can get rid of non integer values with the products $\omega_i y_r_i$.

Remember from Lemma 13 that the Turing machine $M$ can be simulated by a system of ODEs

$$y' = p_u(t, y, \theta_j(y), y_{input(1)}, y_{input(2)}),$$

(13)

denoted by $U$. This system has two variables corresponding to the two external inputs of $M$, $y_{input(1)}$, $y_{input(2)}$, and two variables, denoted by $y_{sgn}$, $y_{abs}$, corresponding to the two outputs of $M$.

Then we construct a system of ODEs as Figure 6 suggests. More formally, the GPAC contains three copies, denoted by $U_0$, $U_1$, and $U_2$ of the system (13), each one with external input $y_{r_i}, n$:

$$U_i : \quad Y_i' = p_u(t, Y_i, \theta_j(Y_i), y_{r_i}, n) \quad i = 0, 1, 2.$$  

In other words, they simulate a Turing machine $M$ with input $(k, n)$ whenever $y_{r_i}(t) = k$. Denote by $y_{sgn_i}$ and $y_{abs_i}$ its two outputs. The problem is that sometimes $y_{r_i}(t) \notin \mathbb{N}$ and hence the outputs $y_{sgn_i}$ and $y_{abs_i}$ of $U_i$ may not have a meaningful value. Thus, we need to have a subsystem of ODEs (the “weighted sum circuit”) that can select “good outputs”. It will be constructed with the help of the “detecting functions” defined by $\omega_i(t) = \theta_j(\sin 2\pi(t - i/3))$, for $i = 0, 1, 2$ (cf. Figure 5).
Fig. 6. Schema of a GPAC that calculates a real computable function \( f : [a, b] \rightarrow \mathbb{R} \). \( x \) is the current argument for \( f \), the two outputs of the “weighted sum unit” give \( \text{sgn}(k, n) \) and \( \text{abs}(k, n) \). The divisor computes Equation (12). The dotted line gives the signal that orders the memory to store the current value and the other GPACs to restart the computation with the new inputs associated to \( n + 1 \).

It is easy to see that for every \( t \in \mathbb{R} \), \( \omega_0(t) + \omega_1(t) + \omega_2(t) > 0 \) and that \( \omega_i(t) > 0 \) iff \( y_r_i(t) \) is an integer and \( t \not\in 1/2\mathbb{Z} + i/3 \) (i.e. \( U_i \) is fed with a “good input”). Hence, in the weighted sum

\[
y_{\text{abs}} = \frac{\omega_0(2^{m(n)}x) y_{\text{abs}0} + \omega_1(2^{m(n)}x) y_{\text{abs}1} + \omega_2(2^{m(n)}x) y_{\text{abs}2}}{\omega_0(2^{m(n)}x) + \omega_1(2^{m(n)}x) + \omega_2(2^{m(n)}x)}
\]

only the “good outputs” \( y_{\text{abs}i} \) are not multiplied by 0 and therefore \( y_{\text{abs}} \) provide \( \text{abs}(k, n) \),\(^{10}\) whatever the value of real variable \( x \) is. Replacing \( \text{abs}_i \) by \( \text{sgn}_i \) provides in a similar way \( \text{sgn}(k, n) \).

Then we use an other subsystem of ODEs for the division in Equation (12), which provides an approximation of \( f(x) \) with error bounded by \( 2^{-n} \), from \( \text{abs}(k, n) \) and \( \text{sgn}(k, n) \)

\[
y_{\text{approx}} = (1 - 2 \times y_{\text{sgn}}) \frac{y_{\text{abs}}}{2^n}
\]

We will see in Section 4.5 that the error done when approaching \( f(x) \) with \( (1 - 2 \times \text{sgn}(k, n)) \text{abs}(k, n)/2^n \) computed with the GPAC is bounded by \( \eta/2^n \), where \( \eta \in \mathbb{N} \) is some fixed constant. Hence we can compute with the GPAC a function \( y_j \) satisfying condition 2 of Definition 5.

\(^{10}\) In reality, it gives a weighted sum of \( \text{abs}(k, n) \) and \( \text{abs}(k-1, n) \). This is because if \( 2^{m(n)}x \in [i, i + 1/6] \), for \( i \in \mathbb{Z} \), we have that both \( y_{r0}(2^{m(n)}x) = i \) and \( y_{r2}(2^{m(n)}x) = i - 1 \) gives “good inputs”. Nevertheless this is not problematic for our concerns, since this only introduces an error bounded by \( 2^{-n} \), that can be easily dealt with (see Section 4.5).
So far, we have shown that the $\theta_j$-GPAC we designed above converges to $f(x)$ as required by condition (1) of Definition 5. Moreover, all its parameters except $x$ are computable as in condition (2).

We now have to deal with the case where $a \leq 0$. This case can be reduced to the previous one as follows: let $k$ be an integer greater than $|a|$. Then consider the function $g : [a + k, b + k] \to \mathbb{R}$ such that $g(x) = f(x - k)$. The function $g$ is computable in the sense of computable analysis, and has only positive arguments. Therefore, by the previous case, $g$ is GPAC-computable. Then, to compute $f$, it is only necessary to use a substitution of variables in the system of ODEs computing $g$. This ends the proof of Lemma 14.

Remark that the previous proof is constructive, in the sense that if we are given a computable function $f$ and the Turing machine computing it, we can explicitly build a corresponding GPAC that computes it.

4 Removing the $\theta_j$’s

To complete the proof of Theorem 11 we need to show that Lemma 14 also holds for GPACs, without using $\theta_j$’s. This is based on the construction presented in [12], where the fundamental idea to remove the $\theta_j$’s is to allow a certain error in the simulation while keeping it bounded.

We follow the steps of the proof of Lemma 14, modifying the construction of the GPAC that simulates the function-oracle Turing machine $M$ where necessary. More precisely, the parts of the $\theta_j$-GPAC in Lemma 14 that use $\theta_j$ are the ones that: memorize the value of Equation (12), increment the value of $n$ and reinitialize the input of $M$, and calculate the weighted-sum of Equation (14). Sections 4.2, 4.3 and 4.4, respectively, deal with those parts of the proof. We don’t need to address again the computation of $\text{sgn}(k, n)$ and $\text{abs}(k, n)$ since we know from Proposition 10 that this can be done with a GPAC. This shows that there is a GPAC satisfying condition (1) of Definition 5. Finally, section 4.5 addresses the bound on the error established by condition (2).

4.1 Preliminary results for removing the $\theta_j$’s

We begin by introducing some tools that were presented in [12]. Namely, we consider two error-contracting functions, which can be easily defined with polynomial ODEs.
(1) We consider the error-contracting function defined by
\[ \sigma(x) = x - 0.2 \sin(2\pi x). \]

already used page 13. This function will be useful to reduce the error around every integer, by a predetermined amount (therefore the error-correction is performed in a *static* way).

(2) Let \( l_2 : \mathbb{R}^2 \to \mathbb{R} \) be given by \( l_2(x, y) = \frac{1}{\pi} \arctan(4y(x - 1/2)) + \frac{1}{2} \). This error-contracting function differs from the previous one on two aspects: (i) it can only contract the error in the vicinity of the integers 0 and 1; (ii) it can contract the error in a *dynamic* way. Indeed \( l_2 : \mathbb{R}^2 \to \mathbb{R} \) has the property that whenever \( \tilde{a} \) is an approximation of \( a \in \{0, 1\} \), with error bounded by \( 1/4 \), then \( |l_2(\tilde{a}, y) - a| < 1/y \), for \( y > 0 \) (more precisely, if \( \tilde{a} \leq 1/4 \), then \( |l_2(\tilde{a}, y) - 0| < 1/y \), and if \( \tilde{a} \geq 3/4 \), then \( |l_2(\tilde{a}, y) - 1| < 1/y \), for \( y > 0 \)).

Next we will show how those error-contracting functions allow us to eliminate the \( \theta_j \)'s in the proofs of Lemmas 12 and 13. Without loss of generality, pick \( \varepsilon = 1/10 \) (any positive value \( < 1/8 \) would also work).

### 4.2 Memorizing a partial result

We will show how to remove \( \theta_j \) from Equations (7) and (10), which together define a variable \( y_{mem} \) that “stores” the last successful computation of \( M \) (i.e. \( sgn \), \( abs \)). We can adapt Equation (7) in the following way: we are only interested in the cases when the value of \( \theta_j(y_{q,\text{clock}} - d + 0.5)/0.5^j \) is 0 or 1 (for integer time values). Therefore, since \( d \) is the larger “exact” value that \( y_{q,\text{clock}} \) can take, it is not difficult to define an analytic function \( T : \mathbb{R} \to \mathbb{R} \) (e.g. an exponential) satisfying

\[
T(y_{q,\text{clock}}) \begin{cases} 
\geq 3/4 & \text{if } M \text{ halted} \\
\leq 1/4 & \text{if } M \text{ hasn’t halted}
\end{cases}
\]

Now we can use this term to replace the term \( \theta_j(y_{q,\text{clock}} - d + 0.5)/0.5^j \) in Equation (7). The Equation (10) can then be adapted to the analytic case, in a similar way as Equation (9). This process is described in [12]. The idea is to correct the error in \( y_{q,\text{clock}} \) with the function \( l_2 \). For instance, in the first line of (10), one should substitute \( y_{q,\text{clock}} \) by a term \( l_2(\cdot, \cdot) \), where the first argument of \( l_2 \) is the amount to be corrected (i.e. \( y_{q,\text{clock}} \)) and the second argument is a bound on \( \lambda_j(mem - y_{mem})^3 \) (e.g. \( \lambda_j((mem - y_{mem})^4 + 1) \)). We can also replace \( y_{q,\text{clock}} (\sin 2\pi t) \) (Notice that we still need a “clock” since the equation describing \( y'_{q,\text{clock}} \) should be “enabled” for values in \([k, k + 1/2]\),

18
where \( k \in \mathbb{Z} \) and “disabled” otherwise).

### 4.3 Reinitializing the TM's

To adapt the proof of Lemma 13, we only need to reinitialize the input of the TM. For example, the first input of the GPAC simulating the Turing machine M should be an approximation of \( \lfloor x^{2^{m(n)}} \rfloor \) with error less than some error bound \( 0 < \varepsilon < 1/10 \) on the time interval \([0, 1/2]\). However, since \( n \) is computed with error bounded by \( \varepsilon \), this error is amplified by the exponential \( 2^{m(n)} \). Hence, we have to increase the precision of calculations for each value of \( n \). This is done in two steps: in each time-unit interval, increase the constant \( \lambda_j \) in Equation (9) (i.e., decrease the “targeting error”. Note that \( \lambda_j \) needs not to be a constant and may be, e.g., the exponential function, which is enough for our purposes), increase the precision of the simulations variables (with a finite number of applications of \( \sigma \)), and in the functions \( l_2(...) \) substituting \( \theta_k \), increase the second argument in each step (e.g. multiplying by the constant \( \lambda_j \)). In this form we can decrease the error linearly in each step, \( \varepsilon_{k+1} = \varepsilon_k / N \), where \( N \in \mathbb{N} - \{0\} \) may be arbitrary, but fixed a priori. To perform the \( n \)th simulation (the one that uses input \( \lfloor x^{2^{m(n)}} \rfloor \)), and supposing that for each \( n \) we need at least \( m(n) \) steps to compute \( 2^{m(n)} \) (to achieve this, it is enough to assume that the Turing machine computing this number, after finishing this computation, counts from 1 to \( 2^{m(n)} \)), we will need at least \( m(n) \) steps from the absolute beginning of the computation. Thus if we begin with error \( \varepsilon \), it will be reduced to \( \varepsilon / N^{m(n)-1} \) at step \( n \). Moreover, from Proposition 10, if \( n \) is computed with precision \( \varepsilon / N^{m(n)-1} \), then we can compute \( m(n) \) with precision \( 2\varepsilon / N^{m(n)-1} \). Therefore, the total error committed in \( x^{2^{m(n)}} \) if we do not use the exact \( n \), but instead \( n + \varepsilon \), is

\[
x^{2^{m(n)+\varepsilon/N^{m(n)-1}}} - x^{2^{m(n)}} = x^{2^{m(n)}(2\varepsilon/N^{m(n)-1} - 1)} < 1/10
\]

for \( N \) large enough (note that \( x \in [a, b] \) is bounded).

### 4.4 Computing the weighted sum

To obtain analytic versions of the \( \omega_i \) we first consider the function \( s : \mathbb{R} \to [-\frac{1}{8}, 1] \) defined by

\[
s(t) = \frac{1}{2} \left( \sin^2(2\pi t) + \sin(2\pi t) \right).
\]

On \([0, 1/2]\), \( s \) ranges between 0 and 1 and on \([1/2, 1]\) \( s \) ranges between \(-\frac{1}{8}\) and 0. Therefore, the function \( W : \mathbb{R} \times \mathbb{R}^+ \to [0, 1] \) defined by

\[
W_0(t, y) = l_2(s(t), y)
\]
gives a good alternative to $\omega_0$. Note that for $t \in [1/2, 1]$, we always have $|W(t, y)| < 1/y$. In other words, $y$ permits to have a control over the error done when $\omega_0$ is 0. Moreover, from simple calculations, we get $s([0.16, 0.34]) \subseteq [3/4, 1]$, i.e. $W_0$ is guaranteed to be non-zero in an interval of length $0.18 > 1/6$. We could proceed similarly to obtain approximations of $\omega_1, \omega_2$. However the “non-zero” parts are not sufficient to cover all the real line. Instead of three functions $\omega_0, \omega_1, \omega_2$, we will need six $W_0, ..., W_5$ to cover adequately the whole real line. Nevertheless, the overall procedure does not change if we pass from 3 to 6 covering functions. Thus, instead of Equation (14), we use

$$y_{abs} = \frac{\sum_{i=0}^{5} W_i(2^{m(n)} x, \gamma \cdot (y_{abs} + \ldots + y_{abs})) y_{abs}}{\sum_{j=0}^{5} W_j(2^{m(n)} x, \gamma \cdot (y_{abs} + \ldots + y_{abs}))}$$

where $\gamma$ is some sufficiently big constant (the second argument of $W_i$ is to ensure that the computations are performed with enough precision). In this manner, $y_{abs}$ gives a weighted sum that is a sufficient approximation of $abs(k, n)$ with error bounded by some value $\varepsilon > 0$. The construction for $y_{sgn}$ is similar.

### 4.5 Bounds on the error

Finally, we check that the bound on the total error committed by the GPAC when computing approximations of $f(x)$ is as claimed. This error consists of two parts: one given by Equation (11), the other caused by the fact the GPAC uses approximations of $n, sgn(k, n)$ and $abs(k, n)$. Because $abs(k, n)$ and $2^n$ are calculated with error bounded by $\varepsilon$, we have that the approximate value of $\frac{abs(k, n)}{2^n}$ belongs to $[\frac{abs(k, n) - \varepsilon}{2^n}, \frac{abs(k, n) + \varepsilon}{2^n}]$. Moreover, since $x \in [a, b]$, the absolute value of $x$ is bounded and therefore the approximation of $\frac{abs(k, n)}{2^n}$ is always bounded by some value $\eta$. Therefore, the error done when calculating numerically Equation (12) is bounded by the product of $\eta$ multiplied by the error of $sgn(k, n)$. Since the error on $sgn(k, n)$ belongs to $O(2^{-n})$ (see Section 4.3), we can see that the error belongs to $O(2^{-n})$ as we wanted.

### 5 Conclusion

In this paper we established some links between computable analysis and Shannon’s General Purpose Analog Computer. In particular, we showed that contrarily to what was previously suggested, the GPAC and computable analysis can be made equivalent from a computability point of view, as long as we take an adequate notion of computation for the GPAC. In addition to those results it would be interesting to answer the following questions. Is is possible to have similar results, but at a complexity level? For instance, using the
framework of [5], is it possible to relate polynomially-time computable functions to a class of GPAC-computable functions where the error $\varepsilon$ is given as a function of a polynomial of $t$? And if this is true, can this result be generalized to other classes of complexity? From the computability perspective, our results suggest that polynomial ODEs and GPACs are very natural continuous-time counterparts to Turing machines.

Acknowledgments. This work was partially supported by Fundação para a Ciência e a Tecnologia and FEDER via the Center for Logic and Computation - CLC, the project ConTComp POCTI/MAT/45978/2002 and grant SFRH/BD/17436/2004. Additional support was also provided by the Fundação Calouste Gulbenkian through the Programa Gulbenkian de Estímulo à Investigação, and by EGIDE and GRICES under the Program Pessoa through the project Calculabilité et complexité des modèles de calculs à temps continu.

References


