The General Purpose Analog Computer and Computable Analysis are two equivalent paradigms of analog computation

Olivier Bournez[1,6], Manuel L. Campagnolo[2,4], Daniel S. Graça[3,4], and Emmanuel Hainry[5,6]

[1] INRIA Lorraine, Olivier.Bournez@loria.fr
[2] DM/ISA, Universidade Técnica de Lisboa, 1349-017 Lisboa, Portugal
mlc@math.isa.utl.pt
[3] DM/FCT, Universidade do Algarve, C. Gambelas, 8005-139 Faro, Portugal
dgraca@ualg.pt
[4] CLC, DM/IST, Universidade Técnica de Lisboa, 1049-001 Lisboa, Portugal
[5] Institut National Polytechnique de Lorraine, Emmanuel.Hainry@loria.fr
[6] LORIA (UMR 7503 CNRS-INPL-INRIA-Nancy2-UHP), Campus scientifique, BP 239, 54506 Vandœuvre-Lès-Nancy, France

Abstract. In this paper we revisit one of the first models of analog computation, Shannon's General Purpose Analog Computer (GPAC). The GPAC has often been argued to be weaker than computable analysis. As main contribution, we show that if we change the notion of GPAC-computability in a natural way, we compute exactly all real computable functions (in the sense of computable analysis). Moreover, since GPACs are equivalent to systems of polynomial differential equations then we show that all real computable functions can be defined by such models.

1 Introduction

In the last decades, the general trend for theoretical computer science has been directed towards discrete computation, with relatively scarce emphasis on analog computation. One possible reason is the fact that there is no Church-Turing thesis for analog computation. In other words, among the many analog models that have been studied, be it the BSS model [2], Moore's $\mathbb{R}$-recursive functions [17], neural networks [23], or computable analysis [20, 13, 25], none can be treated as a "universal" model.

In part, this is due to the fact that few relations between them are known. Moreover some of these models have been argued not to be equivalent, making the idea of a Church-Turing thesis for analog models an apparent utopian goal. For example the BSS model allows discontinuous functions while only continuous functions can be computed in the framework of computable analysis [25].

However, this objective may not be as unrealistic as it seems. Indeed, we will prove in this paper the equivalence of two models of analog computation that were previously considered non-equivalent: on one side, computable analysis and

on the other side, the General Purpose Analog Computer (GPAC). The GPAC was introduced in 1941 by Shannon [22] as a mathematical model of an analog device, the Differential Analyzer [6]. The Differential Analyzer was used from the 30s to the early 60s to solve numerical problems, especially differential equations for example in 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 Fig. 1 page 6, where inputs are functions of an independent variable called the *time*. It will be more precisely described in Subsection 2.2.

Many of the usual real functions are known to be generated by a GPAC, a notable exception is the Gamma function $\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt$ [22]. Since this function is known to be computable under the computable analysis framework [20], it seems and it has often been argued that the GPAC is a weaker model than computable analysis. However, we believe this is mostly due to a misunderstanding, and that this limitation is more due to the notion of GPAC-computability rather than the model itself.

In fact, 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 the Γ function becomes computable as shown recently in [9]. In this paper, the term GPAC-computable will refer to this notion. Notice that this "converging computation" with GPACs corresponds to a particular class of $\mathbb{R}$-recursive functions [17, 18, 4]. As in [4] we only consider a Turing-computable subclass of $\mathbb{R}$-recursive functions, but here, in some sense, we restrict our focus to functions that can be defined as limits of solutions of polynomial differential equations.

In this paper, we extend the result from [9] to obtain the following theorem: A function defined on a compact domain is computable (in the sense of computable analysis) if and only if it is computed by a GPAC in a certain framework.

It was already known [10] that Turing machines can be simulated by GPACs. Since functions computable in the sense of computable analysis are those computed by function-oracle Turing machines [13], this paper shows that the previous result can be extended to such models. This way, it gives an argument to say that the Church-Turing thesis may not be as utopian as it was believed: the Γ function is not generable by a GPAC but computable by a GPAC, and more generally, computable analysis is equivalent to GPAC computability.

This paper is an extended abstract. Detailed proofs can be found in technical report [3].

2 Preliminaries

2.1 Computable Analysis

Recursive analysis or *computable analysis*, was introduced by Turing [24], Grzegorczyk [12], and Lacombe [15].

The idea underlying computable analysis is to extend the classical computability theory so that it might deal with real quantities. See [25] for an up-to-date monograph of computable analysis from the computability point of view, or [13] for a presentation from a complexity point of view.

Following Ko [13], let $\nu_{\mathbb{Q}} : \mathbb{N}^3 \to \mathbb{Q}$ be the following representation of dyadic rational numbers by integers: $\nu_{\mathbb{Q}}(p,q,r) \mapsto (-1)^p \frac{q}{2^r}$.

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

Definition 1. ***(computability)***

1. *A point $\mathbf{x} = (x_1, ..., x_d) \in \mathbb{R}^d$ is said computable (denoted by $\mathbf{x} \in \mathcal{R}ec(\mathbb{R})$) if for all $j \in \{1, ..., d\}$, there is a computable sequence $(y_n, z_n)_{n\in\mathbb{N}}$ of integers such that $(y_n, z_n) \rightsquigarrow x_j$.*[7]
2. *A function $f : X \subseteq \mathbb{R}^d \to \mathbb{R}$, where X is compact, is said computable (denoted by $f \in \mathcal{R}ec(\mathbb{R})$), if there is some d-oracle Turing machine M with the following property: if $\mathbf{x} = (x_1, ..., x_d) \in X$ and $(\alpha_n^j) \rightsquigarrow x_j$, where $\alpha_n^j \in \mathbb{N}^2$, then when M takes as oracles the sequences $(\alpha_n^j)_{n\in\mathbb{N}}$, it will compute a sequence $(\beta_n)_{n\in\mathbb{N}}$, where $\beta_n \in \mathbb{N}^2$, satisfying $(\beta_n) \rightsquigarrow f(\mathbf{x})$. 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 [13, Corollary 2.14]

Theorem 1. *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:*

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*
$$|x - y| \leq 2^{-m(n)} \quad \Longrightarrow \quad |f(x) - f(y)| \leq 2^{-n}$$
2. *For all $(i, j, k) \in \mathbb{N}^3$ such that $\nu_{\mathbb{Q}}(i, j, k) \in [a, b]$ and all $n \in \mathbb{N}$,*
$$\left| \nu_{\mathbb{Q}}(\psi(i, j, k, n)) - f\left((-1)^i \frac{j}{2^k}\right) \right| \leq 2^{-n}.$$

2.2 The GPAC

The GPAC was originally introduced by Shannon in [22], and further refined in [19], [16], [11], [9]. The model basically consists of families of circuits built with the basic units presented in Fig. 1. In general, not all kinds of interconnections are allowed since this may lead to undesirable behavior (*e.g.* non-unique outputs. For further details, refer to [11]).

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 the following definition:

[7] 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.

Definition 2. *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*

$$p\left(t, y, y', ..., y^{(n)}\right) = 0, \qquad on\ I. \tag{1}$$

As a corollary, and noting that the Gamma function $\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt$ is not d.a. [21], we get that

Proposition 1. *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 [19], [16]). However, for the more robust class of GPACs defined in [11], the following stronger property holds:

Proposition 2. *A scalar function $f : \mathbb{R} \to \mathbb{R}$ is generated by a GPAC iff it is a component of the solution of a system*

$$y' = p(t, y), \tag{2}$$

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 ordinary differential equations (ODEs) of the type (2). For a concrete example of the previous proposition, see Fig. 2 page 6. GPAC generable functions (in the sense of [11]) are obviously d.a.. Another interesting consequence is the following (recall that solutions of analytic ODEs are always analytic - *cf.* [1]):

Corollary 1. *If f is a function generated by a GPAC, then it is analytic.*

As we have seen in Proposition 1, 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 following notion of GPAC computability [9]:

Definition 3. *A function $f : \mathbb{R}^n \to \mathbb{R}^k$ is computable by a GPAC via approximations if there exists some polynomial ODE (2) with n components $y_1, ..., y_n$ admitting initial conditions $x_1, ..., x_n$ such that, for some particular components g and ε of $y_1, ..., y_n$, one has $\lim_{t\to\infty} \varepsilon(x_1, ..., x_n, t) = 0$ and*

$$\|f(x_1, ..., x_n) - g(x_1, ..., x_n, t)\| \leq \varepsilon(x_1, ..., x_n, t). \tag{3}$$

More informally, this model of computation consists of a dynamical system $y' = p(y, t)$ with initial condition x. For any x, the component g of the system approaches $f(x)$, with the approximation error being bounded by the other component ε which goes to 0 with time.

One point, behind the initial definitions of GPAC from Shannon, is that nothing is assumed on the constants and initial conditions of the ODE (2). In particular, there can be non-computable reals, and some outputs of a GPAC can *a priori* have some unbounded rate of growth.

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 4. *We say that $f : [a, b] \to \mathbb{R}$ is GPAC-computable iff:*[8]

1. *There is a ϕ computed by a GPAC $\mathcal{U}$ via approximations, with initial conditions $(\alpha_1, \ldots, \alpha_{n-1}, x)$ set at $t_0 = 0$, such that $f(x) = \phi(\alpha_1, \ldots, \alpha_{n-1}, x)$ for all $x \in [a, b]$;*
2. *The initial conditions $\alpha_1, \ldots, \alpha_{n-1}$ and the coefficients of p in (3) are computable reals.*
3. *If y is the solution of the GPAC $\mathcal{U}$, then there exists $c, K > 0$ such that $\|y\| \leq cK^{|t|}$ for time $t \geq 0$.*

We remark that $\alpha_1, \ldots, \alpha_{n-1}$ are auxiliary parameters needed to compute f. The result of [9] can be reformulated as:

Proposition 3 ([9]). *The Γ function is GPAC-computable.*

In this paper, we show that this actually hold for all computable functions (in the sense of computable analysis). 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.

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 [5] (cf. Fig. 3).

Proposition 4. *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 $g : \mathbb{R}^3 \to \mathbb{R}^2$, such that for all $x_0 \in \mathbb{N}$, any solution of ODE $y' = g(y, t)$, with $y_1(0) = x_0$ satisfies $y_1(m) = f^{[m]}(x_0)$ for all $m \in \mathbb{N}$.*[9]

However Branicky's construction involves non-differentiable functions. To avoid this, we follow instead the approach of [7, p. 37] which shows that arbitrarily smooth functions can be used. In a first approach, we use the function $\theta_j : \mathbb{R} \to \mathbb{R}$, $j \in \mathbb{N} - \{0, 1\}$ 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 [8] 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 $x < 0$. This construction can even be done with analytic functions, as shown in [10].

Using the construction presented in [7], it is not difficult to simulate the evolution of a Turing machine. Indeed, it suffices to code each one of its configurations into integers and apply Branicky's trick, *i.e.* $f : \mathbb{N}^k \to \mathbb{N}^k$ gives the

[8] Recall that in the paper, the term GPAC-computable refers to this particular notion. The expression "generated by a GPAC" corresponds to Shannon's notion of "computability".

[9] $f^{[m]}$ denotes the mth iteration of f.

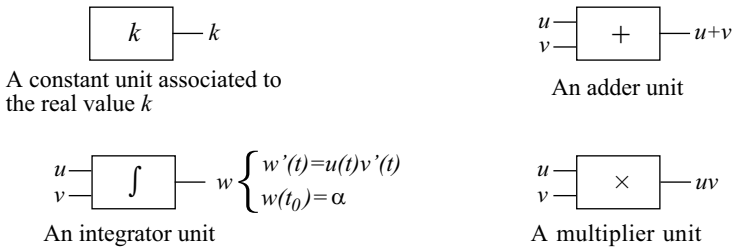

Fig. 1. Different types of units used in a GPAC.

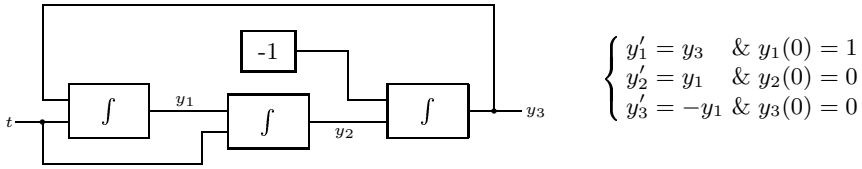

Fig. 2. Generating cos and sin via a GPAC: circuit version on the left and ODE version on the right. One has $y_1 = \cos$, $y_2 = \sin$, $y_3 = -\sin$.

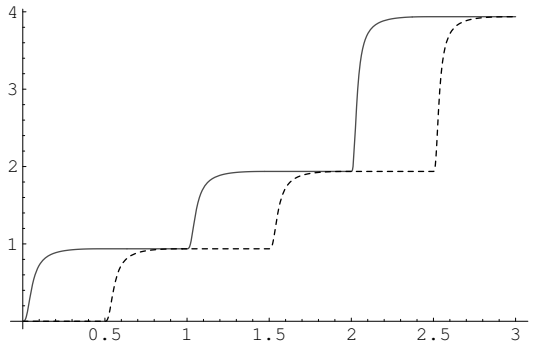

Fig. 3. Simulation of the iteration of the map $f(n) = 2^n$ via ODEs.

transition rule of the TM (note that with a similar construction, we can also iterate vectorial functions with ODEs). In general, if M has l tapes, we can suppose that its transition function is defined over $\mathbb{N}^{2l+1}$: each tape is encoded by 2 integers, plus one integer for the state.

Proposition 5. *Let M be some Turing machine with l tapes and let $x \in \mathbb{N}^{2l+1}$ be the encoding of the initial configuration of M. Then there is an ODE*

$$\begin{array}{ll} z' = g(z,t), & z_i(0) = x_i \text{ for } i \in \{1,...,2l+1\}, \\ & z_i(0) = 0 \text{ otherwise} \end{array} \tag{4}$$

that simulates M as follows: $(z_0(m),...,z_{2l+1}(m)) = f_M^{[m]}(x_1,...,x_{2l+1})$, where $f_M^{[m]}(x_1,...,x_{2l+1})$ gives the configuration of M after m steps. Moreover each component of g can be supposed to be constituted by the composition of polynomials with θ_j's.

3 The result

In this section we present the main result of this article. This result relates computable analysis with the GPAC, showing their equivalence in a certain framework.

Theorem 2 (Main result). *A function $f : [a,b] \to \mathbb{R}$ is computable iff it is GPAC-computable.*

Notice that, by Proposition 2, dynamical systems defined by an ODE of the form $y' = p(t,y)$, where p is a vector of polynomials are in correspondence with GPAC. Then, in a first step, we suppose in the proof that we have access to the function θ_j and refer to the systems

$$y' = p(t, y, \theta_j(y)) \tag{5}$$

where $\theta_j(y)$ means that θ_j is applied componentwise to y, as θ_j-GPACs [9]. Similarly to Def. 4, we can define a notion of θ_j-GPAC-computability. Later, we will see how these functions θ_j's can be suppressed. To prove Theorem 2 we will need to simulate a cyclic sequence of TM computations.

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 arbitrarily 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 GPAC.

Since any Turing machine can be simulated by a θ_j-GPAC [10], then there is also a θ_j-GPAC that can simulate an infinite sequence of computations of a Turing machine M over successive inputs $(k_1,1),(k_2,2),\ldots$, where k_n is some

function of n. Moreover, this θ_j-GPAC can be designed so that it keeps track of the last value computed by M and the corresponding index n, and it "ticks" when M starts a new computation with some input (k_n, n). This is done by adding extra components in Equation (5), which depend on the variables that encode the current configuration of M. More precisely, the following lemma can be proved (see [3]).

Lemma 1. *Let M be a Turing machine with two inputs and two outputs, that halts on all inputs, and let $m : \mathbb{N} \to \mathbb{N}$ be a recursive function. Let $L \in \mathbb{N} - \{0\}$. If (k_n) is a sequence of natural integers*[10] *that satisfies $k_n \leq 2^{m(n)}L$, then there is a θ_j-GPAC*

$$y' = p(t, y, \theta_j(y), u_1, u_2), \tag{6}$$

given $u_1(0) = y_1$, $u_2(0) = 0$, with the following properties:

1. *The θ_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 $u_2(n+1) = n+1$, and so on;*
2. *Three variables of the θ_j-GPAC act as a "memory": they keep the value of the last n where the computation $M(k_n, n)$ was carried out, and the corresponding two outputs;*
3. *There is one variable y_{clock} of the θ_j-GPAC which takes value 1 each time n is incremented, such that if t_n denotes the nth time $y_{clock} = 1$, then for all $k_n \leq 2^{m(n)}L$, $t_{n+1} - t_n \geq t(k_n, n)$.*

3.1 Proof of the "if" direction for Theorem 2

Let $f : [a, b] \to \mathbb{R}$ be 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

$$\begin{aligned} y' &= p(t, y) \\ y(0) &= x \end{aligned}$$

which solution has two components $g : \mathbb{R}^2 \to \mathbb{R}$ and $\varepsilon : \mathbb{R}^2 \to \mathbb{R}$ such that

$$|f(x) - g(x, t)| \leq \varepsilon(x, t) \text{ and } \lim_{t \to \infty} \varepsilon(x, t) = 0$$

From standard error analysis of Euler's algorithm, function g and ε can be computed using Euler's algorithm on ODE $y' = p(t, y)$ up to any given precision 2^{-n}, as soon as we have a bound on the derivative of y. This is provided by the 3rd condition on the Definition 4.

So, given any $n \in \mathbb{N}$, we can determine t^* s.t. $\varepsilon(x, t^*) < 2^{-(n+1)}$ and compute $g(x, t^*)$ with precision $2^{-(n+1)}$. This gives us an approximation of $f(x)$ with precision 2^{-n}.

[10] Notice that (k_n) needs not *a priori* to be a computable sequence of integers.

3.2 Proof of the "only if" direction for Theorem 2

Here we only prove the result for θ_j-GPACs. Indeed, in [10] it was shown how to implement Branicky's construction for simulating Turing machines in GPACs without using θ_j's. The idea is to approximate non-analytic functions with analytic ones, and to control the error committed along the entire simulation. Applying similar techniques, it is possible to remove the θ_j's of the following lemma (see [3] for further details).

Lemma 2. *A function $f : [a, b] \to \mathbb{R}$ computable then it is θ_j-GPAC-computable.*

Proof. By hypothesis, there is an oracle Turing machine M such that for all oracles $(j(n), l(n))_{n\in\mathbb{N}} \rightsquigarrow x \in [a, b]$, the machine outputs a sequence $(z_n) \rightsquigarrow f(x)$, 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 Theorem 1, it is not difficult to conclude that there are computable functions $m : \mathbb{N} \to \mathbb{N}$, abs, $sgn : \mathbb{N}^2 \to \mathbb{N}$ such that given $x \in [a, b]$ and non-negative integers k, n satisfying $\left|k/2^{m(n)} - x\right| < 2^{-m(n)}$, one has

$$\left|(2 \times sgn(k, n) - 1)\, \frac{abs(k, n)}{2^n} - f(x)\right| < \frac{1}{2^n}. \tag{7}$$

Now, given some real x, we would like to design a θ_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 $\left|k/2^{m(n)} - x\right| < 1/2^{m(n)}$;
2. Simulate M to compute $sgn(k, n)$ and $abs(k, n)$;
3. When $sgn(k, n), abs(k, n)$ are obtained, compute

$$(2 \times sgn(k, n) - 1)\, \frac{abs(k, n)}{2^n} \tag{8}$$

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

With Lemma 1, we can implement steps 2, 3, and 4 with a θ_j-GPAC. This GPAC outputs a signal y_{clock} that says each time the computation should be restarted and increases the variable n in step 4. But we still 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: use the integer part function $r : \mathbb{R} \to \mathbb{R}$ defined by

$$r(0) = 0,\ r'(x - 1/4) = c_j\theta_j(-\sin 2\pi x), \tag{9}$$

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. 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 ω_i such that if $\omega_i(t) \neq 0$, then $y_{r_i}(t)$ is an integer (cf Fig. 4). Hence we can get rid of non-integer values with the products $\omega_i y_{r_i}$.

Remember that the Turing machine M can be simulated by an ODE (5)

$$y' = p_{\mathcal{U}}(t, y, \theta_j(y), y_{input(1)}, y_{input(2)}), \tag{10}$$

denoted by $\mathcal{U}$. 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 Fig. 5 suggests. More formally, the GPAC contains three copies, denoted by $\mathcal{U}_0$,$\mathcal{U}_1$ and $\mathcal{U}_2$ of the system (10), each one with external input y_{r_i}, n:

$$\mathcal{U}_i : \qquad Y_i' = p_{\mathcal{U}}(t, Y_i, \theta_j(Y_i), y_{r_i}, n) \qquad i = 0, 1, 2.$$

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 $\mathcal{U}_i$ 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 $\omega_{j,i}(t) = \theta_j(\sin 2\pi(t - i/3))$, for $i = 0, 1, 2$ (*cf.* Fig. 4).

It can be shown that for every $t \in \mathbb{R}$, $\omega_{j,0}(t) + \omega_{j,1}(t) + \omega_{j,2}(t) > 0$ and that if $\omega_{j,i}(t) > 0$, then $y_{r_i}(t)$ is an integer and it fed $\mathcal{U}_i$ with a "good input": $y_{r_i}(t)$ is an integer. Hence, in the weighted sum

$$y_{abs} = \frac{\omega_{j,0}(nx)y_{abs_0} + \omega_{j,1}(nx)y_{abs_1} + \omega_{j,2}(nx)y_{abs_2}}{\omega_{j,0}(nx) + \omega_{j,1}(nx) + \omega_{j,2}(nx)} \tag{11}$$

only the "good outputs" y_{abs_i} are not multiplied by 0 and therefore y_{abs} provide $abs(k, n)$,[11] whatever the value of real variable x is. Replacing abs_i by sgn_i provides in a similar way $sgn(k, n)$.

Then we use an other subsystem of ODEs for the division in equation (8), which provides an approximation y_{approx} of $f(x)$, from $abs(k, n)$ and $sgn(k, n)$, with error bounded by 2^{-n} (this gives the error bound ε). It can be shown that, using the coding of TMs described in [14, 10], we can make the θ_j-GPAC satisfy condition 3 of Definition 4.

Then, to finish the proof of the lemma, we only 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

[11] In reality, it gives a weighted sum of $abs(k, n)$ and $abs(k - 1, n)$. This is because if $nx \in [i, i + 1/6]$, for $i \in \mathbb{Z}$, we have that both $y_{r_0}(nx) = i$ and $y_{r_2}(nx) = i - 1$ gives "good inputs". Nevertheless this is not problematic for our concerns, since this only introduces an error bounded by 2^{-n}.

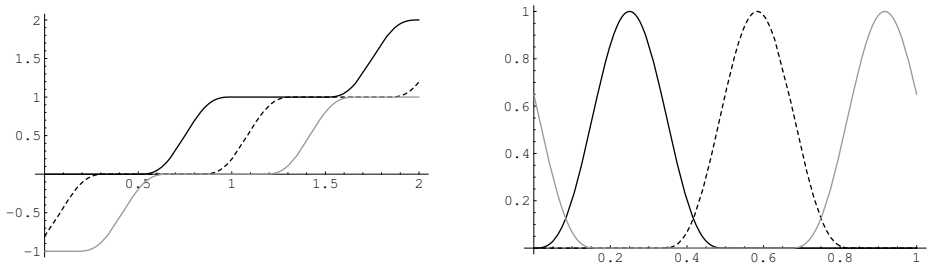

Fig. 4. Graphical representations of functions r_i and ω_i $(i = 0, 1, 2)$.

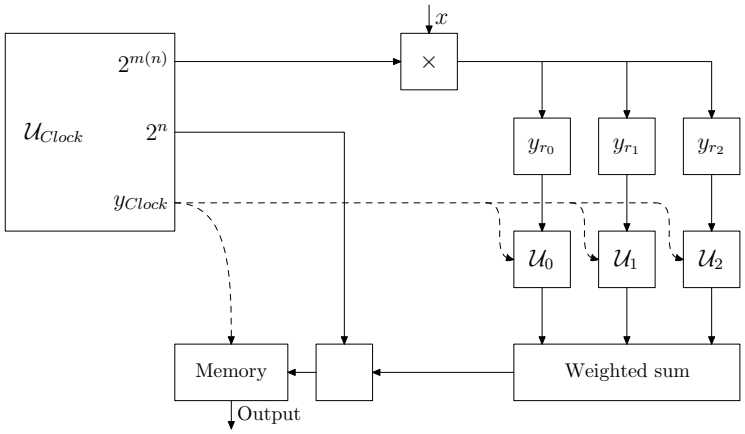

Fig. 5. Schema of a GPAC that calculates a real computable function $f : [a, b] \to \mathbb{R}$. x is the current argument for f, the two outputs of the "weighted sum unit" give $sgn(k, n)$ and $abs(k, n)$. The divisor computes (8). The dotted line gives the signal that orders the memory to store the current value and the other GPACs to restart the computation with the new inputs associated to $n + 1$.

computable analysis, and has only positive arguments. Therefore, by the previous case, g is θ_j-GPAC-computable. Then, to compute f, it is only necessary to use a substitution of variables in the system of ODEs computing g.

We remark that our 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 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 and natural 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 [13], is it possible to relate polynomially-time computable functions to a class of GPAC-computable functions where the error ε 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 the Program *Pessoa* through the project *Calculabilité et complexité des modèles de calculs à temps continu.*

References

1. V. I. Arnold. *Ordinary Differential Equations*. MIT Press, 1978.
2. 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. *Bull. Amer. Math. Soc.*, 21(1):1–46, 1989.
3. O. Bournez, M. L. Campagnolo, D. S. Graça, and E. Hainry. Polynomial differential equations compute all real computable functions. available at `http://www.loria.fr/~hainry/Limits.pdf`.
4. O. Bournez and E. Hainry. Recursive analysis characterized as a class of real recursive functions. to appear in Fund. Inform.
5. M. S. Branicky. Universal computation and other capabilities of hybrid and continuous dynamical systems. *Theoret. Comput. Sci.*, 138(1):67–100, 1995.
6. V. Bush. The differential analyzer. A new machine for solving differential equations. *J. Franklin Inst.*, 212:447–488, 1931.

7. M. L. Campagnolo. *Computational Complexity of Real Valued Recursive Functions and Analog Circuits*. PhD thesis, IST/UTL, 2002.
8. M. L. Campagnolo, C. Moore, and J. F. Costa. Iteration, inequalities, and differentiability in analog computers. *J. Complexity*, 16(4):642–660, 2000.
9. D. S. Graça. Some recent developments on Shannon's General Purpose Analog Computer. *Math. Log. Quart.*, 50(4-5):473–485, 2004.
10. D. S. Graça, M. L. Campagnolo, and J. Buescu. Robust simulations of Turing machines with analytic maps and flows. In S. B. Cooper, B. Löwe, and L. Torenvliet, editors, *CiE 2005: New Computational Paradigms*, LNCS 3526, pages 169–179. Springer, 2005.
11. D. S. Graça and J. F. Costa. Analog computers and recursive functions over the reals. *J. Complexity*, 19(5):644–664, 2003.
12. A. Grzegorczyk. On the definitions of computable real continuous functions. *Fund. Math.*, 44:61–71, 1957.
13. K.-I Ko. *Computational Complexity of Real Functions*. Birkhäuser, 1991.
14. P. Koiran and C. Moore. Closed-form analytic maps in one and two dimensions can simulate universal Turing machines. *Theoret. Comput. Sci.*, 210(1):217–223, 1999.
15. D. Lacombe. Extension de la notion de fonction récursive aux fonctions d'une ou plusieurs variables réelles III. *Comptes Rendus de l'Académie des Sciences Paris*, 241:151–153, 1955.
16. L. Lipshitz and L. A. Rubel. A differentially algebraic replacement theorem, and analog computability. *Proc. Amer. Math. Soc.*, 99(2):367–372, 1987.
17. C. Moore. Recursion theory on the reals and continuous-time computation. *Theoret. Comput. Sci.*, 162:23–44, 1996.
18. J. Mycka and J. F. Costa. Real recursive functions and their hierarchy. *J. Complexity*, 20(6):835–857, 2004.
19. M. B. Pour-El. Abstract computability and its relations to the general purpose analog computer. *Trans. Amer. Math. Soc.*, 199:1–28, 1974.
20. M. B. Pour-El and J. I. Richards. *Computability in Analysis and Physics*. Springer, 1989.
21. L. A. Rubel. A survey of transcendentally transcendental functions. *Amer. Math. Monthly*, 96(9):777–788, 1989.
22. C. E. Shannon. Mathematical theory of the differential analyzer. *J. Math. Phys. MIT*, 20:337–354, 1941.
23. H. T. Siegelmann. *Neural Networks and Analog Computation: Beyond the Turing Limit*. Birkhäuser, 1999.
24. A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.
25. K. Weihrauch. *Computable Analysis: an Introduction*. Springer, 2000.