Abstract geometrical computation and the linear Blum, Shub and Smale model

Jérôme Durand-Lose*

Laboratoire d'Informatique Fondamentale d'Orléans, Université d'Orléans,
B.P. 6759, F-45067 ORLÉANS Cedex 2.

Abstract. Abstract geometrical computation naturally arises as a continuous counterpart of cellular automata. It relies on signals (dimensionless points) traveling at constant speed in a continuous space in continuous time. When signals collide, they are replaced by new signals according to some collision rules. This simple dynamics relies on real numbers with exact precision and is already known to be able to carry out any (discrete) Turing-computation. The Blum, Shub and Smale (BSS) model is famous for computing over $\mathbb{R}$ (considered here as a $\mathbb{R}$ unlimited register machine) by performing algebraic computations.
We prove that signal machines (set of signals and corresponding rules) and the infinite-dimension linear (multiplications are only by constants) BSS machines can simulate one another.

Key-words. Abstract geometrical computation; Analog computation; BSS model; Signal machine.

1 Introduction

There is no agreed analog counterpart of the Church-Turing thesis; relating the models is crucial to understand the differences between the various computing capabilities. For example, Bournez *et al.* related Moore's recursion theory on $\mathbb{R}$ [Moo96], computable analysis [Wei00] and the general purpose analog computer [BH04,BCGH06]. The aim of this paper is to link two analog models of computation. One, abstract geometrical computation deals with regular and automatic drawing on the euclidean plane, while the second, the Blum, Shub and Smale model [BCSS98] relies on algebraic computations over $\mathbb{R}^n$. Let us note that Bournez [Bou99] already provide some relations between linear BSS and Piecewise Constant Derivative systems. The latter also generate Euclidean drawings.

Abstract geometrical computation (ACG) arises from the common use in cellular automata (CA) literature of Euclidean settings to explain an observed dynamics or to design a CA for a particular purpose. But CA operate in discrete time over discrete space, while Euclidean geometry deals with both continuous

* `http://www.univ-orleans.fr/lifo/Members/Jerome.Durand-Lose`, `Jerome.Durand-Lose@univ-orleans.fr`

time and space. This switch of context is justified by the scaling invariance of CA and comes for our preference and ability for thinking in classical continuous terms rather than in discrete terms (for example just think how recent and complex is discrete geometry compared to the Euclidean one). Abstract geometrical computation works in a continuous setting: discrete signals/particles become dimensionless points; the local function of CA, computing the next state of a cell according to the states of neighbouring cells, is replaced by collision rules: which signals emerges from a collision of signals. Signals and rules define *signal machines* (SM). This recent model, even restricted to rational numbers, is able to carry out any (discrete) Turing-computation [DL06c]. With continuous time, Zeno paradox arises: not only are accumulations possible, but they can be used to decide recursively enumerable problems by using the black-hole principle [DL05,DL06a]. Let us note that if accumulations can be generated at will, they can hardly be foreseen [DL06b]. In this paper, we are not interested by accumulations and the super-Turing capability that they bring forth in the discrete computability. We are interested on considering AGC as an analog model, thus there is no rational number restriction and accumulations are not encompassed (they are considered as divergent computations).

In the Blum, Shub and Smale model (BSS), machines computes over any ring. Roughly speaking, polynomial functions can be performed on variables as well as test (according to some order) for branching. Linear BSS [MM97] is the restriction where it is forbidden to multiply two variables, but it is still allowed to multiply by constants. In the case where the dimension of the input is not bounded or the number of registers needed to compute is not bounded, a *shift* operator is provided on order to access any register (finitely many registers hold non zero values since only finitely many registers can be accessed in finite time). We prove the equivalence of AGC and linear BSS over $\mathbb{R}$ in infinite dimension. For the sake of simplicity, we consider that there is no shift operator but indirect addressing through *addresses* (natural counters, the term address is used to distinguish from *real* registers) and that all operations are carried out on an accumulator; this corresponds to the real number unlimited register machine ($\mathbb{R}$-URM) [Nov95] (the arguments for the full BSS translate to the linear case).

To simulate a lin-$\mathbb{R}$-URM with a SM, the value of each register is encoded as the distance between two signals. A lin-$\mathbb{R}$-URM is considered as an assembly language program and we show how to translate each instruction. Since the reader might not be familiar with ACG, the first and simplest constructions are more detailed to provide examples.

To simulate a SM with a lin-$\mathbb{R}$-URM, a configuration is encoded as a finite sequence of, alternatively, signal value and distance to the next one. Although signal machines work with continuous time, the only important discrete dates are when a collision occurs. The simulation goes from a collision date to the next. This is achieved in three steps: compute the next collision time, then update the distances between the signals and finally carry out the collision(s).

Section 2 gives the definition of both models. Section 3 provides the simulation of any lin-$\mathbb{R}$-URM by a SM while Sect. 4 carries the simulation the other way round. Conclusion, remarks and perspective are gathered in Sect. 5.

2 Definitions

Abstract geometrical computations. In this model, dimensionless objects are moving on the real axis. When a collision occurs they are replaces by others. This is defined by the following machines:

Definition 1 A *signal machine* is defined by (M, S, R) where M (*meta-signals*) is a finite set, S (*speeds*) a mapping from M to $\mathbb{R}$ and R (*collision rules*) a partial mapping from the subsets of M of cardinality at least two into subsets of M (speeds must differ in both domain and range).

The elements of M are called *meta-signals*. Each instance of a meta-signal is a *signal*. The mapping S assigns *speeds* to meta-signals. They correspond to the inverse slopes of the segments in space-time diagrams. The *collision rules*, denoted $\rho^- \to \rho^+$, define what emerges (ρ^+) from the collision of two or more signals (ρ^-). Since R is a mapping, signal machines are deterministic. The *extended value set*, V, is the union of M and R plus two symbols: one for void, $\oslash$, and one for an accumulation (which is not addressed here). A *configuration*, c, is a total mapping from $\mathbb{R}$ to V such that the set $\{ x \in \mathbb{R} \mid c(x) \neq \oslash \}$ is finite.

A signal corresponding to a meta-signal μ at a position x, *i.e.* $c(x) = \mu$, is moving uniformly with constant speed $S(\mu)$. A signal must start (resp. end) in the initial (resp. final) configuration or in a collision. These correspond to condition 2 in Def. 2. At a $\rho^- \to \rho^+$ collision signals corresponding to the meta-signals in ρ^- (resp. ρ^+) must end (resp. start) and no other signal should be present (condition 3).

Definition 2 The *space-time diagram* issued from an initial configuration c_0 and lasting for T, is a mapping c from $[0, T]$ to configurations (*i.e.* a mapping from $\mathbb{R} \times [0, T]$ to V) such that, $\forall (x, t) \in \mathbb{R} \times [0, T]$:

1. $\forall t \in [0, T]$, $\{ x \in \mathbb{R} \mid c_t(x) \neq \oslash \}$ is finite,
2. if $c_t(x) = \mu$ then $\exists t_i, t_f \in [0, T]$ with $t_i < t < t_f$ or $0 = t_i = t < t_f$ or $t_i < t = t_f = T$ s.t.:
   - $\forall t' \in (t_i, t_f)$, $c_{t'}(x + S(\mu)(t' - t)) = \mu$,
   - $t_i = 0$ or $c_{t_i}(x_i) \in R$ and $\mu \in (c_{t_i}(x_i))^+$ where $x_i = x + S(\mu)(t_i - t)$,
   - $t_f = T$ or $c_{t_f}(x_f) \in R$ and $\mu \in (c_{t_f}(x_f))^-$ where $x_f = x + S(\mu)(t_f - t)$;
3. if $c_t(x) = \rho^- \to \rho^+ \in R$ then $\exists \varepsilon$, $0 < \varepsilon$, $\forall t' \in [t - \varepsilon, t + \varepsilon] \cap [0, T]$, $\forall x' \in [x - \varepsilon, x + \varepsilon]$,
   - $c_{t'}(x') \in \rho^- \cup \rho^+ \cup \{\oslash\}$,
   - $\forall \mu \in M$, $c_{t'}(x') = \mu \Rightarrow \bigvee \begin{cases} \mu \in \rho^- \text{ and } t' < t \text{ and } x' = x + S(\mu)(t' - t)) \,, \\ \mu \in \rho^+ \text{ and } t < t' \text{ and } x' = x + S(\mu)(t' - t)) \,. \end{cases}$

On space-time diagrams, the traces of signals represent line segments whose directions are defined by $(S(.), 1)$ (1 is the temporal coordinate). Collisions correspond to the extremities of these segments. In the space-time diagrams, time increases upwards.

A configuration is composed of the identities and positions of all the present signals. Since the origin is not relevant (because of the shift invariance), if is enough to have the identities and the distances between signals from left to right. At any time, there are finitely, although unbounded, many signals.

As a computing device, the input is the initial configuration and the output is the final configuration (*e.g.* when no collision can happen).

Linear real number unlimited register machines. We do not use the definition of [BCSS98] (graph with input, output, computation and branch nodes plus a shift node to deal with infinite dimension). Instead, we use the more assembly language like definition of linear $\mathbb{R}$-URM. Each register holds a real number (with exact precision). Inputs as well as outputs are stored in the registers. The machine can add, multiply (by a constant) and copy values. To cope with infinite dimension, address (natural integers) registers are used for indirect addressing. To simplify our constructions we suppose that all real computations are done with one accumulator (which corresponds to a constant slowdown).

Definition 3 A *linear real number unlimited register machine (lin-$\mathbb{R}$-URM)* is described by a sequence of instructions among the following ones:

- $\texttt{inc}\, A_i$, $\texttt{dec}\, A_i$ and $\texttt{if}\ 0{<}A_i\ \texttt{goto}\, n$ for the addresses, and
- $\texttt{load}\, R_i$ $(\texttt{load}\, R_{(i)})$, $\texttt{store}\, R_i$ $(\texttt{store}\, R_{(i)})$, $\texttt{add}\, R_i$ $(\texttt{add}\, R_{(i)})$, $\texttt{mul}\, \alpha$, and $\texttt{if}\ 0{<}X\ \texttt{goto}\, n$ for the registers,

where i is a natural integer, α is a constant real number, n is a line number, A_i is an address, R_i is a register and X is the accumulator. The indirect addressing, $R_{(i)}$, corresponds to R_{A_i}.

The first register has number 0 to avoid any addressing problem. All the operations are done on the accumulator; thus there is no second argument. There is no $\texttt{mul}\, R_i$ since multiplication is only by constant (otherwise it would not be linear). To simplify, there is no $\texttt{add}\, \alpha$, additive constants are supposed to be stored in some registers.

A configuration consists of the line number, the values of the addresses and of the registers. There are finitely many addresses (their number is directly given by the code) and, at any instant, finitely many registers used (but their number is not bounded).

3 Linear $\mathbb{R}$-URM simulation by signal machines

3.1 Encoding a configuration

A lin-$\mathbb{R}$-URM configuration is composed of a line number, n, and values for addresses, $\{A_i\}_{i\in I}$, and registers, $\{R_i\}_{i\in K}$ (I and K are finite initial segments

of $\mathbb{N}$). The set I is a constant of the machine while K may be enlarged during the computation. Since there are finitely many line numbers, each one can be identified by a meta-signal. The rest of the configuration is encoded with speed 0 signals ensuring its stability (parallel signals never interact). Since addresses are only used for indicating registers, they are added as markers on the corresponding registers (again the number of addresses is bounded and as many as needed meta-signals are available from the start).

Registers. A register is encoded as the distance from a base signal to the pairing val signal. There is no absolute scale since signal machines are scale invariant. Two scale signals whose distance amounts for a scale are provided as depicted on Fig. 1. All registers use the same scale. For the value 0, the superposition of base and val is encoded as a single signal nul. This value is never considered in the rest of the paper; the reader is invited to check that it can be easily covered.

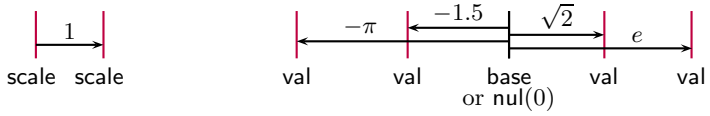


Fig. 1. Encoding: scale and positions of val for values $-\pi$, -1.5, 0, $\sqrt{2}$ and e.

The registers are always encoded with the same meta-signals, base and val. They are regularly displayed as depicted on Fig. 2. The signals base are at a distance, say d, one from the next, such that each val is at distance strictly less than $d/2$ from its corresponding base. Not to tangle one register encoding with another one during the computation, each pair is kept away from the others; if a value becomes too large (which is simple to check since sums and multiplications are done on the accumulator), each distance from val to base is scaled down as well as the scale pair to keep the same values.

The accumulator is encoded like a register and is displayed on the left of the registers. An end marker indicates the right limit of the configuration. It is used in order both to prevent signal from drifting forever on the right and to help enlarging the configuration when needed. The line number is encoded as a signal, line_n, of negative speed which is about to hit the right scale and start the next iteration. A complete encoding is given on Fig. 2.

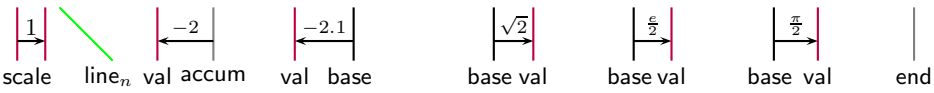


Fig. 2. Scale, line number, accumulator (-2), registers (-2.1, $\sqrt{2}$, $\frac{e}{2}$, $\frac{\pi}{2}$) and end.

Addresses. Since they are used to designate registers, they are encoded by marks on the corresponding base's. This is done by replacing base by any value in

$\{\mathsf{base}_J\}_{J\subseteq I}$ (there are finitely many such meta-signals). Each i of I must appear in exactly one base_J. If needed, dummy null registers are added to cover all the values of the addresses and all registers directly addressed in the code.

3.2 Updating the configuration

The simulation is as follows: line_n bounces on scale and changes to whatever signal is used to carry out the instruction at line n. After the instruction is carried out, extra signals are disposed of and one signal with the new line number is sent to the scale. It remains to deal independently with each possible instruction.

Address manipulations. These are: increasing or decreasing by one and branch if non null. Decreasing corresponds to getting to the base_J holding i and move the element i to the base on the left except when this register is number 0. To achieve this, signal $\overrightarrow{{}_n\mathsf{dec}_i^{0?}}$ is send to the right. The left subscript n is used to record the line number; it is changed when the operation is performed. The superscript 0? indicates that, as far the computation has gone, the value of the address could still be zero. This signal moves to the right leaving every signal as it is until it reaches the first base_J. If i belongs to J then the address is 0 and the signal goes back on the left as line_{n+1} (lower part of Fig. 3); otherwise it turns to $\overrightarrow{{}_n\mathsf{dec}_i}$ and keep going right until it meets the base_J such that i belongs to J. It then turns to $\overleftarrow{{}_n\mathsf{dec}_i}$ and goes on the left; as soon as it reaches a base_J, it replaces it by $\mathsf{base}_{J\cup\{i\}}$ and turns to line_{n+1} (upper part of Fig. 3).

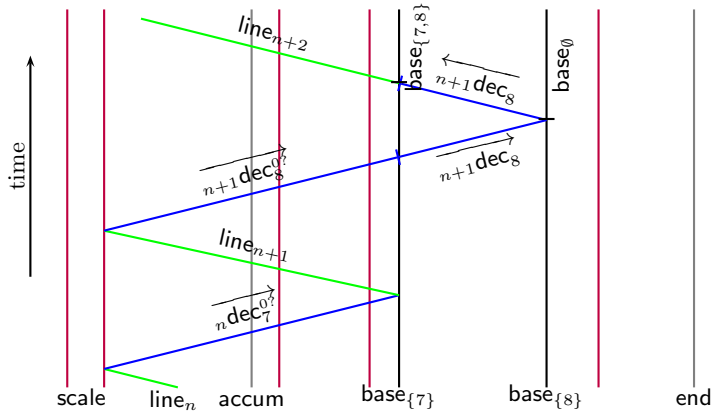


Fig. 3. Example of updating: $\begin{matrix} n\text{: } \mathsf{dec}\ A_7 \\ n{+}1\text{: } \mathsf{dec}\ A_8 \end{matrix}$.

Increasing A_i corresponds to getting to the base_J holding i and move i to the next base on the right (lower part of Fig. 4). When there is no more register on the right, end is reached and used to create a new nul register instead of end and to reposition end one step on the right (upper part of Fig. 4). This time, the

meta-signals used are: $\overrightarrow{{}_n\mathsf{inc}_i^s}$ (searching) and $\overrightarrow{{}_n\mathsf{inc}_i}$ plus two extra signals, $\overleftarrow{\mathsf{end}}$ and $\overrightarrow{\mathsf{end}}$ to regenerate end. The last two meta-signals must be three times faster than $\overrightarrow{{}_n\mathsf{inc}_i}$ in order to ensure the positioning of the new end at the same distance.

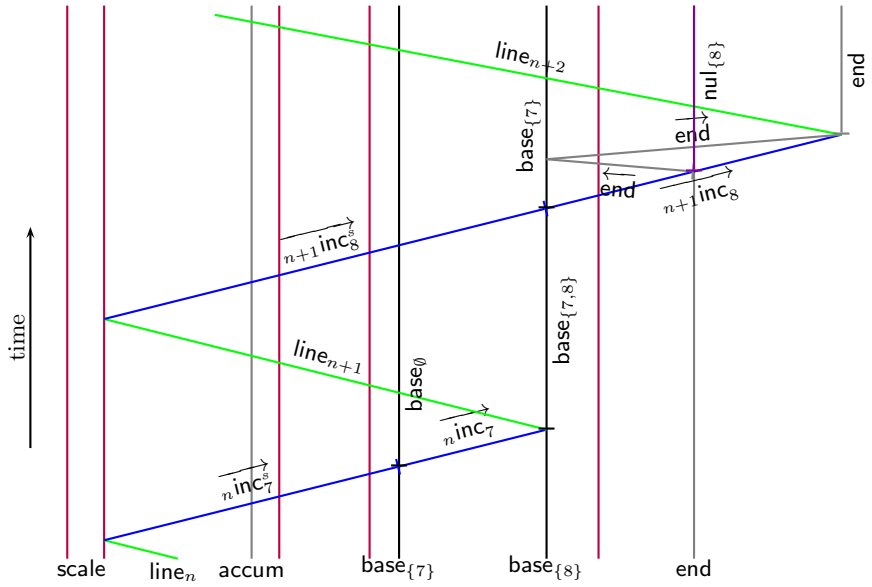


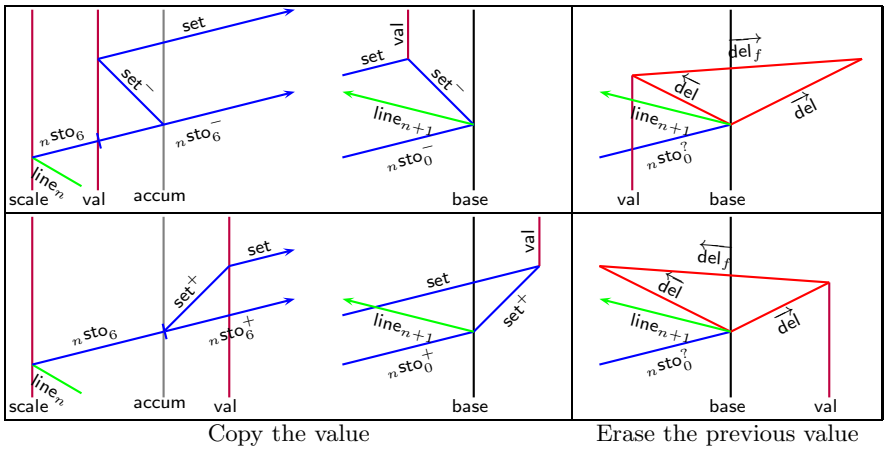
Fig. 4. Example of updating: $\begin{matrix} n\text{: } \texttt{inc}\, A_7 \\ n{+}1\text{: } \texttt{inc}\, A_8 \end{matrix}$.

Branching is very easy, the signal goes on the right, passes the accumulator and get to the first base_J. Depending on whether i belongs to J, it comes back as the following line or the branch line number.

Registers operations. We first present how to move a value from the accumulator to a given register ($\texttt{load}\, R_i$ is similar). The register can be indicated directly or indirectly. In the first case its number is know directly from the code, it is thus possible to make as many as needed meta-signals to count down from i to 0 (meta-signals are used as a finite unary counter). At each base_J crossing, it is decremented until it reaches 0, the designated register is reached. In the second case, a signal is issued that looks for the base_J such that i belongs to J (as in address manipulation).

The first column of Fig. 5 presents how the copy starts from the accumulator and is stored on the corresponding register. The second column shows the erasing of the previous value of the register. The first row deals with negative values and the second row with positive ones (handling 0 is trivial). What happens on the target register is the superposition of the right of left column and right column. There is no risk of collision with the new val since $\overleftarrow{\mathsf{del}}$ and $\overrightarrow{\mathsf{del}}$ are below set^-

and set^+. The value stored is exactly the same since set and ${}_n\mathsf{sto}$ are parallel as well as the pair of set^- (or of set^+).



Copy the value Erase the previous value

Fig. 5. Various cases to achieve `store` R_6.

To handle multiplication, the scale invariance of AGC is used: if starting from two signals at distance 1, they end up at distance α, then starting from two signals at distance d, they end up at distance αd. There are two cases to consider: $\alpha<1$ and $1<\alpha$ (multiplication by 1 is not very interesting). The space-time diagrams and (pre-computed) speeds are given on Fig. 6.

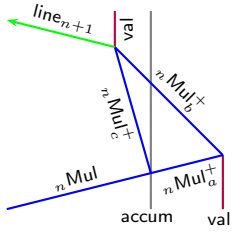
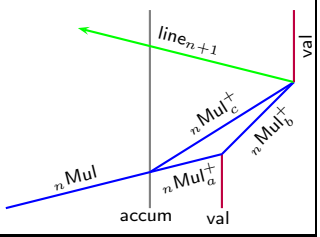
$\alpha < 1$ (here $\alpha = -\frac{1}{2}$)	$1 < \alpha$ (here $\alpha = 2$)
	
Speed of ${}_n\mathsf{Mul}_a^+$: 4	4
Speed of ${}_n\mathsf{Mul}_b^+$: -1	1
Speed of ${}_n\mathsf{Mul}_c^+$: $\frac{4\alpha}{5-4\alpha}$	$\frac{4\alpha}{4\alpha-3}$

Fig. 6. Multiplication scheme.

Let us note that multiplication can produce an overflow: a value so large that it might provoke some entanglement between two registers. This is easy

to detect: two dummy bounding signals are added around accum, whenever the multiplication crosses any of them an overflow flag is raised (*i.e.* some signal switches to a given meta-signal). The multiplication is carried out normally, but the val is replaced by over (to distinguish it from any other val). When an overflow occurs, a special subroutine is launched. It scales down by $1/\alpha$ (for `mul` α with $1<\alpha$) all the registers, the accumulator and the scale. This way, all encoded real values are preserved. All the val and accum remain at their positions, only the left scale, over and all val are moved. The above multiplication scheme is used, this time no overflow can happen.

Addition is not presented, let us just say that it works exactly like load except that val (of the accumulator) is used as origin instead of accum, and deletion of the old val is slightly different. Let us note that addition as an overflow detection like multiplication, in such a case, all is scaled by one half.

4 Signal machines simulation by linear $\mathbb{R}$-URM

This construction is less detailed since it relies on classical construction on register machines and the reader should have now a rather clear picture of what both models are (and also because of the lack of room). A SM configuration consists of an alternating sequence of meta-signals and distances, starting and ending with a meta-signal. This is straight forwardly be translated into a sequence of registers (meta-signals are encoded by integers starting at 1) followed by 0's.

Updating is done in three steps: first find the delay to the next collision, then update the distances and finally process the collision(s) (there might be synchronous ones). Finding the delay is just to go through the sequence and consider signals two by two: compute the delay before next collision (if any) and store the minimum. To achieve this, it loops through the configuration encoding (this is easy with indirect addressing; the loop stops at the first 0 for a meta-signal) and computes the collision delay. For the latter, there is a formula depending on the distance and speeds of meta-signals but it is not linear. Nevertheless, there are finitely many meta-signals and their speeds are known from the signal machine; each case is linear. It only remains to branch to the correct case with a big switch/if then else.

If no collision happens then the machine halts. Otherwise, it "advances" the time by the given duration (*i.e.* the distances are updated) then it goes through the configuration again and process each collision, *i.e.* signals at distance 0. There could be more than two signals involved in a collision (but no more than the number of meta-signals). Again, there are finitely many possible collision rules and they are all given by the signal machine. So to find one, a huge switch has to be provided (there is a case for each rule): first consider how many signals are involved then find the corresponding rule. In-coming signals are replaced by out-going. If their numbers are different, a procedure to compress or enlarge the configuration (exactly like one would do inside any array) is used.

5 Conclusion

We have proved that signal machines are equivalent to lin-$\mathbb{R}$-URM and infinite dimensional linear $\mathbb{R}$-BSS. Let us note that SM restricted to rational speeds and positions are equivalent to lin-$\mathbb{Q}$-URM with the same constructions. The number of rules of the simulating SM is up bounded by an exponential in the number of lines of the lin-$\mathbb{R}$-URM while the number of line of the simulating lin-$\mathbb{R}$-URM is linear in the numbers of collisions and rules.

Considering the number of collisions as a complexity measure on SM, in each case, the slowdown is of the order of the number of signals/register, *i.e.* of *space*. Let us remember that all is constructed with lin-$\mathbb{R}$-URM, not linear BSS and full BSS has also a weak model of complexity [Koi93] so we do not go any further on complexity issues here.

Let us note that reversible and conservative signal machines on rational have full Turing-computability. Would reversible and conservative restrictions already be equivalent to linear BSS? In a rational setting, accumulation was used to climb up the arithmetical hierarchy. We believe that in the real setting, they could be used to provide inner multiplication and thus proved that signal machine could simulate the full BSS model. We believe that in such a case BSS would be a strictly less powerful model (unless some limit operator is provided).

References

[BCGH06] O. Bournez, M. L. Campagnolo, D. S. Graça, and E. Hainry. The general purpose analog computer and computable analysis are two equivalent paradigms of analog computation. In J.-Y. Cai, S. B. Cooper, and A. Li, editors, *Theory and Appliacations of Models of Computations (TAMC '06), Beijing, China*, number 3959 in LNCS, pp. 631–643. Springer, 2006.

[BCSS98] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and real computation*. Springer, New York, 1998.

[BH04] O. Bournez and E. Hainry. An analog characterization of elementarily computable functions over the real numbers. In J. Diaz, J. Karhumäki, A. Lepisto, and D. T. Sannella, editors, *31st Int. Col. on Automata, Languages and Programming (ICALP '04), Turku, Finland*, number 3142 in LNCS, pp. 269–280. Springer, Jul 2004.

[Bou99] O. Bournez. Some bounds on the computational power of piecewise constant derivative systems. *Theory Comput Syst*, 32(1):35–67, 1999.

[DL05] J. Durand-Lose. Abstract geometrical computation for black hole computation (extended abstract). In M. Margenstern, editor, *Machines, Computations, and Universality (MCU '04)*, number 3354 in LNCS, pp. 176–187. Springer, 2005.

[DL06a] J. Durand-Lose. Abstract geometrical computation 1: Embedding black hole computations with rational numbers. *Fund Inf*, 74(4):491–510, 2006.

[DL06b] J. Durand-Lose. Forcasting black holes in abstract geometrical computation is highly unpredictable. In J.-Y. Cai, B. S. Cooper, and A. Li, editors, *Theory and Applications of Models of Computations (TAMC '06)*, number 3959 in LNCS, pp. 644–653. Springer, 2006.

[DL06c] J. Durand-Lose. Reversible conservative rational abstract geometrical computation is Turing-universal. In A. Beckmann and J. V. Tucker, editors, *Logical Approaches to Computational Barriers, 2nd Conf. Computability in Europe (CiE '06)*, number 3988 in LNCS, pp. 163–172. Springer, 2006.

[Koi93] P. Koiran. A weak version of the Blum, Shub & Smale model. In *34th Annual Symp. on Foundations of Computer Science (FOCS '93)*, pp. 486–495. IEEE, 1993.

[MM97] K. Meer and C. Michaux. A survey on real structural complexity theory. *Bulletin of the Belgian Mathematical Society*, 4:113–148, 1997.

[Moo96] C. Moore. Recursion theory on the reals and continuous-time computation. *Theoret Comp Sci*, 162(1):23–44, 1996.

[Nov95] E. Novak. The real number model in numerical analysis. *J. Complex.*, 11(1):57–73, 1995.

[Wei00] K. Weihrauch. *Introduction to computable analysis*. Texts in Theoretical computer science. Springer, Berlin, 2000.