On the computational power of a continuous-space optical model of computation

Thomas J. Naughton and Damien Woods

TASS Research Group,
Department of Computer Science,
National University of Ireland, Maynooth, Ireland.
Email: {tomn, dwoods}@cs.may.ie
URL: http://www.cs.may.ie/TASS

Date: January 2001

Technical Report: NUIM-CS-TR-2001-01

Key words: model of computation, analog computation, real computation, computability, non-Turing computation, continuous space, Type-2 machine, optical information processing, optical computing architectures and algorithms

Abstract

Our continuous-space model of computation was developed for the analysis of analog optical computing architectures and algorithms. We show a lower bound on the computational power of this model by Type-2 machine simulation. We view each Type-2 computation as a sequence of repeated instantiations of a single halting Turing machine. We also illustrate, by example, a problem solvable with our model that is not Type-2 computable.

1 Introduction

In this paper we introduce to the theoretical computer science community a continuous-space model of computation. The model was developed for the analysis of (analog) Fourier optical computing architectures and algorithms, specifically pattern recognition and matrix algebra processors [4]. The functionality of the model is limited to operations routinely performed by optical scientists. The model uses a finite number of two dimensional (2-D) images of finite size and infinite resolution for data storage. The model's data processing unit (finite control) can navigate, copy, and perform other optical operations on its images. A useful analogy would be to describe the model as a random access machine, without conditional branching and with registers that each hold an image of infinite resolution. This model has previously [3, 2] been shown to be at least as computationally powerful as a universal Turing machine (TM). However, its exact computational power has not yet been characterised. To demonstrate a lower bound on computational power we simulate a Type-2 machine with the model. Interestingly, we have evidence that the model can decide at least one language that a Type-2 machine can not. In Sect. 2, we give a more formal introduction to the optical model of computation. In Sect. 3, we outline some relevant points from Type-2 theory of effectivity, and present our working view of Type-2 machines. In Sect. 4, we present our simulation of a Type-2 machine and finish with a discussion and conclusion (Sects. 5 and 6).

2 The optical computational model

Each instance of our machine [3, 2] consists of a memory containing a program (an ordered list of operations) and an input. The memory structure is in the form of a 2-D grid of rectangular elements, as shown in Fig. 1(*a*). The grid has finite size and a scheme to address each element uniquely. Each grid element is a 2-D continuous complex-valued image. Three of these images are represented in the machine by the identifiers **a**, **b**, and **sta** (two global storage locations and a program start location, respectively). The program is a list of instructions for the data processing unit and is stored in memory with the input. The most basic operations available to the programmer, **ld** and **st** (both parameterised by two column addresses and two row addresses), copy rectangular $m \times n$ ($m, n \in \mathbb{N}$, $m, n \geq 1$) subsets of the grid into and out of image **a**, respectively. Upon such loading and storing the image information is rescaled to the full extent of the target location (as depicted in Fig. 1(*b*)). Two additional real-valued parameters z_{lower} and z_{upper}, specifying lower and upper cut-off values, filter the rectangle's contents by amplitude before rescaling,

$$f(i,j) = \begin{cases} z_{\text{lower}} & : \quad \text{Re}\left[f(i,j)\right] < z_{\text{lower}} \\ z_{\text{upper}} & : \quad \text{Re}\left[f(i,j)\right] > z_{\text{upper}} \\ f(i,j) & : \quad \text{otherwise} \end{cases} .$$

Other atomic operations perform horizontal and vertical 1-D Fourier transforms (**h** and **v**, respectively) on the 2-D image **a**, multiply ($\cdot$) **a** by **b** (point by point), perform a complex addition (+) of **a** and **b**, and produce the complex conjugate ($*$) of the image in **a**. By default, the result of any such operation will be found in **a**. Finally, there are two control flow commands **br** and **hlt**, which unconditionally branch to another part of the program, and halt execution, respectively. A more formal specification of the

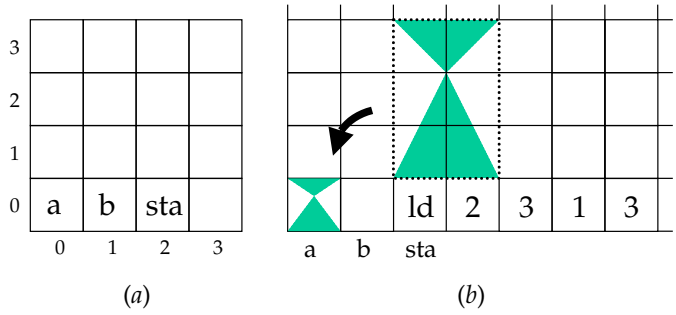

Figure 1: Schematics of (*a*) the grid memory structure of our model of computation, showing example locations for the 'well-known' addresses **a**, **b** and **sta**, and (*b*) loading (and automatically rescaling) a subset of the grid into grid element **a**. The program ld | 2 | 3 | 1 | 3 | . . . | *hlt* instructs the machine to load into default location **a** the portion of the grid addressed by columns 2 through 3 and rows 1 through 3.

model including sample machines and citations to previous optical models can be found in [3, 2].

As might be expected for an analog processor, its programming language does not support comparison of arbitrary image values. Fortunately, not having such a comparison operator will not impede us from simulating a branching instruction (see Sect. 4). In addition, address resolution is possible since (*i*) our set of possible images is finite (each memory grid has a fixed size), and (*ii*) we anticipate no false positives (we will never seek an address not from this finite set). Each syntactically-correct program must form a word in the language defined by the following grammar,

$$
\begin{aligned}
S &\rightarrow MS \mid FS \mid M \mid F \\
M &\rightarrow \text{ld}A \mid \text{st}A \mid \text{br}N;N; \mid \text{hlt} \mid \square \\
F &\rightarrow \text{h} \mid \text{v} \mid * \mid \cdot \mid + \\
A &\rightarrow N;N;N;N;Q;Q; \\
N &\rightarrow ND \mid D \\
Q &\rightarrow N/N \\
D &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \qquad ,
\end{aligned}
$$

where we use capital letters for nonterminals and lowercase letters for terminals. Some explanation of the symbols follows. S, by convention, is the first nonterminal. Memory and control operations, M, and optical processing operations, F, can be combined sequentially. Load (**ld**) and store (**st**) operations address a portion of the memory using two column, two row, and the two real-valued numbers mentioned previously. (In this work, these real-valued numbers are expressed as quotients.) Branching requires row and column coordinates. The optical F operations require no parameters; they act on images **a** and **b** by default. The symbol $\square$ represents an empty or undefined image. Although not part of the programmer's set of operations, empty or undefined grid elements can appear in the absence of a programming symbol. The symbol N denotes an image that encodes a binary number and can be used to specify a row or column of the memory grid. Such an address encoding scheme would have to be determined by the designer of any physical realisation of the theoretical machine. The symbol '$/$' is used to separate the numerator and denominator when specifying the amplitude filter $(z_{\text{lower}}, z_{\text{upper}})$

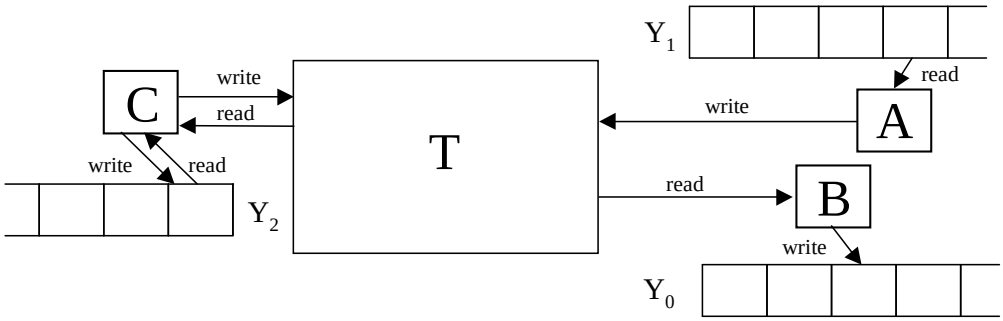

Figure 2: Our working view of a Type-2 machine: (*T*) a halting TM; (Y_0) the output tape; (Y_1) the input tape; (Y_2) the nonvolatile 'work tape'. Controls *A*, *B*, and *C* represent the functionality to read from Y_1, write to Y_0, and read/write to Y_2, respectively.

with rational numbers. The symbol ';' may be required to separate symbols under some encoding schemes.

3 Type-2 Theory of Effectivity

Standard computability theory [6] describes a set of functions that map from one countably infinite set of finite sequences to another. In the "Type-2 Theory of Effectivity" (TTE) [7], "computation" refers to processing over infinite sequences of symbols, that is, infinite input sequences are mapped to infinite output sequences. If we use two or more symbols the set of such sequences is uncountable; TTE describes computation over uncountable sets and their subsets. The following is a definition of a Type-2 machine as taken from [7].

> **Definition 1.** *A Type-2 machine M is a Turing machine with k input tapes together with a type specification* $(Y_1, \ldots, Y_k, Y_0)$ *with* $Y_i \in \{\Sigma^*, \Sigma^\omega\}$*, giving the type for each input tape and the output tape.*

In this definition, Σ is a finite alphabet of two or more symbols, Σ^* is the set of all finite length strings over Σ, Σ^ω is the set of all infinite length strings over Σ. There are two possible input/output tape types, one holds words from Σ^* and the other from Σ^ω.

Input tapes are one-way read only and the output tape is one-way write only. If the output tape restriction was not in place any part of an infinite output would not be guaranteed to be fixed as it could possibly be overwritten at a future time. Hence, finite outputs from Type-2 computations are useful for approximation or in the simulation of possibly infinite processes. A Type-2 machine either finishes its computation in finite time with a finite number of symbols on its output tape, or computes forever writing an infinite sequence. Machines that compute forever while outputting only a finite number of symbols are undefined in Type-2 theory [7].

3.1 A new view of Type-2 computations

We maintain that a Type-2 machine can be viewed as a repeatedly instantiated halting TM that has an additional (read-only) input tape Y_1 and an additional (write-only) output tape Y_0. (Without loss of generality, the finite number of input tapes from Def. 1 can be

mapped to a single tape.) The machine also has a nonvolatile 'work tape' Y_2 that stores symbols between repeated instantiations ('runs') of the halting TM. This is illustrated in Fig. 2. *T* is the halting TM. Control *A* represents the functionality to read from Y_1. Control *B* represents the functionality to write to Y_0. Control *C* represents the functionality to write to and read from Y_2.

A Type-2 computation will proceed as follows. *T* is instantiated at its initial state with blank internal tape(s). It reads a symbol from Y_1. Combining this with symbols (if any) left on Y_2 by the previous instantiations, it can, if required, write symbols on Y_2 and Y_0. *T* then halts, is instantiated once more with blank internal tape(s), and the iteration continues. The computation will either terminate with a finite sequence of symbols on the output tape or compute forever writing out an infinite sequence. In this light, an infinite Type-2 machine computation corresponds to an infinite sequence of instantiations of a single halting TM (plus extra computation steps for controls *A*, *B*, and *C*).

4 Simulation

We use simulation as a technique to measure computational power. If we can show that machine B can simulate every operation that machine A performs, we can say that B is at least as powerful as A, without ever having to explicitly compare functionality. Universality for our machine has already been proved [3, 2] following Minsky's arithmetization of TMs [1] (representing a TM in terms of quadruples of integers). Four images were used to represent Minsky's four registers, **s**, **m**, **n**, and **z**. Image **s** was the symbol under the TM tape head, image **m** encoded a stack holding all symbols on the tape to the left of the tape head, **n** encoded a stack holding all symbols on the tape to the right of the tape head, and **z** was used for temporary storage. We have since significantly simplified our technique of TM simulation.

In general, a TM could be simulated by a look-up table and the two stacks **m** and **n**, as shown in Fig. 3. A given TM (e.g. Fig. 3(*a*)) is written in the imperative form illustrated in Fig. 3(*b*), where the simulation of state changes and TM tape head movements can be achieved with two stacks and two variables as shown in Fig. 3(*c*).

In order to simulate a stack we previously effected indirect addressing with a combination of program self-modification and direct addressing. We also simulated conditional branching by combining indirect addressing and unconditional branching [3, 2]. This was based on a technique by Rojas [5] that relied on the fact that our set of symbols is finite. Without loss of generality, in our simulation we will restrict ourselves to three possible symbols, '0', '1' and a blank symbol 'b'. Then, the conditional branching instruction "if (α='1') then jump to address X, else jump to Y" is written as the unconditional branching instruction "jump to address α". We are only required to ensure that the code corresponding to addresses X and Y is always at addresses '1' and '0', respectively. In a 2-D memory, multiple such branching instructions are possible. The data in the stack can be encoded as a sequence of images, compressed recursively into a single grid image.

q	s	s'	d	q'
0	0	1	R	1
0	1	0	R	1
0	b	b	R	2
1	b	b	L	3

0: initial state
1: moving left
2: rejecting halt
3: accepting halt

(*a*)

```
q := initial_state;
halt := false;
while (halt = false) {
  select case (q, s) {
    (0,0):  fn(1,R,1);
    (0,1):  fn(0,R,1);
    (0,b):  fn(b,R,2);
    (1,b):  fn(b,L,3);
    else:  halt := true;
  }
}
```

(*b*)

```
void fn(s',d,q') {
  if(d = R) {
    m.push(s');
    s := pop(n);
  } else {
    m.push(s');
    s := pop(m)
  }
  q = q';
}
```

(*c*)

Figure 3: Figure showing (*a*) an example TM table of behaviour. This machine flips the binary value at its tape head and halts in an accepting state. If there is a blank at its tape head it halts in a rejecting state; (*b*) an illustration of how an arbitrary TM table of behaviour might be simulated with pseudocode; (*c*) how one might effect a TM computation step with stacks **m** and **n**.

4.1 Push and pop routines

In Sect. 2 we mentioned the images **a** and **b** that are in every machine. For the push and pop routines, we make use of a third image **c**. When the pop routine, illustrated by the pseudocode below, is called the column number of the intended stack will have already been copied to **a**,

pop:
st(&□) // overwrite four blanks (□) below with symbol in **a**
ld(□ □ r r 0/1 1/1)
st(ab) // rescale contents of **a** over both registers **a**, **b**
st(c)
ld(b)
st(□ □ r r 0/1 1/1)
ld(c)

(All stacks will be located on a 'well-known' row. Therefore, the row number, depicted by 'r' in the routine above, can be hardcoded into the machine.) In the first three statements, we use self-modification to load the contents of the stack into **a** and rescale it over both **a** and **b**. Image **a** now contains the top element and **b** contains the remaining contents of the stack. The top element is temporarily stored in **c**, the contents of the stack stored back in its original location, and the top element returned to **a** before the routine ends.

The following pseudocode for the push routine,

psh:
st(&□)
ld(□ □ r r 0/1 1/1)
st(b)
ld(c)
ld(ab)
st(□ □ r r 0/1 1/1)

is called with the address of the intended stack in **a** and the new element in **c**. The contents of the stack are copied into **b** and the new element moved to **a**. Then both **a**

and **b** are rescaled into one image, pushing the new element onto the top of the stack, and the contents stored back in the stack's original location. We use these push and pop routines to simulate the movement of the TM tape head as illustrated in Fig. 3(*c*).

4.2 Shorthand conventions

To facilitate persons reading and writing programs, a shorthand notation is used. Such notations used in our simulation are summarised in Fig. 5. Note that in this shorthand, instead of having to specify exact addresses, we give images a temporary name (such as 'x1') and refer to the address of that image with the ampersand ('&') character. Expansion from this shorthand to the long-form programming language is a mechanical procedure that could be performed as a 'tidying-up' phase by the programmer or by a preprocessor. Unless otherwise stated, we assume that the bounds on image amplitude values are $z_{\text{MIN}} = 0$ and $z_{\text{MAX}} = 1$. The load and store commands contain $0/1$ $(= 0)$ and $1/1$ $(= 1)$ for their z_{lower} and z_{upper} parameters, respectively, indicating that the complete image is to be accessed. As a convention we use boldface and underlining in program grid elements whose images can be modified by the machine and italics to highlight points of machine termination within the grid.

4.3 Type-2 machine simulation

An arbitrary Type-2 machine is incorporated into our simulation as follows. Firstly, transform the Type-2 machine into a Type-2 machine that operates over our alphabet. Then rewrite the machine to conform to the form shown in Fig 2. For the purposes of this simulation we represent Y_2 with *T*'s internal tape (effectively using the semi-infinite tape to the left of the tape head). When *T* halts it will either be in an accepting or rejecting state. *T*'s accepting state is equivalent to the simulator's initial state (i.e. *T* passes control back to the simulator when it halts). At the simulator's initial state it checks if *T*'s tape head was at a non-blank symbol when *T* halted. If so it writes that symbol to Y_0. All symbols to the left of the tape head (equivalent to the contents of Y_2) will be retained for the next instantiation of *T*. Next, the simulator reads a symbol from Y_1 and writes it on *T*'s tape in the cell being scanned by *T*'s tape head. It then passes control to *T*, by going into *T*'s initial state. If at any time *T* halts in a rejecting state we branch to the simulator's halt state. In Fig. 4, we provide a specific example of a Type-2 machine that flips the bits of its binary input. If the input is an infinite sequence it computes forever, writing out an infinite sequence of flipped bits. If the input is finite it outputs a finite sequence of flipped bits.

4.4 Explanation of Fig. 4

The simulation by our model is shown in Fig. 4. It consists of two parts (separated in the diagram for clarity). The larger is the simulator (consisting of functionality *A*, *B*, and *C* from Fig. 2, stacks Y_1 and Y_0, and a universal TM). A TM table of behaviour must be inserted into this simulator [the example TM is that from Fig. 3(*a*)]. It has a straightforward encoding. Notice how for each row of the table of behaviour $\langle q, s, s', d, q' \rangle$ an ordered triple $\langle s', d, q' \rangle$ is placed at the location addressed by the coordinates (q, s). Stacks Y_1 and Y_0 represent the one-way tapes from Fig. 2. We pop from Y_1 and push to Y_0. The stack **m** encodes all symbols on *T*'s tape to the left of tape head, and the

Figure 4: Simulating Type-2 machines on our model of computation. The machine is in two parts for clarity. The larger is a universal Type-2 machine simulator and the smaller is its halting TM table of behaviour. [The example TM we use here is that in Fig. 3(*a*).] The simulator is written in a compact shorthand notation. The expansions into sequences of atomic operations are shown in Fig. 5.

(a)

Pp	Y_1	→	ld	Y_1	br	pop		
Pp	m	→	ld	m	br	pop		
Pp	n	→	ld	n	br	pop		
Ph	m	→	st	c	ld	m	br	psh
Ph	n	→	st	c	ld	n	br	psh
Ph	Y_0	→	st	c	ld	Y_0	br	psh

(b)

br	q0	*s	→	ld	s	st	&y1	br	q0	**y1**
br	0	*s	→	ld	s	st	&y2	br	0	**y2**

(c)

‘b’	R	q2	→	ld	‘b’	br	mvr	br	q2	*s
‘0’	R	q1	→	ld	‘0’	br	mvr	br	q1	*s
‘1’	R	q1	→	ld	‘1’	br	mvr	br	q1	*s
‘b’	L	q3	→	ld	‘b’	br	mvl	br	q3	*s

(d)

ld	Y_1	→	ld	5	5	99	99	0	/	1	1	/	1
ld	Y_0	→	ld	7	7	99	99	0	/	1	1	/	1
ld	m	→	ld	0	0	99	99	0	/	1	1	/	1
ld	n	→	ld	3	3	99	99	0	/	1	1	/	1
st	Y_1	→	st	5	5	99	99	0	/	1	1	/	1
st	Y_0	→	st	7	7	99	99	0	/	1	1	/	1
st	m	→	st	0	0	99	99	0	/	1	1	/	1
st	n	→	st	3	3	99	99	0	/	1	1	/	1
ld	s	→	ld	2	2	99	99	0	/	1	1	/	1
ld	b	→	ld	21	21	99	99	0	/	1	1	/	1
ld	c	→	ld	22	22	99	99	0	/	1	1	/	1
st	s	→	st	2	2	99	99	0	/	1	1	/	1
st	b	→	st	21	21	99	99	0	/	1	1	/	1
st	c	→	st	22	22	99	99	0	/	1	1	/	1
ld	ab	→	ld	20	21	99	99	0	/	1	1	/	1
st	ab	→	st	20	21	99	99	0	/	1	1	/	1
ld	‘0’	→	ld	9	9	99	99	0	/	1	1	/	1
ld	‘1’	→	ld	10	10	99	99	0	/	1	1	/	1
ld	‘b’	→	ld	11	11	99	99	0	/	1	1	/	1

(e)

br	pop	→	br	0	8
br	psh	→	br	0	7
br	mvr	→	br	0	6
br	mvl	→	br	0	5
br	*acc*	→	br	0	4
br	*rej*	→	br	0	3

(f)

st	&x1	→	st	9	9	8	8	0	/	1	1	/	1	
ld	**x1**	**x2**	→	ld	**x1**	**x2**	99	99	0	/	1	1	/	1
st	**x3**	**x4**	→	st	**x3**	**x4**	99	99	0	/	1	1	/	1

Figure 5: Time-saving shorthand conventions when programming the model of computation. These specific examples from the simulator in Fig. 4 serve to illustrate the idea. (*a*) Setting up calls to the **psh** and **pop** routines (loading the appropriate stack address into **a** in advance of a pop and loading the stack address into **a** and storing the appropriate symbol in **c** in advance of a push). In advance of a **psh** the element to be pushed will be in **a**. After a **pop**, the popped element will be in **a**. (*b*) Branching to an address where the row is specified by the symbol currently scanned by the tape head. (*c*) Calling the tape head movement routines (simulating the execution of a row of the TM table of behaviour). (*d*) Loading from and storing to locations specified at runtime, and to/from the ‘well-known’ locations on row 99. (*e*) Branching to subroutines. (*f*) All labels are eventually given absolute addresses by a preprocessor. After a first pass of the preprocessor (expanding the shorthand) the modifiable references are updated with hardcoded addresses.

stack **n** encodes all symbols on *T*'s tape to the right of the tape head. Image **s** encodes the symbol currently being scanned by *T*'s tape head. Execution of the simulator begins with an input encoded in stack Y_1 [grid element $(6, 99)$] and the finite control pointing at **sta** [grid element $(17, 99)$]. Control flow always proceeds from left to right unless one of **br** or **hlt** is encountered. The stacks are indirectly addressed to allow push and pop operations to take a stack as an argument (e.g. the image at address labelled **m** contains the column number of stack **m**). The simulation is written in a shorthand whose long form is given in Fig. 5.

5 Discussion

Type-2 machines do not describe all of the computational capabilities of our model. The model's atomic operations operate on a continuum of values in constant time (independent of input size) and would not have obvious TM or Type-2 machine implementations.

Consider the language L defined by the following characteristic function $f : \Sigma^\omega \to \{0, 1\}$, where

$$f(p) := \begin{cases} 1 & : \quad \text{if } p \neq 0^\omega \\ 0 & : \quad \text{otherwise} \end{cases}$$

and where p is an infinite sequence over alphabet $\{0, 1\}$. This language is acceptable but not decidable by a Type-2 machine [7] (Ex. 2.1.4.6). In our model, we encode a boolean value in an image by letting a δ-function at its origin denote a '1' and an empty image (or an image with low background noise) denote a '0'. An infinite sequence of boolean-valued images appear concatenated together in one image without loss of information (by definition, images in our machine have infinite spatial resolution). An off-centre peak can be centred for easy detection through Fourier transformation (using the following shorthand program ld Y_1 h v st b $*$ $\cdot$). This uses the property that the term at the origin of a Fourier transform of an image, the dc term, has a value proportional to the energy over the entire image. Our model could Fourier transform the continuous input image and measure the value of the dc term in unit time. A peak would indicate that there is some energy (and therefore at least one '1') somewhere in the image; the corresponding word is in L. An absence of a peak at the origin indicates that there is not a '1' in the image; the corresponding word is not in L.

6 Conclusion

In a previous paper [2] a method for arbitrary TM simulation was shown. Here, we have shown how our model can simulate any Type-2 machine. TMs and Type-2 machines are both infinite state machines; however, Type-2 computability theory is an extension of standard Turing computability theory. In this theory "computation" includes mappings between infinite sequences. In order to make the jump from TM to Type-2 machine simulation we refined our TM simulator, making it efficient, and allowed it to make better use of our underlying model by permitting infinite input sequences. We were not required to alter the underlying model (even when deciding a Type-2 undecidable language) and it remains a faithful interpretation of Fourier optical information processing architectures. This motivates further research into the exact characterisation of the computational power of our model.

Acknowledgements

We gratefully acknowledge advice and assistance from J. Paul Gibson and the Theoretical Aspects of Software Systems group, NUI Maynooth.

References

[1] Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Series in Automatic Computation. Prentice Hall, Englewood Cliffs, New Jersey, 1967.

[2] Thomas J. Naughton. Continuous-space model of computation is Turing universal. In Sunny Bains and Leo J. Irakliotis, editors, *Critical Technologies for the Future of Computing*, Proceedings of SPIE vol. 4109, San Diego, California, August 2000. To appear.

[3] Thomas J. Naughton. A model of computation for Fourier optical processors. In Roger A. Lessard and Tigran Galstian, editors, *Optics in Computing 2000*, Proceedings of SPIE vol. 4089, pages 24–34, Quebec, Canada, June 2000.

[4] Thomas Naughton, Zohreh Javadpour, John Keating, Miloš Klíma, and Jiří Rott. General-purpose acousto-optic connectionist processor. *Optical Engineering*, 38(7):1170–1177, July 1999.

[5] Raúl Rojas. Conditional branching is not necessary for universal computation in von Neumann computers. *Journal of Universal Computer Science*, 2(11):756–768, 1996.

[6] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society ser. 2*, 42(2):230–265, 1936. Correction in vol. 43, pp. 544–546, 1937.

[7] Klaus Weihrauch. *Computable Analysis*. Springer, Berlin, 2000.