Continuous-space model of computation is Turing universal

Thomas J. Naughton

Department of Computer Science, National University of Ireland/Maynooth, Maynooth, Ireland

ABSTRACT

Our model of computation (theoretical machine) was designed for the analysis of analog Fourier optical processors, its basic data unit being a continuous image of unbounded resolution. In this paper, we demonstrate the universality of our machine by presenting a framework for arbitrary Turing machine simulation. Computational complexity benefits are also demonstrated by providing a $O(\log_2 n)$ algorithm for a search problem that has a lower bound of $n-1$ on a Turing machine.

Keywords: analog computation, real computation, optical information processing, computability, computational complexity, models of computation, Fourier transform processor, general-purpose optical computer

1. INTRODUCTION

As optical information processing architectures and algorithms become more widespread and sophisticated, the need for formal mathematical tools to compare them will increase. The ability of Fourier processors to take constant-time Fourier transforms and constant-time image multiplication (but little else) requires us to construct a specialized model of computation. We recently presented a novel and simple theoretical model of computation[1] that captures what we believe are the most important characteristics of an optical Fourier transform processor. Our theoretical machine was shown to be suitable for the representation (and subsequent analysis) of a range of Fourier optical information processing algorithms (including matched filtering[2] and joint transform correlation[3]).

In our idealistic model (summarized in Sect. 2), the basic data unit is a continuous image of unbounded resolution. The model has image copying functionality, constant-time Fourier transformation/multiplication/complex conjugation, and minimal flow control in the form of unconditional branching (`goto`). Since this very restricted instruction language has neither conditional branching (`if`) nor iteration (`for`, `while`) we have been able to argue that algorithms describable with this model should have optical implementations that do not require a digital electronic computer to act as a master unit.

We now use this abstract model to reason about the computational properties of the physical systems it describes. In Sect. 3 we present our framework for simulation of an arbitrary Turing machine[4,5] (TM). The TM is a universal model of computation; its computational power (in terms of the set of functions it can realize) is no less than that of any instruction-based processor (including modern digital electronic computers). In Sect. 4, we look at the computational complexity gains that are achievable with our model for one well known searching problem.

2. SUMMARY OF THE MODEL

Before it became obvious that we needed a new model of computation to analyze the algorithms of our Fourier optical processors,[6,7] we looked at existing models from computer theory[8] and optical information processing literature.[9–12] All were found unsuitable, in the main due to their discrete nature and use of fixed-resolution space.[1]

Each instance of our machine[1] consists of a memory, 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 image. Three of the images are known to the machine by the identifiers **a**, **b**, and **sta** (two global storage locations and a program start location, respectively). The program is stored in memory with the input. The most basic operations available to the programmer, **ld** and **st** (both parameterized by two column addresses and two row addresses), copy rectangular $m \times n$ subsets of the grid ($m, n \in \mathbb{N}$, $m, n \geq 1$) 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

Further author information. Fax: +353-1-7083848 Email: `tomn@cs.may.ie`

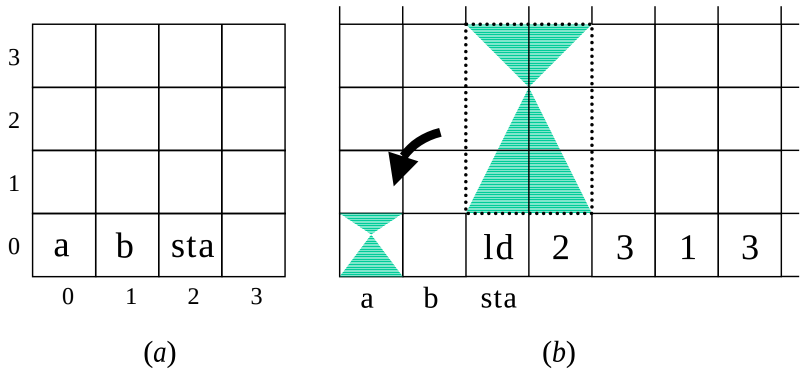

Figure 1. Schematics of (a) the grid memory structure of our theoretical machine, showing possible locations for the 'well known' addresses **a**, **b** and **sta**, and (b) loading (and automatically rescaling) a subset of the grid into grid element **a**. The program $\boxed{\text{ld}}\boxed{2}\boxed{3}\boxed{1}\boxed{3}\boxed{.\;.\;.}\boxed{\mathit{hlt}}$ 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.

additional real-valued parameters z_{lower} and z_{upper}, specifying upper and lower 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 FTs (**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.

As might be expected for an analog processor, its programming language does not support comparison of arbitrary image values; correlation cannot be used to definitively compare two arbitrary analog values in a finite number of steps. Fortunately, not having such a comparison operator will not impede us from simulating a branching operation (see Sect. 3). In addition, correlation can be used for address resolution 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).

2.1. Grammar for the machine's programming language

There is a restriction on the possible sequences of programming symbols that can appear in any instance of our theoretical machine. Each syntactically-correct program must form a word in the language accepted by whatever compiler or interpreter is employed for an implementation. This language is generated from the following context-free grammar[5] in Backus normal form notation,

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

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 processing operations,

F, can be combined sequentially. Load and store operations address a portion of the memory using two column, two row, and the two real-valued numbers that are expressed here as quotients. Branching requires row and column coordinates. F operations require no parameters; they act on images **a** and **b** by default. The symbol □ 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 and so may be encountered by the preprocessor. The symbol N denotes an image that encodes a natural number and is used to specify a row or column of the memory grid. Such an encoding scheme would have to be determined by the designer of any physical realization of the theoretical machine. The symbol / is used to separate the numerator and denominator when specifying the amplitude filter $(z_{\text{lower}}, z_{\text{upper}})$ with rational numbers. The symbol ; may be required to separate naturals under some encoding schemes.

3. UNIVERSALITY OF THE THEORETICAL MACHINE

We often use a technique called simulation to measure computational power. If we can show that machine B can simulate every operation that A performs, we can say that B is at least as powerful as A, without ever having to explicitly compare functionality. We have already investigated the general-purpose properties of our machine[1] by simulating a standard (albeit non-universal) model of computation from computer theory: the push-down automaton[5] (PDA). A 2-stack PDA (2PDA) is no less powerful than a TM. A 2PDA can simulate any k-tape TM in the following manner. First, convert the k-tape TM into a 1-tape TM. Then, to simulate the movement of the tape head in one direction we pop the top item off one stack and push it onto the other stack. To simulate the movement of the head in the opposite direction we reverse the procedure.

The arithmetization of TMs (representing a TM in terms of quadruples of integers) has been shown by Minsky.[13] Our simulation follows Minsky's approach. We use four images to represent Minsky's four registers, **s**, **m**, **n**, and **z**. Image **s** is the symbol under the TM tape head, image **m** encodes a stack holding all symbols on the tape to the left of the tape head, **n** encodes a stack holding all symbols on the tape to the right of the tape head, and **z** is used for temporary storage.

In order to simulate a stack we needed to effect indirect addressing. In a previous paper[1] we showed how to simulate indirect addressing with a combination of self-modification and direct addressing. We were also able to simulate conditional branching by combining indirect addressing and unconditional branching. This was based on a technique by Rojas[14] that relied on the fact that our set of symbols is finite. Without loss of generality, we will restrict ourselves to only two possible symbols, 0 and 1. Then, the conditional branching instruction "if (α=1) then jump to address X, else jump to Y" is written as 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 then be encoded in an image as a sequence of sub-images, compressed recursively into a single grid element.

3.1. Push and pop routines

For the push and pop routines, we require a third register **c**. When the pop routine is called the column number of the intended stack will have already been copied to **a**,

```
Pop:
  st(&□) // overwrite the four blanks (□) in code below with column number stored at a
  ld(□ □ Y Y 0 1)
  st(ab) // rescale contents of a over both registers a, b
  st(c)
  ld(b)
  st(□ □ Y Y 0 1)
  ld(c)
```

(All stacks will be located on a "well-known" row. Therefore, the row number, depicted by 'Y' in the routine, 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.

```
      z := 0
C0: case pop(m)
            '0': br(C1)
            '1': push(z,'1')
                 push(z,'1')
                 br(C0)
C1: m := z + s
      z := 0
      s := 0
C2: case pop(n)
            '0': br(C3)
            '1': s := 1
                 case pop(n)
                       '0': br(C3)
                       '1': s := 0
                            push(z,'1')
                            br(C2)
C3: n := z
```

(*a*)

```
function Search(input, location)
   // input and location are two images
   if input contains only one sub-image
      return location
   else
      rescale input over a and b
      if FT of a has a peak at the origin
         append symbol '0' to location
         Search(a, location)
      else
         append symbol '1' to location
         Search(b, location)
      end
   end
end Search
```

(*b*)

Table 1. Examples of pseudocode for the machine. (*a*) An algorithm to simulate a move to the right by the TM tape head. Labels C1,C2,C3 are used by the branch (br) instructions. The case commands also branch to one of two possible instructions (labeled '0' and '1') based on the results of a pop operation. (*b*) A recursive algorithm that performs a $O(\log_2 n)$ search on an unsorted list of Boolean values.

The following push routine,

```
Psh:
   st(&□)
   ld(□ □ Y Y 0 1)
   st(b)
   ld(c)
   ld(ab)
   st(□ □ Y Y 0 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 the push and pop routines to simulate the movement of the TM tape head. Table 1(*a*) contains the algorithm for a rightwards movement of the tape head, showing the required symbol updates to **m** and **n**. A movement to the left follows much the same procedure. Both are encoded in rows 2 through 5 of our machine in Fig. 2.

3.2. Shorthand conventions

To facilitate persons reading and writing programs, a shorthand notation is used. This is summarized in Fig. 3. Note that in this shorthand, instead of having to specify exact addresses, we give images a temporary name (such as 't1') 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 values for theoretical machines are $z_{\mathrm{MIN}} = 0$ and $z_{\mathrm{MAX}} = 1$. 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.

		m		n		z		s	‘0’	‘1’	‘2’	‘A’	‘R’	sta			acc			rej			a	b	c
	99	1	**0**	3	**0**	5	**∅**	**0**	0	1	2	1	0	br	qS	0	ld	‘A’	hlt	ld	‘R’	hlt	**∅**	**∅**	**∅**
	:																								
pop	31	st	&x1	st	&x2	st	&x3	st	&x4	ld	**x1**	**x2**	st	ab	st	c	ld	b	st	**x3**	**x4**	ld	c	ret	
psh	30	st	&x5	st	&x6	st	&x7	st	&x8	ld	**x5**	**x6**	st	b	ld	c	ld	ab	st	**x7**	**x8**	ret			
	:																								
mvr	5	ld	m	st	&t1	st	&t2	st	&t3	st	&t4	br	C0	4											
	4	ld	n	st	&t5	st	&t6	st	&t7	st	&t8	st	&t9	st	&tA	br	C0	1							
mvl	3	ld	n	st	&t1	st	&t2	st	&t3	st	&t4	br	C0	2											
	2	ld	m	st	&t5	st	&t6	st	&t7	st	&t8	st	&t9	st	&tA	br	C0	1							
	1	:=	z	‘0’	br	C0	0				Psh	z	‘1’	Psh	z	‘1’	br	C0	0						
	0	Pop	**t1**	**t2**	st	&y1	br	C1	**y1**		br	C5	*s												
		C0									C1														

1	Psh	z	‘1’	br	C6	0																		
0	br	C6	0				:==	**t3**	**t4**	z	:=	z	‘0’	=	s	‘0’	Pop	**t5**	**t6**	st	&y2	br	C3	**y2**
	C5						C6										C2							

1	=	s	‘1’	Pop	**t7**	**t8**	st	&y3	br	C4	**y3**		=	s	‘0’	Psh	z	‘1’	br	C2	0
0	:==	**t9**	**tA**	z	ret								br	C3	0						
	C3												C4								

1					R	‘1’	q0		R	‘1’	q1		L	‘0’	q3		L	‘1’	q3		*br*	*acc*
0	br	q0	s		R	‘1’	q1		L	‘0’	q2		*br*	*rej*			R	‘0’	q4		*br*	*acc*
	qS				q0				q1				q2				q3				q4	

Figure 2. Theoretical machine simulating a Turing machine. The majority of rows 0 and 1 appear below the main body of the simulator (rather than to the right of it) for formatting reasons.

(a)	Pop	**t1**	**t2**	→	ld	**t1**	**t2**	br	pop			
	Psh	z	‘1’	→	ld	‘1’	st	c	ld	z	br	psh

(b)	ld	**t1**	**t2**	→	ld	**t1**	**t2**	99	99	0	/	1	1	/	1
	st	**t1**	**t2**	→	st	**t1**	**t2**	99	99	0	/	1	1	/	1
	ld	z	→	ld	4	4	99	99	0	/	1	1	/	1	
	st	c	→	st	23	23	99	99	0	/	1	1	/	1	
	st	ab	→	st	21	22	99	99	0	/	1	1	/	1	
	ld	‘1’	→	ld	8	8	99	99	0	/	1	1	/	1	

(c)	st	&y1	→	st	&y1	&y1	0	0	0	/	1	1	/	1
			→	st	27	27	0	0	0	/	1	1	/	1
	br	C0	1	→	br	0	1							
	br	C1	**y1**	→	br	29	**y1**							

(d)	br	pop	→	br	0	31
	br	psh	→	br	0	30
	br	*rej*	→	br	18	99

(e)	=	s	‘0’	→	ld	‘0’	st	s								
	:=	z	‘0’	→	ld	z	st	&r1	st	&r2	ld	‘0’	st	**r1**	**r2**	
	:==	**t1**	**t2**	z	→	ld	z	st	&r1	st	&r2	ld	**r1**	**r2**	st	**t1**	**t2**

(f)	R	‘0’	q1	→	=	s	‘0’	br	mvr	br	q1	*s
	L	‘0’	q1	→	=	s	‘0’	br	mvl	br	q1	*s

(g)	br	mvr	→	br	0	5
	br	mvl	→	br	0	3

(h)	br	q1	*s	→	ld	s	st	&y1	br	q1	**y1**

Figure 3. Shorthand conventions when programming the theoretical machine. (a) Push and pop shorthand. (b) Loading from and storing to locations specified at runtime, and “well-known” locations on row 99. (c) All constant references are eventually given absolute addresses by the preprocessor. After the first pass of the preprocessor (expanding the shorthand) the modifiable references are updated with hardcoded addresses. (d) Branching to subroutines. (e) Three types of assignment: direct assignment to an image (=),assignment to the address of an image (:=), and assignment from the address of an image (:==). (f) Calling the tape head movement routines. (g) Hardcoding the addresses of these routines at “well-known” rows 5 and 3. (h) Branching to an address specified by the symbol currently scanned by the tape head.

3.3. Simulation

An arbitrary TM is incorporated into our simulation as follows. First convert the TM into a TM that operates on binary symbols. Then renumber the set of states such that for each row of the table of behavior $\langle q, s, s', q', d\rangle$ an ordered triple $\langle d, s', q'\rangle$ can be placed at the location addressed by (q, s). As an example, the following transducer TM to add two unary numbers is simulated by the machine,

q	s	s'	q'	d
0	1	1	0	R
0	#	1	1	R
1	1	1	1	R
1	#	#	2	L
2	1	#	3	L
3	1	1	3	L
3	#	#	4	R

where q is the current state, s is read symbol, s' is the written symbol, q' is the new state, and d is the direction of tape head movement. State 0 is the initial state and state 4 is the final state. Computation begins with the tape head at the leftmost digit of the first number and the numbers are separated by a single blank. For our simulation we use symbol 0 for the TM's blank symbol #. The simulation is shown in Fig. 2 with the encoding of this particular TM illustrated in lower section of rows 0 and 1.

4. COMPUTATIONAL COMPLEXITY

We demonstrate the computational complexity benefits of our machine through the example of a common search problem. Given a list L and an element $X \in L$, where in L is X positioned? The list has n elements, but we can be sure of nothing more about L (i.e. we must assume it is unsorted). With a comparison based model of computation (where we may only get ordering information about L and X through comparison) the lower bound on the worst cost is $n-1$ comparisons. This is the cost associated with solving the problem on a modern (sequential) digital electronic computer. A more restricted form of this problem searches for the position of a 1 in a list otherwise populated with 0s. This restricted problem is generalizable. The Boolean values could represent keys $C(l_i) = 1$ or $C(l_i) = 0$ indicating whether list element l_i satisfied property C or not, respectively. The problem then becomes a search for the one element that satisfies that property or condition. It also has a lower bound on the worst cost of $n-1$ comparisons for a digital computer.

We encode Boolean values in our machine by letting a δ-function at the origin of an image denote a 1 and an empty image (or image with low background noise) denote a 0. A list of Boolean valued images could be concatenated together in the one image without loss of information (by definition, images in our machine have infinite spatial resolution). Table 1(b) contains the algorithm. The image containing the remainder of the list is rescaled over the two adjacent images **a** and **b**. Searching continues with the image that contains the peak, and the other image is discarded. An off-center peak can be centered for easy detection through Fourier transformation. This uses the fact that the term at the origin of a FT, the dc term, has a value proportional to the energy over the entire image. When the algorithm terminates, the location image will encode the address of the sought element. This address could then be used to locate the original object in the list that this list of Boolean keys was derived from. Analysis of this algorithm shows that it has a time complexity of $O(\log_2 n)$ (measured in list accesses). The address image can also be used to locate the original object in $O(\log_2 n)$ timesteps. We observe that our machine seems to swap time complexity for RESOLUTION complexity[1] (this algorithm requires $O(n)$ RESOLUTION complexity).

The algorithm requires a little polishing. For example, a method to determine that there is only one sub-image in the list, and functionality to take care of situations where there are an odd number of elements in the list. These problems can be overcome since the magnitude of n is known in advance.

Finding a simulation of our machine on a TM (efficiently or otherwise) appears problematic. Since our machine works with continuous images, it is possible that an input could have infinite resolution. Consider the following decision problem. Given a possibly infinite list of Boolean values, is there are least one '1' in the list? By definition, this problem is undecidable on a TM (the halting problem can be reduced to it). In theory, our machine could FT the continuous input image and measure the dc value in unit time. A peak would indicate that there was some energy (and therefore a 1) in the list.

5. CONCLUSION

We have presented a general framework for simulating TMs on our Fourier optical model of computation. Such a result is important for several reasons: (*i*) it shows that analog optical processors are not restricted to being co-processors to digital master units – they are as powerful in terms of the set of functions they compute. (*ii*) a suitable universal model of computation will allow us to analyze, compare, and develop efficient analog optical algorithms. (*iii*) the machine's functionality has been restricted to operations routinely demonstrated optically (image multiplication/convolution/transformation, but no equality testing). This means that, in principle, our machine could be realized physically with optoelectronic hardware. (*iv*) the model admits solutions with interesting complexity and computability gains to problems other than those routinely associated with analog optical information processing.

In proving universality for our theoretical analog optical processor, we investigate if analog optics has anything to offer the mainly digital approach to general-purpose optical computer design. As such, we hope to illuminate another avenue of research in the investigation of whether a primarily-optical general-purpose computer can be built or not.

ACKNOWLEDGMENTS

Many thanks to Damien Woods and Paul Gibson (both at NUI Maynooth) for fruitful discussions.

REFERENCES

1. T. J. Naughton, "A model of computation for Fourier optical processors," in *Optics in Computing 2000*, R. A. Lessard and T. Galstian, eds., *Proc. SPIE* **4089**, pp. 24–34, 2000.
2. A. VanderLugt, "Signal detection by complex spatial filtering," *IEEE Trans. Information Theory* **IT-10**, pp. 139–145, 1964.
3. C. S. Weaver and J. W. Goodman, "A technique for optically convolving two functions," *Appl. Opt.* **5**, pp. 1248–1249, 1966.
4. A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society ser. 2* **42**, pp. 230–265, 1936. Correction in **43**, pp. 544–546, 1937.
5. D. I. A. Cohen, *Introduction to Computer Theory*, Wiley, New York, second ed., 1997.
6. T. Naughton, Z. Javadpour, J. Keating, M. Klíma, and J. Rott, "General-purpose acousto-optic connectionist processor," *Opt. Eng.* **38**, pp. 1170–1177, 1999.
7. T. J. Naughton, M. Klíma, and J. Rott, "Improved joint transform correlator performance through spectral domain thresholding," in *Optical Engineering Society of Ireland/Irish Machine Vision and Image Processing Joint Conference*, D. Vernon, ed., pp. 199–214, 1998.
8. J. C. Shepherdson and H. E. Sturgis, "Computability of recursive functions," *J. ACM* **10**(2), pp. 217–255, 1963.
9. J. H. Reif and A. Tyagi, "Efficient parallel algorithms for optical computing with the discrete Fourier transform (DFT) primitive," *Appl. Opt.* **36**, pp. 7327–7340, 1997.
10. A. Louri and A. Post, "Complexity analysis of optical-computing paradigms," *Appl. Opt.* **31**, pp. 5568–5583, 1992.
11. R. A. Athale, M. W. Haney, J. J. Levy, and G. W. Euliss, "Minimum complexity optical architecture for look-up table computation in the residue number system," in *Optical Technology for Microwave Applications VI and Optoelectronic Signal Processing for Phased-Array Antennas III*, S.-K. Yao and B. M. Hendrickson, eds., *Proc. SPIE* **1703**, pp. 411–418, 1992.
12. A. P. Goutzoulis, "Complexity of residue position-coded lookup table array processors," *Appl. Opt.* **26**, pp. 4823–4831, 1987.
13. M. L. Minsky, *Computation: Finite and Infinite Machines*, Series in Automatic Computation, Prentice Hall, Englewood Cliffs, New Jersey, 1967.
14. R. Rojas, "Conditional branching is not necessary for universal computation in von Neumann computers," *J. Universal Computer Science* **2**(11), pp. 756–768, 1996.