



NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

National University of Ireland, Maynooth
MAYNOOTH, CO. KILDARE, IRELAND.

DEPARTMENT OF COMPUTER SCIENCE
TECHNICAL REPORT SERIES

NUIM-CS-TR-2003-04

Analog recurrent neural network simulation and $\Theta(\log_2 n)$ unordered search
with an optically inspired model of computation

Damien Woods and Thomas J. Naughton

Analog recurrent neural network simulation and $\Theta(\log_2 n)$ unordered search with an optically inspired model of computation

Damien Woods and Thomas J. Naughton

TASS Research Group, Department of Computer Science,
National University of Ireland, Maynooth, Ireland.

Email: {dwoods,tomn}@cs.may.ie

Date: 26 March 2003

Technical Report: NUIM-CS-TR2003-04

Key words: continuous space machine, unconventional model of computation, analog computation, optical computing, computability, computational complexity, analog recurrent neural network, Fourier transform, binary search, unordered search.

Abstract

We prove computability and complexity results for an original model of computation called the continuous space machine. Our model is inspired by the theory of Fourier optics. We prove our model can simulate analog recurrent neural networks, thus establishing a lower bound on its computational power. We also define a $\Theta(\log_2 n)$ unordered search algorithm with our model.

This report supercedes an older report (NUIM-CS-TR2001-06) dated 03 September 2001.

1 Introduction

In this paper we prove some computability and complexity results for an original continuous space model of computation called the continuous space machine (CSM). The CSM was developed for the analysis of (analog) Fourier optical computing architectures and algorithms, specifically pattern recognition and matrix algebra processors [1,2]. The functionality of the CSM is inspired by operations routinely performed by optical information processing scientists and engineers. The CSM operates in discrete timesteps over a finite number of two dimensional (2D) complex-valued images of finite size and infinite spatial resolution. A finite control is used to traverse, copy, and perform other optical operations on the images. A useful analogy would be to describe the

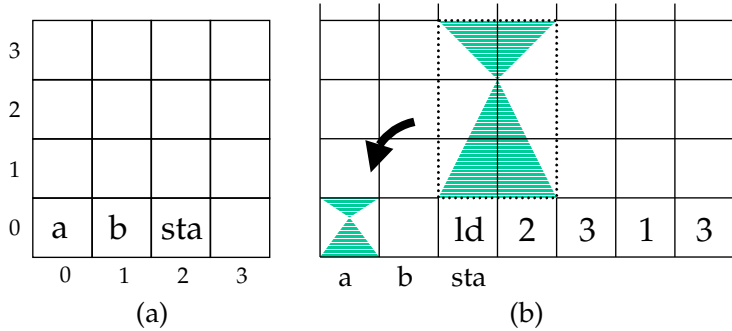


Fig. 1. Schematics of (a) the grid memory structure of the CSM, showing example locations for the ‘well-known’ addresses **a**, **b**, and **sta**, and (b) loading (and automatically rescaling) a subset of the grid into address **a**. The program

ld	2	3	1	3	hit
----	---	---	---	---	-----

 instructs the machine to load into default address **a** the portion of the grid addressed by columns 2 through 3 and rows 1 through 3.

CSM as a random access machine, without conditional branching and with registers that hold continuous complex-valued images. It has recently been established [3,4] that the CSM can simulate Turing machines and Type-2 machines [5]. However, the CSM’s exact computational power has not yet been characterised.

In Sect. 2, we define our optical model of computation and give the data representations that will be used subsequently. In Sect. 3 we demonstrate a lower bound on computational power by proving the CSM can simulate a type of dynamical system called analog recurrent neural networks (ARNNs) [6,7]. This simulation result proves our analog model can decide any language (of finite length words over a finite alphabet) in finite time. In Sect. 4, a $\Theta(\log_2 n)$ binary search algorithm that can be applied to certain unordered search problems is presented.

2 CSM

Each instance of the CSM consists of a memory containing a program (an ordered list of operations) and an input. Informally, the memory structure is in the form of a 2D 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 holds a 2D infinite spatial resolution complex-valued image. There is a program start address **sta** and two well-known addresses labelled **a** and **b**. The model has a number of operations that effect optical image processing tasks. For example, two operations available to the programmer, *st* and *ld* (parameterised by two column addresses and two row addresses), copy rectangular subsets of the grid out of and into image **a**, respectively. Upon such loading and storing the image contents are rescaled to the full extent of the target

location [as depicted in Fig. 1(b)]. The other operations are image Fourier transform, conjugation, multiplication, addition, amplitude thresholding, and some control flow operations. We now give the formal definition of our model.

2.1 CSM definition

Before defining the CSM we define its basic data unit and some of the functions it implements.

Definition 1 (Complex-valued image) *A complex-valued image (or simply, an image) is a function $f : [0, 1] \times [0, 1] \mapsto \mathbb{C}$, where $[0, 1]$ is the real unit interval.*

We let \mathcal{I} be the set of all complex-valued images. We now define six functions that are implemented in six of the CSM's ten operations. Let each $f \in \mathcal{I}$ be parameterised by orthogonal dimensions x and y ; we can indicate this by writing f as $f(x, y)$. The function $h : \mathcal{I} \mapsto \mathcal{I}$ gives the 1D Fourier transformation (in the x -direction) of its 2D argument image f . The function h is defined as

$$h(f(x, y)) = h'(F(\alpha, y)), \quad (1)$$

where $F(\alpha, y)$ is the Fourier transform in the x -direction of $f(x, y)$, defined as [1,2]

$$F(\alpha, y) = \int_{-\infty}^{\infty} f(x, y) \exp[i2\pi\alpha x] dx,$$

where $i = \sqrt{-1}$, and where $h'(F(\alpha, y)) = F(\theta\alpha, y)$. Here, h' uses the constant θ to linearly rescale its argument F so that F is defined over $[0, 1] \times [0, 1]$. The function $v : \mathcal{I} \mapsto \mathcal{I}$ gives the 1D Fourier transformation (in the y -direction) of its 2D argument image f , and is defined as

$$v(f(x, y)) = v'(F(x, \beta)), \quad (2)$$

where $F(x, \beta)$ is the Fourier transform in the y -direction of $f(x, y)$, defined as [1,2]

$$F(x, \beta) = \int_{-\infty}^{\infty} f(x, y) \exp[i2\pi\beta y] dy,$$

and where $v'(F(x, \beta)) = F(x, \theta\beta)$. The function $*$: $\mathcal{I} \mapsto \mathcal{I}$ gives the complex conjugate of its argument image,

$$*(f(x, y)) = f^*(x, y), \quad (3)$$

where f^* denotes the complex conjugate of f . The complex conjugate of a scalar $c = a + ib$ is defined as $c^* = a - ib$. The function \cdot : $\mathcal{I} \times \mathcal{I} \mapsto \mathcal{I}$ gives the pointwise complex product of its two argument images,

$$\cdot(f(x, y), g(x, y)) = f(x, y)g(x, y). \quad (4)$$

The function $+ : \mathcal{I} \times \mathcal{I} \mapsto \mathcal{I}$ gives the pointwise complex sum of its two argument images,

$$+(f(x, y), g(x, y)) = f(x, y) + g(x, y). \quad (5)$$

The function $\rho : \mathcal{I} \times \mathcal{I} \times \mathcal{I} \mapsto \mathcal{I}$ performs amplitude thresholding on its first image argument using its other two real valued $(z_l, z_u : [0, 1] \times [0, 1] \mapsto \mathbb{R})$ image arguments as lower and upper amplitude thresholds, respectively,

$$\rho(f(x, y), z_l(x, y), z_u(x, y)) = \begin{cases} z_l(x, y), & \text{if } |f(x, y)| < z_l(x, y) \\ |f(x, y)|, & \text{if } z_l(x, y) \leq |f(x, y)| \leq z_u(x, y) \\ z_u(x, y), & \text{if } |f(x, y)| > z_u(x, y) \end{cases}. \quad (6)$$

The amplitude of an arbitrary $c \in \mathbb{C}$ is denoted $|c|$ and is defined as $|c| = \sqrt{cc^*}$.

We let \mathbb{N} be the set of non-negative integers and for a given CSM we let \mathcal{N} be a finite set of images that encode that CSM's addresses (see Sect. 2.5 for an example encoding).

Definition 2 (Continuous space machine) *A continuous space machine is a quintuple $M = (D, L, I, P, O)$, where*

$D = (m, n)$, $D \in \mathbb{N} \times \mathbb{N}$: grid dimensions

$L = ((s_\xi, s_\eta), (a_\xi, a_\eta), (b_\xi, b_\eta))$: addresses **sta**, **a**, and **b**

$I = \left\{ (\iota_{1_\xi}, \iota_{1_\eta}), \dots, (\iota_{k_\xi}, \iota_{k_\eta}) \right\}$: addresses of the k input images

$P = \left\{ (\pi_1, p_{1_\xi}, p_{1_\eta}), \dots, (\pi_r, p_{r_\xi}, p_{r_\eta}) \right\}$, $\pi_j \in (\{h, v, *, \cdot, +, \rho, st, ld, br, hlt\} \cup \mathcal{N}) \subset \mathcal{I}$: the r programming symbols and their addresses

$O = \left\{ (o_{1_\xi}, o_{1_\eta}), \dots, (o_{l_\xi}, o_{l_\eta}) \right\}$: addresses of the l output images.

Also, $(s_\xi, s_\eta), (a_\xi, a_\eta), (b_\xi, b_\eta), (\iota_{k'_\xi}, \iota_{k'_\eta}), (p_{r'_\xi}, p_{r'_\eta}), (o_{l'_\xi}, o_{l'_\eta}) \in \{0, \dots, m-1\} \times \{0, \dots, n-1\}$ for all $k'_\xi, k'_\eta \in \{1, \dots, k\}$, $r'_\xi, r'_\eta \in \{1, \dots, r\}$, $l'_\xi, l'_\eta \in \{1, \dots, l\}$.

Addresses whose contents are not specified by P in a CSM definition are assumed to contain the constant image $f(x, y) = 0$.

We adopt a few notational conveniences. For the remainder of the current section, in a given CSM the addresses c and (γ, δ) are both elements from the set $\{0, \dots, m-1\} \times \{0, \dots, n-1\}$, and g, u , and w are sequences of elements from the set $\mathcal{I} \times \{0, \dots, m-1\} \times \{0, \dots, n-1\}$. In a CSM the image at address c is denoted \hat{c} . In the case where \hat{c} represents an integer from $\{0, \dots, |\mathcal{N}|-1\}$, that integer is denoted $\hat{\hat{c}}$.

Definition 3 (CSM configuration) *A configuration of a CSM M is a pair $\langle c, g \rangle$, where $c \in \{0, \dots, m-1\} \times \{0, \dots, n-1\}$ is an address called the control. Also, $g = ((i_{00}, 0, 0), \dots, (i_{m-1, n-1}, m-1, n-1))$ is a $m.n$ -tuple that contains M 's $m.n$ images and each of their addresses, with $i_{\gamma\delta} \in \mathcal{I}$ being the image at address (γ, δ) . The tuple g is ordered by address. Given configuration*

$\langle(\gamma, \delta), g\rangle$, the address of the next image after $i_{\gamma\delta}$ in g is

$$\nu(\gamma, \delta) = \begin{cases} (0, \delta - 1) & \text{if } \gamma = m - 1 \\ (0, n - 1) & \text{if } (\gamma, \delta) = (m - 1, 0) \\ (\gamma + 1, \delta) & \text{otherwise.} \end{cases}$$

The notation $\nu^n(c)$ is shorthand for n recursive applications of ν to c , e.g. $\nu^2(c) = \nu(\nu(c))$. An *initial configuration* of M is a configuration $C_s = \langle c_s, g_s \rangle$, where $c_s = (s_\xi, s_\eta)$ is the address of **sta**, and g_s contains all elements of P and elements $(\varphi_1, \iota_{1\xi}, \iota_{1\eta}), \dots, (\varphi_k, \iota_{k\xi}, \iota_{k\eta})$ (the k input images at the addresses given by I). A *final configuration* of M is a configuration of the form $\widehat{C}_h = \langle(\gamma, \delta), (u, (hlt, \gamma, \delta), w)\rangle$, where u and w are given above. Notice that $\widehat{(\gamma, \delta)} = hlt$.

In Def. 4 we adopt the following notations. At a given configuration $\langle c, g \rangle$ we let $p_n = \widehat{\widehat{\nu^n(c)}}$, (p_n represents the integer encoded in the image at address $\nu^n(c)$). We let the scaling relationships for st and ld be $x' = (x + \gamma - p_1)/(p_2 - p_1 + 1)$ and $y' = (y + \delta - p_3)/(p_4 - p_3 + 1)$. We let $a(x, y)$ be the image stored in address **a**. Finally, the symbol \square denotes an arbitrary image that is not a programming symbol; $\square \in \mathcal{I}$, $\square \notin \{h, v, *, \cdot, +, \rho, st, ld, br, hlt\}$. Recall that (a_ξ, a_η) is the address of **a**.

Definition 4 (\vdash_M) *Let \vdash_M be a binary relation on configurations of CSM M containing exactly the following eleven elements.*

$$\begin{aligned} \langle c, (u, (i_{a_\xi a_\eta}, a_\xi, a_\eta), w) \rangle \vdash_M \langle \nu(c), (u, (h(i_{a_\xi a_\eta}), a_\xi, a_\eta), w) \rangle, & \text{ if } \widehat{c} = h \\ \langle c, (u, (i_{a_\xi a_\eta}, a_\xi, a_\eta), w) \rangle \vdash_M \langle \nu(c), (u, (v(i_{a_\xi a_\eta}), a_\xi, a_\eta), w) \rangle, & \text{ if } \widehat{c} = v \\ \langle c, (u, (i_{a_\xi a_\eta}, a_\xi, a_\eta), w) \rangle \vdash_M \langle \nu(c), (u, (* (i_{a_\xi a_\eta}), a_\xi, a_\eta), w) \rangle, & \text{ if } \widehat{c} = * \\ \langle c, (u, (i_{a_\xi a_\eta}, a_\xi, a_\eta), w) \rangle \vdash_M \langle \nu(c), (u, (\cdot (i_{a_\xi a_\eta}, i_{b_\xi b_\eta}), a_\xi, a_\eta), w) \rangle, & \text{ if } \widehat{c} = \cdot \\ \langle c, (u, (i_{a_\xi a_\eta}, a_\xi, a_\eta), w) \rangle \vdash_M \langle \nu(c), (u, (+ (i_{a_\xi a_\eta}, i_{b_\xi b_\eta}), a_\xi, a_\eta), w) \rangle, & \text{ if } \widehat{c} = + \\ \langle c, (u, (i_{a_\xi a_\eta}, a_\xi, a_\eta), w) \rangle \vdash_M \langle \nu(c), (u, (\rho(i_{a_\xi a_\eta}, \widehat{\nu(c)}, \widehat{\nu^2(c)}), a_\xi, a_\eta), w) \rangle, & \text{ if } \widehat{c} = \rho \end{aligned}$$

$$\begin{aligned} \langle c, (u_{\gamma\delta}, (i_{\gamma\delta}(x, y), \gamma, \delta), w_{\gamma\delta}) \rangle \vdash_M \langle \nu^5(c), (u_{\gamma\delta}, (a(x', y'), \gamma, \delta), w_{\gamma\delta}) \rangle, \\ \forall \gamma, \delta \text{ s.t. } p_1 \leq \gamma \leq p_2, p_3 \leq \delta \leq p_4, \forall (x, y) \in [0, 1] \times [0, 1], & \text{ if } \widehat{c} = st \end{aligned}$$

$$\begin{aligned} \langle c, (u, (a(x', y'), a_\xi, a_\eta), w) \rangle \vdash_M \langle \nu^5(c), (u, (i_{\gamma\delta}(x, y), a_\xi, a_\eta), w) \rangle, \\ \forall \gamma, \delta \text{ s.t. } p_1 \leq \gamma \leq p_2, p_3 \leq \delta \leq p_4, \forall (x, y) \in (0, 1] \times (0, 1], & \text{ if } \widehat{c} = ld \end{aligned}$$

$$\begin{aligned} \langle c, (u) \rangle \vdash_M \langle (\widehat{\widehat{\nu(c)}}, \widehat{\widehat{\nu^2(c)}}), (u) \rangle, & \text{ if } \widehat{c} = br \\ \langle c, (u) \rangle \vdash_M \langle c, (u) \rangle, & \text{ if } \widehat{c} = hlt \\ \langle c, (u) \rangle \vdash_M \langle \nu(c), (u) \rangle, & \text{ if } \widehat{c} = \square \end{aligned}$$

The first six elements of \vdash_M define the CSM's implementation of the functions defined in Eqs. (1) through (6). Notice that in each case the image at well-known address \mathbf{a} is overwritten by the result of applying one of $h, v, *, \cdot, +$ or ρ to its argument (or arguments). The value of the control c is then simply incremented to the next address, as defined in Def. 3. The seventh element of \vdash_M defines how the store operation copies the image at well-known address \mathbf{a} to a 'rectangle' of images specified by the st parameters p_1, \dots, p_4 . The eighth element of \vdash_M defines how the load operation copies a rectangle of images specified by the ld parameters p_1, \dots, p_4 to the image at well-known address \mathbf{a} . In this element of \vdash_M we restrict both x and y to have values from the interval $(0, 1]$ to resolve ambiguity when loading adjacent images. Elements nine, ten, and eleven of \vdash_M define three control flow operations: branch, halt, and ignore. When the image at the address specified by the control c is br , the value of c is updated to the address encoded by the two br parameters. The hlt operation always maps a final configuration to itself. Finally, if the image at the address specified by the control c is \square (that is, it is not an element of $\{h, v, *, \cdot, +, \rho, st, ld, br, hlt\}$) then c is simply incremented to the next address, as defined in Def. 3.

For convenience, we use an informal 'grid' notation when specifying programs for the CSM, see for example Fig. 1. In our grid notation the first and second elements of an address tuple refer to the horizontal and vertical axes of the grid, respectively, image $(0, 0)$ is at the bottom left-hand corner of the grid. The images in a grid must have the same orientation as the grid. Hence in a given image i , the first and second elements of an address tuple refer to the horizontal and vertical axes of i , respectively, and the coordinate $(0, 0)$ is located at the bottom left-hand corner of i . Figure 2 informally explains the elements of \vdash_M , as they appear in this grid notation.

Let \vdash_M^* denote the reflexive and transitive closure of \vdash_M . A halting computation by M is a finite sequence of configurations beginning in an initial configuration and ending in a final configuration: $C_s \vdash_M^* C_h$. After giving some data representations in Sect. 2.3 we will then define language decision by CSM. First we give some complexity measures.

2.2 Complexity measures

Computational complexity measures are used to analyse CSM instances. The TIME complexity of a CSM computation is the number of configurations in that computation sequence. The SPACE complexity of a CSM M 's computation is the number of image elements in M 's grid. The SPACE complexity of a CSM computation is always constant. The RESOLUTION of an image is a measure of the spatial compression of the image relative to some specified unit image.

The RESOLUTION of a CSM computation is the maximum RESOLUTION of the grid images in that computation sequence. In this paper unary symbol images, binary symbol images, and real number images (see Sect. 2.3) each have unit RESOLUTION. A final measure is RANGE. The RANGE complexity of an image $f : [0, 1] \times [0, 1] \mapsto f' \subseteq \mathbb{C}$ is the number of bits required to represent the values in the set f' . In signal/image processing terms it corresponds to the quantisation and dynamic range of a signal/image. The only two values for RANGE complexity of interest in this paper are constant RANGE and infinite RANGE. Unary and binary images (see Sect. 2.3) have constant RANGE of 1. An image with a value $f(x, y) = c$, where c is an arbitrary complex value, would have infinite RANGE complexity (real number images defined in Sect. 2.3 have infinite RANGE complexity). The RANGE complexity of a CSM computation is the maximum RANGE of the images in that configuration sequence.

2.3 Representing data as images

Let $\Sigma = \{0, 1\}$, let $\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$, let $\Sigma^+ = \bigcup_{i=1}^{\infty} \Sigma^i$ and, unless otherwise stated, let a language $L \subseteq \Sigma^+$. There are many ways to represent elements of finite, countable, and uncountable sets as images. We give a number of techniques that will be used later in the paper. The symbol 1 is represented by an image having value 1 at its centre and value 0 everywhere else. An image that has value 0 everywhere represents the symbol 0.

Definition 5 (Binary symbol image) *The symbol $\psi \in \Sigma$ is represented by the binary symbol image f_ψ ,*

$$f_\psi(x, y) = \begin{cases} 1, & \text{if } x, y = 0.5, \psi = 1 \\ 0, & \text{otherwise .} \end{cases}$$

We extend this representation scheme to binary words using ‘stack’ and ‘list’ images.

Definition 6 (Binary stack image) *The word $w = w_1 w_2 \cdots w_k \in \Sigma^+$ is represented by the binary stack image f_w ,*

$$f_w(x, y) = \begin{cases} 1, & \text{if } x = 1 - \frac{3}{2^{k-i+2}}, y = 0.5, w_i = 1 \\ 0, & \text{otherwise ,} \end{cases}$$

where $w_i \in \Sigma, 1 \leq i \leq k$. Image f_w is said to have length k and the pair (f_w, k) uniquely represents w .

Definition 7 (Binary list image) *The word $w = w_1 w_2 \cdots w_k \in \Sigma^+$ is rep-*

resented by the binary list image f_w ,

$$f_w(x, y) = \begin{cases} 1, & \text{if } x = \frac{2i-1}{2^k}, y = 0.5, w_i = 1 \\ 0, & \text{otherwise} \end{cases},$$

where $w_i \in \Sigma, 1 \leq i \leq k$. Image f_w is said to have length k and the pair (f_w, k) uniquely represents w .

If $\Sigma = \{1\}$ we replace the word “binary” with the word “unary” in Defs. 5, 6, and 7. In Defs. 6 and 7 each unary/binary symbol in w is represented by a corresponding value of 0 or 1 in f_w . Notice that, in the binary stack image, w 's leftmost symbol (w_1) is represented by the rightmost value in the sequence of values representing w in f_w ; this means that w_k is represented by the topmost stack element. We represent a single real value r by an image with a single peak of value r .

Definition 8 (Real number image) *The real number $r \in \mathbb{R}$ is represented by the real number image f_r ,*

$$f_r(x, y) = \begin{cases} r, & \text{if } x, y = 0.5 \\ 0, & \text{otherwise} \end{cases}.$$

To represent a $R \times C$ matrix of real values we define $R.C$ peaks that represent the matrix values and use both dimensions of a stack-like image.

Definition 9 ($R \times C$ matrix image) *The $R \times C$ matrix A , with real-valued components $a_{ij}, 1 \leq i \leq R, 1 \leq j \leq C$, is represented by the $R \times C$ matrix image f_A ,*

$$f_A(x, y) = \begin{cases} a_{ij}, & \text{if } x = 1 - \frac{1+2k}{2^{j+k}}, y = \frac{1+2l}{2^{i+l}} \\ 0, & \text{otherwise} \end{cases},$$

where

$$k = \begin{cases} 1, & \text{if } j < C \\ 0, & \text{if } j = C \end{cases}, \quad l = \begin{cases} 1, & \text{if } i < R \\ 0, & \text{if } i = R \end{cases}.$$

This matrix image representation is illustrated in Fig. 3(f).

The representations given in Defs. 5 through 9 are conveniently manipulated in the CSM using a programming technique called ‘rescaling’. Binary symbol images can be combined using stepwise rescaling (creating a binary stack image) or with a single rescale operation (creating a binary list image). A stack representation of the word 11 could be generated as follows. Take the image f_0 (having value 0 everywhere), representing an empty stack, and a unary symbol image f_1 that we will ‘push’ onto the stack. A push is accomplished by placing the images side-by-side with f_1 to the left and rescaling both into a

single image location. The image at this location is a (binary or unary) stack image encoding the word 1. This concept is illustrated in Fig. 3(a); a unary symbol image is placed at address **a** and an empty stack image is placed at address **b**. The command $\boxed{\text{ld}_{\text{ab}}}$ pushes the symbol onto the empty stack, and by default the result is stored in address **a**. Take another unary symbol image f_1 , place it to the left of the stack image, and rescale both into the stack image location once again. The (binary or unary) stack image contains two peaks at particular locations that testify that it is a representation of the word 11, as illustrated in Fig. 3(b). To remove a 1 from the stack image, a ‘pop’ operation is applied. Rescale the stack image over any two image locations positioned side-by-side. The image to the left will contain the symbol that had been at the top of the stack image (f_1) and the image to the right will contain the remainder of the stack image, as illustrated in Fig. 3(c). The stack image can be repeatedly rescaled over two images popping a single image each time. Popping an empty stack [Fig. 3(d)] results in the binary symbol image representing 0 and the stack remaining empty.

We can interpret a unary stack image as a nonnegative integer. Push and pop can then be interpreted as increment and decrement operations, respectively. As a convenient pseudocode, we use statements such as `c.push(1)` and `c.pop()` to increment and decrement the unary word represented by the stack image at address **c**. Binary representations of nonnegative integers would be represented in a similar manner. A unary stack representation of the integer 2 could be regarded as a binary stack representation of the integer 3. Our convention is to represent words with the rightmost symbol at the top of the stack. Therefore, if the second f_1 in the preceding example had been instead f_0 the resulting push operation would have created a stack image representing the word 10 (or alternatively, the binary representation of the integer 2). Pushing (or popping) p binary or unary symbol images to (or from) a binary or unary stack image requires TIME $\Theta(p)$, constant SPACE, RESOLUTION $\Theta(2^p)$ and constant RANGE 1. For CSM algorithms that use stack representations, RESOLUTION is of critical concern.

In the list image representation of a unary or binary word, each of the rescaled binary symbol images are equally spaced (unlike the stack image representation). The binary list image representation of a word $w \in \Sigma^+$, $|w| = k$, involves placing k symbol images (representing the k symbols of w) side-by-side in k contiguous image locations and rescaling them into a single image in a ld operation. For example in Fig. 3(e) a unary list representation of the unary word 111 is accomplished by the command $\boxed{\text{ld}_{\text{3}|\text{5}|\text{7}|\text{7}}}$. Rescaling p binary or unary symbol images to form a binary or unary list image, or rescaling a binary or unary list image to form p binary or unary symbol images both require constant TIME, constant SPACE, RESOLUTION $\Theta(p)$ and constant RANGE 1.

The $R \times C$ matrix image representation can be manipulated using image rescaling not only in the horizontal direction (as push and pop given above), but also in the vertical direction. In the matrix representation an initial empty image (to push to) is not required. Pushing (or popping) p real number images to (or from) a $p \times 1$ or $1 \times p$ matrix image requires TIME $\Theta(p)$, constant SPACE, RESOLUTION $\Theta(2^{p-1})$, and infinite RANGE ω . Pushing (or popping) q $p \times 1$ or q $1 \times p$ 1-D matrix images to (or from) a $p \times q$ or $q \times p$ matrix image requires TIME $\Theta(q)$, constant SPACE, RESOLUTION $\Theta(2^{p+q-2})$, and infinite RANGE ω .

2.4 Language deciding by CSM

Definition 10 (Language deciding by CSM) *CSM M_L decides $L \subseteq \Sigma^+$ if M_L has initial configuration $\langle c_s, g_s \rangle$ and final configuration $\langle c_h, g_h \rangle$, and the following hold:*

- *sequence g_s contains the two input elements $(f_w, \iota_{1_\xi}, \iota_{1_\eta})$ and $(f_{1^{|w|}}, \iota_{2_\xi}, \iota_{2_\eta})$*
- *g_h contains the output element $(f_1, o_{1_\xi}, o_{1_\eta})$ if $w \in L$*
- *g_h contains the output element $(f_0, o_{1_\xi}, o_{1_\eta})$ if $w \notin L$*
- *$\langle c_s, g_s \rangle \vdash_M^* \langle c_h, g_h \rangle$, for all $w \in \Sigma^+$.*

Where f_w is the binary stack image representation of $w \in \Sigma^+$, $f_{1^{|w|}}$ is the unary stack image representation of the unary word $1^{|w|}$. Images f_0 and f_1 are the binary symbol image representations of the symbols 0 and 1, respectively.

In this definition addresses $(\iota_{1_\xi}, \iota_{1_\eta}), (\iota_{2_\xi}, \iota_{2_\eta}) \in I$ and address $(o_{1_\xi}, o_{1_\eta}) \in O$, where I and O are as given in Def. 2. We use the stack image representation of words. The unary input word $1^{|w|}$ is necessary for M_L to determine the length of input word w . (For example the binary stack image representations of the words 00 and 000 are identical.)

2.5 Transformation from continuous image to finite address

Our model uses symbols from a finite set in its addressing scheme and employs an address resolution technique to effect decisions (see Sect. 2.6). Therefore, during branching and looping, variables encoded by elements of the uncountable set of continuous images must be transformed to the finite set of addresses. In one of the possible addressing schemes available to us, we use symbols from the set $\{0, 1\}$. We choose $B = \{w : w \in \{0, 1\}^{\max(m,n)}, w \text{ has a single } 1\}$ as our underlying set of address words. Each of the m column and n row addresses will be a binary word from the finite set B . An ordered pair of such binary words identifies a particular image in the grid. Each element of B will have a unique image encoding. \mathcal{N} is the set of encoded images, with $|\mathcal{N}| = \max(m, n)$.

In order to facilitate an optical implementation of our model we cannot assume to know the particular encoding strategy for the set \mathcal{N} (such as the simple binary stack or list representations of Sect. 2.3). We choose a correlation based address resolution technique. The address resolution technique chosen (the transformation from \mathcal{I} to B) must be general enough to resolve addresses that use any reasonable encoding (see Sect. 2.5.1).

Given an image $s \in \mathcal{I}$ we wish to determine which address word in B is encoded by s . In general, comparing a continuous image s with the elements of \mathcal{N} to determine membership is not guaranteed to terminate. However, for each s that our addressing scheme will be presented with, and given a reasonable encoding for \mathcal{N} , we can be sure of the following restrictions on \mathcal{N} : (i) $s \in \mathcal{N}$, (ii) $|\mathcal{N}|$ is finite, and (iii) \mathcal{N} contains distinct images (no duplicates). Given these restrictions, we need only search for the single closest match between s and the elements of \mathcal{N} . We choose a transformation based on cross-correlation (effected through a sequence of Fourier transform and image multiplication steps) combined with a thresholding operation.

The function $t : \mathcal{I} \times \mathcal{I} \mapsto \mathcal{N}$ is defined as

$$t(s, P) = \tau (\otimes(P, s)) \quad , \quad (7)$$

where s encodes the unknown addressing image to be transformed, P is a list image formed by rescaling all the elements of \mathcal{N} (in some predefined order) into a single image using one *ld* operation, \otimes denotes the cross-correlation function, and τ is a thresholding operation. The cross-correlation function [1,2] produces an image $f_{\text{corr}} = \otimes(P, s)$ where each point (u, v) in f_{corr} is defined

$$f_{\text{corr}}(u, v) = \int_0^1 \int_0^1 P(x, y) s^*(x + u, y + v) dx dy \quad , \quad (8)$$

where s^* denotes the complex conjugate of s , where (x, y) specifies coordinates in P and s , and where $(+u, +v)$ denotes an arbitrary relative shift between P and s expressed in the coordinate frame of f_{corr} . In Eq. (8), let s have value 0 outside of $[0, 1] \times [0, 1]$. Let image f_{corr} be defined only over $[0, 1] \times [0, 1]$. In the CSM, $f_{\text{corr}}(u, v)$ would be produced in image **a** by the code fragment

ld	P	h	v	st	b	ld	s	*	h	v	.	h	v
----	---	---	---	----	---	----	---	---	---	---	---	---	---

, where a multiplication in the Fourier domain is used to effect cross-correlation [1,2]. According to Eq. (8), and given a reasonable encoding for \mathcal{N} (implying the three restrictions outlined above), f_{corr} will contain exactly one well resolved maximum amplitude value. This point of maximum amplitude will be a nonzero value at a position identical to the relative positioning of the list element in P that most closely matches s . All other points in f_{corr} will contain an amplitude less than this value.

We define the thresholding operation of Eq. (7) for each point (u, v) in f_{corr}

as

$$\tau(f_{\text{corr}}(u, v)) = \begin{cases} 1, & \text{if } |f_{\text{corr}}(u, v)| = \max(|f_{\text{corr}}(u, v)|) \\ 0, & \text{if } |f_{\text{corr}}(u, v)| < \max(|f_{\text{corr}}(u, v)|) . \end{cases}$$

This produces an image with a single nonzero value at coordinates $u = (2i + 1)/[2 \times \max(m, n)], v = 0.5$ for some positive integer i in the range $[0, \max(m, n) - 1]$. From the definition of a binary list image (Def. 7), we can see that these unique identifiers are exactly the images that represent the binary words corresponding to the integers $\{2^0, 2^1, 2^2, \dots, 2^{[\max(m, n) - 1]}\}$. Therefore, t is a function from continuous images to the set of image representations of the finite set B defined earlier.

2.5.1 Reasonable encodings of \mathcal{N}

A note is required on what constitutes a reasonable encoding for \mathcal{N} , such that t will correctly transform s to an image representation of the appropriate element in B . There are two considerations which one needs to bear in mind when designing an encoding for \mathcal{N} . Firstly, Eq. (8) is not a normalised cross-correlation. Therefore, \mathcal{N} has to be chosen such that the autocorrelation of each element of \mathcal{N} has to return a larger maximum value than the cross-correlation with each of the other elements of \mathcal{N} .

Secondly, one may choose f_0 (the image with zero everywhere) as an element of \mathcal{N} . We can see from Eq. (8) that this will result in a cross-correlation of $f_{\text{corr}} = f_0$ when we try to match $s = f_0$ with P . We can resolve this special case (without the need for an explicit comparison with f_0) with the following rule. Given that \mathcal{N} is a reasonable encoding, if no single well resolved maximum amplitude value is generated from \otimes , we assume that $s = f_0$. (In all cases other than when $s = f_0$, f_{corr} will contain a well resolved point of maximum amplitude, as explained above.)

2.6 Conditional branching from unconditional branching

Our model does not have a conditional branching operation as a primitive; it was felt that giving the model the capability for equality testing of continuous images would rule out any possible implementation. However, we can effect indirect addressing through a combination of program self-modification and direct addressing. We can then implement conditional branching by combining indirect addressing and unconditional branching. This is based on a technique by Rojas [8] that relies on the fact that $|\mathcal{N}|$ is finite. Without loss of generality, we could restrict ourselves to two possible symbols 0 and 1. Then, the conditional branching instruction “if ($i=1$) then jump to address X , else jump to Y ” is written as the unconditional branching instruction “jump to address

i ". We are required only to ensure that the code corresponding to addresses X and Y is stored at addresses 1 and 0, respectively. In a 2D memory (with an extra addressing coordinate in the horizontal direction) many such branching instructions are possible in a single machine.

2.7 A general iteration construct

Our bounded iteration construct is based on the conditional branching instruction outlined in Sect. 2.6. Consider a loop of the following general form, written in some unspecified language,

```
SX
while (e > 0)
  SY
  e := e - 1
end while
SZ
```

where variable e contains a nonnegative integer specifying the number of remaining iterations, and SX , SY , and SZ are arbitrary lists of statements. Without loss of generality, we assume that statements SY do not write to e and do not branch to outside of the loop. If e is represented by a unary stack image (where the number of represented 1s equals the value of e), this code could be rewritten as

```
SX
while (e.pop() =  $f_1$ )
  SY
end while
SZ
```

and compiled to a CSM as shown in Fig. 4. In this CSM, e specifies the number of remaining iterations in unary and is represented by a unary stack image. A second address d , unused by the statements in the body of the loop, holds the value popped from e and must be positioned immediately to the left of e . Address a' is used to store and restore the contents of address a before and after, respectively, decrementing the loop counter e . The fragment $\boxed{\text{br } 0 \mid \widehat{d}}$ is shorthand for a piece of indirect addressing code, and means "branch to the address at the intersection of column 0 and the row specified by the image at address d ".

The while routine in Fig. 4 has TIME complexity $6 + i(s + 6)$, constant SPACE complexity, RESOLUTION complexity $\max(2^i, r)$, and RANGE complexity r' , where $i \in \mathbb{N}$ is the number of times the body of the while loop is executed, s

is the number of operations in the body of the while loop, and finally r and r' are the maximum RESOLUTION and maximum RANGE, respectively, of any image accessed during execution of the while loop.

3 Computability results

In this section we prove the CSM can simulate ARNNs, which are neural networks that compute over the set of real numbers. As an immediate corollary our model can decide any language $L \subseteq \Sigma^+$.

3.1 Boolean circuits and ARNNs

Informally, a Boolean circuit, or simply a circuit, is a finite directed acyclic graph where each node is an element of one of the following three sets: $\{\wedge, \vee, \neg\}$ (called *gates*, with respective in-degrees of 2,2,1), $\{x_1, \dots, x_n\}$ ($x_i \in \{0, 1\}$, *inputs*, in-degree 0), $\{0, 1\}$ (*constants*, in-degree 0). A circuit family is a set of circuits $C = \{c_n : n \in \mathbb{N}\}$. A language $L \subseteq \Sigma^*$ is decided by the circuit family C_L if the characteristic function of the language $L \cap \{0, 1\}^n$ is computed by c_n , for each $n \in \mathbb{N}$. It is possible to represent a circuit by a finite symbol sequence, and a circuit family by an infinite symbol sequence. When the circuits are of exponential size (with respect to input word length and where circuit size is the number gates in a circuit), for each $L \subseteq \Sigma^*$ there exists a circuit family to decide L . For a more thorough introduction to circuits we refer the reader to [9].

ARNNs are finite size feedback first-order neural networks with real weights [6,7]. The state of each neuron at time $t + 1$ is given by an update equation of the form

$$x_i(t + 1) = \sigma \left(\sum_{j=1}^N a_{ij}x_j(t) + \sum_{j=1}^M b_{ij}u_j(t) + c_i \right) , \quad i = 1, \dots, N \quad (9)$$

where N is the number of neurons, M is the number of inputs, $x_j(t) \in \mathbb{R}$ are the states of the neurons at time t , $u_j(t) \in \Sigma^+$ are the inputs at time t , and $a_{ij}, b_{ij}, c_i \in \mathbb{R}$ are the weights. An ARNN update equation is a function of discrete time $t = 1, 2, 3, \dots$. The network's weights, states, and inputs are often written in matrix notation as A, B and $c, x(t)$, and $u(t)$, respectively.

The function σ is defined as

$$\sigma(x) = \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{if } 0 \leq x \leq 1 \\ 1, & \text{if } x > 1 . \end{cases}$$

A subset P of the N neurons, $P = \{x_{k_1}, \dots, x_{k_p}\}$, $P \subseteq \{x_1, \dots, x_N\}$, are called the p output neurons. The output from an ARNN computation is defined as the states $\{x_{k_1}(t), \dots, x_{k_p}(t)\}$ of these p neurons over time $t = 1, 2, 3, \dots$.

3.2 Deciding languages using formal nets

ARNN input/output (I/O) mappings can be defined in many ways [6]. In this paper we give a CSM that simulates the general form ARNN which has the update equation given by Eq. (9). A specific type of ARNN called a formal net [6], decides languages. Formal nets are ARNNs with the following input and output encodings. A formal net has two binary input lines, called the input data line (D) and the input validation line (V), respectively. If D is *active* at a given time t then $D(t) \in \Sigma$, otherwise $D(t) = 0$. $V(t) = 1$ when D is active, and $V(t) = 0$ thereafter (when D is deactivated it never again becomes active). An input to a formal net at time t has the form $u(t) = (D(t), V(t)) \in \Sigma^2$. The input word $w = w_1 \dots w_k \in \Sigma^+$ where $w_i \in \Sigma$, $1 \leq i \leq k$, is represented by $u_w(t) = (D_w(t), V_w(t))$, $t \in \mathbb{N}$, where

$$D_w(t) = \begin{cases} w_t & \text{if } t = 1, \dots, k \\ 0 & \text{otherwise} \end{cases}, \quad V_w(t) = \begin{cases} 1 & \text{if } t = 1, \dots, k \\ 0 & \text{otherwise} \end{cases}.$$

A formal net has two output neurons $O_d, O_v \in \{x_1, \dots, x_N\}$, called the output data line and output validation line, respectively. Given a formal net \mathcal{F} with an input word w and initial state $x_i(1) = 0$, $1 \leq i \leq N$, w is *classified* in time τ if the output sequences have the form

$$O_d = 0^{\tau-1}\psi_w 0^\infty, \quad O_v = 0^{\tau-1}10^\infty,$$

where $\psi_w \in \Sigma$. If $\psi_w = 1$ then w is accepted, if $\psi_w = 0$ then w is rejected. We now give a definition of language deciding by ARNN (from [6]). Let $\mathbf{T} : \mathbb{N} \mapsto \mathbb{N}$ be a total function.

Definition 11 (Language deciding by formal net) *The language $L \subseteq \Sigma^+$ is decided in time \mathbf{T} by the formal net \mathcal{F} provided that each word $w \in \Sigma^+$ is classified in time $\tau \leq \mathbf{T}(|w|)$ and $\psi_w = 1$ if $w \in L$ and $\psi_w = 0$ if $w \notin L$.*

In [6], Siegelmann and Sontag prove that for each language $L \subseteq \Sigma^+$ there exists a formal net \mathcal{F}_L to decide the membership problem for L , hence proving the ARNN model to be computationally more powerful than the Turing machine model. \mathcal{F}_L contains one real weight. This weight encodes a circuit family C_L that decides L . Let $S_{C_L} : \mathbb{N} \mapsto \mathbb{N}$ be the size of C_L . For a given input word $w \in \Sigma^+$, \mathcal{F}_L retrieves the encoding of circuit $c_{|w|}$ from its real weight and simulates this encoded circuit on input w to decide membership in L , in time $\mathbf{T}(|w|) = O(|w|[S_{C_L}(|w|)]^2)$. Given polynomial time, formal nets decide the nonuniform language class P/poly . Given exponential time, formal nets decide all languages.

3.3 Statement of main result

Theorem 1 *There exists a CSM \mathcal{M} such that for each ARNN \mathcal{A} , \mathcal{M} computes \mathcal{A} 's I/O map, using our ARNN I/O representation.*

PROOF. The proof is provided by the ARNN simulation program for the CSM given in Fig. 5. The simulation is written in a convenient shorthand notation. The expansions into sequences of atomic operations are given in Fig. 6. The ARNN I/O representation is given in Sect. 3.4. The simulation program is explained in Sects. 3.5 through 3.7. A computational complexity analysis of the simulation program is given in Sect. 3.8. \square

3.4 ARNN representation

As a convenient notation we let $\bar{\kappa}$ be the image representation of κ . We now give the I/O representation used in Theorem 1. The inputs to \mathcal{M} fall into three categories: inputs that represent \mathcal{A} , inputs that represent \mathcal{A} 's input, and some constant inputs. Recall that our representation of matrices by images was defined in Def. 9 and illustrated in Fig. 3(f).

The ARNN weight matrices A , B , and c are represented by $N \times N$, $N \times M$, and $N \times 1$ matrix images \bar{A} , \bar{B} , and \bar{c} , respectively. The state vector x is represented by a $1 \times N$ matrix image \bar{x} . The set of output states P are represented by the image \bar{P} (described below). The values $N - 1$ and $M - 1$ need to be given as input to the simulator in order to bound the loops. They are represented by unary stack images $\overline{N - 1}$ and $\overline{M - 1}$, representing the unary words $1^{(N-1)}$ and $1^{(M-1)}$, respectively. These seven input images define ARNN \mathcal{A} . The constant images $f(x, y) = 0$ and $f(x, y) = 1$, denoted $\mathbf{0}$ and $\mathbf{1}$, respectively, are also given as input. Images $\mathbf{0}$ and $\mathbf{1}$ are used to parameterise ρ (see Lemma 2 below).

For ARNN timestep t , the ARNN input vector $u(t)$ is represented by a $1 \times M$ matrix image \bar{u} . In an initial configuration of our simulation program we assume an input stack image I encodes all input vectors $u(t)$ for all $t = 1, 2, 3, \dots$. At ARNN timestep t , the top element of stack image I is a $1 \times M$ matrix image representing the input vector $u(t)$.

The p output neurons are represented by a $1 \times N$ matrix image \bar{P} . We use \bar{P} to extract our representation of the p output states from the N neuron states represented by \bar{x} . The image \bar{x} contains N (possibly nonzero) values at specific coordinates defined in Def. 9. p of these values represent the p ARNN output states and have coordinates $(x_1, y_1), \dots, (x_p, y_p)$ in \bar{x} . In the image \bar{P} , each of the coordinates $(x_1, y_1), \dots, (x_p, y_p)$ has value 1 and all other coordinates in \bar{P} have value 0. We multiply \bar{x} by \bar{P} . This image multiplication results in an output image o that has our representation of the p ARNN outputs at the coordinates $(x_1, y_1), \dots, (x_p, y_p)$. o has value 0 at all other coordinates. The simulator then pushes o to an output stack image O . This process is carried out at the end of each simulated state update.

3.5 ARNN simulation overview

From the neuron state update equation Eq. (9), each $x_j(t)$ is a component of the state vector $x(t)$. From $x(t)$ we define the $N \times N$ matrix $X(t)$ where each row of $X(t)$ is the vector $x(t)$. Therefore $X(t)$ has components $x_{ij}(t)$, and for each $j \in \{1, \dots, N\}$ it is the case that $x_{ij} = x_{i'j}, \forall i, i' \in \{1, \dots, N\}$. From $u(t)$ we define the $N \times M$ matrix $U(t)$ where each row of $U(t)$ is the vector $u(t)$. Therefore $U(t)$ has components $u_{ij}(t)$, and for each $j \in \{1, \dots, M\}$ it is the case that $u_{ij} = u_{i'j}, \forall i, i' \in \{1, \dots, N\}$. Using $X(t)$ and $U(t)$ we rewrite Eq. (9) as

$$x_i(t+1) = \sigma \left(\sum_{j=1}^N a_{ij} x_{ij}(t) + \sum_{j=1}^M b_{ij} u_{ij}(t) + c_i \right) , \quad i = 1, \dots, N . \quad (10)$$

In the simulation we generate $N \times N$ and $N \times M$ matrix images \bar{X} and \bar{U} representing $X(t)$ and $U(t)$, respectively. We then simulate the affine combination in Eq. (10) using our model's $+$ and \cdot operators. We use the CSM's amplitude filtering operation ρ to simulate the ARNN σ function.

Lemma 2 *The CSM operation ρ simulates $\sigma(x)$.*

PROOF. From the definition of ρ in Eq. (6), we set $z_l(x, y) = 0$ (denoted $\mathbf{0}$) and $z_u(x, y) = 1$ (denoted $\mathbf{1}$) to give

$$\rho(f(x, y), \mathbf{0}, \mathbf{1}) = \begin{cases} 0, & \text{if } |f(x, y)| < 0 \\ |f(x, y)|, & \text{if } 0 \leq |f(x, y)| \leq 1 \\ 1, & \text{if } |f(x, y)| > 1 \end{cases} .$$

Under our representation of ARNN state values by images, $\rho(\bar{x}, \mathbf{0}, \mathbf{1})$ simulates $\sigma(x)$. \square

3.6 ARNN simulation algorithm

For brevity and ease of understanding we outline our simulation algorithm in a high-level pseudocode, followed by an explanation of each algorithm step.

- (i) $\bar{u} := I.\text{pop}()$
- (ii) $\bar{X} := \text{push } \bar{x} \text{ onto itself vertically } N - 1 \text{ times}$
- (iii) $\overline{AX} := \bar{A} \cdot \bar{X}$
- (iv) $\Sigma \overline{AX} := \Sigma_{i=1}^N (\overline{AX}.\text{pop}_i())$
- (v) $\bar{U} := \text{push } \bar{u} \text{ onto itself vertically } N - 1 \text{ times}$
- (vi) $\overline{BU} := \bar{B} \cdot \bar{U}$
- (vii) $\Sigma \overline{BU} := \Sigma_{i=1}^M (\overline{BU}.\text{pop}_i())$
- (viii) $\text{affine-comb} := \Sigma \overline{AX} + \Sigma \overline{BU} + \bar{c}$
- (ix) $\bar{x}' := \rho(\text{affine-comb}, \mathbf{0}, \mathbf{1})$
- (x) $\bar{x} := (\bar{x}')^T$
- (xi) $O.\text{push}(\bar{P} \cdot \bar{x})$
- (xii) goto step (i)

In step (i) we pop an image from input stack I and call the popped image \bar{u} . \bar{u} is a $1 \times M$ matrix image representing the ARNN's inputs at some time t . In step (ii) we generate the $N \times N$ matrix image \bar{X} by vertically pushing $N - 1$ identical copies of \bar{x} onto a copy of \bar{x} . In step (iii), \bar{X} is point by point multiplied by matrix image \bar{A} . This single multiplication step efficiently simulates (in linear TIME) the matrix multiplication $a_{ij}x_j$ for all $i, j \in \{1, \dots, N\}$ (as described in Sect. 3.5). Step (iv) simulates the ARNN summation $\sum_{j=1}^N a_{ij}x_j$. Each of the

N columns of \overline{AX} are popped and added (using the $+$ operation), one at a time, to the previous popped image.

In step (v) we are treating \overline{u} in a similar way to our treatment of \overline{x} in step (ii). In step (vi) we effect $\overline{B} \cdot \overline{U}$, efficiently simulating (in linear TIME) the multiplication $b_{ij}u_j$ for all $i \in \{1, \dots, N\}, j \in \{1, \dots, M\}$. Step (vii) simulates the ARNN summation $\sum_{j=1}^M b_{ij}u_j$ using the same technique used in step (iv).

In step (viii) we simulate the addition of the three terms in the ARNN affine combination. In our simulator this addition is effected in two simple image addition steps. In step (ix) we simulate the ARNN's σ function by amplitude filtering using the CSM's ρ function with the lower and upper threshold images $(\mathbf{0}, \mathbf{1})$ (as given by Lemma 2). The resulting $N \times 1$ matrix image is transformed into a $1 \times N$ matrix image (we simply transpose the represented vector) in step (x). We call the result of this amplitude filtering and transformation \overline{x} ; it represents the ARNN state vector $x(t+1)$. In step (xi) we multiply \overline{x} by the output mask \overline{P} (as described in Sect. 3.4). The result, which represents the ARNN output at time $t+1$ is then pushed to the output stack O . The final step in our algorithm sends us back to step (i). Notice that our algorithm never halts as ARNNs are defined for time $t = 1, 2, 3, \dots$.

3.7 Explanation of Figs. 5 and 6

The ARNN simulation with our model is shown in Fig. 5. The numerals (i)–(xii) are present to assist the reader in understanding the program; they correspond to steps (i)–(xii) in the high-level pseudocode in Sect. 3.6. In our ARNN simulator program addresses are written in a shorthand notation that are expanded using Fig. 6. Before the simulator begins executing a simple preprocessor or compiler could be used to update the shorthand addresses to the standard long-form notation.

Addresses t_1 , t_2 , and t_3 are used as temporary storage locations during a run of the simulator [note: address t_3 is located at grid coordinates (10, 14)]. In the simulator our $\overline{\kappa}$ notation not only denotes the image representation of κ , but also acts as an address identifier for the image representing κ . Addresses \overline{x} and \overline{u} are used to store our representation of the neurons' states and inputs, respectively, during a computation. The temporary storage addresses $\overline{\Sigma AX}$ and $\overline{\Sigma BU}$ are used to store the results of steps (iv) and (vii), respectively. Addresses $\overline{N-1}$ and $\overline{M-1}$ store our representation of the dimensions of x and u , respectively (necessary for bounding the while loops). The address identifiers \overline{A} , \overline{B} , and \overline{c} store the image representation of the corresponding ARNN matrices, and \overline{P} stores our mask for extracting the p output states from the N neuron states, as described in Sect. 3.4. Code fragments of the

form

whl	i	...	end
-----	-----	-----	-----

 are shorthand for code to initialise and implement the while loop given in Sect. 2.7. The instructions between i and **end** are executed i times. The notation $\widehat{\mathbf{0}}$ is shorthand for the “image at address $\mathbf{0}$ ”.

At ARNN timestep t , our representation of the ARNN input $u(t)$ is at the top of the input stack image I . This input is popped off the stack and placed in address \bar{u} . The computation then proceeds as described by the high-level pseudocode algorithm in Sect. 3.6. The output memory address O stores the sequence of outputs as described in Sect. 3.4. Program execution begins at well-known address **sta** and proceeds according to the rules for our model’s programming language defined in Def. 4 and explained in Fig. 2.

3.8 Complexity analysis of simulation algorithm

If the ARNN being simulated is defined for time $t = 1, 2, 3, \dots$, has M as the length of the input vector $u(t)$ and has N neurons, and k is the number of stack image elements used to encode the active input to our simulator, then \mathcal{M} requires TIME

$$T(N, M, t, k) = (49N + 11M + 28)t + 1 .$$

\mathcal{M} requires TIME linear in N , M , and t , and independent of k . It requires constant SPACE, and exponential RESOLUTION

$$R(N, M, t, k) = \max(2^{(k+M-1)}, 2^{(2N-2)}, 2^{(N+M-2)}, 2^{(t+N-1)}) .$$

Finally, \mathcal{M} requires infinite RANGE ω in order to represent real-valued ARNN weight matrices.

3.9 CSM deciding Languages by formal net simulation

Corollary 3 *There exists a CSM \mathcal{D} that decides the membership problem for each $L \subseteq \Sigma^+$.*

PROOF. The proof relies on two facts. Firstly, for each $L \subseteq \Sigma^+$ there exists a formal net \mathcal{F}_L that decides the membership problem for L [6]. Secondly there exists a CSM \mathcal{M} that simulates each ARNN (Theorem 1). CSM \mathcal{D} is given in Fig. 7, its I/O format and a brief complexity analysis are given in the remainder of the current section. \square

To decide membership of $w \in \Sigma^+$ in L , \mathcal{D} simulates formal net \mathcal{F}_L on input w . \mathcal{D} is a language deciding CSM, hence \mathcal{D} 's I/O format is consistent with Def. 10 (language deciding by CSM). In Fig. 7, rows 2 to 13 are exactly rows 2 to 13 from CSM \mathcal{M} in Fig. 5, the remaining extra functionality is necessary to properly format the I/O. Given the problem instance of deciding membership of $w \in \Sigma^+$ in L , CSM \mathcal{D} has thirteen input images and a single output image. Input images f_w and $f_{1^{|w|}}$ are the binary and unary stack image representations of the words w and $1^{|w|}$, respectively. Images $\mathbf{0}$ and $\mathbf{1}$ are the constant images $f(x, y) = 0$ and $f(x, y) = 1$, respectively. Formal net \mathcal{F}_L is completely defined by the following seven input images: $\overline{A}, \overline{B}, \overline{c}, \overline{P}, \overline{x}, \overline{N-1}$, and $\overline{M-1}$. These images have the format described above in Sect. 3.4. When simulating a formal net the input images $\overline{M-1}$ and \overline{x} are constant (as $M = 2$ and $x(1)$ is a vector of zeros). Images $\overline{O_d}$ and $\overline{O_v}$ are unary stack images representing the unary words 1^d and 1^v , respectively. Here d and v are the indices of the output data and output validation neurons, respectively, in the N vector of neurons. Images $\overline{O_d}$ and $\overline{O_v}$ are used to extract the output ‘decision’ of \mathcal{F}_L . There is one output image denoted f_{ψ_w} .

Let us assume we are simulating a formal net \mathcal{F}_L that decides language L in time \mathbf{T} . Hence, on input word $w \in \Sigma^+$, \mathcal{F}_L decides if w is in L in t timesteps for some $t \leq \mathbf{T}(|w|)$. Let N, M, d , and v be as given above. CSM \mathcal{D} requires in the worst case linear TIME

$$T(N, M, \mathbf{T}(|w|), |w|, d, v) = 12|w| + 7d + (49N + 7v + 67)\mathbf{T}(|w|) + 22$$

exponential RESOLUTION

$$R(N, M, \mathbf{T}(|w|), |w|, d, v) = \max(2^{|w|}, 2^{(2N-2)})$$

and constant SPACE to decide membership of w in L . We also require infinite RANGE ω , as one of \mathcal{F}_L 's weight matrices contains a real-valued weight encoding the (possibly infinite) circuit family C_L . When deciding a language from the class P/poly the formal net time complexity function \mathbf{T} is polynomial in input word length, in the worst case. When deciding an arbitrary language the function \mathbf{T} is exponential in input word length, in the worst case [6]. By way of formal net simulation the CSM decides any language $L \subseteq \Sigma^+$ with these complexity bounds.

4 Unordered search

Sorting and searching [10] provide standard challenges to computer scientists in the field of algorithms, computation, and complexity. In this paper we focus on a binary search algorithm. With our model this algorithm can be applied to unordered lists. Consider an unordered list of n elements. For a given property

P , the list could be represented by an n -tuple of bits, where the bit key for each element denotes whether or not that element satisfies P . If, for a particular P , only one element in the list satisfies P , the problem of finding its index becomes one of searching an unordered binary list for a single 1. The problem is defined formally as follows.

Definition 12 (Needle in haystack problem) *Let $L = \{w : w \in 0^*10^*\}$. Let $w \in L$ be written as $w = w_0w_1 \dots w_{n-1}$ where $w_i \in \{0, 1\}$. Given such a w , the needle in haystack (NIH) problem asks what is the index of the symbol 1 in w . The solution to NIH for a given w is the index i , expressed in binary, where $w_i = 1$.*

This problem was posed by Grover in [11]. His quantum computer algorithm requires $O(\sqrt{n})$ comparison operations on average. Bennett et al. [12] have shown the work of Grover is optimal up to a multiplicative constant, and that in fact any quantum mechanical system will require $\Omega(\sqrt{n})$ comparisons. Algorithms for conventional models of computation require $\Theta(n)$ comparisons in the worst case to solve the problem. We present an algorithm that requires $\Theta(\log_2 n)$, in the worst case, with a model of computation that has promising future implementation prospects.

Before presenting a CSM instance of the algorithm, we give a pseudocode version (see Fig. 8). This pseudocode algorithm consists of a single loop. It is formatted to conform to the iteration construct presented in Sect. 2.7. The algorithm takes two arguments, one is a list image and the other is a stack image. (Stack images and list images were defined in Defs. 6 and 7.) The first argument, **i1**, is a binary list image representing w . We assume that n is a power of 2. The second argument, **i2**, is a unary stack image of length $\log_2 n$, and is used to bound the iteration of the algorithm's loop. The algorithm uses address **c** as it constructs, one binary image at a time, a binary stack image of length $\log_2 n$. At halt, the binary stack image at address **c** represents the index i of the 1 in w . This index is returned through **a** when the algorithm terminates. To aid the reader, each line of the pseudocode algorithm in Fig. 8 is prepended with a pair of coordinates that relate the pseudocode to the beginning of the corresponding code in the CSM version of the algorithm. The CSM version of the algorithm is given in Fig. 9.

Definition 13 (Comparison in CSM) *A comparison in a CSM computation is defined as a conditional branching instruction.*

Theorem 4 *There exists a CSM that solves NIH in $\Theta(\log_2 n)$ comparisons for a list of length n , where $n = 2^c, c \in \mathbb{N}, c \geq 1$.*

PROOF. The proof is provided by the algorithm in Fig. 9. Correctness: The correctness is most easily seen from the pseudocode algorithm in Fig. 8 and

the following inductive argument. (Figure 8 contains a mapping from pseudocode statements to the CSM statements of Fig. 9.) The two inputs are a binary list image representation of w (image **i1**) and a unary stack image of length $\log_2 n$ (image **i2**). During the first iteration of the loop, a single image f_1 is popped from **i2**, and **i1** is divided equally into two list images (a left-hand image and a right-hand image). The nonzero peak (representing the 1 in w) will be either in the left-hand list image or the right-hand list image. In order to determine which list image contains the peak in a constant number of steps, the left-hand list image is transformed [the Fourier transform was defined in Eq. (1)] such that its centre will contain a weighted sum of all of the values over the whole list image. Effectively, the list image is transformed to an element of the set $\{f_0, f_1\}$. If the left-hand list image is transformed to f_1 (if the centre of this transformed list image contains a nonzero amplitude) then the left-hand list image contained the peak. In this case, the right-hand image is discarded, and f_0 is pushed onto stack image **c**. Otherwise, the right-hand list image contained the peak, the left-hand list image is discarded, and f_1 is pushed onto **c**. After the first iteration of the loop, the most significant bit of the solution to the problem is represented by the top of stack image **c**, and **i1** has been reduced to half its length. For the second iteration of the loop, a second image f_1 is popped from counter **i2**, the list image is divided in two, and the appropriate half discarded. The algorithm continues in this binary search fashion until the image popped from **i2** is f_0 . Image **c** is copied to image **a** and the algorithm halts. At halt, the index (in binary) of the 1 in w is represented by the stack image in **a** of length $\log_2 n$.

Complexity: The loop in the algorithm makes exactly $\log_2 n$ iterations, corresponding to $\log_2 n + 1$ evaluations of the loop guard. Inside the loop, there is a single comparison. In total, the CSM algorithm makes $2\log_2 n + 1$ comparisons to transform the binary list image representation of w (of length n) into the binary stack image representation of index i . \square

Theorem 4 states computational complexity in terms of number of comparisons, so that the result can be directly compared with the lower bound analyses from classical algorithm theory and quantum complexity theory. This simplification hides constant overheads only, as the following corollary shows.

Corollary 5 *There exists a CSM that solves NIH in $\Theta(\log_2 n)$ TIME, constant SPACE, constant RANGE, and $\Theta(n)$ RESOLUTION for a list of length n , where $n = 2^c$, $c \in \mathbb{N}$, $c \geq 1$.*

PROOF. The proof is provided by the algorithm in Fig. 9. Correctness: The correctness follows from Theorem 4.

Complexity: Each iteration of the loop requires constant TIME. The total TIME from problem instance to solution is $23\log_2 n + 11$. The maximum length required of any stack image during the computation is $\log_2 n + 1$ (for image

c). This results in RESOLUTION complexity of $2n$. The CSM requires RANGE complexity of 1 (to represent values 0 and 1). From Sect 2.2, all CSMs require constant SPACE. \square

5 Conclusion

We have presented the CSM, an analog model of computation inspired by the field of optical information processing. This model does not support arbitrary equality testing of images, and so is closer in spirit to models found in [6,13,14] than (say) the Blum, Shub, and Smale model [15,16]. We have given some insight into the computational power of the CSM by proving it can simulate ARNNs (this simulation includes linear time matrix multiplication), and by giving a $\Theta(\log_2 n)$ unordered search algorithm. For future work it would be interesting to prove further computability and complexity results for the CSM and to investigate (computationally less powerful) variants of the model.

References

- [1] A. VanderLugt, *Optical Signal Processing*, Wiley Series in Pure and Applied Optics, Wiley, New York, 1992.
- [2] J. W. Goodman, *Introduction to Fourier Optics*, 2nd Edition, McGraw-Hill, New York, 1996.
- [3] T. J. Naughton, A model of computation for Fourier optical processors, in: R. A. Lessard, T. Galstian (Eds.), *Optics in Computing 2000*, Proc. SPIE vol. 4089, Quebec, Canada, 2000, pp. 24–34.
- [4] T. J. Naughton, D. Woods, On the computational power of a continuous-space optical model of computation, in: M. Margenstern, Y. Rogozhin (Eds.), *Machines, Computations and Universality: Third International Conference*, Vol. 2055 of *Lecture Notes in Computer Science*, Chişinău, Moldova, 2001, pp. 288–299.
- [5] K. Weihrauch, *Computable Analysis: An Introduction*, Texts in Theoretical Computer Science, Springer, Berlin, 2000.
- [6] H. T. Siegelmann, E. D. Sontag, Analog computation via neural networks, *Theoretical Computer Science* 131 (2) (1994) 331–360.
- [7] H. T. Siegelmann, *Neural networks and analog computation: beyond the Turing limit*, Progress in theoretical computer science, Birkhäuser, Boston, 1999.

- [8] R. Rojas, Conditional branching is not necessary for universal computation in von Neumann computers, *Journal of Universal Computer Science* 2 (11) (1996) 756–768.
- [9] J. L. Balcázar, J. Díaz, J. Gabarró, *Structural Complexity*, Vol. 1 of EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin, 1988.
- [10] D. Knuth, *The Art of Computer Programming*, Vol 3: Sorting and Searching, Addison-Wesley, 1973.
- [11] L. K. Grover, A fast quantum mechanical algorithm for database search, in: *Proc. 28th Annual ACM Symposium on Theory of Computing*, 1996, pp. 212–219.
- [12] C. H. Bennett, E. Bernstein, G. Brassard, U. Vazirani, Strengths and weaknesses of quantum computing, *SIAM Journal on Computing* 26 (5) (1997) 1510–1523.
- [13] C. Moore, Recursion theory on the reals and continuous-time computation, *Theoretical Computer Science* 162 (1) (1996) 23–44.
- [14] M. L. Campagnolo, Computational complexity of real valued recursive functions and analog circuits, Ph.D. thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa, submitted (2001).
- [15] L. Blum, M. Shub, S. Smale, A theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines, *Bulletin of the American Mathematical Society* 21 (1989) 1–46.
- [16] L. Blum, F. Cucker, M. Shub, S. Smale, *Complexity and real computation*, Springer-Verlag, New York, 1997.

h	: perform a horizontal 1-D Fourier transform on the 2D image in a . Store result in a .					
v	: perform a vertical 1-D Fourier transform on the 2D image in a . Store result in a .					
*	: replace a with the complex conjugate of a .					
·	: multiply (point by point) the two images in a and b . Store result in a .					
+	: perform a complex addition of a and b . Store result in a .					
<table border="1" style="display: inline-table; vertical-align: middle;"> <tr> <td style="padding: 2px;">ρ</td> <td style="padding: 2px;">z_l</td> <td style="padding: 2px;">z_u</td> </tr> </table>	ρ	z_l	z_u	: $z_l, z_u \in \mathcal{I}$; filter the image in a by amplitude using z_l and z_u as lower and upper amplitude threshold images, respectively.		
ρ	z_l	z_u				
<table border="1" style="display: inline-table; vertical-align: middle;"> <tr> <td style="padding: 2px;">st</td> <td style="padding: 2px;">$p1$</td> <td style="padding: 2px;">$p2$</td> <td style="padding: 2px;">$p3$</td> <td style="padding: 2px;">$p4$</td> </tr> </table>	st	$p1$	$p2$	$p3$	$p4$: $p1, p2, p3, p4 \in \mathbb{N}$; copy the image in a into the rectangle of images whose bottom left-hand corner address is $(p1, p3)$ and whose top right-hand corner address is $(p2, p4)$.
st	$p1$	$p2$	$p3$	$p4$		
<table border="1" style="display: inline-table; vertical-align: middle;"> <tr> <td style="padding: 2px;">ld</td> <td style="padding: 2px;">$p1$</td> <td style="padding: 2px;">$p2$</td> <td style="padding: 2px;">$p3$</td> <td style="padding: 2px;">$p4$</td> </tr> </table>	ld	$p1$	$p2$	$p3$	$p4$: $p1, p2, p3, p4 \in \mathbb{N}$; copy into a the rectangle of images whose bottom left-hand corner address is $(p1, p3)$ and whose top right-hand corner address is $(p2, p4)$.
ld	$p1$	$p2$	$p3$	$p4$		
<table border="1" style="display: inline-table; vertical-align: middle;"> <tr> <td style="padding: 2px;">br</td> <td style="padding: 2px;">$p1$</td> <td style="padding: 2px;">$p2$</td> </tr> </table>	br	$p1$	$p2$: $p1, p2 \in \mathbb{N}$; unconditionally branch to the image at address $(p1, p2)$.		
br	$p1$	$p2$				
hlt	: halt.					
	: move to the next grid image (ignore images that do not represent a programming symbol).					

Fig. 2. The set of CSM operations, given in our informal grid notation. For formal definitions see Def. 4.

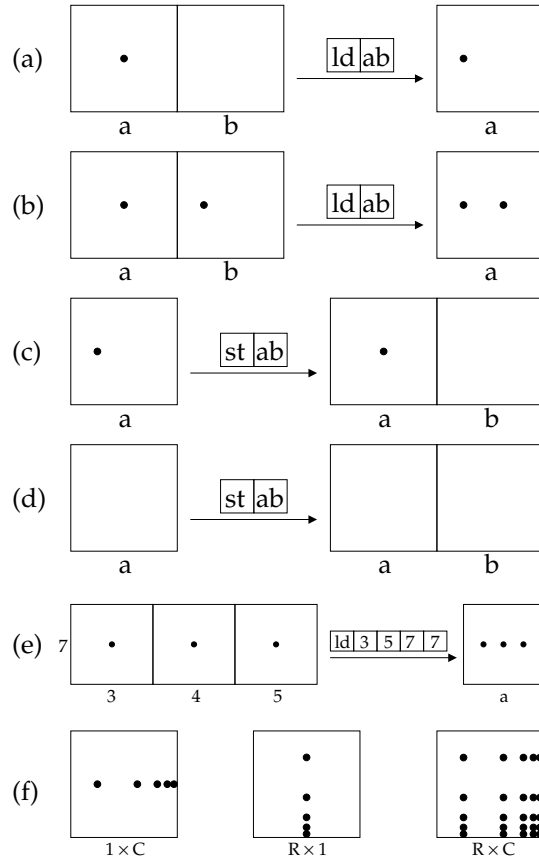


Fig. 3. Representing data by images through the positioning of peaks. The nonzero peaks are coloured black and the white areas denote value 0. (a) Pushing a unary symbol image onto an empty stack image. (b) Pushing a unary symbol image onto a stack image representing the word 1. (c) Popping a stack encoding the word 1 to create the representation of 11. (d) Popping a stack encoding the word 1, resulting in a popped unary symbol image (in **a**) and an empty stack (in **b**). (e) Popping an empty stack. (f) Rescaling three adjacent unary symbol images into a single unary list image (in **a**) representing 111. (f) $1 \times C$, $R \times 1$, and $R \times C$ matrix images where $R = C = 5$.

	sta	0	1	2	3	4	5	6	7	8	9	10	...	\bar{x}	$\bar{N}-1$	$\bar{M}-1$	\bar{A}	\bar{B}	\bar{c}	0	1	
99	br	0		14																		
(i)	ld	I	st	t_1a	st	I	ld	t_1	st	\bar{u}												
(ii)	ld	\bar{x}	whl	$\bar{N}-1$	st	t_3	ld	\bar{x}	ld	at_3	end											
(iii)	st	b	ld	\bar{A}	.																	
(iv)	st	t_2	ld	$\mathbf{0}$	whl	$\bar{N}-1$	st	t_1	ld	t_2	st	bt_2	ld	t_1	+	end						
10		st	b	ld	t_2	+	st	$\Sigma\bar{A}\bar{X}$														
(v)	ld	\bar{u}	whl	$\bar{N}-1$	st	t_3	ld	\bar{u}	ld	at_3	end											
(vi)	st	b	ld	\bar{B}	.																	
(vii)	st	t_2	ld	$\mathbf{0}$	whl	$\bar{M}-1$	st	t_1	ld	t_2	st	bt_2	ld	t_1	+	end						
6		st	b	ld	t_2	+	st	$\Sigma\bar{B}\bar{U}$														
(viii)	ld	$\Sigma\bar{A}\bar{X}$	st	b	ld	$\Sigma\bar{B}\bar{U}$	+	st	b	ld	\bar{c}	+										
(ix)	ρ	$\hat{\mathbf{0}}$	st	t_3	ld	$\mathbf{0}$	ld	st	t_1													
(x)	whl	$\bar{N}-1$	ld	t_3	st	at_3	ld	t_1a	st	t_1	end	ld	ld	t_3	st	ab	ld	t_1a	st	st	t_1	
2	whl	$\bar{N}-1$	ld	t_1	st	t_1a	ld	ab	st	b	end	ld	ld	t_1	st	t_1a	ld	ab	st	st	\bar{x}	
(xi)	st	b	ld	\bar{P}	.	st	t_1	ld	ld	t_1a	st											
(xii)	br	0	14																			

note: address t_3 is located at grid coordinates (10, 14)

Fig. 5. CSM \mathcal{M} that simulates any ARNN \mathcal{A} . The simulator is written in a convenient shorthand notation, (see Fig. 6 for the expansions into sequences of atomic operations). The simulation program is explained in Sect. 3.7.

(a)

I	\rightarrow	15	15	99	99
t_1a	\rightarrow	9	10	99	99
t_1	\rightarrow	9	9	99	99
\bar{u}	\rightarrow	5	5	99	99
\bar{x}	\rightarrow	12	12	0	0
t_3	\rightarrow	10	10	14	14
at_3	\rightarrow	10	10	14	99
b	\rightarrow	11	11	99	99
\bar{A}	\rightarrow	15	15	0	0
ab	\rightarrow	10	11	99	99
bt_2	\rightarrow	11	12	99	99
t_2	\rightarrow	12	12	99	99
$\Sigma\bar{A}\bar{X}$	\rightarrow	6	6	99	99
\bar{B}	\rightarrow	16	16	0	0
$\Sigma\bar{B}\bar{U}$	\rightarrow	7	7	99	99
\bar{c}	\rightarrow	17	17	0	0
\bar{P}	\rightarrow	18	18	0	0
O	\rightarrow	14	14	99	99

(b)

whl	$\overline{N-1}$...	end
whl	$\overline{M-1}$...	end

Fig. 6. Time-saving shorthand notation used in the simulator in Fig. 5: (a) shows shorthand addresses, and (b) expands to initialisation instructions and the while loop code given in Fig. 4.

	sta	0	1	2	3	4	5	6	7	8	...	\overline{O}_d	\overline{O}_v	\bar{x}	$\overline{N-1}$	$\overline{M-1}$	\overline{A}	\overline{B}	\bar{c}	0	1	
			\bar{u}	ΣAX	ΣBU	t_1	a	b	t_2	f_{ψ_w}	f_w	$f_{1^{ \omega }}$										
99	br	0	18																			
18	ld	f_w	st	b	ld	0	st	t_2														
17	whl	$f_{1^{ \omega }}$	ld	b	st	t_1a	st	b	ld	t_2	ld	t_1a	st	t_2	end	st	f_w					
16	ld	f_w	st	t_1a	st	f_w	ld	t_1	st	13	16	14	14	14	14	17	14	14				
15	st	b	ld	$f_{1^{ \omega }}$	st	t_1a	st	$f_{1^{ \omega }}$	ld	t_1	st	13	16	14	14							
14	ld	12	15	14	14	+	st	\bar{u}	br	0	13											
...																						
1	st	b	ld	\overline{P}	.	st	t_3	st	t_1	whl	\overline{O}_v	st	t_1a	end	ld	t_1	br	0	\hat{a}			
f_1	ld	t_3	whl	\overline{O}_d	st	t_1a	end	ld	t_1	st	f_{ψ_w}	hlt										
f_0	br	0	16																			
		0	1	2	3	4	5	6	7	8	...	\overline{O}_d	\overline{O}_v	\bar{x}	$\overline{N-1}$	$\overline{M-1}$	\overline{A}	\overline{B}	\bar{c}		\overline{P}	

note: address t_3 is located at grid coordinates (10, 18)

Fig. 7. Language deciding CSM \mathcal{D} . Shorthand notation follows the format given in Fig. 6. Rows 2 to 13 are exactly rows 2 to 13 from CSM \mathcal{M} in Fig. 5.

```

(8,99) procedure search(i1, i2)
(0,3)   e := i2
(4,3)   c :=  $f_0$ 
(0,w)   while (e.pop() =  $f_1$ )
(0,1)     rescale i1 over both image a and image b
(0,2)     FT, square, and FT image a
(8,2)     if (a =  $f_1$ )
(8,1)       i1 := LHS of i1
(14,1)      c.push( $f_0$ )
(8,2)     else /* a =  $f_0$  */
(8,0)       i1 := RHS of i1
(16,0)      c.push( $f_1$ )
            end if
            end while
(0,0)     a := c
            end procedure

```

Fig. 8. Pseudocode algorithm to search for a single 1 in a list otherwise populated with 0s. Line numbers give locations of the corresponding piece of code in the CSM machine in Fig. 9. FT: Fourier transform. Images f_0 and f_1 were defined in Def. 5.

