Typeset using the Legrand Orange Book LaTeXTemplate Version 2.2 (30/3/17), downloaded from

http://www.LaTeXTemplates.com.

Original template author:

Mathias Legrand (legrand.mathias@gmail.com) with modifications by: Vel (vel@latextemplates.com). License: CC BY-NC-SA 3.0

(http://creativecommons.org/licenses/by-nc-sa/3.0/)

First printing, September 2017

Preface

This book is, in many ways, a continuation of work that was begun in the 1980s by Jimmy Hargrove, Larry Henry, Murali Raju and myself. In the midst of conversations about artificial intelligence, data flow machines, microcode and architecture tuning, self-awareness, the halting problem, and a variety of other topics, an idea emerged for a new type of computational building block: a reconfigurable, *self*-configurable platform named the PIG ("Processing Integrated Grid").

In 1999, the first patent on the PIG was issued. Later that year, the PIG was renamed to the Cell Matrix, and Cell Matrix Corporation was founded by Lisa Durbeck and Nick Macias.

Over the next 3 decades, research and development on self-configurable systems led to a variety of work in self-repair, autonomous self-assembly, introspective computing, embryonic electronics, and other bio-inspired methodologies. Some of this work led to the development of an analog, continuous-valued version of the Cell Matrix, called the "Songline Processor." Throughout all this work, a set of core ideas emerged, and eventully, the Cell Matrix and the Songline Processor felt less like the end of the journey, and more like steps along a much longer path.

In 2016, EEXIST was developed. This was not a simple step forward, but rather a case of taking many steps backwards, and then moving forward in a very different direction, using past lessons as a guide through this new territory. The result is a system that barely resembles a computational engine. EEXIST is an abstraction of a highly simplified machine, infused with the central ideas of the Cell Matrix and Songline Processor, modified to incorporate concepts from the real world such as continuity of space and time. The result is a system that is difficult to understand, currently impossible to program in any traditional sense, but seemingly very rich in the behaviors it can exhibit.

It is an ongoing challenge to understand the nature of EEXIST, and to figure out how to work with it in its most general, abstract form. This effort is only just beginning, yet already seems to suggest an interesting, useful architecture, applicable to a number of different problem areas. The present work is a description of some aspects of this early work. What is contained herein is not the end of the story, it is only the beginning. My hope is that the interested reader may perhaps find a starting point here for their own research in this area.

I am indebted to many friends and colleagues for numerous discussions and brainstorming sessions related to this and similar topics. My students at Clark College have been a particular source of insight and inspiration in this work, especially Jordan Curry, Stefanie LoSchiavo, Saulius Braciulis and Lydia Brynmoor.

Nicholas Macias
Vancouver, Washington, USA
September 2017

Contents

II Part Two - Experiments

Part One - Theory

1. Introduction

This manuscript describes a work in progress. It is not a complete story, nor is it a story whose conclusion has been reached. It is a description of an ongoing research effort that, in many ways, started 30 years ago.

The ideas presented herein may not seem useful at first. The system – EEXIST – is simultaneously difficult to use and control, and limited in demonstrated applications. It does not do any one thing better than existing systems; it is not a drop-in replacement for a von Neumann architecture. Nonetheless, it is a potentially promising direction in which to explore interesting concepts related to extending our notions of computation.

The point of this work is thus to expose new ideas, new directions in the field of computing. While EEXIST feels like a very different approach to computation and control, it is not a randomly-conceived architecture. It has been deliberately designed, based on lessons learned and observations of the real world. Successful application of

the system to a variety of problems is therefore interesting *beyond the solutions themselves*. It is more the fact that a seemingly unusual architecture, which is so difficult to imagine controlling, can in fact perform these algorithms. This is an important point to remember throughout: it is not the solutions themselves that are intriguing, *it is the challenge of understanding the system at large* that is the main focus of this research.

1.1 Quick View of EEXIST

While carefully reading this manuscript (at least Part One) should give a fairly complete picture of EEXIST, there are doubtless some readers who hope to glean at least a high-level view of the system from this introduction. To that end, here is an attempt at a one-paragraph synopsis of EEXIST.

The idea of this architecture is to define a system where time and space are continuous; where the effect of each action is felt immediately everywhere; and where there is no distinction between subject and object, because everything is at all moments acting on and being acted on by everything in the system.

1.2 Chapter Breakdown

Since the goal of this text is to expand on the above description, to help the reader understand and explore EEXIST (rather than just apply it to problems), it is thus important to understand the background of the system. Part 1 of this text discusses the theory of EEXIST through a number of topics. Chapter 2 describes a very simple type of computer: a Transfer Machine, also called a Zero-Bit Computer (ZBC), presented as a starting point for the development of EEXIST. Chapter 3 reviews relevant background on the Cell Matrix and the Songline Processor, which are used in Chapter 4 to specify design goals for an extended ZBC. Chapter 5 presents a model for this extended system, based on a system of chemical transfers.

Part 2 covers a set of experiments involving application of EEXIST to different problems, including implementation of digital logic; frequency detection and generation; game playing; and control of creatures in a simulated competitive environment.

Part 3 discusses open questions and next steps.

As with this text as a whole, each of these chapters, rather than being a final word on any of these topics, is a launching-off point for future research. The particulars explored in this manuscript represent one set of possibilities in a much higher-dimensional design space.

1.3 Exercises

Some chapters include exercises at the end of the chapter. These are things to think about or try that may help in understanding the material presented in the chapter.

2. Transfer Machines

What is a computer? Early computers were humans: "computer" was originally a job description. The term "electronic computer" was used to differentiate artificial computers from human ones. Computers perform calculations, but more generally they execute *algorithms*. This however is a high-level view of what computers *do*. It does not answer the question "What is a computer?"

At a lower level, computers are systems that store information and process information based on stored instructions. At a machine-code level, this is still a reasonable description; but at a lower level – closer to the hardware – the description changes. Consider a microcoded machine, such as the VAX-11/780 [2]. Inside the CPU, below the level of the machine code, is a *micromachine*. A series of 96-bit microinstructions direct the behavior of the hardware in order to *interpret* the binary machine code. The microinstructions have one main purpose: *to direct the movement of data through the components of the datapath* .

2.1 A Zero-Bit Computer

Note that most of the code in this chapter can be found here [26].

A standard question in a Digital Logic class is: "how many different instructions can be represented with n bits?" The answer (2^n) suggests that more than a few bits is often sufficient, at least in terms of *opcodes*. A related question (sometimes posed in my Discrete Structures class) is: "what is the fewest number of bits required for coding an opcode in a generally-useful computer architecture?" If we reduce this to the case of a transfer machine, the somewhat surprising answer is 0: the system only requires one instruction ("transfer") and, being the only instruction, requires no bits to code it. Each instruction is understood to be a transfer, and thus only the operands need to be expressed. Hence the term "Zero-Bit Computer" (or "ZBC") is sometimes used to describe such a system.

A ZBC can be viewed as a large memory, containing pairs of addresses, each pair containing a source ("SRC") and a destination ("DST") address. The understanding is that, beginning with a first pair, this memory describes a series of transfer operations that are to take place, copying the contents of the SRC location to the DST location. For reasons that will be made clear below, there is only a single memory, containing both the transfer instructions and the data being transferred. In other words, *the transfer instructions are themselves potentially subject to transfer*.

To do useful work on a ZBC, we require a way to do things such as arithmetic operations. Computational blocks – Arithmetic Logic Units (ALUs), multipliers, etc. – are memory-mapped into the address space, so that, for example:

- transferring to address 1011 might copy a value to the A input of an ALU;
- transferring to address 1012 might copy a value to the B input of an ALU;
- transferring to address 1010 might copy a value (meaning *add*) to the FUNCTION SELECT input of an

ALU; and

- transferring from address 1013 might copy the result of the specified function of the two inputs (i.e. their sum).

By having computation and input/output (I/O) blocks mapped into the ZBC's address space, one can write code that performs various procedural steps. For example (using the above-described ALU), listing 2.1 shows code that will add the contents of locations 100 and 101 and store their sum in location 102.

Listing 2.1: ZBC Code to Add Two Variables

```
Address   Contents
   0         100     ; read the first number
   1        1011     ; copy to ALU(A)
   2         101     ; read the second number
   3        1012     ; copy to ALU(B)
   4        2000     ; use loc 2000 to save a constant (1)
   5        1010     ; copy the literal 1 to ALU(FUNC)
   6        1013     ; read the ALU's output
   7         102     ; and save in location 102
2000           1     ; this is a literal used to specify ADD
```

This code assumes the ZBC begins executing by reading locations 0 and 1 from memory, and copying from the address specified in location 0 to the address specified in location 1. If we view MEM as an array describing the memory's contents, the exact behavior is $MEM[MEM[0]] \rightarrow MEM[MEM[1]]$. In other words, all memory references are indirect: the transfer is not from $MEM[0] \rightarrow MEM[1]$ but rather from $MEM[100] \rightarrow MEM[1001]$. This has the desired effect of copying the first variable to be added (stored in memory location 100) to the ALU's A input.

Following this transfer, the ZBC reads memory locations 2 and 3, and transfers from $MEM[101] \rightarrow MEM[1012]$. This process continues with successive pair of memory locations. After the pair at $MEM[6]$ and $MEM[7]$ are read and processed, the sum of the data in $MEM[100]$ and $MEM[101]$ will have been stored in $MEM[102]$. The ZBC has performed an addition of two variables.

Note that if the ZBC executes 1000 consecutive pairs of instructions, it will then attempt to execute the transfer request stored in $MEM[2000]$ and $MEM[2001]$. Since $MEM[2000]$ was being used to store a literal value, exe-

cuting a transfer based on that stored value would be undesirable. This is, of course, a familiar situation in most stored-program computers with a single memory: *there is no intrinsic differentiation between code and data*. More generally, in Listing 2.1, it's unspecified what should happen after memory locations 6 and 7 are read and processed: whatever happens to be in memory next will be executed. This raises the need for some sort of branch capability.

2.2 Branching in a ZBC

Branching is handled very simply, by **mapping the program counter (PC) itself into the memory of the ZBC**. In the simulated implementation, the PC is mapped to memory address 65,535. Thus, the following modification to listing 2.1 causes the code to loop repeatedly:

Listing 2.2: ZBC Code to Add Two Variables and loop forever

```
Address   Contents
      0        100    ; read the first number
      1       1011    ; copy to ALU(A)
      2        101    ; read the second number
      3       1012    ; copy to ALU(B)
      4       2000    ; use loc 2000 to save a constant (1)
      5       1010    ; copy the literal 1 to ALU(FUNC)
      6       1013    ; read the ALU's output
      7        102    ; and save in location 102
      8       2001    ; read the literal 0
      9      65535    ; and copy to the PC
   2000          1    ; this is a literal used to specify ADD
   2001          0    ; this is a literal used for branching
```

The additional instruction (at memory locations 8 and 9) specifies a transfer from $MEM[2001] \rightarrow MEM[65535]$, which copies a 0 to the PC, thus causing the instructions to be re-executed beginning with location 0.

2.3 Conditionals in a ZBC

With procedural statements and looping, we almost have a complete programming language: the last piece we need is a way to do conditionals. This is already available though, since we can manipulate the PC based on the value of a variable. For example, listing 2.3 shows a simple if/then/else statement. If the contents of memory location 100 is 0, the

code will jump to location 10; if the contents is 1, the code will jump to location 20.

Listing 2.3: ZBC Code for Conditional Execution

```
Address   Contents
      0        100    ; read the branch variable (must be 0 or 1)
      1       1011    ; copy to ALU(A)
      2       2002    ; read the constant 2004
      3       1012    ; copy to ALU(B)
      4       2000    ; read the constant 1
      5       1010    ; copy to ALU(FUNC)
      6       1013    ; read ALU's output (=either 2004 or 2005)
      7          8    ; and save in next instruction SRC
      8       0000    ; placeholder - copy from 2004 or 2005
      9      65535    ; to the PC. This causes a jump to 10 or 20
   2000          1    ; this is a literal used to specify ADD
   2002       2004    ; another literal
   2004         10    ; branch target if MEM[100]=0
   2005         20    ; branch target if MEM[100]=1
```

Listing 2.3 uses the same ALU operations to add the contents of $MEM[100]$ to the constant 2004, giving a sum of either 2004 or 2005. That sum is saved in MEM[8], so that when that instruction is executed, it specifies either a transfer from $MEM[2004] \rightarrow MEM[65535]$ or $MEM[2005] \rightarrow MEM[65535]$, which copies either 10 or 20 to the PC, causing the next instruction executed to come from either location 10 or 20.

One complication to the above code is that $MEM[100]$ must be exactly 0 or 1 (since it's basically being used as an index into a branch table). Using a comparator (mapped into the ZBC's address space, just like the ALU) will work well here, to generate a 1 or 0 based on a specified comparison.

2.4 A ZBC Programming Language

This is all perhaps a bit awkward, but can nonetheless be used to methodically code conditional statements which, along with branches, allow implementation of most any algorithm. To write code more easily, a simple programming language can be defined as follows:

- two numbers separated by a colon (:) represent a location/contents pair to store in memory;
- a single number represents a number to be stored in the next successive location in memory;

- everything between a semicolon (;) and the end of the line is a comment;
- blank lines are ignored

So, for example, the code in listing 2.3 could be written as shown in listing 2.4 (comments have been dropped):

Listing 2.4: ZBC Code for Listing 2.3

```
0:100
1011
2002
1012
2000
1010
1013
8
0
65535
2000:1
2002:2004
2004:10
20
```

2.5 A ZBC Interpreter

Using this shorthand, we can write code suitable for interpretation in a simple ZBC interpreter. Listing 2.5 shows a short C program that ingests code such as shown in listing 2.4 and interprets its execution. You can find this program in the file "xm.c" available here [26]

Listing 2.5: Code for ZBC Interpreter

```
#include <stdio.h>

int mem[65536]; // main mem!
#define PC mem[65535] // PC is actually stored at end of mem
#define DEBUG (mem[65534]) // set to 1 to turn on debugging

main(int argc, char **argv)
{
  FILE *fp;
  char buffer[120];
  int addr=0,data,t1,t2,from,to,status;

// load program into memory
  if (argc==2) fp=fopen(argv[1],"r"); else fp=stdin;
  while (NULL != fgets(buffer,120,fp)){
    if (2==(status=sscanf(buffer,"%d:%d",&t1,&t2))){ // addr:data
      addr=t1;data=t2;
    } else data=t1; // just data
    if (status>=1) mem[addr++]=data;
  }

// main processing loop...
  while (1){
    if (DEBUG) printf("%d:%d[%d]=>%d\n",PC,mem[PC],
                      memRead(mem[PC]),mem[PC+1]);
    from=mem[PC++];PC&0xffff;
    to=mem[PC++];PC=PC&0xffff;
    memWrite(to,memRead(from));
```

```
  }
}

// memory-mapped-hardware simulation

int alufunc=0; // 1=+,2=-,3=*,4=/
int aluA=0,aluB=0;
int compfunc=0; //1=> 2==
int compA=0,compB=0;

memWrite(int loc, int data)
{
  switch (loc){
// 1000=DISPLAY
    case 1000:display(data);break;

// 1010=ALU
    case 1010:alufunc=data;break;
    case 1011:aluA=data;break;
    case 1012:aluB=data;break;

//1020=COMPARATOR
    case 1020:compfunc=data;break;
    case 1021:compA=data;break;
    case 1022:compB=data;break;

// If not HW, then just write to memory
    default:mem[loc]=data;
  }
}

memRead(int loc)
{
  switch(loc){
// ALU
    case 1013: // ALU
      return(aluvalue());

// COMPARATOR
    case 1023: // comparator
      return(compvalue());

// Not HW: just read from memory
    default:return(mem[loc]);
  }
}

// support code

display(int data)
{
  printf("%d ",data);fflush(stdout);
}

aluvalue() // calculate ALU value
{
  switch(alufunc){
    case 1:return(aluA+aluB);
    case 2:return(aluA-aluB);
    case 3:return(aluA*aluB);
    case 4:if (aluB != 0) return(aluA/aluB);
           return(0);
  }
  return(0);
}

compvalue() // calculate comparator value
{
  switch(compfunc){
    case 1:return((compA>compB)?1:0);
    case 2:return((compA==compB)?1:0);
  }
  return(0);
}
```

The main processing loop of this code is only 3 lines,

reflecting the very simple nature of the ZBC's architecture. The code also includes some memory mapped hardware, including:

- a display at $MEM[1000]$;
- an ALU at $MEM[1010-1013]$;
- a comparator at $MEM[1020]-MEM[1023]$;
- the PC at $MEM[65535]$; and
- a debug flag at $MEM[65534]$.

2.6 Sample ZBC Programs

Given this very simple architecture, programs for performing even modest tasks can be fairly lengthy. Listing 2.6 shows code for counting down from 10 to 0, displaying each value on an output port (the display mapped to ZBC address 1000).

Listing 2.6: ZBC Code to Count Down From 10 to 0

```
65535:0 ; Set PC=0
65534:0

; constants
511:1
512:10
513:0
514:2
515:150 ; for infinite loop
520:110
500:0   ; X

; start of program
0:512
500     ; x=10

; LOC 2
500
1000    ; Display X
514     ; constant=2
1010    ; ALU func=sub
500
1011
511
1012
; now 1013=X-1
1013
500     ; X=X-1

; see if X=0
514
1020    ; comparator "=" function
500
1021    ; A
513     ; 0
1022    ; 1023 shows X==0

511
1010    ; ALU "+"
520
1011    ; 110
```

```
1023     ; 0 or 1
1012     ; to B...[1013]=110(x<>0) or 111 (X==0)

; branch to loc 100
99
65535
99:100

; here is loc 100
100:1013
101:102
102:000
103:65535 ; if X<>0 this branches to [110] else [111]

110:120 ; branch to 120 if x<>0
111:150 ; branch to 150 if x==0

120:514
65535    ; LOOP!

150:515
65535    ; stay here
```

A more complex example is shown in listing 2.7, which shows code for generating prime numbers.

Listing 2.7: ZBC Code to Generate Prime Numbers

```
65535:0 ; initial PC=0
65534:0 / disable debugging :)

;*** HW addresses
;1000: DISPLAY

;1010: ALU func (1:C=A+B;2:C=A-B;3:C=A*B;4:C=A/B)
;1011: A
;1012: B
;1013: C

;1020 Comparator relation R (1:>,2:=)
;1021: A
;1022: B
;1023: A r B

500:3    ; A
501:1    ; B
502:0    ; X

; Constants
511:1
512:2
513:3
514:4

; Program starts at 0

0:511
1:501    ; b=1

; B=B+2
2:511
1010 ; ALU set to +
501
1011
512
1012
1013
501 ; B=B+2

; X=A/B
514
1010 ; ALU set to /
500
1011
501
```

```
1012
1013
502 ; X=A/B

; is B>X?
511
1020 ; comparator set to >
501
1021
502
1022
; 1023 is 1 if b>x else 0
; use to to determine a branch address

511
1010 ; ALU=+
1023
1011
100 ; =101
1012
; 1013 (adder output) is either 102 (B>X) or 101

; jump to 110 (just so we know our PC!)
109
65535 ; read from loc 109 (=110) to PC
109:110 ; loc 109 contains 110

100:101
101: 150 // B <= X code goes at 150
102: 120 // B>X code goes at 120

; (loc 110: continue here)
110: 1013
111: 112 ; loc 112 contains either 101 or 102
112: 0 ; contains 102 if B>X
65535  ; now the PC is 120 if B>X

; PRIME!
120: 500
1000 ; display the prime number
511
1010
512
1011
500
1012
1013
500 ; A=A+2

511
501 ; B=1 (again)

510 ; constant 0
65535 ; Jump to loc 0


; See if X==int(X)
150:513
1010 ; ALU func=*
502
1011 ; X
501
1012 ; B
; [1013]=B*X
512
1020 ; COMP r is "="
1013
1021
500
1022
; [1023]=1 for composite, 0 for continue
179
65535 ; branch to [179] (=180)

; setup for conditional branch to 200 (1023=0) or 220 (1023=1)
165:167 ; constant
166:0 ; either 167(cont) or 168 (comp)
167: 200 ; (continue)
```

```
168: 220 ; (comp)

179:180
; branched here...
1023 ; 0 or 1 (continue or composite)
1011
165
1012
511
1010 ; [1013]=167 or 168
1013
188
; here is loc 188
0 ; filled in with either 167 or 168
65535 ; branch to that loc

; continue
200:512 ; branch to loc 2 (B=B+2)
65535

; composite
220:511
1010
500
1011
512
1012
1013
500 ; A=A+2
510
65535    ; branch to loc 0 (B=1)
```

As with many things at this level, this code is easier (though not easy) to write than to read! While the ZBC is not very practical for general purpose programming, it represents a pared down, minimalist architecture that serves as an ideal starting point from which to re-build our notion of computation. Before we can do this, we must review some of the ideas that drove this redesign: specifically, the Cell Matrix and the Songline Processor (SLP)[3]. These are the topic of the next chapter.

2.7 Exercises

1. Download the ZBC code (xm.c), compile it, and run the count and prime test files (all of these are available here [26].
2. Write a ZBC program for adding two numbers.
3. Write a ZBC program for printing the larger of 2 numbers.
4. Write a ZBC program for adding a set of numbers.
5. What is a minimal set of memory-mapped hardware that allows a ZBC to be used for general-purpose computing?

3. Cell Matrix/SLP Background

Extensive details on the Cell Matrix and the Songline Processor can be found at the main research website http://songlinesystems.com [1]. What is presented here is a high-level summary, specifically of those features that led to the development of EEXIST.

3.1 Cell Matrix

The idea of reconfigurable logic is straightforward. Software is changeable, morphable, able to be modified with its behavior changing accordingly (that's the "soft" aspect of it). Hardware, in contrast, is rigid, fixed in form and function, and generally difficult to modify without some sort of invasive procedure (de-soldering, re-wiring, etc.) (that's the "hard" aspect of it). Reconfigurable hardware combines the best of these two models, offering the speed of a hardware system with the flexibility of a software system. Thus devices such as field programmable gate arrays [14]

("FPGAs") became popular in the 1980s.

While FPGAs extend the notion of software to the hardware domain, early devices lacked one important aspect of software: the ability of software to examine and modify itself. This aspect – which is a hallmark of the stored program computer [15] – offers many advantages over a system whose configuration is controlled only from outside the system.

The Cell Matrix [16] addresses this by endowing the basic reconfigurable elements of the system with the ability to directly read and write the configuration of other elements. Before discussing this ability for *self modification*, it will be useful to discuss the overall structure of the Cell Matrix, as well as its basic configurability.

3.1.1 Cell Matrix Architecture

The Cell Matrix is comprised of a large grid of simple, identical elements called "cells." Each cell has a set of inputs and outputs, connecting it to a fixed set of *neighbors*. The inputs to a cell are processed by the cell's internal program, and the cell generates outputs accordingly. Figure 3.1 shows a set of connected cells.

The program inside each cell is a simple truth table, which combinatorially maps inputs to outputs. In the 2-D example shown in figure 3.1, each cell has 4 neighbors, and thus continually receives a total of 4 bits of input from its neighbors. The cell also produces a single output bit to each neighbor, thus requiring 4 output bits. This mapping can be defined by a truth table, as shown in figure 3.2.

This truth table can itself be defined by the 64 bits in the output columns. These are stored in a per-cell memory and define the basic input-to-output mapping of a cell. Note that *the mapping from input to output is unclocked*: when any inputs change, the outputs change in response as soon as possible. Nothing here is synchronized to any sort of global clock. While this may make the design process more complicated than a synchronous one, it offers advantages in speed and flexibility (and synchronicity can also be added

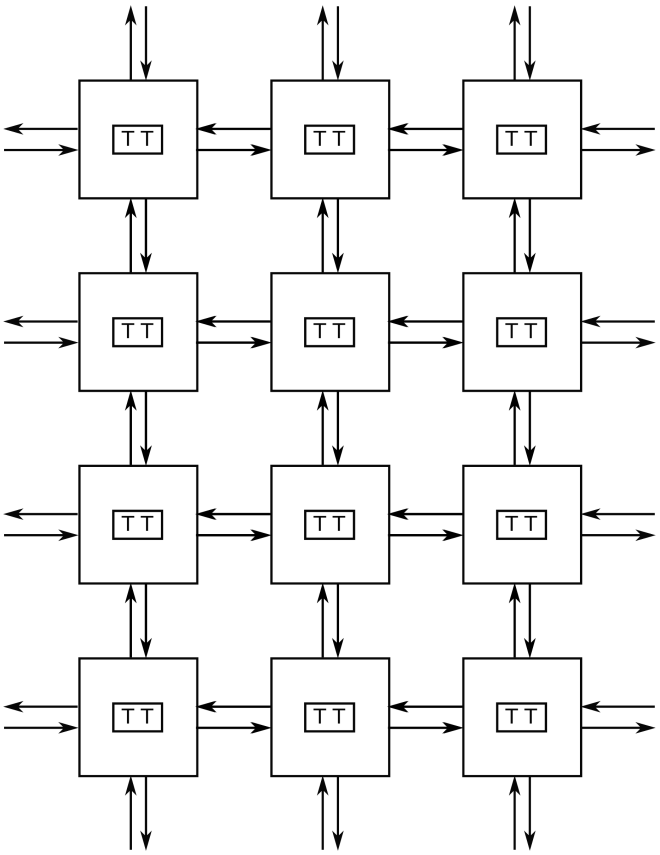


Figure 3.1: Two-Dimensional Matrix of Cells. All cells are identical except for the program stored inside each one (in the block labeled "TT"). Connections to neighbors are shown with arrows.

via additional mechanisms described below).

3.1.2 Sample Cell Matrix Circuits

Designing circuits with these cells is fundamentally no different from designing with standard digital logic blocks. One can, for example, configure cells to act as logic gates (AND, OR, etc.). Cells can also be configured to act as simple wires. By combining these in the right way, general digital circuits can be constructed. Figures 3.3 - 3.5 show some sample circuits, along with the truth tables used to

INPUTS				OUTPUTS			
N	S	W	E	N	S	W	E
0	0	0	0	D_{03}	D_{02}	D_{01}	D_{00}
0	0	0	1	D_{07}	D_{06}	D_{05}	D_{04}
0	0	1	0	D_{11}	D_{10}	D_{09}	D_{08}
				...			
1	1	1	1	D_{63}	D_{62}	D_{61}	D_{60}

Figure 3.2: Truth Table For a Cell Connected to Four Neighbors. Neighbors are referenced by their compass directions ("N," "S," "W" or "E") relative to the cell. D_i refers to the i^{th} bit of the truth table.

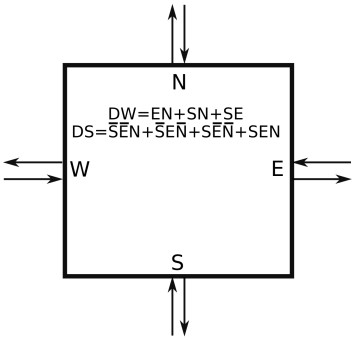
configure each cell.

In figure 3.3, a single cell is being used as a one-bit full adder : the truth table is simply set up to produce the proper outgoing sum and carry bits in response to the inputs. Note that the cell has unused inputs and outputs, but they still appear in the truth table.

Figure 3.4 shows a set of 8 such one-bit adders, situated side-by-side so one adder's outgoing carry is fed to the next adder's incoming carry, thus creating an 8-bit ripple-carry adder. The inputs are fed in parallel to the north and south, and the parallel sum appears to the south.

Note that this layout could, in theory, be extended to any number of cells/adders/bits: 1024 cells would produce a 1024-bit adder. While the maximum propagation delay from a 1024-bit ripple carry adder is likely to be prohibitively large, the principle is a general one: by carefully designing blocks of cells to be modular, larger circuits can sometimes be constructed simply by placing these blocks together in the matrix. This is one key to *autonomous circuit synthesis*.

Figure 3.5 shows a different type of Cell Matrix circuit, this one comprised of two cells. This configuration imple-



INPUTS				OUTPUTS			
W	S	E	N	W	S	E	N
0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	0
0	0	1	1	1	0	0	0
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	0
0	1	1	1	1	1	0	0
1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0
1	0	1	0	0	1	0	0
1	0	1	1	1	0	0	0
1	1	0	0	0	1	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	1	0	0

Figure 3.3: A Single Cell Setup As a one-Bit Full Adder. Incoming bits are supplied to the north and south; the incoming carry is applied to the east; the sum is presented to the south; and the outgoing carry appears on the west. The Boolean equations for the cell's truth table are shown inside the cell.

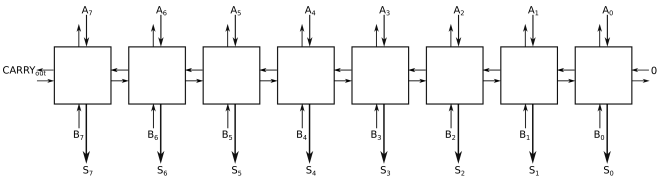


Figure 3.4: 8 Cells Setup As an 8-Bit Adder. Inputs A and B are supplied to the north and south, and the sum S is presented to the south. Each cell is identical to the one shown in figure 3.3.

ments a simple D flip flop: a one-bit storage element. The setup is straightforward: the cell on the left either sends an incoming bit from the west to the east, or echoes an incoming bit from the east back to the east. The cell on the right provides feedback, reflecting whatever is sent from the cell on the left. When the gate is 1, the incoming data bit is sent to the cell on the right; when the gate drops to 0, that bit becomes trapped inside the cells, being passed from one to

the other repeatedly.

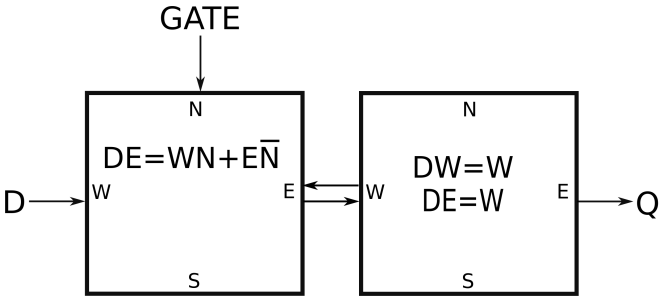


Figure 3.5: A Data Flip Flop. The D input comes from the west, the clock is presented on the north; Q comes out from the east.

Thus it is possible to design sequential circuits that operate with a clock and synchronize operations using e.g. standard state machine design techniques. Figure 3.6 shows a more-complex circuit employing logic blocks, flip flops and wires (wires are implemented exactly like other functions, i.e. the equation $D_E \rightarrow D_W$ copies data from the eastern D input to the western D output). This circuit implements a small RAM. One of 16 rows can be addressed with the inputs A_3, A_2, A_1 and A_0. When the WRITE input is high, the 4 data bits $D_3 - D_0$ (from the top of the circuit) are loaded into the selected row of flip flops. When the READ input is high, the selected row's flip flops supply outputs to $D_3 - D_0$ at the bottom of the circuit.

3.1.3 Configuration of Cells: C-Mode

The above description of cells covers their behavior in terms of *data processing*, i.e., transforming inputs to outputs. While this is sufficient for implementing standard digital circuits, it doesn't allow for introspection as described in the start of this chapter: in particular, it does not explain *how cells are configured*. To allow cells to be configured by other cells, we add an additional input and output to each side of the cell, giving a cell as shown in figure 3.7.

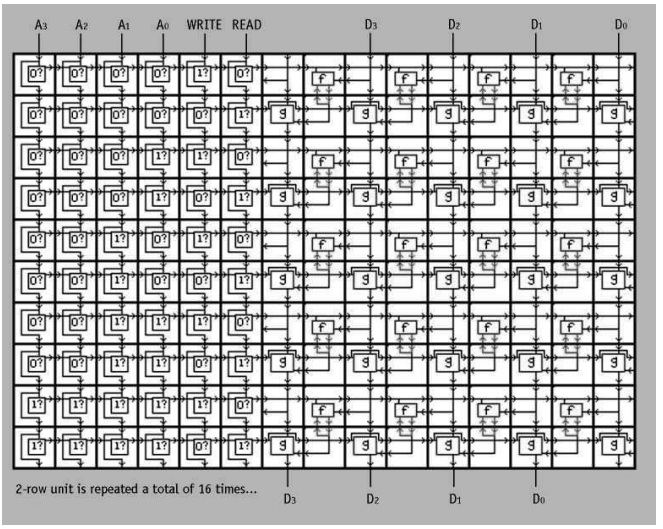
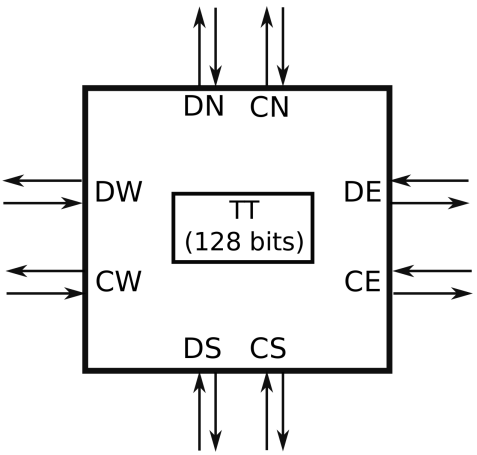


Figure 3.6: More-Complex Cell Matrix Circuit. This layout implements a small memory. The left side of the circuit routes address inputs ($A_3 - A_0$) from top to bottom; the 4^{th} column outputs a 1 to the right when the address matches that row's address. The match output is combined with the READ and WRITE inputs to drive the array of flip flips (on the right side of the circuit). Each flip flop is comprised of a cell ("f") and a feedback circuit to its right (similar to the circuit in figure 3.5. Inputs are loaded in response to a match from the address block combined with a WRITE signal. Flip flop outputs are sent to the "g" blocks, which either pass the prior block's output from north to south (if there is no match); or pass the flip flop's output to the south (if the flip flop is being addressed by the $A_3 - A_0$ inputs).

In this cell, the C inputs are used to control the configuration of the cell, as follows:

- if $C = 0$ (called "D-mode"), then the D inputs are processed as described above: they reference a row of the cell's truth table, which contains the outputs that are sent to neighboring cells;
- if $C = 1$ (called "C-mode"), then the corresponding D input is used to supply truth table bits, i.e., to populate the cell's truth table. The corresponding D output is



INPUTS				OUTPUTS							
N	S	W	E	CN	CS	CW	CE	DN	DS	DW	DE
0	0	0	0	D_{007}	D_{006}	D_{005}	D_{004}	D_{003}	D_{002}	D_{001}	D_{000}
0	0	0	1	D_{015}	D_{014}	D_{013}	D_{012}	D_{011}	D_{010}	D_{009}	D_{008}
0	0	1	0	D_{023}	D_{022}	D_{021}	D_{020}	D_{019}	D_{018}	D_{017}	D_{016}
				...							
1	1	1	1	D_{127}	D_{126}	D_{125}	D_{124}	D_{123}	D_{122}	D_{121}	D_{120}

Figure 3.7: Full Cell Matrix Cell. There are two *inputs and outputs ("C" and "D") connecting this cell to each neighbor. The D lines are used for regular data processing, while the C lines are used for configuring the cell.*

used for reading the truth table's current contents.

C-mode operations are clocked via a system-wide clock (which is only used in C-mode, but can be tapped into and utilized by D-mode circuitry).

Figure 3.8 shows the interaction of C- and D-modes, as well as a typical read/modify/write operation. When the cell (configured as a simple wire $DE \rightarrow DW$) is in D-mode, its eastern output reflects its western input, regardless of the state of the system clock. When the cell enters C-mode, the D output changes to reflect the "first" bit (D_{000} according to the pre-defined ordering shown in figure 3.7) in the cell's truth table. On the rising edge of the clock, the D input

is sampled and saved inside the cell. On the falling edge, that saved value replaces the first bit in the truth table, and the D output now reflects the second truth table bit. When the C input returns to 0, the cell re-enters D-mode, and the D output now reflects the results of applying the cell's D inputs to its new truth table.

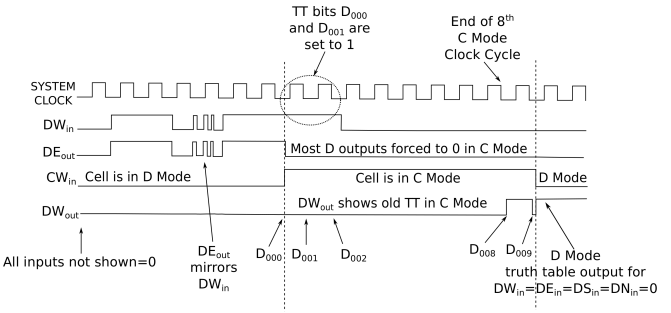


Figure 3.8: Interactions of C-mode, D-mode and the System Clock. In D-mode, the system clock has no effect on the cell. In C-mode, reading and writing of the cell's truth table is synchronized to the system clock.

This simple setup allows a number of interesting circuits to be implemented. For example, figure 3.9 shows a *cell reader*. The cell on the right places the target cell into C mode by asserting a 1 to its CW output (which is the target cell's CE input). The target cell sends its current truth table bits out its DE output, which the cell reader ingests from its DW input. The cell reader copies those old truth table bits back to its DW output, which loads them back into the target cell's truth table (thus effecting a non-destructive read). The target cell also copies those truth table bits to its own DS output, where another cell could pick them up and process them.

Figure 3.10 shows a *cell replicator* , which is similar to the cell reader, but with one small change: the CS output is set to 1. With this change, the bits being read from the target cell's truth table will be copied into the cell to the south, thus making that cell an exact copy of the target cell.

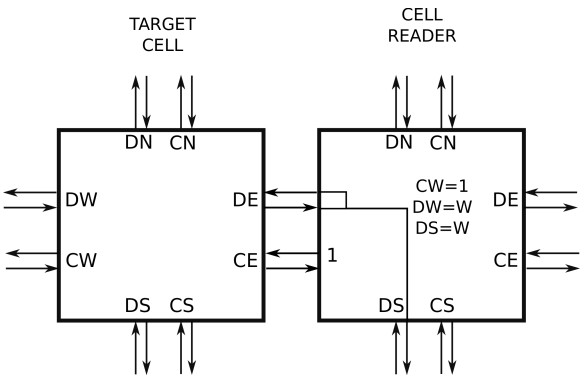


Figure 3.9: A Cell Reader. The target cell's truth table is read by the cell reader. As bits are received, they are re-sent to the target cell, making the read non-descructive.

Figure 3.11 shows a further embellishment to the cell replicator. In this circuit, the cell replicator is connected to a horizontal row of 3 circuits which copy bits from the west to the south and east, while asserting their CS output. The result is that four copies of the target cell are created (in the second row of the circuit). Note that these copies are created in parallel: the writing of all 4 truth tables occurs at the same time. This is thus a *parallel cell replicator*.

It should be noted that the 3 cells on the right of the top row are identical to each other. This means that these cells could themselves have been configured in parallel. Of course, that would require *another* circuit to perform *that* parallel replication, so it would seem parallel replication of n cells always requires (at least) n operations to set it up. In fact, this is not true: done properly, it takes on the order of n operations to configure n^2 cells in a 2D matrix. On a 3D matrix, n operations are sufficient for configuring n^3 cells. Circuits for doing these better-than-parallel builds are called *"Medusa Circuits"* [17].

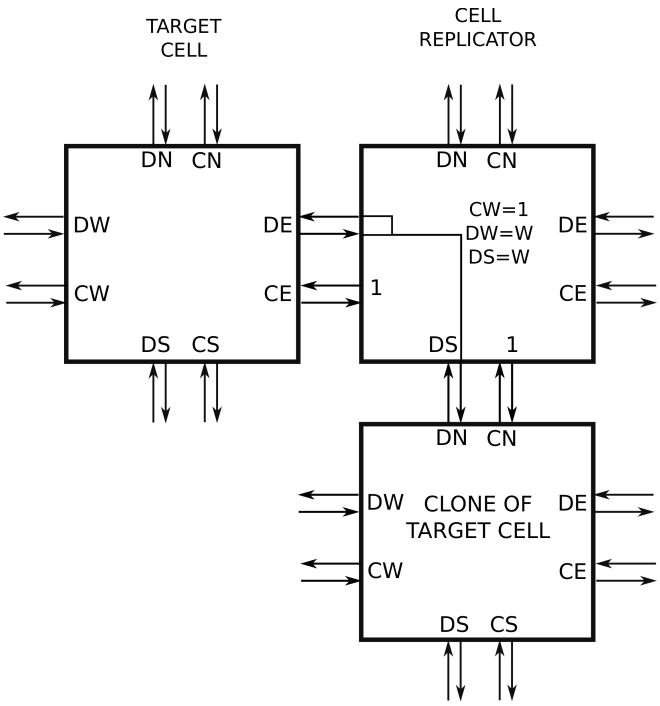


Figure 3.10: A Cell Replicator. The cell replicator reads the target cell's truth table bits, but copies those bits into the truth table of the cell to the south, thus making it an exact copy of the target cell.

3.1.4 Applications of C-mode

Some of the most important aspects of the Cell Matrix are the following:

- the process of configuring cells is intrinsic to the overall architecture;
- control over cells can be realized *from within the system itself*;
- the control is distributed throughout the system;
- controlled and controlling entities are interchangeable ("non-dualism"); and
- the homogeneity of the cellular organization makes the architecture highly scalable.

By properly utilizing these features, a variety of behav-

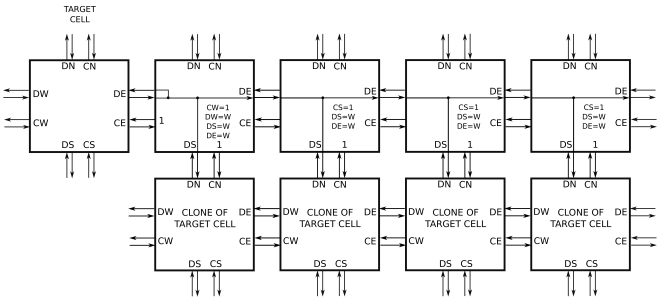


Figure 3.11: A Parallel Cell Replication Circuit. The top row of cells copies truth table bits from the target cell to the four cells on the second row. The replication occurs in parallel: all 4 truth tables are written simultaneously.

iors can be realized in circuits built on the Cell Matrix. The following is a partial list of some of the areas where the Cell Matrix is particularly well-suited:

- Highly-parallel processing. If a problem can be divided into a set of identical, independent sub-tasks, then groups of cells can be configured to perform one set of sub-tasks, and then replicated to perform multiple sets in parallel. While overhead such as control and communication potentially obviate such performance gains, the ability to build custom control and communication channels (in parallel) makes it possible to realize significant speedups. This is especially true for so-called "embarrassingly parallel" problems. Examples include massive search problems; simulation of 2-D and 3-D systems; finite element analysis; and so on.
- Fault tolerant computing. Much work has been done on using the Cell Matrix's self-configurability to build and run test circuits in order to check for hardware defects. The trick – being able to handle potential errors in the test circuits themselves – plys the self-configurability and non-dualism of the system. Moreover, parallel testing is possible, as is parallel synthesis of the test circuits themselves: see this final report

[18] on a NASA SBIR investigating use of the Cell Matrix for autonomous fault handling.

- Adaptable Computing. There are domains where being able to change the micro-architecture of a system at run-time may be useful, for example:
  - if the same operation is being performed on the elements of a large dataset, multiple hardware instances can be created to operate on the data in parallel;
  - if an array processor needs to switch between integer, floating point and character processing, the underlying hardware can be re-purposed to the required datatype;
  - if a set of instructions are repeatedly executed more frequently than other sections of code, they can be cast into hardware, allowing for faster execution (just-in-time hardware synthesis).

  These are but a few examples of where adaption of the hardware may be useful. In all cases though, it is the ability of the hardware to *manage the modification of itself* that makes the Cell Matrix a good fit.
- More generally, the interplay between C and D mode allows for the possibility of, say, compiling high-level code into a mix of software and hardware. This is no different from a compiler that takes advantage of a hardware floating point accelerator (FPA): it simply adds a layer of designing and implementing the FPA alongside the machine code.
- Simulation of a Cell Matrix (yes, this is a thing!) While simulation in software is straightforward, it is also inherently slow, as performance degrades in proportion to the number of cells being simulated. With a hardware-based simulation, there is no inherent slowdown as more cells are added to the simulation. But being able to simulate a matrix allows, for example:
  - the ability to freeze the system, or to step it forward slowly while observing its state;
  - the ability to examine internal cells (whose in-

puts and outputs are normally only available to neighboring cells);
  - the ability to address individual cells and interact directly with their truth tables;
  - the ability to reset the state of part of the system without performing a system-wide reset;
  - the ability to efficiently record a history of system states, with an option for rolling back to a prior state;

  and so on.
- The Cell Matrix can be used as a *process improvement driver* for support of aggressive manufacturing. By targeting the matrix and using its ability for self-analysis, it can help diagnose manufacturing errors within itself, thus helping to debug the manufacturing process itself. One recent theoretical example of this is using self-assembly to build a physical matrix from a collection of 3D cells. The cells are manufactured in parallel, and then allowed to self-assemble into a 3D array. In so doing, their orientation is uncontrolled, meaning the final matrix has cells whose orientation is effectively random. This makes it impossible to load truth tables in a meaningful way: building a wire to copy data from a given side to another requires knowing whether those sides are on the north, south, etc. of the cell, and without a fixed physical orientation, this is unknown. However, the cells' ability to introspect allows for circuits and algorithms that let the cells discover and correct-for their own orientation (in parallel). See http://journal.frontiersin.org/article/10.3389/frobt.2016.00002/full [19] for details.
- The Cell Matrix can be embedded alongside other hardware (e.g. MEMS-based systems), and used as a distributed control and communication network, *with all the inherent advantages of massive parallelism, fault tolerance and scalability*.

3.2 Songline Processor

The central themes of the Cell Matrix – self-configuration, homogeneity of the cells, lack of centralized control, nearest-neighbor interconnect – have proven interesting and useful in different ways. Nonetheless, the underlying architecture still derives from more standard models of computation: the system is binary in nature; it employs a clock for reading and writing cell memories; and, despite the interchangeability of modes, each cell is, at any given moment, *either* in D or C mode, i.e., executing its program or being programmed.

The Songline Processor (SLP) is an attempt to retain those unique features of the Cell Matrix, while freeing the architecture from the more standard characteristics. This changes the C and D mode behaviors of the Cell Matrix as follows:

- The values being passed from cell to cell are real-valued vs. binary. This in itself is not such a big difference. In a typical digital system, the binary values are actually voltages, but the circuitry simply restricts those voltages to one of two ranges ("high" and "low"; or 1 and 0). To pass a real-valued signal, just think perhaps of a voltage that can vary anywhere between 0V and 5V. So the D inputs and outputs are real-valued.
- The *C* inputs and outputs are also real-valued. This requires a reinterpretation of how a cell's mode affects its behavior. Instead of mode being a binary state (say 1=C Mode and 0=D Mode), the mode is now a mix of C and D modes. In practice, this means a cell's truth table may be partially perturbed by incoming D values; and D outputs may be a mix of a cell's program output and the program itself.

Implementing these changes requires changing the nature of a cell's truth table. In a cell with binary inputs and outputs, it is possible to exhaustively describe all possible input combinations with a finite number of rows. Adding real-valued outputs doesn't in itself change this: the entries in the truth table's output columns are simply real numbers

instead of binary digits (bits). But since one cell's outputs are connected to another cell's inputs, the *inputs* to a truth table are now also real-valued. This means the set of possible input combinations is (uncountably) infinite. One cannot write a table with a finite collection of rows to represent all possible input conditions: even for a single input, the set of possible input values has the same cardinality as the set of real numbers.

Viewed differently though, this is really not so mysterious: the "truth table" is now simply a real-valued function of real variables (e.g. $z = sin(x^2 + y^2)$). Such a function, however, is not easily stored in a table (unless the input space is discretized). Moreover, even if the function were somehow stored (say, as a curve or a surface), it's not clear how to perform the C mode operations of reading and writing this function. For a curve (e.g. a function of a single variable), one may sweep the input from its minimum value to its maximum value, and thus read/write the entire essence of the function in a fixed amount of time. For 2 or more inputs, there's no obvious (continuous) way to sweep the entire space of input combinations in a finite amount of time. Utilizing some sort of space-filling curve may work, but this is currently an unsolved problem.

By abandoning the system clock though, there are other ways to transfer these functions: for example a 2D surface can be used to transform a second surface so as to mimic the first (imagine a 2D sheet of plastic, shaped to represent the Z value of $z = f(x,y)$, and used as a mold to shape a second target piece of plastic).

There are other possibilities as far as implementation: further details are available here [3]. Despite these issues, simulation is still feasible, and has been used to develop and test some sample applications of the Songline Processor. For example, a single cell can be used to multiply two inputs: this is effectively an amplifier. Differentiation, integration, sample-and-hold, amplitude modulation, and other types of signal processing are each easily achieved with just a few cells. This suggests that the Songline Processor is,

in some ways, fundamentally different from a traditional digital/von Neumann machine. These differences suggest ways to extend the Zero-Bit Computer, as discussed in the next chapter.

3.3 Exercises

1. Write the Boolean equations for a single-cell 2-1 selector on the Cell Matrix.
2. Design a T flip flop on the Cell Matrix.
3. Design a T flip flop without using D flip flops. This can be done with only 3 cells.
4. Design a 4 bit register.
5. Design a 4 bit register using only 5 cells (note that the clock will need to be sent to multiple cells, and the wires for routing the clock signal are not included in the 5 cells).
6. Design a single cell for swapping the truth tables of two neighbors.
7. Design a circuit on the Songline Processor for measuring how similar two signals are to each other.
8. Design an SLP circuit for mathematically composing two functions.
9. Design an SLP circuit for composing two functions, and programming a cell with that new composite function.

4. Enhancing the ZBC

Recall that the ZBC is basically a transfer machine, programmed by naming pairs of SRC/DST addresses. The machine repeatedly performs copies from $SRC \rightarrow DST$, but despite this perhaps peculiar architecture, the system is still essentially a von Neumann machine: it has a program counter (PC), which is used to pull instructions from memory, which perform read/modify/write operations on the memory. The PC is automatically incremented from instruction to instruction, but can also be explicitly loaded.

4.1 Removing the Ego

This architecture (as with most stored-program computers) already contains one important aspect of the Cell Matrix: the interchangeability of code and data. The contents of memory can be interpreted as data or as instructions, and in general, it is impossible from simply looking at the contents itself to tell whether that contents is data or code. We wish to

preserve this interchangeability (this is the "egoless" aspect of EEXIST).

Despite this interchangeability, there is still an imperfection in this non-dualism. While a cell within the Cell Matrix can operate in *either* C or D mode, it is, at any given time, in only one of those modes. It's not possible for a cell to be in *both* C and D mode. The Songline Processor makes some progress in this area, as its cells' C inputs accept a real-valued number, allowing a cell to operate in a partial-C/partial-D mode. This is still imperfect: perfect non-dualism suggests each cell be **fully** in both C-mode and D-mode at all times, i.e. that it can be both a subject and an object simultaneously. The shortcoming is not really in the architecture *per se*, but rather in the fact that instructions operate one at a time, so that (as with the Cell Matrix), some piece of memory may first be a piece of code, but then may later be treated as a piece of data, and then still later as, again, a piece of code. This is our first hint that sequential execution will need to be abandoned.

4.2 Continuity of Time

Both the Cell Matrix and the Songline Processor have a schism in how they process information: in (pure) D-mode, everything happens asynchronously. Inputs cause outputs to change immediately, and those outputs flow into connected inputs, and so on. All this happens without a clock. But in C-mode, there is a system-wide clock which is used to control the sampling and presentation of inputs and outputs representing the new and old code. If we are to perfectly merge C and D mode so that every piece of the system is at all times (potentially) both a subject and object of a transfer, we will need to decide whether everything is clocked or everything operates in a dataflow mode. Using the natural world as a model, it seems most natural to have the entire system operate without a clock. This however makes the notion of a PC mostly unusable. Consider a typical description of the PC: "the PC is incremented at the beginning of

an instruction's execution cycle." This description is very clock-centric: the notions of "beginning" and "execution cycle" each convey a sense of clocked operation.

But if the PC is eliminated, what determines which instruction is being executed? The answer is simple: *all instructions are executed simultaneously*. This is perhaps a difficult thing to imagine. In the simple ZBC, each instruction specifies a transfer from $SRC \rightarrow DST$, and this transfer is an atomic unit: it begins, the transfer takes place, and then the instruction is finished. Rather than viewing this as clocked, we may view it as *discretized*. Following that line of thought, envisioning this as a clockless process means viewing the transfer as a *continuous* process. Thus, we imagine that the transfer from SRC to DST happens over a period of time[1], as opposed to at a single instant (e.g. the edge of a clock tick).

4.3 Continuity of Space

The most significant difference between the Cell Matrix and the Songline Processor is the change from discrete values (1 or 0) to continuous values (coming from the set of real numbers between 1 and 0). This continuity certainly makes modeling and interfacing with with real world seem more natural, and, being in some sense a superset of the discrete version, seems reasonable to apply to the ZBC.

At first glance, this seems straightforward: let the values stored in memory be real-valued, so that transferring from, say, $MEM[100] \rightarrow MEM[105]$ copies a real value instead of just an integer. So if $MEM[100]$ contained the value π, then after the transfer, so does $MEM[105]$ But because code and data are identical, what happens when the instruction pair stored at locations 100 and 101 is executed? That would imply a transfer from $MEM[\pi]$, whatever that means...

[1] "clockless" is very different from "timeless." We still expect there to be a time component to the operation of EEXIST: things do not happen instantaneously, even though they are unclockes

Here again, the real issue is discretization, in particular discretization of space. The conclusion is clear: *space* must also be made continuous. In particular, the address space in which instructions are stored should be addressable with real values. This is a very different kind of memory system from a typical von Neumann machine, as well as from the Cell Matrix and the Songline Processor.

4.4 Extended Effects: Karma (κ)

Assuming that a (theoretical) memory can be designed to be addressed by real numbers, there is an additional issue that arises. Suppose that (just for the sake of argument) an instruction specifies a transfer: $MEM[1.414] \rightarrow MEM[2.71828]$ In the context described so far, the effect of this instruction is like a *Dirac delta function*, in that it specifies a transfer from precisely one point to another. For example, while the data at location 2.71828 would be loaded with the contents of location 1.414, the data at location 2.7182800001 would be left unchanged, as would the data at 2.7182799999.

This seems a bit unnatural, at least in terms of everyday experience. It feels discontinuous, borderline chaotic: because if the instruction at $MEM[2.7182799999]$ is unrelated to the instruction at $MEM[2.7182800000]$, a slight perturbation in the coding of $MEM[1.414] \rightarrow MEM[2.71828]$ could have a significant change on the system's behavior.

In response to these notions, the idea of an *extended effect* is introduced. Basically, each transfer instruction $MEM[SRC] \rightarrow MEM[DST]$ is interpreted as not only requesting that data be copied from SRC to DST, but also from $SRC - \delta$ to $DST - \delta$, where δ ranges over some interval. However, the *strength* of that transfer weakens as $|\delta|$ increases. The exact meaning of "strength" will be defined in the next chapter. For now, suffice to say that not all transfers have the same impact on memory: some will barely change the contents of $MEM[DST]$, while others may impact the contents significantly.

This distributed impact is termed *Karma* (κ), and ultimately relates to the impact a transfer instruction may have on itself. Suppose, for example, $MEM[10]$ contains the instruction $MEM[20] \rightarrow MEM[11]$. The instruction at location 10 is requesting data be copied from location 20 to location 11. Normally, only the contents of $MEM[11]$ would be affected by this. In the presence of karma, locations near address 11 would also be affected. If κ is large enough, then $MEM[10]$ would also be affected, i.e., the instruction's requested action is impacting itself. In other words, there is an intermingling of effect with cause.

This completes the basic catalog of characteristics we wish to have in the enhanced ZBC. What is required next is a model for implementing these characteristics. The chosen model is not necessarily practical, but will give us a simulation target with which we can explore this system. This model is called "*EEXIST*": **E**goless **EX**tended effect cont**I**nuous **S**pace **T**ime.

5. EEXIST

Exploring this proposed architecture – studying its behavior, potential applications, and so on – requires some sort of model that captures the characteristics described above. The proposed model ("EEXIST") is neither unique nor necessarily ideal, but it has worked well enough to begin exploring these concepts.

5.1 Memory Structure

A traditional memory subsystem, viewed as a collection of indexed holding bins, does not extend to a spatially-continuous layout as needed for EEXIST. Instead, imagine a porous substrate, capable of holding liquid at any place throughout its extent: something like a long, rectangular sponge with its largest dimension labeled "x" (figure 5.1). The amount of liquid present at some location x would represent the value stored at that address , i.e. $MEM[x]$. Let's clarify the notion of "amount of liquid": if x is truly a

point location, then the *volume* of liquid at x tends towards 0 (as $\delta x \to 0$ in our volume calculation). But consider, instead, the *height* of the liquid at address x, as shown in figure 5.1.

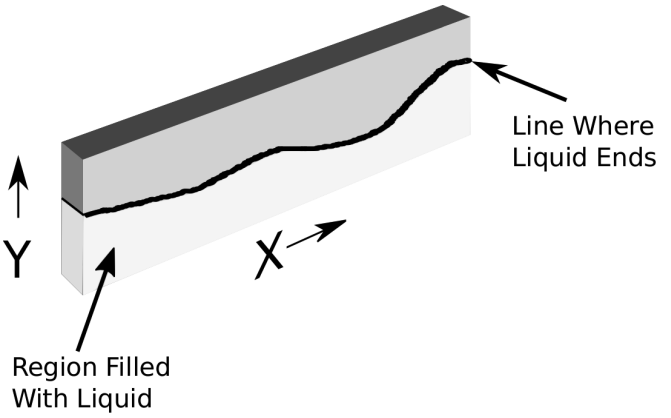


Figure 5.1: Image of EEXIST Memory as a Liquid-Filled Sponge. The line shows the height of the liquid at different x positions, which codes the value of $MEM[x]$.

The contents of $MEM[x]$ is now simply the height of the liquid at position x along the horizontal dimension. Not that while this appears to change smoothly in the figure, no assumption is made about continuity of values.

Since memory will be addressed by a continuum of values, there is no longer a notion of "next" or "adjacent" addresses: we cannot treat memory addresses in pairs as we can with a normal memory. So instead of coding $SRC \to DST$ instructions in pairs of addresses, we store one transfer instruction per address. To do this, we consider two different liquids, named "SRC" and "DST." Throughout, SRC will be show as a red chemical, and DST as a blue chemical. At any given position x in the medium, the mixing of these chemicals is irrelevant, as is their location along the y dimension. For simplicity, we will usually show the SRC chemical as situated below the DST chemical; but all that is relevant is the height of the regions containing the SRC and

DST chemicals at a given x location.

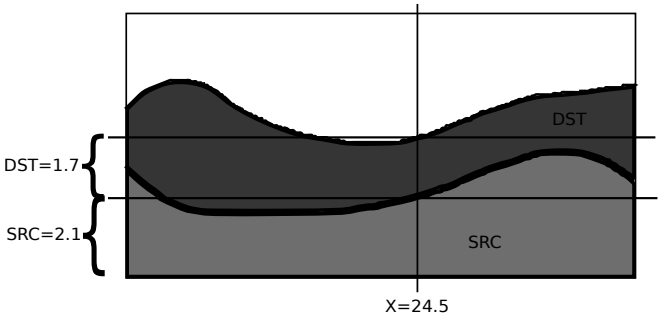


Figure 5.2: Front-view of the Chemical Memory. The red region corresponds to the SRC chemical, and the blue region corresponds to the DST chemical. The relative position of SRC and DST (e.g. which is on top) is irrelevant: the height *of each region (top to bottom) codes the desired information.*

Figure 5.2 shows a front view of this memory setup. The lower red region shows the values of SRC, while the upper blue region shows DST. For the position marked x, it is the thickness of the SRC region that codes the actual SRC address. Similarly, the thickness of the DST region codes the DST address. In the figure, at the point $x = 24.5$, we have a SRC value of 2.1 and a DST value of 1.7. Hence, $MEM[24.5]$ codes a transfer instruction $MEM[2.1] \rightarrow MEM[1.7]$.

5.2 Discretization of Memory

In this example, since SRC and DST vary smoothly, the instruction at, say, $MEM[4.51]$ would be similar to what's at $MEM[4.5]$. While there is no requirement for continuity of the boundaries of the SRC and DST regions, having these change smoothly allows us to approximate the system with a series of discrete tubes[1], as shown in figure 5.3. Here we are

[1]This discretization does not contradict our original motivation for making the spatial dimension continuous: it's merely a nod to ease of simulation.

breaking the x dimension into a series of thin regions, where each region has a given height of SRC and DST chemicals (where again, SRC and DST are color coded red and blue, respectively).

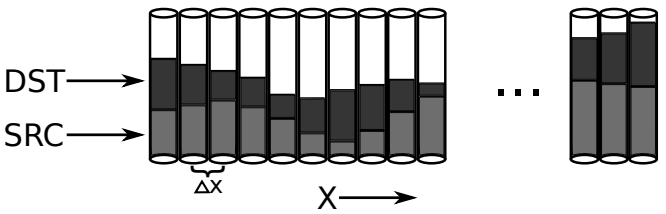


Figure 5.3: Discretized View of SRC/DST Memory. The x dimension is broken into a set of small intervals, each viewed as a thin tube containing both SRC and DST chemicals. SRC is shown in red, DST is shown in blue.

This is only an approximation of an ideal EEXIST memory; but if we let these tubes grow thinner and more numerous, then this model approaches the ideal. This discretized model makes it easier to simulate the system though, and is the model we'll work with throughout this text. It remains an open question whether this model changes smoothly to the ideal in the limit, or if the behavior of the system somehow fundamentally changes in the pure-continuous case. See Part III for this and other discussions of future-work.

In this context, for example, a memory transfer instruction such as $MEM[5] \rightarrow MEM[6]$ would request that the contents of memory at location $x = 5$ be transferred to location $x = 6$. This differs from a usual memory copy in two ways:

1. The instruction actually specifies a *transfer* of contents rather than a simple copy. In EEXIST, chemicals (both SRC and DST chemicals) are moved from the SRC location to the DST location; and
2. the transfer is not instantaneous, but rather occurs over a period of time.

If the instruction executes for enough time, eventually a complete transfer will have taken place, and $MEM[SRC]$

will be empty, its contents completely moved to $MEM[DST]$. In practice, this may not happen, for two reasons:

1. The transfer instruction itself is likely to change, in response to other transfer requests elsewhere in the system (or possibly in response to itself); and
2. other transfer requests may continually be loading chemicals *into* $MEM[SRC]$.

5.3 Extended Transfer Effects: Karma (κ)

Viewing the memory of EEXIST in this way, transfer instructions themselves can be coded precisely (since the level of SRC and DST chemicals are real-valued), but the *action* of those transfers is approximated, since space is a discritized approximation of a continuum. For example, suppose the x domain ranges from 0 to 40 (inclusive), and is broken into tubes of width 0.03125. Suppose also $MEM[0]$ specifies a transfer $MEM[5.51] \rightarrow MEM[6.02]$. There are tubes corresponding to 5.50000 and 5.53125, but not to exactly 5.51. Similarly, there are tubes corresponding to 6.00000 and 6.03125, but not to exactly 6.02. This raises a question of how to map such a transfer to our discretized space.

Rather than just rounding transfer addresses to the nearest tube, EEXIST distributes the effect of a transfer request to a region of tubes. Continuing the above example of a transfer from $MEM[5.51] \rightarrow MEM[6.02]$, this instruction would be interpreted as transferring from a region of memory locations *centered at 5.51* to a region of memory locations centered at 6.02. Nearby regions will also be affected, but to a lesser extent.

Figure 5.4 shows a profile of how much a transfer request from $MEM[5.51]$ will affect nearby regions. Ignoring discretization of space, such an instruction will have the greatest impact on $MEM[5.51]$, but will also have a (lesser) impact on $MEM[5.50]$ and $MEM[5.52]$. The request will also affect (to an even lesser degree) $MEM[5.49]$ and $MEM[5.53]$, and so on. The impact curve could be

made bell-shaped, but for simplicity and speed of simulation, a simple linear profile has been chosen.

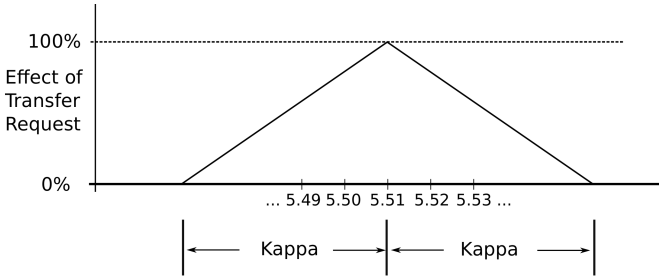


Figure 5.4: Curve Showing Effect of a Transfer Request. The most prominent effect will be centered at $MEM[5.51]$, but nearby locations will also be affected, with the effect dropping off linearly for a distance of κ on either side of 5.51. For a transfer request centered at location p, the effect at location x is denoted $D_\kappa(x, p)$.

κ controls the extent of this distributed effect (the "karma" of the system). A transfer request $MEM[x] \rightarrow MEM[y]$ will actually affect addresses from $MEM[x-\kappa]$ to $MEM[x+\kappa]$, transferring chemicals from those addresses to addresses between $MEM[y-\kappa]$ and $MEM[y+\kappa]$. The most pronounced effect will be at $MEM[x]$ and $MEM[y]$, with the effect tapering off to zero at e.g. $MEM[x-\kappa]$ and $MEM[y-\kappa]$. This turns out to be useful far beyond the simple question of mapping in discretized space: the karma of the system fundamentally affects the large-scale behavior of the system.

The extreme values for κ are worth considering:

- If $\kappa = 0$, transfers have a purely local effect. Each transfer specifies a point-source and point-destination. This in some ways mimics the behavior of a traditional (discretely-address) memory.
- If $\kappa = X_{max}$, then any transfer will affect *every* region of the memory, though the effect diminishes away from the specified SRC and DST.
- If $\kappa = \infty$, then any transfer affects all regions equally, i.e. a transfer request from $MEM[SRC] \rightarrow MEM[DST]$

causes an equivalent transfer
$MEM[SRC-\Delta] \rightarrow MEM[DST-\Delta]$
for all Δ.

κ is generally fixed for the entire system, but in some experiments it has been modified over the course of a run. It could also theoretically change from spatial point to point.

5.4 Discretization in Time

To allow for transfer requests to interact with one another, chemical transfers do not occur instantaneously, but rather over a period of time. For example, the transfer request $MEM[5] \rightarrow MEM[10]$ says to move chemicals from location 5 to location 10. One may view this as connecting a pipe between those locations, and allowing chemicals to be siphoned off from location 5 to location 10. This transfer happens at a finite rate, so that over time the chemical level at $MEM[5]$ drops while the level at $MEM[10]$ rises. The details of this transfer are described below in the *Simulation Mechanics* section.

Such a transfer can be approximated by repeatedly incrementing time by a small amount Δt, where $\Delta t > 0$. The goal of course is to have Δt approach 0, but in practical terms, the smaller Δt is, the longer the simulation takes. If we imagine Δx and Δt approaching 0, we can describe the state of the system with a pair of equations.

We idealize κ (karma) by using a curve for the effect $D_\kappa(x,p)$ (the "effective diameter due to κ") on $MEM[x]$ of a transfer request at location p, and letting $D_\kappa(x,p) = e^{-\frac{(x-p)^2}{\kappa}}$ If $\kappa = 0$ then $D_\kappa(x,p)$ is 0 everywhere except where $x = p$. As κ increases, the effect $D_\kappa(x,p)$ still has a maximum value where $x = p$, and tapers-off (but is non-zero) everywhere else. As $\kappa \rightarrow \infty$, $D_\kappa(x,p)$ flattens out, and in the limit, is equal to 1 everywhere. See figure 5.5.

Let $SRC(p,T)$ and $DST(p,T)$ be the amount of SRC and DST chemicals (respectively) at position p and time T. We can now describe the theoretical behavior of SRC and DST with a pair of integral equations:

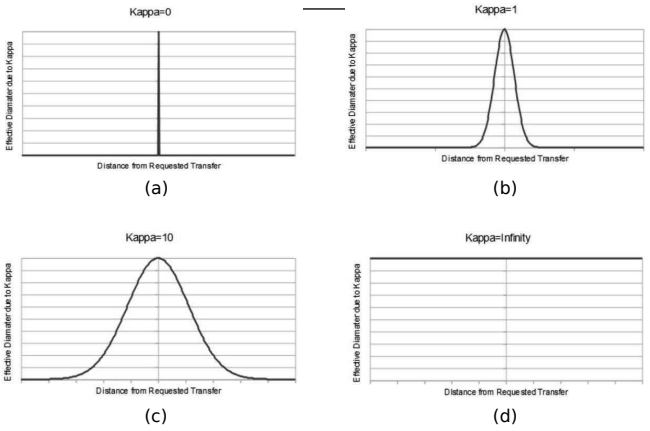


Figure 5.5: Idealized Effect of κ on $D_\kappa(x,p)$. For $\kappa = 0$, this is an impulse function (a). As κ increases, it becomes a bell-shaped curve (b), distributing transfer effects over more and more space. As κ approaches ∞, the curve widens (c) until it is a flat line (d).

$$SRC(p,T) = SRC(p,0) + \int_0^T \int_{-\infty}^{\infty} e^{-\frac{(DST(x,t)-p)^2}{\kappa}} - e^{-\frac{(SRC(x,t)-p)^2}{\kappa}} dxdt$$

$$DST(p,T) = DST(p,0) - \int_0^T \int_{-\infty}^{\infty} e^{-\frac{(DST(x,t)-p)^2}{\kappa}} - e^{-\frac{(SRC(x,t)-p)^2}{\kappa}} dxdt$$

5.5 Simulation Mechanics

Simulation of EEXIST has changed little since the beginning of this project. This doesn't suggest that the chosen knob settings are ideal; rather, it shows (perhaps) the relative insensitivity of the system to the particular choice of settings.

Current settings are as follows:

- range of x: 0.0 to 40.0

- Δx: 0.03125
- Δt: 0.05
- κ: typically 5, but sometimes valued between 2 and 10

If a transfer instruction requests a transfer to (or from) location p, the effect on $MEM[x]$ is defined by $D_\kappa(x,p)$, which is approximated with a set of linear functions, as follows:

$$D_\kappa(x,p) = \begin{cases} 0, & \text{if } x \leq p-\kappa \\ \frac{\kappa-|p-x|}{\kappa}, & \text{if } p-\kappa < x < p+\kappa \\ 0, & \text{if } x \geq p+\kappa \end{cases} \quad (5.1)$$

5.5.1 Transfer Addressing

The first step in simulating the effect of transfer requests is to determine the actual source and destination addresses for each transfer. Suppose $MEM[X]$ contains the instruction: $MEM[SRC] \rightarrow MEM[DST]$. This can be interpreted two ways:

1. *SRC* and *DST* can be the *absolute* source and destination addresses; or
2. *SRC* and *DST* can be *relative* addresses, i.e., $X+SRC$ and $X+DST$ are the actual addresses.

The choice of relative or absolute addressing is selectable in the EEXIST API. It turns out though this distinction is immaterial: the introduction of *bias* (discussed later in this chapter) allows the emulation of relative addressing using absolute addressing. Unless stated otherwise, all addressing is absolute throughout this text.

5.5.2 Transfer Type

Karma (and $D_\kappa(x,p)$) and the choice of transfer addressing (absolute or relative) determine the final SRC and DST addresses of a transfer. Once those addresses have been determined, there are two ways in which the actual transfer can take place:

1. "SD flow," which specifies a flow rate from *SRC* to *DST* based on the *amount* of SRC chemical; and
2. "Equilibrium flow," which modulates the flow rate based on the *difference* in the amount of chemicals at SRC and DST.

The main difference is that SD flow will eventually drain the SRC location, whereas equilibrium flow will eventually cause the chemical levels at SRC and DST to be the same. The choice of transfer type is selectable in the EEXIST API, though almost all work to date has been done under an assumption of SD Flow.

5.5.3 Main Simulation Loop

The basic simulation update loop proceeds as follows:

1. Look at each transfer instruction from $p = 0$ to $p = 40$; suppose $MEM[p] : MEM[SRC] \rightarrow MEM[DST]$;
2. calculate the corresponding change to $MEM[SRC]$ and $MEM[DST]$. For SD flow, the equations are:
   - $\Delta MEM[SRC] = -MEM[SRC] * D_{\kappa}(x,p) * \Delta t * \Delta x$
   - $\Delta MEM[DST] = MEM[SRC] * D_{\kappa}(x,p) * \Delta t * \Delta x$

   For equilibrium flow, the equations are:
   - $\Delta MEM[SRC] = -(MEM[SRC] - MEM[DST]) * D_{\kappa}(x,p) * \Delta t * \Delta x$
   - $\Delta MEM[DST] = (MEM[SRC] - MEM[DST]) * D_{\kappa}(x,p) * \Delta t * \Delta x$
3. accumulate all these Δ's, **being careful not to let any amounts go negative**;
4. after evaluating all transfer instructions, apply the accumulated Δ's to each memory location.

Remember that $MEM[x]$ has 2 values associated with it: a SRC amount and a DST amount. So each Δ calculated above (for example, $\Delta MEM[SRC]$) describes *a pair* of changes: a change to the amount of SRC chemical at $MEM[SRC]$ and a change to the amount of DST chemical at $MEM[SRC]$.

Figure 5.6 (a) shows an example of an absolute SD transfer operation. The instruction at $MEM[5]$ specifies the

transfer $MEM[2] \rightarrow MEM[10]$. Assuming for simplicity $\kappa = 0$ (so $D_\kappa(x,p) = 1\ \forall x,p$), $\Delta x = .1$ and $\Delta t = .1$, this single instruction requests a transfer of $8*1*.1*.1 = .08$ SRC from $MEM[2]$ to $MEM[10]$, and a transfer of $9*1*.1*.1 = .09$ DST from $MEM[2]$ to $MEM[10]$, resulting in the concentrations shown in figure 5.6(b).

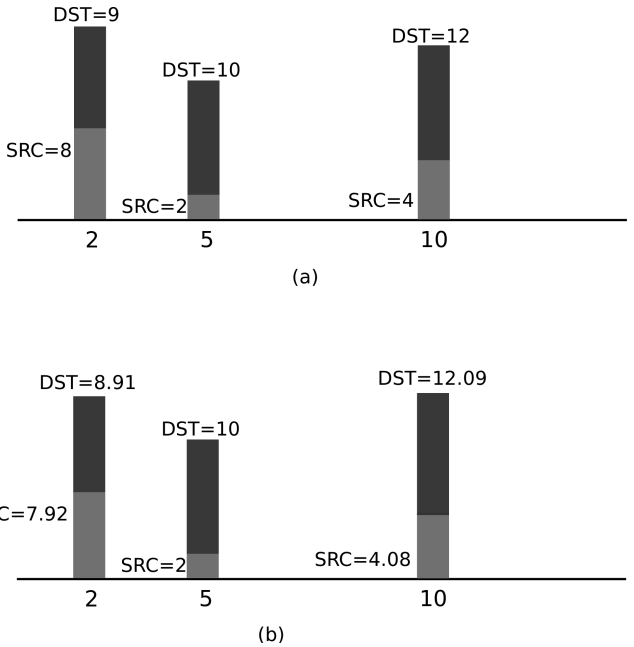


Figure 5.6: Sample Set of Transfer Requests. (a) shows an initial configuration, where $MEM[5]$ specifies a transfer from $MEM[2] \rightarrow MEM[10]$. (b) shows the results of that single transfer in absolute, SD flow mode with $\kappa = 0$. Locations 2 and 10 also specify transfer requests, but the effect of those are not shown.

Note the following:

1. Figure 5.6 only shows the effect of the instruction at $MEM[5]$; the instructions at $MEM[2]$ and $MEM[10]$ also affect memory.
2. The instruction at $MEM[5]$ does not affect the contents of $MEM[5]$. The SRC and DST chemicals only specify where a transfer should take place; those chemicals are not directly affected by the transfer,

unless the instruction (perhaps taking karma into account) refers to its own address.

Assuming absolute addressing and SD flow, this will transfer SRC and DST chemicals from location 2 to location 10. Under equilibrium flow, this would transfer SRC chemicals from 2 to 10, and DST chemicals from 10 to 2. With relative addressing, the instruction requests transfers from 7 to 15 ($5+2$ to $5+10$).

5.6 Effect of Karma

It's interesting to look at the effect of changing κ on the behavior of the system. For this analysis, a configuration that implements a 3 input exclusive-or (XOR) gate was employed. The nature of the results shown here seem typical though, regardless of the specific system configuration. Figures 5.7 - 5.18 show the chemical distribution in this system after a number of timesteps have passed, for different values of κ.

A nominal value of $\kappa = 5$ has been used for most work to date. There's no particular reason for this choice; it was simply the initial choice, and it just hasn't changed. Figure 5.7 shows the chemical balance in the evolved XOR system. The distribution is stable: it does not change over time, unless the chemical balance is perturbed from outside the system.

In figure 5.8, κ has been increased to 6. The chemical balance changes, and the system stabilizes again.

Similar behavior is observed as κ changes to 8 (figure 5.9 and 15 (figure 5.10.

At $\kappa = 25$ (figure 5.11), the chemical distribution begins to flatten out. While still stable over time, a small perturbation can be seen near the left end of the system.

When κ increases to 27 (figure 5.12), the perturbation seen in figure 5.11 begins to change. Figure 5.12 shows two snapshots of the system at different points in time.

At $\kappa = 30$ (figure 5.13), multiple small regions are changing between two states.

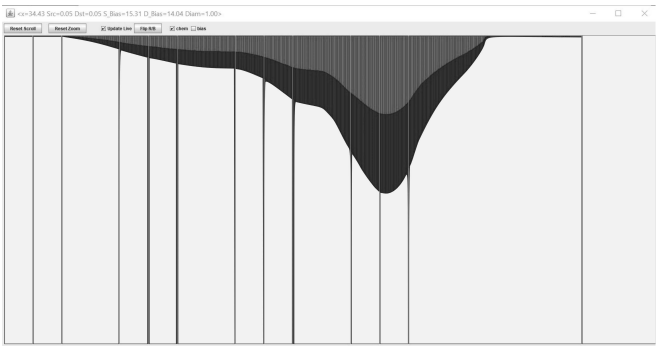


Figure 5.7: System Behavior With $\kappa = 5$. The chemical balance is stable, unchanging as the simulation is advanced.

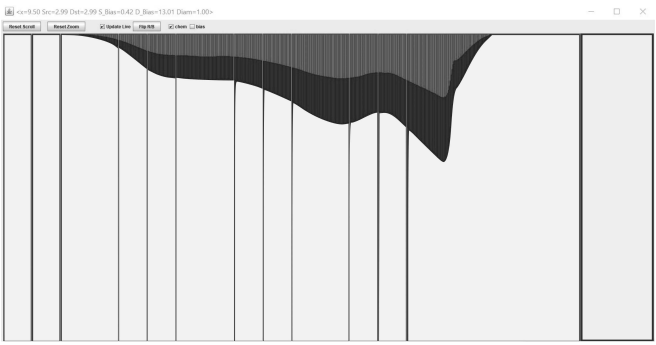


Figure 5.8: System Behavior With $\kappa = 6$. The chemical balance has changed from $\kappa = 5$ but has stabilized again.

As κ continues to increase, the chemical balance changes to a series of peaks and troughs, which shift in position over time (figures 5.14 (a)-(c)).

Figure 5.15 shows the system when $\kappa = 40$. The behavior is the same as that shown in figure 5.14, but the shape of the peaks has become more uniform.

These behaviors are not entirely unexpected. The mechanics of EEXIST instruction interpretation basically link cause to effect; *karma (κ) allows those effects to come back and affect the cause*. This allows the creation of feedback, which one typically expects to lead to either stability

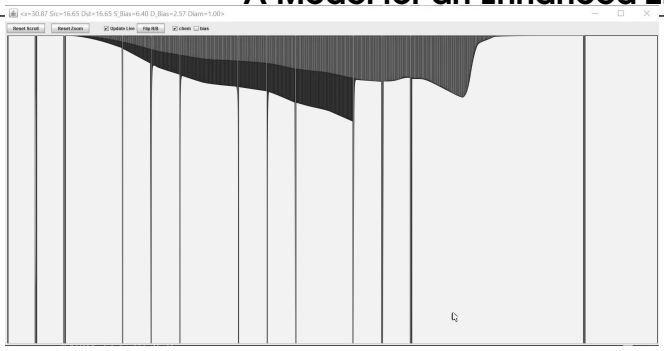


Figure 5.9: System Behavior With $\kappa = 8$.

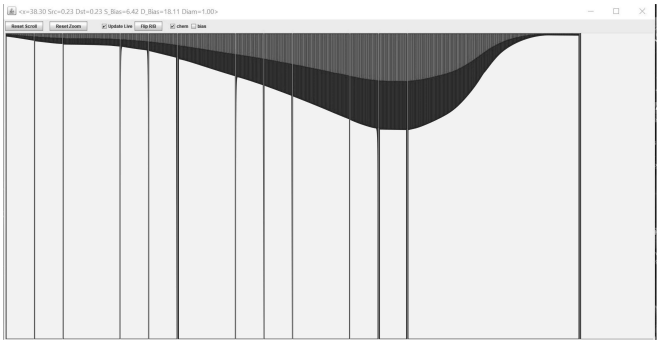


Figure 5.10: System Behavior With $\kappa = 15$.

(as seen when κ is between say 5 and 25) or oscillation ($\kappa > 25$).

It is also interesting to look at what happens as κ approaches 0. Figure 5.16 shows the system with $\kappa = 2$. Here, the chemical balance is beginning to change over time.

At $\kappa = 1$, the system looks non-periodic. Figure 5.17 (a)-(d) shows 4 snapshots of the system at different (not immediately-successive) timesteps.

When $\kappa = 0$ the system looks fundamentally different, and appears very disorganized and random (possibly chaotic?). Figure 5.18 (a)-(d) show 4 snapshots of the system, with no obvious patterns in the chemical balance.

This behavior is also not entirely unexpected: without

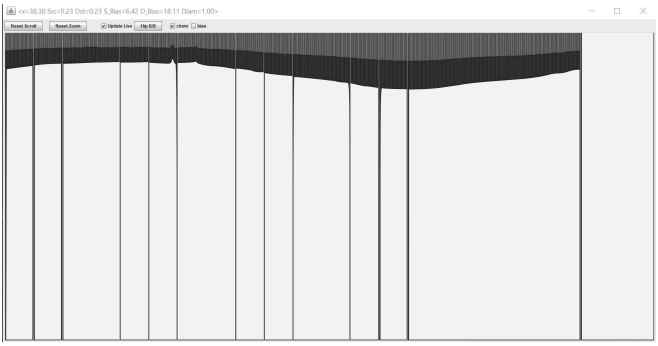


Figure 5.11: System Behavior With $\kappa = 25$. The chemical distribution seems to be flattening out, but a small perturbation appears near the left.

any feedback in the system, each instruction modified the system's chemical balance, but all instructions act entirely on their own, without any direct consequence on themselves. One might expect this to lead to a large number of seemingly unrelated changes, which is one way of interpreting figure 5.18.

5.7 Bias, Diameters

There are two useful embellishments to the above mechanisms for EEXIST. One allows an offset ("bias") to be applied to each location; the other allows specification of a flow restriction ("diameter") to each location.

Bias is a set of pairs of offsets (Bias.SRC and Bias.DST) which are added to the SRC and DST locations specified by an instruction. Normally, $Bias(x) = 0\ \forall x$, meaning an instruction such as $MEM[2] \rightarrow MEM[10]$ refers to addresses (centered at) 2 and 10. If that instruction is itself stored at address 5 (as in figure 5.6 (a)), and $Bias.SRC(5) = 20$ and $Bias.DST(5) = -1$ then the instruction actually requests a transfer $MEM[2+20] \rightarrow MEM[10-1]$. As mentioned above, Bias can be used to emulate relative addressing in an absolute-addressed system: by setting $Bias(x) = x\ \forall x$, the

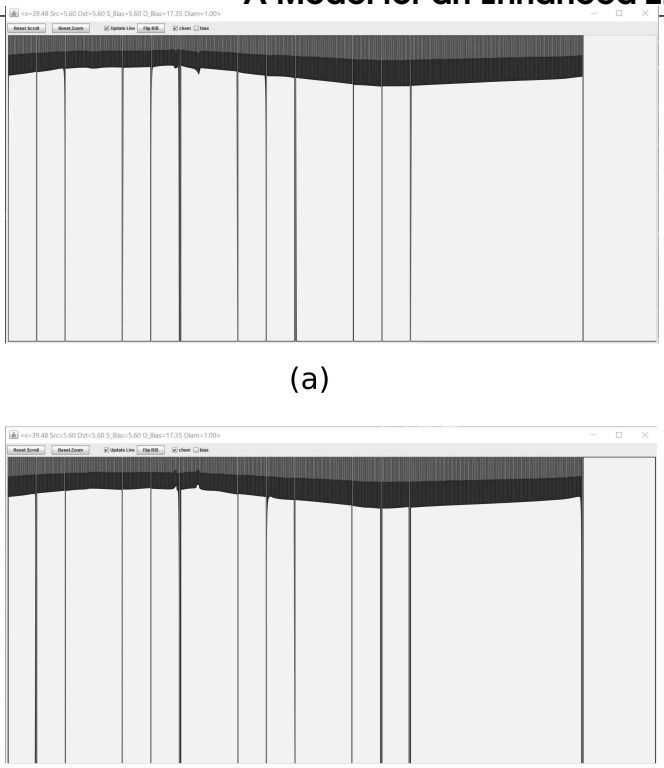
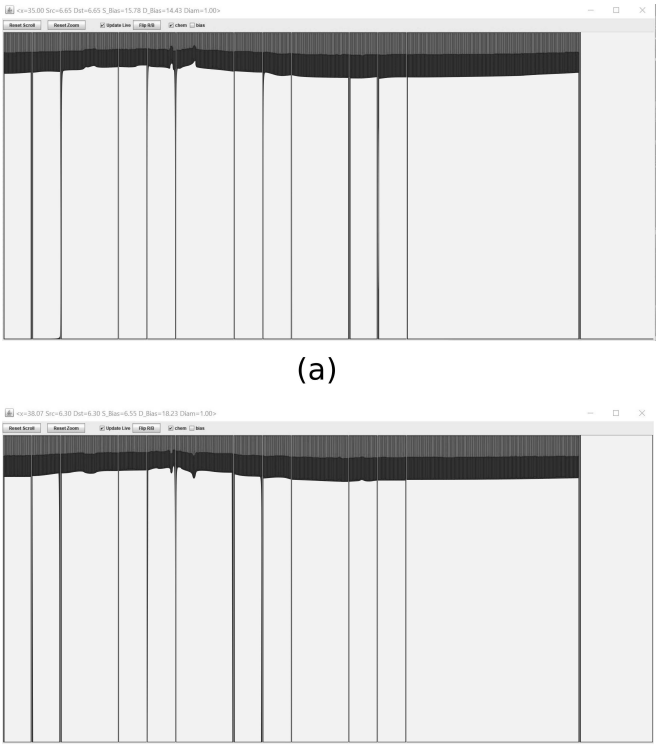


Figure 5.12: System Behavior With $\kappa = 27$. Figures (a) and (b) show two states between which the system alternates.

SRC and DST specified by a transfer instruction are added to the address of the instruction, thereby acting as if they were relative addresses.

$Diameter(x)$ specifies a relative flow metric for each spatial location. In a transfer from $MEM[SRC] \rightarrow MEM[DST]$, the final flow (based on D_κ, Δx, Δt, etc.) is multiplied by $Diameter(SRC) * Diameter(DST)$. Normally, $Diameter(x) = 1\ \forall x$. Setting $Diameter < 1$ causes less of a flow rate than normal; setting $Diameter = 0$ causes no flow to occur. Note that since $Diameter(SRC)$ and $Diameter(DST)$ are multiplied, setting *either* to 0 stops any flow between $MEM[SRC]$ and $MEM[DST]$. This is useful, for example, in specifying



(a)

(b)

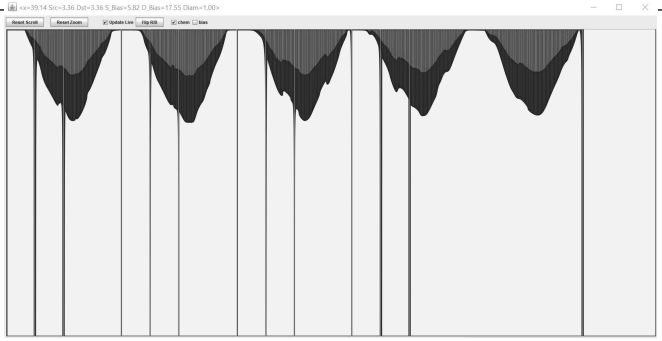
Figure 5.13: System Behavior With $\kappa = 30$. Multiple regions are changing across time.

input regions, whose chemical levels are set by external sensors. Such regions can still affect other regions of the system, but the chemical levels in those regions themselves do not change under transfer requests from inside the system. This will be discussed further in Part II.

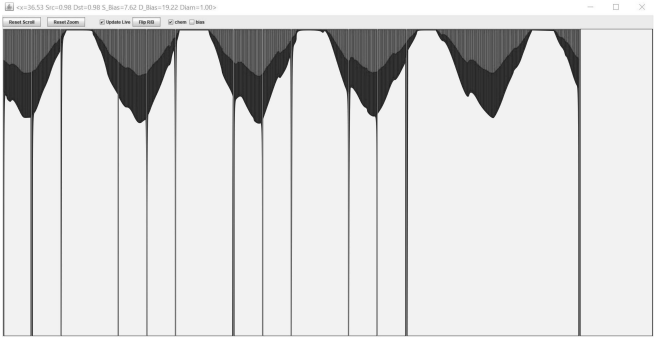
Note that a negative diameter could theoretically be used to reverse the direction of chemical flow: this has not been explored so far.

5.8 Exercises

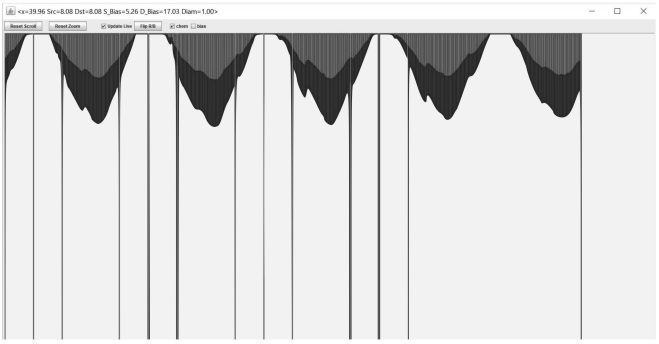
1. Sketch a distribution of SRC and DST chemicals to transfer all chemicals from $[0,4)$ to $[20,24)$, assuming



(a)



(b)



(c)

Figure 5.14: System Behavior With $\kappa = 35$. The chemical balance shows a series of peaks and troughs, whose positions shift over time.

$\kappa = 0.$

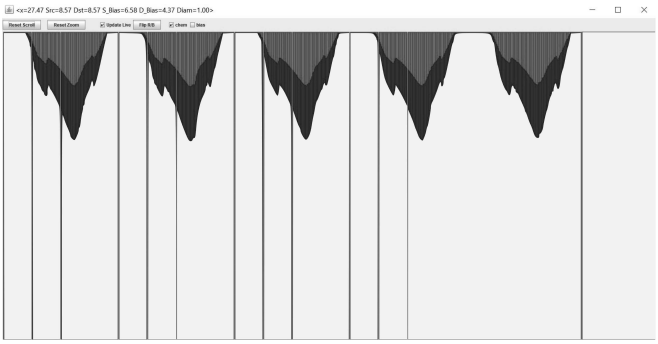


Figure 5.15: System Behavior With $\kappa = 40$.

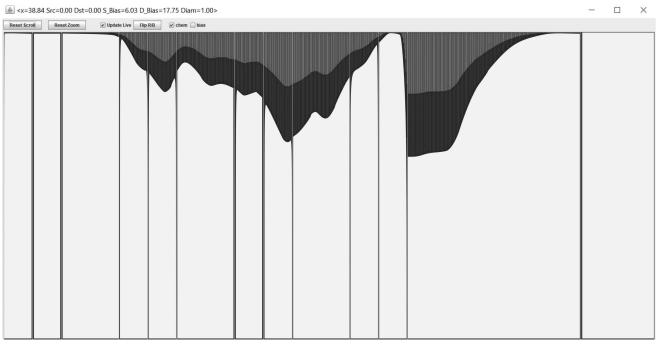


Figure 5.16: System Behavior With $\kappa = 2$. The chemical balance is changing over time.

2. Repeat the above, but with $\kappa = 4$. How does the system differ from the case where $\kappa = 0$? What assumptions do you need to make?
3. What would be the general effect on the system if $\kappa < 0$?
4. What is the general effect of setting $SRC = DST$?
5. What is the effect of setting $BIAS_{SRC} = BIAS_{DST} = C$ where C is a constant? What if instead of being constant, C is the location where the biases are being set, i.e. at any location x, $BIAS_{SRC} = BIAS_{DST} = x$?

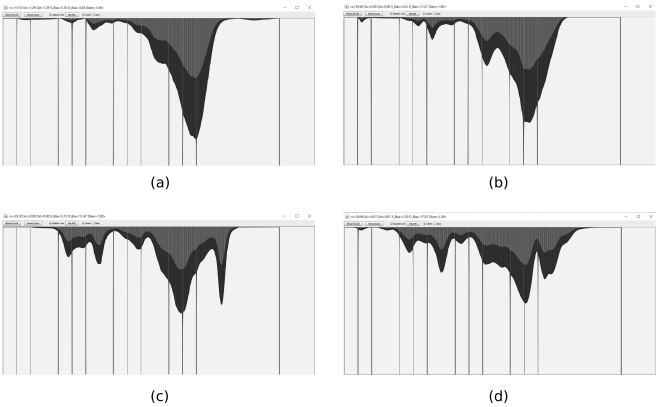


Figure 5.17: System Behavior With $\kappa = 1$. Figures (a)-(d) show 4 different timesteps. The system is changing over time, but doesn't appear to be periodic.

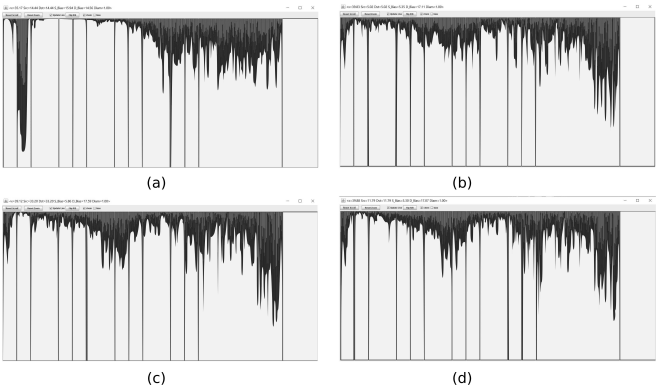


Figure 5.18: System Behavior With $\kappa = 0$. (a)-(d) show 4 snapshots of the system's chemical balance. Without the effects of karma, the system is running essentially open-loop, with each instruction modifying the system but acting independently from all other instructions.

Part Two - Experiments

6. Overview, Links to Software

With any proposed architecture, a natural question is: "what can one do with it?" While EEXIST was developed as an extension of more traditional computing systems, it's not immediately clear how to do anything "useful" with this system. Constructing individual transfers is simple (provided karma is ignored), but since all transfers occur simultaneously, orchestrating a sequence of actions seems difficult. While some work has been done in setting up rudimentary behaviors (nominal cyclic changes, for example), general schemes for configuring EEXIST for specific behaviors have yet to be discovered.

One of the early lessons in this search was that the balance of chemicals is only part of the story. In trying to configure a system that, for example, can perform logic operations, finding a particular configuration of chemicals that leads to the desired behavior is difficult. This led to the introduction of bias and diameter, described at the end of Part I.

At that point, rather than trying to explicitly design configurations for certain behaviors, an evolutionary approach was adopted: using genetic algorithms to *discover* configurations, rather than engineering those configurations. This approach has been used for all the work presented in the experiments in Part II.

It is important to remember that at this point, the main goal of this work is *to explore and understand what EEXIST is capable of*. Even though an evolutionary approach has been employed, this is not fundamentally research about evolvable systems or GAs. The goal is to see what behaviors EEXIST can exhibit.

6.1 Software Setup

6.1.1 Links

While this book is intended to primarily stand by itself, there are references to software (mainly on GITLab) and web pages throughout the discussion of experiments. These can be clicked-on directly in the PDF version of this text. For the printed copy, the reference numbers (e.g. [1]) refer to links on http://book.songlinesystems.com (which means you don't need to copy long URLs from the text).

All this code is built on top of an API [20] which provides access to a full EEXIST simulator, including graphical displays of system activity. There is standard JavaDoc available for this API here [21]. More generally, the GIT repository for the API is available here [22]

A more-general webpage that includes links to videos discussing the EEXIST API is available here [23].

The root of the entire GIT repository is available here [24], and contains not only the API but also the code and data for all experiments described in this text.

6.1.2 General Code Organization

Within each directory of the GIT repository (e.g. EA2), there are a number of files:

- README which often (but not always) contains helpful information about the contents of the directory;
- raw.txt and variations thereof. These usually contain genome information for individuals in evolving populations. The file is human-readable, but not easily (it's really a .CSV file). raw.txt is often tagged with additional information inside the filename itself (e.g. raw.xor.5.22.txt is the raw.txt from evolving an XOR gate, and individual 22 generation 5 is considered noteworthy for some reason);
- various human-readable files containing output from past runs, sometimes following manual processing;
- various scripts for monitoring output as it's being generated (especially for the tic tac toe work);
- classes Gene and Genome which handle the basic genetic aspects of individuals;

The Java code is built on top of the EEXIST API, and uses the Gene and Genome classes. Beyond that, the code is problem-specific, but there is a common structure to the code, comprised of the following pieces:

- a pair of scripts ("c" and "r") for compiling and running Java code, respectively;
- a set of code for running the evolutionary part of the system, i.e. for developing a population that performs well according to some criteria. This consists of the following code:
  - Main.java which is the main class for the evolution process; and
  - Core.java which contains the particular code for the experiments related to this directory.
- a set of code for *analyzing* the evolved population. This code generally reads from a raw.txt-type file to clone from saved genetic information into a test EEXIST system, and allows the user to interact with that EEXIST system in various ways. Typical pieces include:
  - Analyze.java which is the main class (analogous to Main.java)

  - AnalyzeCore.java which contains the particular code for analyzing the results of this set of experiments (analogous to Core.java); and
  - AnalyzeControl.java which contains code related to user-interactions with the cloned EEXIST system.
- a set of .jar files if the experiments require interaction with a server (e.g. LL.jar for simulating the lunar lander game, and VEco.jar for simulating the virtual ecosystem).

6.2 Genetic Setup

Early attempts at evolving EEXIST systems didn't work very well. Part of this seemed to be related to the "special nature" of certain areas of the memory. As chemicals are drained from tubes, the instructions referenced by such tubes refer to smaller and smaller SRC and DST addresses; whereas upper addresses (e.g. close to 40) are rarely referenced unless a tube is very full. The larger issue may have been that chemicals were serving two purpose, being used both for inputs/outputs and for somehow coding the genetic signature of an individual. It is in light of this latter consideration that further parameters were introduced into an individual's genome, in order to foster a genetic signature independent of SRC/DST chemical levels in the system.

The basic genetic structure consists of a partition of EEXIST's address space (currently $[0, 40]$) into a set of intervals (called "genes"), and within each gene, having some sort of coded variation of chemical levels (SRC and DST), biases (again SRC and DST) and diameters. The original vision for this allowed a number of different codings: fixed levels, linearly-changing levels, levels described by sinusoidal functions, and so on. The first implementation of a gene used a simplified version of this, consisting of just the following:

- a specification of a single variable: either chemical amounts, bias amounts or diameter;

- an initial value for the chosen parameter at the beginning of the gene's region; and
- the slope of the linear change in the parameter across the gene.

Provisions were made to allow genes to have a variable size, and to allow for a small amount of random variation in each of these parameters.

In practice though, only the bias amounts were modulated as part of a system's genetic signature. Additionally, all genes are the same length (4), and no variation from the specified linear change was allowed within each gene. Thus, each gene is comprised of a pair of *bias gradients*, and the entire genome is a set of 10 such equally-sized bias gradients.

Genes can be initialized with random values, or can be cloned from existing genes.

Mating occurs by simple point-by-point averaging of values (bias, etc.) from each parent. Mutation occurs (on a gene-by-gene basis) parameter-by-parameter: a random value is generated, and if it is less than the given mutation rate (e.g. 10%), then the parameter is scaled by a random amount between 0.5 and 1.5.

Most GA experiments involve an initial randomly-generated population of individuals. Each individual is assessed on some set of tasks, and scored based on its performance. The top n (typically 10) individuals are retained verbatim in the next generation, while all other individuals are removed. The survivors are randomly mated pairwise (with a possibility of mutation) to create the next generation.

6.3 Exercises

1. Use API calls to set up an EEXIST system that is configured to do a single transfer. Run the code and observe the behavior in the graphical displays.
2. Use the API calls to configure an EEXIST system with a random initial configuration. Introduce more chemicals into the system and observe how the chem-

ical balance changes over time. What sorts of behaviors can you produce?

3. Change the karma in the system while running the above experiments.
4. Look at the code for Genome.java and Gene.java. Where is mating performed? How would you change the code to select genes from one parent or the other instead of averaging them? How is mutation introduced? What controls the number of genes?
5. How can you change this code to use something other than bias gradients as the genetic signature of an individual?

7. Experiments in Digital Logic

7.1 Basic Setup

The basic goal in this set of experiments was to develop EEXIST configurations that perform digital logic functions. The general setup for each of these experiments was roughly the same:

- space extends from coordinates 0.0 to 40.0, with $\Delta x = .0625$;
- a maximum value of 40 (for each of SRC and DST) is imposed throughout the memory;
- individuals are distinguished by their genome, which consists of 10 regions of *bias gradients*, evenly spaced between 0 and 40;
- an initial population of 250 individuals is generated with random bias gradients;
- as each individual is loaded into EEXIST (one at a time), its input-to-output behavior is monitored, and compared to the desired function;
- inputs are fed in the regions $[0,4]$, $[8,12]$ and $[16,20]$;

- an input of 1 is coded as a value of $SRC = DST = 20$; an input of 0 is coded as a value of $SRC = DST = 5$;
- all other locations are initially devoid of chemicals;
- the region $[24, 28]$ is considered the output region. Chemical levels at each location in this region are translated to a logic value: a chemical level of $SRC + DST \geq 15$ is considered a logic 1; $SRC + DST < 10$ is considered logic 0; anything else is considered invalid;
- an individual's score is incremented for each location in the output region (stepped by $\Delta x = 0.0625$) with the correct logic level;
- after injecting input chemicals, the system is stepped for 50 steps;
- the system is then stepped an additional 25 steps, during which the output is monitored;
- for each address within the output region, the individual's score (initialized to 0) is incremented if the output has the correct value. Since the output region's width is 4 and $\Delta x = 0.625$, there are 64 addresses in the output region, yielding a maximum score increment of 64 per timestep. Across 25 timesteps, this gives a maximum increment of 1600 per test;
- for a 3-input system, all 8 possible input combinations are tested; this gives a maximum total score of 12,800.
- after all individuals have been assessed, the top 10 are selected as survivors;
- pairs of randomly-selected survivors are mated by averaging the start and gradient of each bias region, until the new population is 250;
- a 5% mutation rate is applied to mating.

During the evolutionary process, the genome of each individual is written to a raw output file (raw.txt). This allows individuals to be reconstructed for later testing.

The main code for the logic gate tests can be found on https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/EA2 [4].

7.2 3-Input Exclusive Or Gate

The first target circuit was a three-input XOR gate. This is, in some ways, an easier target circuit than an AND, OR, NAND or NOR gate. An AND gate, for example, produces a 0 in 87.5% of all test cases: so a NOP system (one that always outputs 0) would still score 11200 out of 12800. Thus, a NOP circuit (one which never transfers any chemicals into the output zone) might out-perform a promising but not-yet-fully-evolved AND gate.

Surprisingly, EEXIST evolved a perfect XOR gate in only 5 generations! Individual 22 in generation 5 scored a perfect 12800/12800 across all 8 possible input combinations. Figure 7.1 shows the bias gradients for this individual. Each triple of vertical lines shows an input or output region. SRC bias is in red (the upper region, at the top of the graph), DST bias is in blue (the lower region). $BIAS = 0$ at the top of the graph, with *BIAS* increasing towards the bottom. For reference, the titlebar shows the SRC and DST bias values at the point indicated by the cursor arrow (which is generally the point in the graph where $BIAS_{SRC} + BIAS_{DST}$ has its maximum value). **These conventions are used anytime a bias graph is shown**.

7.3 Nand Gate

The next evolve target was a 3-input NAND gate. A perfect system emerged during the 9^{th} generation (individual 19). The raw file (raw.nand.10.0.txt) contains the genome for each individual. Figure 7.2 shows the bias gradients for a perfect individual.

7.4 Nor Gate

The next target circuit was a 3-input NOR gate. This was expected to be the most challenging of the three logic gates, because a NOR gate almost always outputs 0, so a circuit that never transfers chemicals to the output region would score 87.5%. This means an actual NOR circuit will need

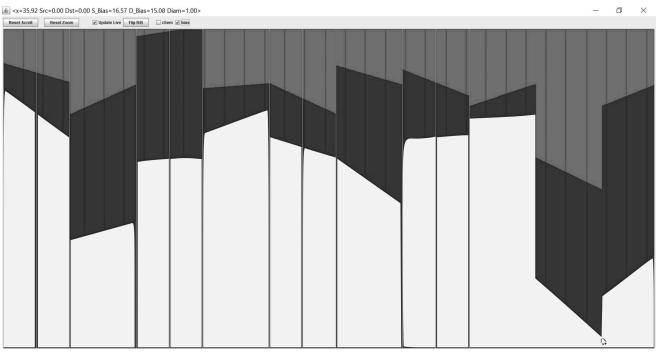


Figure 7.1: Bias Gradient For a Perfect XOR Gate. Source bias is colored red, in the region near the top of the graph; destination bias is colored blue, and shown below the source region. Bias has a value of 0 at the top of the graph, and increases towards the bottom. For reference, the cursor (where $BIAS_{SRC} + BIAS_{DST}$ has its maximum value) shows a point where $BiasSRC = 16.57$ and $BiasDST = 15.08$. Each triple of vertical bars represents an input or output region.

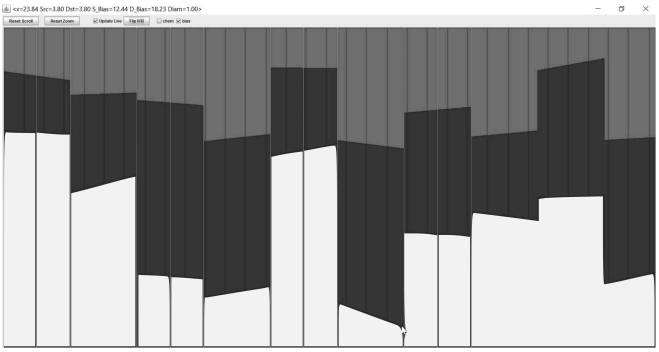


Figure 7.2: Bias Gradient For a Perfect Nand Gate. Source bias is colored red; Destination bias is colored blue. Values at the cursor are $Bias_{SRC} = 12.44$ and $Bias_{DST} = 18.23$.

to score higher than this to compete with NOPs. This is significantly more challenging than a NAND gate, where a NOP circuit will score 12.5%.

A perfect system emerged during the 56^{th} generation

(individual 212). This took 5 times longer to evolve than a NAND gate, and 10 times longer to evolve than an XOR gate. The raw file (raw.nor.56.212.txt) contains the genome for each individual. Figure 7.3 shows the bias gradients for a perfect individual.

Figure 7.3: Bias Gradient For a Perfect Nor Gate. Source bias is colored red; Destination bias is colored blue. Values at the cursor are $Bias_{SRC} = 19.27$ *and* $Bias_{DST} = 23.06$.

7.5 Frequency Discrimination

Given that EEXIST seems to be configurable to perform logic operations, the next task was to see if it could respond to varying inputs over time. This test took the form of a frequency discriminator (possibly inspired by Adrian Thompson's work [6]). The main code for the frequency discriminator tests can be found on https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/EA3 [5]. raw1000 contains the history of individuals' genomes for this experiment.

The idea was to define an input region, and toggle the input at one of two frequencies, hoping to generate an output of 1 or 0 based on the input frequency. The test setup was as follows:

- $\kappa = 5$;
- Two input frequencies are considered:
  - high frequency changes every 4 timesteps

  - low frequency changes every 8 timesteps
- The input region is $[0,4]$: logic 1 is $SRC = DST = 20$; logic 0 is $SRC = DST = 5$;
- The output region is $[24,28]$: chemical levels at each location in the output region (stepped by $\Delta x = .0625$) are translated to logic levels: $SRC + DST > 15$ is interpreted as logic 1; $SRC + DST < 5$ as logic 0; anything else is an undefined logic level;
- an individual's score is incremented for each output that is at the correct logic level. This means a maximum possible increment of 64 per timestep;
- The input is supplied for an initial 64 timesteps;
- for the next 1000 timesteps, the output is analyzed as the input continues to toggle;
- The output goal is 0 for high-frequency input, 1 for low-frequency input;
- a perfect score is 64×1000 steps $\times 2$ tests $= 128000$;

Evolution proceeded slowly, since each individual required 1064 timesteps per test. Nonetheless, after 8 generations, a near-perfect individual emerged (individual 37) with a score of 127009/128000. Despite the imperfect score, all the errors were in the first 19 timesteps: meaning the last 981 timesteps were perfect across the entire output region (subsequent timesteps were also perfect). While further evolution might have resulted in a perfect score under the original criteria, the original requirements (initialization of 64 timesteps, followed by a test period of 1000 timesteps) were essentially arbitrary, and thus the obtained result (initialization of 83 timesteps followed by a test period of 1000 timesteps) was deemed "good enough."

Figure 7.4 shows a display of the test results. In this display, time is drawn vertically (top-to-bottom, then wrapping back to the top), and space horizontally. SRC/DST amounts are color-coded (SRC is red, DST is blue). The intensity of each color reflects the amount of chemicals at that position in space and time. Since $SRC = DST$ throughout, the only color is different intensities of purple.

The vertical lines delimit the input and output regions.

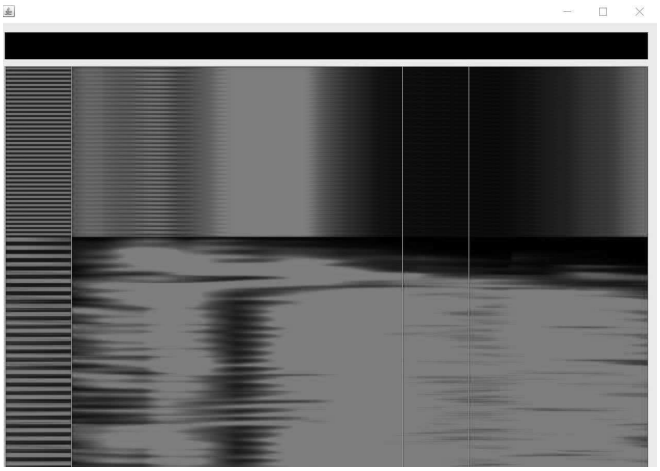


Figure 7.4: Space (x) and Time (y) Display of Frequency Detection Test. Time runs from top to bottom (and re-wraps to the top). Intensity relates to chemical amounts. The input region is on the left, and reflects the periodic input at a high (top half) or low (bottom half) frequency. Output appears towards the right (between the two vertical lines) and shows the desired output of 0 *for a high frequency input, and* 1 *for low frequency.*

The input region (on the left) shows the alternation between high and low chemical levels: this reflects the input signal, which has a high frequency in the top of the display, and a low frequency in the bottom. The output region (marked by two vertical lines near the right of the display) shows the corresponding output:

- In the top of the display, the output is low. While faint traces of the input pattern are visible, the chemical amounts translate to a clean logic 0 throughout the test.
- When the input frequency switches to low (approximately halfway down the display), the output first goes low (dark), but then raises to a high level (bright purple) for the remainder of the time to the bottom of the display.

Figure 7.5 shows the bias gradients associated with this

individual.

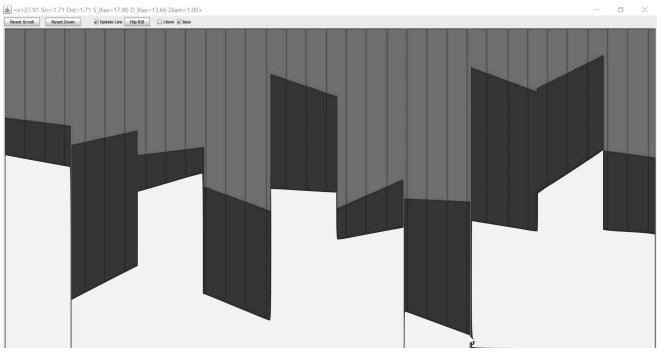


Figure 7.5: Bias Gradients for Individual 37, Generation 8. Values at the cursor are $Bias_{SRC} = 17.90$ *and* $Bias_{DST} = 13.66$.

7.6 Frequency Generation

The final experiment in this set of digital logic-related behaviors was to evolve a circuit that generates an oscillating output. The setup was similar to other experiments, with a few subtle differences:

- The input region was $[0,4]$ and the output region was $[24,28]$;
- The system was stimulated for 16 timesteps by setting the $SRC = DST = 20$ throughout the input region;
- For the next 1000 steps, the output region was monitored. In this case, the *average* chemical level ($SRC + DST$) was calculated across the region, and an average of 15 or more was considered a logic 1, otherwise the output was considered logic 0;
- An individual's score was incremented any time the output changed. Across 1000 timesteps, the maximum possible score would thus be 1000.

The code and files for this experiment can be found on https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/EA5 [7]. The file "raw" contains the genome

history for each individual in this experiment. After 4 generations, individual 90 earned a score of 224, representing on average an output change every 4 timesteps. This was considered successful enough to conclude the experiment. The output from this system is shown in listing 7.1:

Listing 7.1: System Output, Generation 4, Individual 90

```
10000000000000000000000000011110000000000000001111
10000000000000000000001100000000011000000001000000
00100000001000000001000000010000000010000000110000
00010000000110000000100000011000000010000000011000
00001000000011000000010000001000000001000000001100
00000100000001000000001000000110000000100000000100
00000010000000100000000100000011000000010000000010
00000001000000010000000100000011000000010000000001
10000000100000001100000001000000110000001000000000
11000000010000000100000000100000011000000010000000
01100000001000000010000000100000011000000010000000
00110000000100000001000000001000000010000000010000
00011000000010000000100000000100000001100000001000
00001100000001000000011000000010000001000000000100
00000100000000100000011000000010000001000000000010
00000011000000010000001100000001000000110000000001
00000001100000001000000011000000010000001100000000
10000000110000000100000001100000001000000011000000
01000000011000000010000000110000000100000001100000
00100000001100000001000000011000000010000000110000
```

Figure 7.6 shows the display-view output of this run (the display is similar to figure 7.4). As usual, the input region is at the left, and the output is between the yellow vertical lines towards the right. As can be seen (though not easily in the printed version of this figure), after the initial input, the input region is mostly devoid of chemicals, while the output region begins to toggle between high and low shortly thereafter.

Figure 7.7 shows the bias gradients for this same individual.

7.7 Another Look at Karma

Before moving on to the next set of experiments, it's interesting to re-visit the effect of karma (κ) on the system: in particular, to see if evolution is possible without karma. Figure 7.8 shows the results of an experiment designed to explore this. The system is trying to evolve a 3-input XOR gate, using the usual general- and survivor-population sizes (250 and 10), genome structure, etc. With $\kappa = 0$, the system

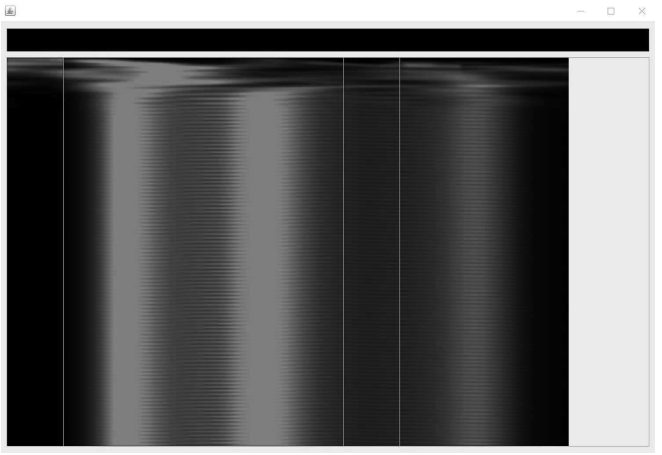


Figure 7.6: Display Window for Generation 4, Individual 90. The system is seeded with a dose of chemicals in the input region (left side). Shortly thereafter, the output region (to the right of center) begins to toggle between high and low amounts of chemicals. The output toggles 224 times during 1000 timesteps (not all timesteps are shown).

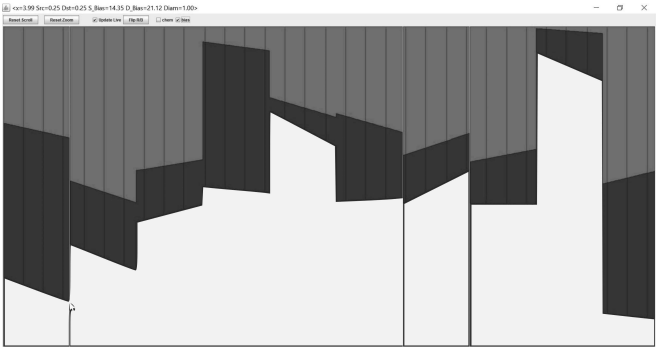


Figure 7.7: Bias Gradients for Individual 90, Generation 4. Values at the cursor are $Bias_{SRC} = 14.35$ *and* $Bias_{DST} = 21.12$.

runs for over 1000 generations, achieving a most-fit individual with a score of 56% (i.e., it gives the correct result in 56% of the test cases). Remember that a NOP – a circuit that always outputs 0 – will achieve a score of 50%. Hence

the best individual after 1000 generations is performing barely better than a circuit that always outputs 0.

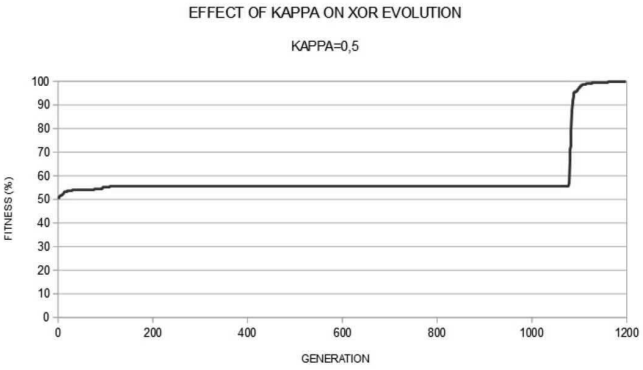


Figure 7.8: Effect of Karma on Evolution of an XOR Gate. κ is initially set to 0 and the system runs for over 1000 generations. Evolution fails to produce an XOR gate that behaves much better than a NOP. When κ is changed to 5, the system quickly evolves an almost-perfect XOR in just a few generations.

During generation 1079, κ is changed to 5. Within the current population at the time of the change, one individual already performs at around 57%. Within 10 generations, the best individual is performing above 90%; and after another 21 generations, the best individual is at 99%. Thus karma seems to have a beneficial impact on evolvability.

7.8 Exercises

1. Run the Analyze code on the file raw.xor.5.22.txt, and test individual 22, generation 5. Confirm that it functions as an XOR.
2. Analyze individual 0, generation 1. Compare its behavior to the above.
3. Look at the raw file, find the entry for individual 22 generation 5, and compare the data in the file with the bias gradients shown in the simulator.
4. How fragile are the bias gradients? Can you truncate to 6 digit and still get an OR? 4 digits? 2 digits?

5. Try evolving an XOR with truncated bias gradients.
6. Change the code to evolve an XNOR (look for code in Core.java around the comment that says "Test Function Here"). Run the analysis code to confirm your system performs an XNOR.
7. Try evolving with different values of κ. Try values such as 10, 20, 30 and 40.
8. Try evolving some nonstandard logic functions.
9. Try evolving a 4-input gate.

8. A Tic Tac Toe Player

Following a series of experiments in emulating behavior related to digital logic, the next series of experiments centered around seeing if EEXIST could learn to play tic tac toe. A number of experiments were performed, with a lot of variation in setup, fitness assessment, scoring, and so on. The system quickly proved able to play a game of tic tac toe; was able to perform better than a random opponent, and was able to sometimes make good moves against an expert opponent; but it never evolved to where it could play without losing. The reasons for this will be discussed below.

The code and files for these experiments can be found on https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/EA6 [8].

8.1 Setup

The basic setup for all experiments was similar:

- The board is viewed as a grid of 9 squares, numbered

as in figure 8.1;

- Each square also corresponds to a region in the EEXIST address space: region 1 is $[0,4)$; region 2 is $[4,8)$ and so on, up to region 8 which is associated with addresses $[32,36)$. These are considered output regions;
- All 9 of these regions are initially devoid of all chemicals;
- The region $[36,40]$ is used to start the system and evoke EEXIST's first move (EEXIST always goes first in these tests). To start the system, the region $[36,40]$ is filled with $SRC = DST = 15$;
- The system is then stepped, and the output regions are monitored for a move request (average of $SRC + DST \geq 10$). If a region which isn't yet marked on the board signals a move request, that is considered EEXIST's move. The square is given EEXIST's mark, and it becomes the opponent's turn;
- Once the opponent has selected a move, it is indicated to EEXIST by saturating the selected region with $SRC = DST = 15$ (the same as for the initiation in $[36,40]$);
- Play continues until someone wins, or there are no spaces left (in which case the game is a draw).

Note that no check is made to prevent illegal moves by the opponent.

There is a maximum number of timesteps the system will wait for EEXIST to move (typically 250 steps). If no move is detected in that time, EEXIST is considered to have forfeited.

The genetic algorithm was run with an initial population of 250 random configurations, a survivor size of 10 individuals, and a mutation rate of 5% (the same as in the digital experiments). The genome was again a set of 10 evenly-spaced bias gradients. Mating was performed by averaging the parents' biases, point-by-point.

EEXIST was always "X" and the opponent was always "O" (this doesn't affect anything, but is useful to know when one is looking at the code).

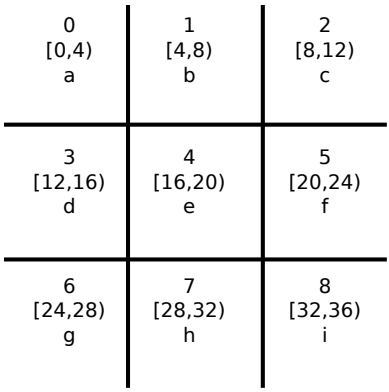


Figure 8.1: Layout of Tic Tac Toe Board. Regions denoted $[x_0, x_1)$ *are the EEXIST address range associated with that square. Indexes* 0 *through* 8 *are used internally in the code; letters "a" through "i" are used in playing an interactive game.*

A lot of output was generated during these evolutionary runs. Besides writing the genome of each individual (the "raw..." files), a graphical display of the board was also produced. This was usually saved to a file, and read offline.

This was also the first set of experiments to be run on an external server. Since the server had no X interface, a headless version of the code was produced (these are the "...HL" variants of the directories).

8.2 Goals

It's important to remember that the overarching goal of this work is not to play tic tac toe (a simple BASIC program will do that); nor is it to study genetic algorithms or evolvability *per se*. Rather, the goal is *to learn what types of behaviors EEXIST is capable of, to better understand its capabilities and limitations*. With that in mind, trying to evolve a system that plays tic tac toe seemed an interesting endeavor; and as usual, rather than trying to manually engineer EEXIST to do this, an evolvable approach was explored.

The immediate goal was, as always, to evolve individu-

als with higher and higher fitness measures. However, there are a number of possible ways to assess fitness. For example, the following are all slightly different goals for a tic tac toe playing system:

- play better than a random opponent;
- play better than an "average" opponent;
- win more than you lose;
- never lose (but possibly draw sometimes);
- play a perfect game

An exact understanding of the goal(s) will help define the fitness metric. However, there are tensions among the above goals:

- training against random opponents may lead to a system that scores well, i.e., has a great fitness measure, but fails miserably against a skilled opponent. For example, winning 90% of all games against random opponents might still allow for 100% loss against a skilled opponent.
- training only for wins ignores the fact that not all games are winnable. By moving first, one can be guaranteed to never lose, but a skilled opponent can always force a DRAW. Therefore, if training against a perfect opponent, one would never see a single win. This intuitively goes against the idea of penalizing DRAWs.
- If the system is trained against every possible game, and WIN is ranked higher than DRAW, it's possible a system that is undefeatable by a perfect opponent might not score as well overall as one that wins many games against imperfect opponents.

In many cases, the system seemed to gravitate towards local maxima, tending to find excellent performers among groups of sub-optimal individuals.

Scoring was based on the outcome of each game, with the following possibilities:

- winning;
- losing;
- ending in a draw; and

- forfeiting (no move made),

along with the following intermediate metrics:

- failing to take a winning move when presented with the possibility; and
- failing to block an opponent's winning move when presented with the possibility.

Different scoring metrics were used, based on the above measures. For example:

- $WIN = +5$
- $DRAW = +2$
- $LOSE = -1$
- $FORFEIT = -\infty$

Forfeiting was always treated as immediate dismissal of the individual (they were given a final score that was negative). This was largely due to practical concerns: whereas many moves are made after only a few timesteps, forfeiting means waiting (e.g.) 250 timesteps before giving up on EEXIST's making a move. This simply slowed down the evolutionary process too much, hence such individuals were quickly removed from the population by assigning them a negative score.

Evolution depends on the nature of the population of opponents:

1. in a totally random population, opponents make random (legal) moves;
2. in a population of smart individuals, opponents make winning moves if possible, but otherwise make random moves;
3. in a population of smarter individuals, opponents block EEXIST's winning moves if possible, but otherwise make random moves;
4. in a population of perfect individuals, opponents always play a perfect game (and thus never lose);
5. in an exhaustive population, all possible games are played.

Of course, combinations of these (such as 2 and 3) are possible.

8.3 Results

Directories from these runs are available on GITLab, generally in subdirectories underneath https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI In general, there are two sets of code:

- Main.java and Core.java are the heart of the *evolutionary* system (along with support code: Game, Gene and Genome);
- Analyze and AnalyzeCore are the counterparts to Main and Core, but are used for *analysis* of individuals (along with AnalyzeControl, plus Game, Gene and Genome).

The evolutionary code displays gameplay while it runs, but also writes a raw file of individuals' genomes. This file can be loaded into the analysis code, which allows the user to play against an EEXIST-controlled opponent. In this mode, the user specifies moves by naming the square (using the letters shown in figure 8.1).

A number of different tests were run; only two are described below.

8.3.1 EA7HL

The EA7HL directory contains the code and results for the following test setup:

- 50 games per individual
- 250 timesteps maximum wait time for a move from EEXIST
- Scoring: Win=2, Draw=1, Loss=0, Bad Move=-1 (where a "Bad Move" means failing to take a winning move, or block an opponent from winning their next turn)
- Final score is the simple sum of each game's score

With 50 games, and a maximum score per game of 2 (for winning), the best possible score for an individual is 100. Against a well-trained opponent, the best possible score is 50 (50 draws). The best individual in this set of experiments (file "rawgood" generation 28 individual 0)

scored 61. Figure 8.2 shows the bias gradients for this individual.

This individual always moves in the middle square ("e") first. Based on the opponent's first move, the outcomes are as follows:

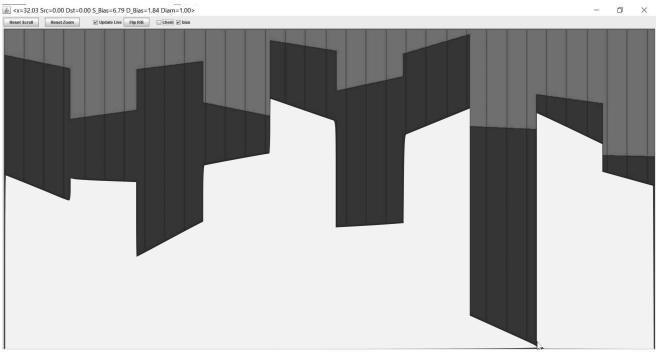


Figure 8.2: Bias Gradients For EA7HL, Individual 0, Generation 28. Values at the cursor are $Bias_{SRC} = 6.79$ *and* $Bias_{DST} = 1.84$.

- a: game is a draw (but if the user doesn't block EEXIST, they can actually win)
- b: EEXIST finds 2 ways to win, and eventually wins, *as long as the user tries to block*
- c: game is a draw (but if the user doesn't block EEXIST, they can win)
- d: user ends up with a choice of 2 moves: one leads to a win for the user, the other leads to a draw
- f: game is a draw
- g: game is a draw
- h: EEXIST finds 2 ways to win: blocking one leads to EEXIST winning; blocking the other leads to a draw
- i: game is a draw

This set of experiments was the first clue to the system's sensitivity to training data. For example, when the user's first move is "a," EEXIST eventually is one move from winning: but if the user doesn't block that move, EEXIST may fail to take that move its next turn.

Figures 8.3 - 8.5 show three possible outcomes for this

initial user move. In 8.3, everyone plays as expected, and the game is a draw. This is the expected behavior. In 8.4, the user fails to block EEXIST from taking a winning move, and in fact EEXIST wins on that next move. In 8.5, the user blocks EEXIST the first time, but fails to block EEXIST's second possible winning move. In this case, EEXIST fails to take that winning move, allowing the user to win on their next turn.

```
          a|b|c                                   O|b|c
          -+-+-                                   -+-+-
          d|X|f                                   d|X|f
          -+-+-                                   -+-+-
          g|h|i                                   g|h|i

1. EEXIST starts in the middle       2. User moves in upper-left
-------------------------------------------------------------------
          O|b|c                                   O|b|c
          -+-+-                                   -+-+-
          d|X|X                                   O|X|X
          -+-+-                                   -+-+-
          g|h|i                                   g|h|i

3. EEXIST moves in f and is          4. User blocks EEXIST in d
   ready to win
-------------------------------------------------------------------
          O|b|c                                   O|b|O
          -+-+-                                   -+-+-
          O|X|X                                   O|X|X
          -+-+-                                   -+-+-
          X|h|i                                   X|h|i

5. EEXIST blocks user in g           6. User blocks in c
-------------------------------------------------------------------
          O|X|O                                   O|X|O
          -+-+-                                   -+-+-
          O|X|X                                   O|X|X
          -+-+-                                   -+-+-
          X|h|i                                   X|O|i

7. EEXIST blocks in b                8. User blocks in h
-------------------------------------------------------------------
          O|X|O
          -+-+-
          O|X|X
          -+-+-
          X|O|X

9. Game is a draw
```

Figure 8.3: Generation 28, Individual 0. If the user plays predictably (i.e. as a "smart" user would), the game will eventually be a draw.

This illustrates some of the challenges of training the system: if the training opponent always blocks, then EEXIST may be tripped up by an opponent who doesn't block;

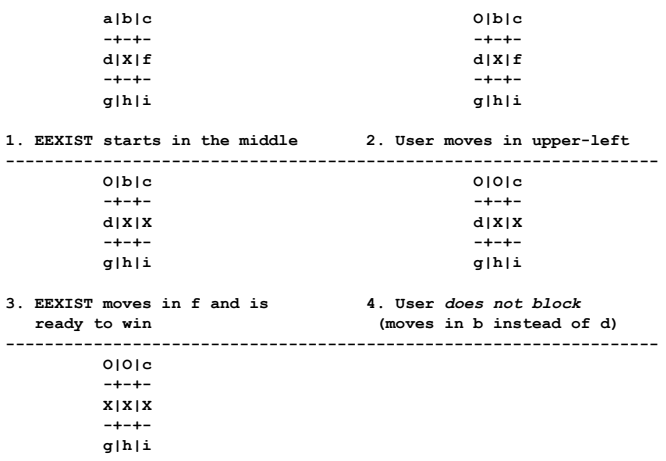


Figure 8.4: Generation 28, Individual 0. If the user fails to block when EEXIST is one move from winning, EEXIST takes the winning move and wins the game.

but if the training opponent misses some blocks, EEXIST may score unreasonably high due to these overly-simple games.

8.3.2 EA8HL

The "EA8HL" directory contains code and output for a series of experiments where EEXIST plays against all sets of possible opponent moves. Specifically, for each move EEXIST makes, every legal opponent move will be considered. In theory, since there are 9 squares, and EEXIST moves first, there are 8 possible first moves by the opponent; after EEXIST's 2^{nd} move, there are 6 open squares for the opponent to choose from; and so on. Thus there are a *maximum* of $8 \times 6 \times 4 \times 2 = 384$ possible games. In practice, the number is fewer than this, since some of these games may end before the opponent has made 4 moves.

An array ("moves[]") is used to record the current set of moves an opponent will make, and this is incremented after each game. If a game ends before a total of 9 moves, the

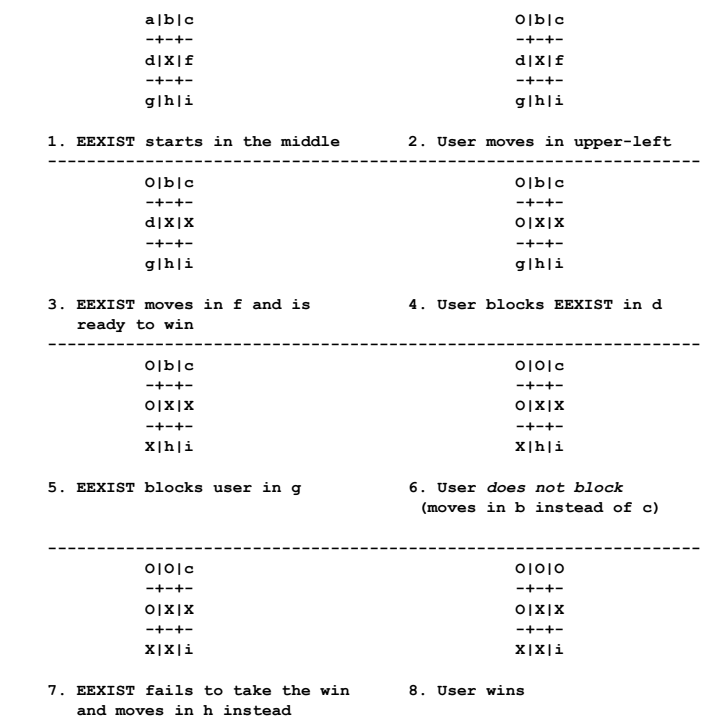


Figure 8.5: Generation 28, Individual 0. In this case, the user blocks EEXIST's first winning opportunity, but fails to block its second winning opportunity. EEXIST fails to claim that move, allowing the user to win on their next move.

game tree is pruned to remove non-viable options.

Scoring was based on the following scale:

- Win=+2
- Draw=1
- Lose=-2
- forfeit=-1 (immediate end of testing, with a final score of 0 for the individual)
- badmove=-1 (immediate end of testing, with a final score of 0 for the individual); a "badmove" is failure to block or failure to take a winning move

(In fact, various other scoring criteria were explored, mostly non-scientifically. The above were the final values used).

Given such a potentially large number of games per run, evolution proceeded slowly in this set of experiments. As such, only 19 generations were developed. See the file "raw18," generation 19, individual 0 for the top performer. Figure 8.6 shows the bias gradients for this individual. Using the reference letters from figure 8.1, the only way the user can win is by moving in squares "f," "a," "b" and then "c"; any other set of moves leads to a draw (assuming the user makes "smart" moves, i.e., blocks when EEXIST is one move from winning). Failing to block sometimes (but not always) allows EEXIST to win.

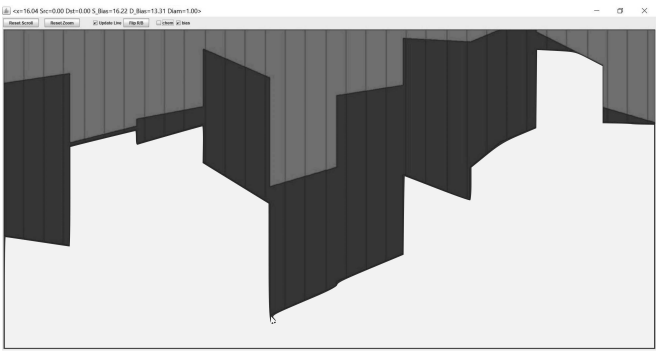


Figure 8.6: Bias Gradients For EA8HL, Individual 0, Generation 19. Values at the cursor are $Bias_{SRC} = 16.22$ *and* $Bias_{DST} = 13.31$.

8.4 Lessons and Next Steps

One lesson learned from these experiments was the importance of choosing a good survival metric. In the end, there didn't seem to be a single best way to score an individual in a feasible amount of time. Some of the setups were allowed to run for a few weeks, with the evolution continually producing higher-scoring individuals, but the improvement was very slow, and it wasn't clear if these improvements were more than incremental. This was, in large part, a failure in the scoring system (which was somewhat biased to

faster execution, i.e., not allowing individuals who forfeit to survive, when in fact their genome might have eventually contributed to a more-fit individual).

A second issue was the mapping of the board to EEXIST's spatial domain. The decision to have EEXIST's space run from 0 to 40 was made in the earliest days of the simulation, and persisted throughout this work. Breaking that range into 10 equal-sized intervals and using characteristics of each interval as a genome was proposed in the first genetic work on EEXIST, and has been used ever since. However, given 9 squares on the tic tac toe board (and an obvious mapping from board squares to regions in the genome), and a 10^{th} region for initiation of the game, there was, in some sense, no space left over for calculations other than $[36,40]$. Any intermediate work being done via chemical transfer was likely signaling (or contributing directly to the eventual signaling of) a desired move. It may be that having more unallocated space would lead to better performance.

At this point though, it's worth going back and reconsidering the basic question: "What are we trying to do?"; and again, the answer is neither "Play tic tac toe," nor "Study genetic algorithms for game playing." The goal is to study EEXIST's capabilities. The fact that it will play a tic tac toe game at all is somewhat remarkable; the fact that it can lead most games to a draw, and sometimes force a win, is perhaps even more interesting. The takeaway message though is that EEXIST *does* seem able to exhibit a range of behaviors: nothing that can't be done on a von Neumann machine, but given its peculiar nature and the difficulty of manually configuring the system, the fact that there *are* configurations that do interesting things is noteworthy, and suggests further investigation may be worthwhile.

In continuing to pursue a genetic/evolutionary approach though, the question of a "good" survival metric needed to be addressed. Rather than ponder different weightings for different actions, or how to combine individual test scores into a composite score for an individual, a different ap-

proach was adopted: testing individuals in an environment where they might survive or perish; and using their survival as the sole metric for passing on their genes. In other words, test individuals in a system where they will either survive or perish. As long as they survive, they can mate. If they perish, their genes no longer contribute directly to future generations of individuals.

A few years ago, I had a student named Jordan Curry, who developed a *Virtual Eco System* ("VEco") for a multi-term project. When he demonstrated this to me, it seemed like a powerful environment in which to explore various algorithms for survival and development. In discussing this later, in the light of EEXIST work, we decided it might be interesting to drive each of his individual creatures with an EEXIST. The system already incorporated a semblance of mating, so this seemed like a natural framework for exploring EEXIST.

To do this required two main developments though:

1. A new version of VEco, into which could be tied a population of EEXIST individuals; and
2. A way to feed inputs to EEXIST over a long period of time without saturating the system.

Item 1 simply required time and coordination. Item 2 is more of a general issue. Inputs are being modeled by injecting chemicals into a region of EEXIST's memory; but since those chemicals may be drained off to other locations (due to various transfer commands), continually restoring an input region's chemical levels to a fixed value could result in more and more chemicals being injected into the system, leading to saturation. This may or may not be a problem: but it felt like it would lead to a weakening of a system's effects over time. This was remedied using the *diameter* option (discussed previously).

As a preamble to VEco, an intermediate task was undertaken: using EEXIST to control a simplified lunar landing game. This is the topic of the next chapter.

8.5 Exercises

1. Run the analyzer and see how well EEXIST plays. Try playing as a skilled player, a random player, or a deliberately unskilled player.
2. Come up with a new grading scheme and try to evolve the system.
3. Change the size of input regions and re-evolve.
4. Make a 2^{nd} EEXIST system and play EEXIST against itself. Does it get better, or plateau at some nominal skill level?

9. Lunar Lander Control

As an exploration of a different problem space, experiments were performed next to see how EEXIST might control a simplified lunar lander game. The setup was similar to the old 1970s Lunar Lander video game [9], but simplified in a few ways:

- the landing surface was flat;
- thrust is a simple binary control (on/off); and
- there is no horizontal control, only vertical.

The code and outputs for these experiments can be found at https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/EA10HL [10] and https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/EA11HL [11]. The system is setup as a client/server pair (another preparatory step for VEco).

The code for the landing simulator (server) itself can be found here [12] while the usual Main/Core/etc. in EA10HL and EA11HL contain the client code driven by EEXIST.

9.1 Simulation Setup

The simulator itself runs as a server: the code is in https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/LunarLander [12]. It creates a ServerSocket on port 1213, through which the user or client code can send initial conditions, step the simulation, turn the thrust on or off, and request information about the system's status (altitude, speed and remaining fuel). The simulator also shows a graphical display of the ship's progress, and a crude visual display of landing speed (a red circle whose diameter corresponds to landing speed). The simulator is runable as a jar file ("LL.jar") in https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/EA10HL [10].

The simulation begins with a craft at a given initial altitude (e.g. 300), being pulled down under a constant gravitational pull (-9.8, which of course sounds reasonable on Earth, not so much on the moon), with a given initial amount of fuel (30). Thrust is initially off, but fuel is burned at a constant rate of 1 (unit) per second while thrust is applied. The craft's initial velocity is 0. A simulated timestep of $\Delta t = .125$ is used to step the simulation.

The following equations summarize typical initial conditions:

- $a_g = -9.8$ (acceleration due to gravity)
- $a_T = 12$ (acceleration due to thrust)
- $f = 30$ (fuel)
- $s = 300$ (initial altitude)
- $v = 0$ (initial velocity)
- $\Delta t = 0.125$ (simulation timestep)

At each timestep, the system is updated as follows:

- $f = f - \Delta f \times \Delta t$ if thrust is on
- $a = a_g$ if thrust is off; $a = a_g + a_T$ if thrust is on and $f > 0$ (total acceleration)
- $v = v + a \times \Delta t$
- $s = s + v \times \Delta t$

The simulation ends in one of two cases:

- $s < 0$ In this case, the craft has hit the surface. The system's score for this run is $|1/v|$ (so a lower impact

speed gives a higher score);

- $s > 1000$ In this case, the system has likely locked the thrust on, will exhaust its fuel and eventually crash into the surface. The system's score for this run is $1/s_{min}$, the reciprocal of the minimum altitude (so the closer it got to the surface before reversing velocity, the higher the score).

9.2 EEXIST Interface

The lunar lander simulation can be driven manually, via a telnet connection to the server's port. A simple command-line interface allows setting of thrust on or off, as well as interrogation of current system system (altitude, velocity and remaining fuel). The goal of this set of experiments was to have EEXIST control the lander. This means supplying current system information to EEXIST, and having it output thrust commands (ON or OFF). The basic mechanism was similar to that used in the tic tac toe player: certain regions were defined as input and output regions, and chemical levels corresponded to input and output values.

Three input regions are defined:

- FUEL: the amount of fuel remaining.
  - Typical range: 30 to 0
  - Input address range: $[0,4)$
  - Formula: $SRC = DST = fuel \times 1.5$
- ALTITUDE: the current altitude of the lander
  - Typical range: 300 to 0
  - Input address range: $[8,12)$
  - Formula: $SRC = DST = altitude/60$
- VELOCITY: the current velocity of the lander
  - Typical range: 10 (rising) to -30 (falling)
  - Input address range: $[16,20)$
  - Formula: $SRC = DST = velocity + 30$

There is also an initialization region ("GO"), defined at $[36,40)$, which is initialized with $SRC = DST = 20$ at the beginning of the experiment.

There is one output region defined: THRUST, which is a

binary output, defined at $[24,28)$. $SRC+DST$ is examined at all points throughout that region, and the average value is interpreted as follows:

$$THRUST = \begin{cases} ON & \text{if } average \geq 10 \\ OFF & \text{if } average < 10 \end{cases} \tag{9.1}$$

Simulation begins by populating the input regions, injecting the GO signal, and initially stepping EEXIST one timestep. Simulation then proceeds as follows:

1. the THRUST value is determined from the output region;
2. the corresponding thrust command is sent to the simulator;
3. the simulator is stepped forward one timestep;
4. the simulator is queried as to the current state of the system;
5. EEXIST's input regions are updated accordingly; and
6. EEXIST is stepped forward one timestep

The above steps are repeated until the craft lands, or its altitude exceeds 1000 (these conditions are reported by the simulation server).

If the craft has landed, the score is $|1/v|$ where v is the impact velocity. If the altitude exceeds 1000, then score is $0.1/alt_{min}$ where alt_{min} is the minimum altitude achieved during the run (note that at alt_{min} the velocity must have reached 0).

If multiple tests are run, the scores for all tests are multiplied to give the final score for the individual.

9.3 Sets of Experiments

Whereas in the tic tac toe system an individual was tested against hundreds of opponents, in the lunar lander game controller each individual was (in some cases) scored on a single test only. The system was given control of the craft's thrust; allowed to land the craft; and then scored on its performance.

A wide range of experiments were performed, with various modifications tried, including:

- cutting thrust if the craft is close to the surface (since it often seemed to get close and then reverse direction);
- changing the mutation rate during breeding (e.g. breeding 50 individuals with no mutation; 50 individuals at a mutation rate of 5%; 125 at 10%; and 25 at 25%);
- changing the scaling for scoring based on minimum altitude;
- changing the population size;
- randomizing the initial altitude at the beginning of each population test (but leaving it the same for all members of that population); and
- testing each individual at a variety of altitudes.

It was this last variation that produced the most interesting results, which are described in the next section.

9.4 Some Results

Among the experiments performed on this system, the most interesting are contained in the files raw8-raw11 (with corresponding output files out8-out11). All these runs were made with initial fuel=30, g=-9.8, thrust=12.0, and $\Delta t = 0.125$.

raw8 was developed using an initial altitude of 300. Individual 0 (generation 87) lands at an impact speed of -0.931 (this is considered a successful landing). However, the performance is extremely sensitive to initial velocity: at an initial altitude of 299, the impact velocity is -2.4; while at an initial altitude of 301, the impact velocity is -42.8!

Figure 9.1 shows the impact velocity for a range of initial altitudes. As can be see, while the performance is good at an initial altitude of 300, it degrades quickly away from that point. Figure 9.2 shows the bias gradients for this individual.

In file raw9, individuals were assessed at a range of altitudes, from 295 to 305 (by 1s, i.e. 11 different initial altitudes). Each run was scored, and the individual's final

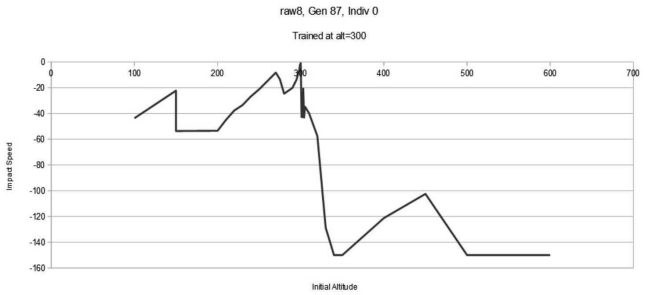


Figure 9.1: Impact Velocity vs Initial Altitude, raw8, Generation 87, Individual 0.

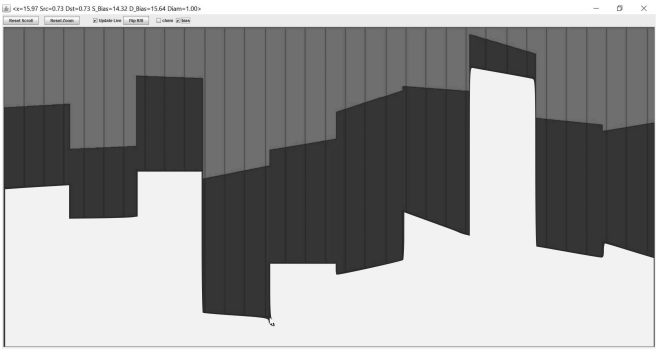


Figure 9.2: Bias Gradients for raw8, Individual 0, Generation 87. Values at the cursor are $Bias_{SRC} = 14.32$ *and* $Bias_{DST} = 15.64$.

score (which was used for the evolutionary process) was the product of scores from each run.

Figure 9.3 shows the impact velocity for different initial altitudes. In this case, the performance seems to drop off linearly away from 300, continually degrading below 300, while initially degrading and then improving towards 400. Figure 9.4 shows the bias gradients for this individual.

In raw10, each individual was assessed at 11 different initial altitudes, from 275 to 325 (by 5's). In this case (see figure 9.5), the performance remains good at values below 300, all the way down to an initial altitude of 200; while at altitudes greater than 300, the performance degrades

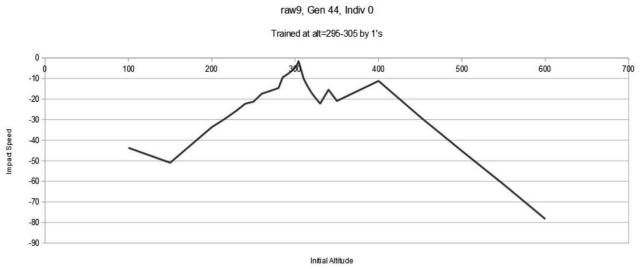


Figure 9.3: Impact Velocity vs Initial Altitude, raw9, Generation 44, Individual 0.

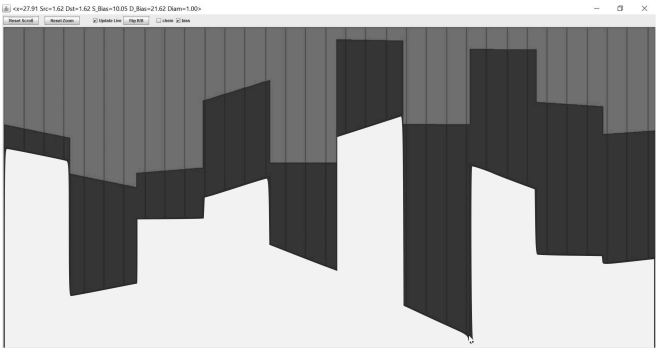


Figure 9.4: Bias Gradients for raw9, Individual 0, Generation 44. Values at the cursor are $Bias_{SRC} = 10.05$ *and* $Bias_{DST} = 21.62$.

steadily (but remains good in the training range of 275-325). Figure 9.6 shows the bias gradients for this individual.

If raw11, each individual was assessed on a range of initial altitudes, from 200 to 400 (by 10's). As can be see in figure 9.7, the performance is good throughout that entire range, except for a few curious spikes near 300. Figure 9.8 shows the bias gradients for this individual.

9.5 Diameter Restriction

EA11HL (https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/EA11HL [11] contains similar experiments, except that the diameter of the input regions

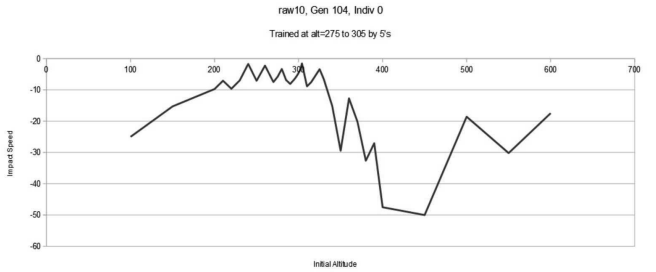


Figure 9.5: Impact Velocity vs Initial Altitude, raw10, Generation 104, Individual 0.

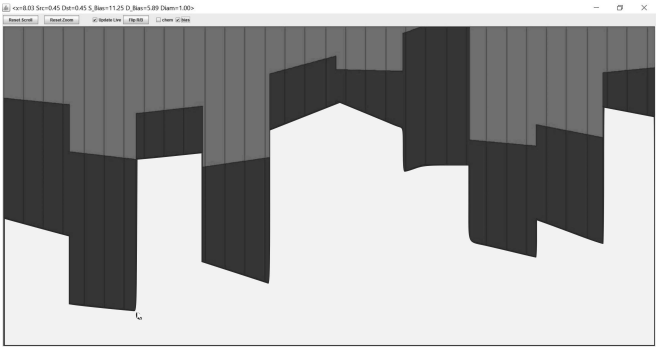


Figure 9.6: Bias Gradients for raw10, Individual 0, Generation 104. Values at the cursor are $Bias_{SRC} = 11.25$ and $Bias_{DST} = 5.89$.

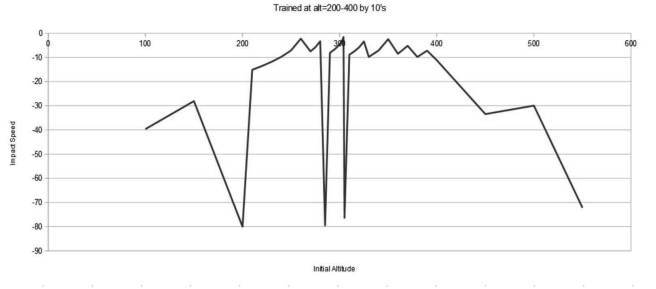


Figure 9.7: Impact Velocity vs Initial Altitude, raw11, Generation 56, Individual 0.

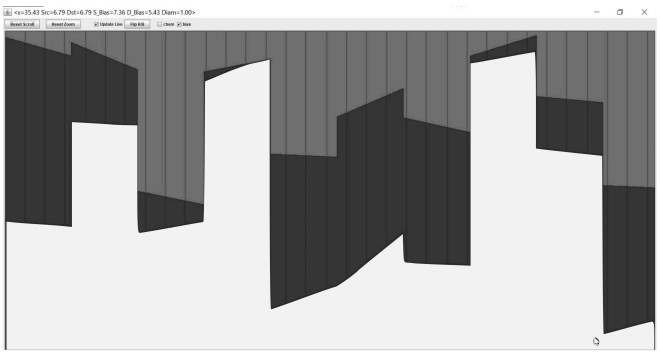


Figure 9.8: Bias Gradients for raw11, Individual 0, Generation 56. Values at the cursor are $Bias_{SRC} = 7.36$ *and* $Bias_{DST} = 5.43$.

has been set to 0. With a diameter of 0, no chemicals can flow into or out of those regions. This does not however mean that those regions don't affect anything: the chemical amounts (offset by the biases) still dictate transfers from $SRC \rightarrow DST$; but the chemicals *which code those instructions* do not themselves change, and are never transferred anywhere.

This restriction prevents system saturation. Consider, for example, the fuel input region. At each timestep, the chemical levels in this region should reflect the amount of remaining fuel. Suppose though that somewhere there are transfer instructions that are removing chemicals from that region. The external system will continually add new chemicals to the system in order to set the fuel region's chemicals to the appropriate level. As those chemicals are transferred to other regions, the total amount of chemicals in the system will increase. After a long enough run, the system may become so flooded with chemicals that it is no longer possible for it to function properly. Similarly, if chemical are transferred *into* an input region, then they will be removed from the system, and the memory may become chemical-starved. Restricting the diameter to 0 eliminates these possibilities.

At first, it seemed that this diameter restriction made

evolution difficult: but in fact, the system evolves as well as with the unrestricted diameters. See raw0diam, generation 141, individual 0, which was trained at a single initial altitude of 300. Figure 9.9 shows the behavior of individual 0, generation 141, when tested against different initial altitudes. As can be seen, the behavior degrades slowly below 300; but above 300, it drops off rapidly. In fact, the system's best performance is at an initial altitude of 304. At 305, it gets very close to the surface, and then engages the thrust fully, runs out of fuel, and free-falls back to the surface at a high impact speed. This is the behavior at larger initial altitudes. In this case, the "script" seems to be well-established: it's trying to descend a distance of 304, achieving a velocity of 0 at the bottom. At an initial altitude of 305, it drops down to a minimum altitude just above the surface, but instead of landing, then engages the thrust until all fuel is spent, and then crashes to the surface.

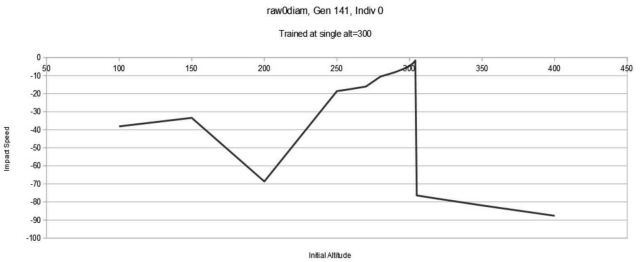


Figure 9.9: Behavior of EEXIST-Controlled Lander with Input Diameters Set to 0.

9.6 Conclusions

It does seem EEXIST is able to control a lunar lander game, though the experiments tended to wander into the "examination of GA dynamics" area rather than studying EEXIST itself. Nonetheless, it's interesting to observe that with the right set of bias gradients, the system can successfully land

for a range of initial altitudes. As usual, there's no yet-apparent reason for the particular pattern of bias gradients emerging from different training methods.

The system clearly looks non-linear, i.e., the performance doesn't always change smoothly with changes in initial altitude. It's also clear the system "doesn't understand what it is doing." It doesn't seem to engage the thrust in direct response to speed or altitude, for example. This is evidences by making small changes in the acceleration due to thrust a_T, which wildly degrades the performance of the system. Similarly, decreasing the timestep causes the craft to come to a near stop at a higher altitude, and then free-fall with the thrust disengaged. This suggests the system may be essentially "following a script" rather than directly responding to input parameters (speed, altitude and remaining fuel). On the other hand, changing the initial amount of fuel (say increasing it, which should have no impact on the lander's performance) significantly changes the system's behavior: suggesting that the amount of remaining fuel *is* being factored into the system's calculations in some manner.

Part of the above can be explained by the limited size of the training sets (comprised of 1, 11 or 21 different initial altitudes). The next set of experiments – the Virtual Eco System ("VEco") – will address this shortcoming, by continually training each individual.

9.7 Exercises

1. Run the LL server and talk to it with telnet. Try to land the ship manually.
2. Run the analyzer and explore the different "raw" files (see README for details on how each file was evolved). Test these with the conditions they were evolved with, then test them with different initial conditions (fuel, altitude, initial speed).
3. Try evolving with very little initial fuel; can you get the system to evolve a more nuanced solution?

4. Try evolving with a variety of initial fuels, and see if you get a more robust system (one that is less sensitive to the initial amount of fuel).

10. Virtual Ecosystem

For an online synopsis of this project, see http://songlinesystems.com/VEco.html [25].

10.1 General Idea

The next set of experiments were based on the idea of a virtual ecosystem ("VEco"). The basic setup consisted of a virtual world in which creatures would:

- move around, expending energy as they move;
- absorb energy ("food") if they encounter it;
- attack other creatures; and
- mate with other creatures.

Each creature was controlled by its own EEXIST system. Information about a creature's immediate vicinity was supplied as inputs to EEXIST, and an output region was monitored to read EEXIST movement requests.

The setup was similar to the lunar lander game, in that a client/server model was used. The server handled all

bookkeeping on the environment (appearance/consumption of food), individual creatures (birth, location, energy level and death) and their interactions (attacking, mating).

Note that there is no need for a separate survival metric in this setup: survival *is* the metric. If a creature runs out of energy, it dies and is removed from the population. If it still has energy, it remains in the population, and may eventually mate, passing on its genes to a new generation.

Most action occurs when a creature moves forward. Attempted movement onto an empty square simply changes the creature's location, and decreases the creature's energy by 1 unit. If this completely depletes a creatures energy, the creature dies.

Movement onto a square containing food increases the creature's energy. Movement into a square occupied by another creature has one of two possible effects:

- if the two creatures are facing each other, mating will potentially occur, with a new offspring appearing behind the creature that is trying to move;
- if the two creatures are *not* facing each other, the moving creature will attack the other creature, absorbing energy from it.

All code and outputs for these experiments are found on the VEco sub-directory, at https://gitlab.com/nickmacias/ChemComp/tree/master/ChemCompAPI/VEco [13].

10.2 Client/Server Setup

The server maintains an $n \times n$ grid of squares comprising the universe in which creatures exist. A simple command line interface allows a client to interact with the universe, using the following commands:

- R - reset the simulation
- B 1 or 0 - allow or disallow breeding
- E - deposit a random amount of energy on a randomly-selected square
- C - create a new creature. The server returns the new creature's integer ID

- id T R G B BL - set the color of the individual whose id is "id." RGB are the red, green and blue colors; BL is 1 for blinking, 0 for non-blinking
- id O - mark this individual as old (yellow eyes)
- id L or id R - indicate that individual "id" wishes to turn left or right
- id F - indicate that individual "id" wishes to move forward one square. The return value from the server is one of:
  - K id - the moved individual killed another creature, whose ID was id
  - M idparent idnew - the movement resulted in mating. idparent is the ID of the other creature involved in the mating; idnew is the ID of the new creature. Note that this information can be used by the client to merge parent genomes for the offspring.
  - OK - nothing special happened
- id Q - query the individual. The server returns a set of information about the individual and its surroundings (detailed below).
- D daylight - sets the daylight level of the system according to the value of "daylight" (0-25)
- Q - shutdown the server and exit

10.2.1 Query Response

The "Q" command asks the server to convey information about the area surrounding a creature. The response string is a single line, consisting of the following:

- the creature's current energy level (0 if the creature is dead)
- a single space
- 24 characters, describing each square in a 5×5 region centered at the creature. Possible characters are:
  - "-" for an empty square
  - "*" for a wall
  - "F" for a square containing food
  - "N," "S," "W" or "E" for a square containing a

creature; the specific value indicates the direction in which that creature is facing.

The order in which this information is returned is shown in figure 10.1. The creature's position is represented by "C" in the middle of the region. In the return string from the Q command, the first of the 24 characters returned corresponds to the square labeled "1," followed by the character corresponding to the square labeled "2," and so on.

12	13	14	15	16
11	2	3	4	17
10	1	C	5	18
9	8	7	6	19
24	23	22	21	20

Figure 10.1: Ordering of Neighbors for Return String from the Query Command. The "Q" command returns the creature's energy, followed by a space, followed by 24 characters representing the state of each of the 24 regions shown. Information is presented for square "1" followed by "2" and so on.

10.3 Additional VEco Mechanics

The above description covers the basic movement, breeding and death of creatures within VEco, most of which are handled by the server. Beyond these details, there are many variations possible, some of which were explored, some left for future work. Note that some of these are handled by the client code.

- Walls are currently impenetrable. A wrap-around model is another possibility, but this has not been explored.

- No energy is expended in turning, but a turn is always followed by a request to move forward. If the forward move is blocked though, the total energy expenditure remains 0.
- Mating is only allowed if both parents have at least a certain amount of energy.
- In some cases, mating is only allowed among parents who have reached a certain age.
- In some cases, mating can occur without parents coming into proximity of each other. At random intervals, new creatures are introduced into the population, but their genome is set to be a mix of the genomes of two randomly-selected parents.
- Creatures are processed in round-robin fashion, but newly-created creatures are always positioned at the head of the processing queue.

10.4 EEXIST Interface

Each creature has an EEXIST system associated with it. There is a client that keeps track of each individual creature (as does the server), and is responsible for communicating with the server. The client conveys information about a creature's neighborhood to its associated EEXIST system; steps EEXIST; and reads any requested action indicated by EEXIST.

The usual genome structure is employed, consisting of 10 equally-spaced regions, each with its own pair of SRC/DST bias gradients. The address space is split into five input and one output region, as follows:

- $[0,4)$ input region R1 (figure 10.2);
- $[4,8)$ input region R2;
- $[8,12)$ input region R3;
- $[12,16)$ input region R4;
- $[16,20)$ input region R5; and
- $[24,28)$ output region.

Regions R1-R5 are used to inject information about a creature's environment into EEXIST. R1, R3 and R5 de-

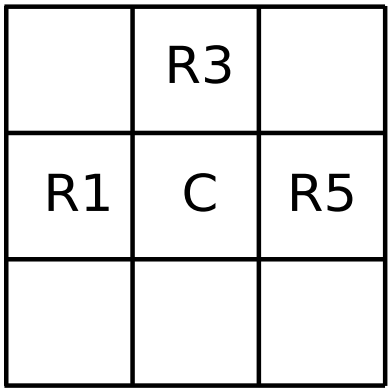


Figure 10.2: Input Regions for VEco. For a creature centered at "C," information about R1, R3 and R5 are sent into the creature's corresponding EEXIST system.

scribe the state of the corresponding regions shown in figure 10.2. For each of these regions, the following SRC and DST chemical levels are set:

- $SRC = DST = 5$ is the square is empty;
- $SRC = DST = 25$ is the square contains a creature that is facing you (and thus is a potential predator or mate, depending on which way you're facing); and
- $SRC = DST = 15$ otherwise (square contains a wall, food, or a creature that is neither a threat nor a potential mating partner).

Initially, R2 and R4 conveyed further neighborhood information; but these were later changed to allow input of more-general information. Region R2 is a general-area input. For the 24 squares around the creature, the number of squares that are not empty (i.e. contain a wall, food, or a creature (in any orientation)) are counted, and SRC and DST in region R2 are set to that count. Thus, R2 shows, roughly, how crowded the area is (though that crowding could be beneficial, dangerous or benign).

In the initial runs of the system, creatures tended to move to the edges of the universe and sit there. If all inputs are based on the contents of the surrounding squares, and noth-

ing is moving, then none of those inputs ever changed, and thus creatures that stopped moving (often, but not always, because they were facing a wall) would never start moving again. This might be rectified by having energy levels drop over time, while also giving a creature information about its own energy level. This however felt a bit "rigged." Instead, a new input was introduced: a cyclic "daylight" variable. This variable runs repeatedly from 0 to 25 and back to 0, changing every 50 ticks of the simulated universe (one tick per creature update). In region R4, $SRC = DST = daylight$. This helps keep the system from becoming stagnant, by changing the inputs to a creature even if the creature is not moving and nothing is changing in nearby squares.

There is a single output region defined at $[24, 28)$, which is interpreted as a movement request by the creature. The average $SRC + DST$ is calculated for this region, and used to determine an output as follows:

- $5 \leq SRC + DST < 10$ turn left and move forward;
- $10 \leq SRC + DST < 15$ turn right and move forward;
- $15 \leq SRC + DST$ just move forward;
- otherwise take no action.

10.5 Experiments

As usual, a number of experiments were run, with a lot of variations in the experimental setup. Throughout the run, genetic information was written to a file, as was a synopsis of birth and death events. The general setup for an experiment was to initially seed the population with a set of individuals with randomly-generated genes, and then allow them to interact throughout time. The server presents a graphical display of the universe, such as shown in figure 10.3.

The client has a user interface, where command-line instructions can be given **by the user** (note that this is different from the command-line interface to the server, which the client controls). The following commands are available to the user:

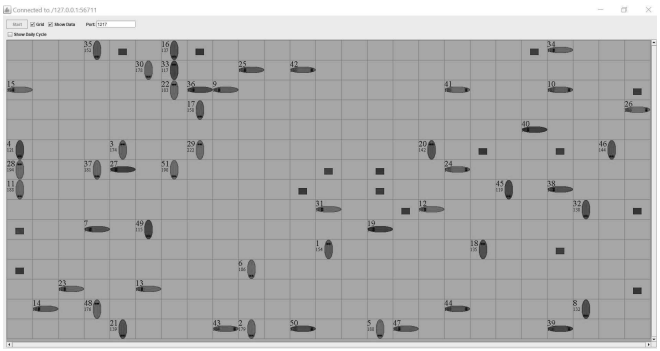


Figure 10.3: Sample Graphical Server Output. Creatures are shown in red; eyes are drawn to indicate which way the creature is facing. Green squares represent food. ID numbers are shown near each creature, and the creature's energy level is shown below the ID (smaller text). The intensity of the creature also correlates to its remaining energy: creatures with more energy are drawn brighter than those with less energy.

- ? - display a help message
- t id - tag an individual (default coloring)
- t id R G B - tag an individual with the given color
- e id - show the general EEXIST display window for the given individual's EEXIST
- e2 id - show the detailed SRC/DST window for the given individual
- f filename - open the given filename for reading genetic information
- c src dst - clone genetic material from individual "src" to individual "dst"
- breed on - enable breeding
- breed off - disable breeding
- pause - pause the simulation
- run - resume the simulation
- d - inject a drone (see the subsection below)
- @filename - run commands from the named file
- reset - reset the simulation (client and server)
- #anything - create a comment (not interpreted)

- Q - quit immediately

In addition to a text/command input area, the client's control panel includes sliders for adjusting:

- the system karma (this affects the karma of all existing individuals, as well as creatures created in the future);
- the initial population size;
- a delay between updates (to make it easier to see what's happening in the graphical output); and
- the number of EEXIST timesteps used to update one creature before moving on to the next creature.

There is also a checkbox for pausing the system (note that the state of this check box is not affected by the pause and run commands), as well as buttons for resetting the system and for exiting the simulation.

10.5.1 Drone Interaction

After letting a population of individuals develop and evolve, it seemed like it might be useful to be able to interact with the population, so a *drone* capability was added to the system. The "d" command requests injection of a drone into the population. The drone is highlighted, and can be maneuvered with the f (forward), l (turn left) and r (turn right) commands. Note that unlike other commands, these keystrokes do not need to be followed by ENTER: they are single-key commands (but holding the key does not successfully register as multiple key presses). This makes it relatively easy to manipulate the drone inside the simulated ecosystem, allowing the user-controlled drone to eat food, to approach and attack other creatures, and so on. The "q" key is used to exit the drone-control mode, and return to the regular command line interface (where ENTER is required to execute a command). Upon leaving drone-control mode, the drone remains in the population, but does not move nor mate.

It is difficult to meaningfully quantify the behavior of the population in response to drone actions, but subjectively, it appears that the population *does* respond to the drone's movements. In some cases, the older creatures seemed to move away as the drone approached. This behavior was

interesting enough to encourage further experimentation, as described in the next sections.

10.5.2 Longevity Data

As the system runs, new creatures are given sequential ID numbers, thus the ID shows relative age of individuals (creatures with higher IDs being younger than those with lower IDs). An initial analysis of the population can be made by looking at the distribution of individuals' IDs.

Figures 10.4-10.11 show graphs of age vs. ID number at different points during the development of the ecosystem. Figure 10.4 shows the population at 100,001 cycles. The population has some individuals with an age around 100,000 (these are likely first-generation individuals), as well as a number of younger individuals (those with IDs above 100).

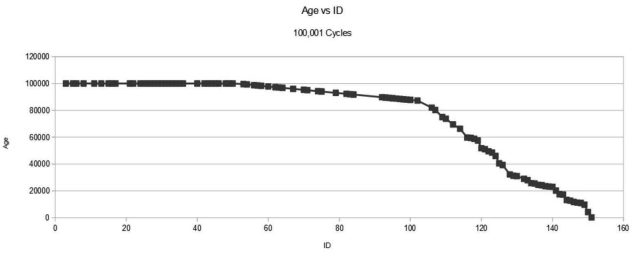


Figure 10.4: Graph of Individual IDs vs. Age, Cycle 100001. A number of members have survived from the beginning of the run.

Figure 10.5 shows the system at cycle 200001. The older individuals remain, but there are more younger creatures. This trend continues through figure 10.8, at cycle 1000001.

If figure 10.9 (cycle 2000001), the oldest individual is younger than 2000001 cycles. It appears the previous longest survivors have died, and the age distribution of the remaining population is becoming linear.

In figure 10.10 (cycle 3,000,001), the oldest individual has an age below 1,000,000, and the age distribution looks linear. Figure 10.11 shows the system at cycle 6,000,0001.

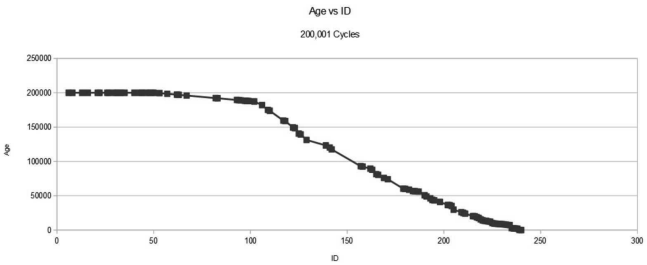


Figure 10.5: Graph of Individual IDs vs. Age, Cycle 200001.

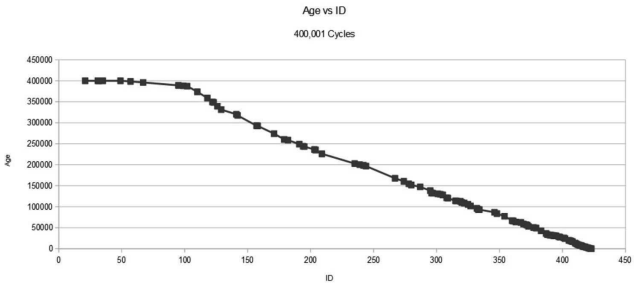


Figure 10.6: Graph of Individual IDs vs. Age, Cycle 400001. A lot of the original population has died.

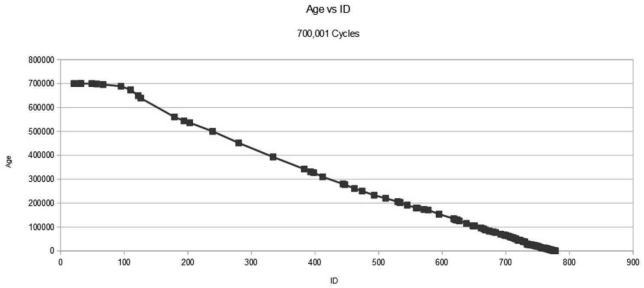


Figure 10.7: Graph of Individual IDs vs. Age, Cycle 700001. Most of the population's ages follow a roughly linear distribution.

The oldest individual has an age below 500,000, and age distribution is close to linear.

The trend seems to be that as the system ages, the oldest

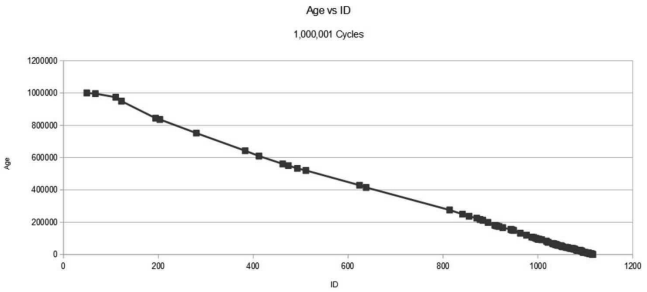


Figure 10.8: Graph of Individual IDs vs. Age, Cycle 1000001. Most of the population is younger than 400000.

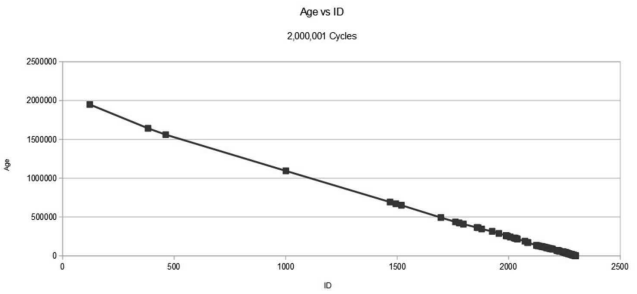


Figure 10.9: Graph of Individual IDs vs. Age, Cycle 2000001.

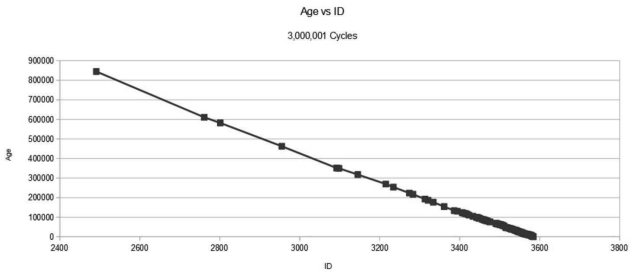


Figure 10.10: Graph of Individual IDs vs. Age, Cycle 3000001. All first-gen members have now died; the oldest members of the population are aged around 850000.

individuals get younger and younger (this was observed in multiple tests). There are (at least) two possible interpreta-

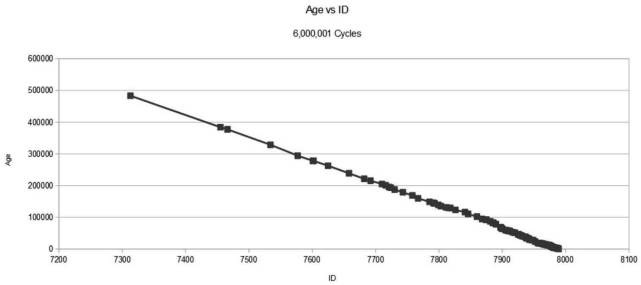


Figure 10.11: Graph of Individual IDs vs. Age, Cycle 6000001. The oldest members are aged under 500000, while most of the population is under 200000.

tions of this:

1. random individuals in the original population happened to be well-suited to survive, but only against the rest of the initial population. As more and more individuals came into the population, the advantages possessed by these first-generation individuals turned out to be nothing special; or
2. the population really *is* learning to survive better, and over time, the entire population has developed survival mechanisms, thus giving none of the individuals any particular advantage over the others.

The most interesting explanation would be #2. In order to test for this possibility, a new experiment was designed: mixing a trained population with a randomly-generated population.

10.5.3 Trained Vs Untrained Population

The goal in this set of experiments was to test the above hypothesis: namely, that a trained population would survive better than an untrained population. To test this, a population of 50 random creatures was created. Then the 25 oldest creatures from an aged population (the one that was 6 million cycles old, from figure 10.11) were cloned into 25 members of the random population. Mating was turned

off, and the ecosystem was simulated, allowing the creatures to interact. The results were consistent across multiple experiments: almost all of the initial deaths were from the young/random population of creatures, while the old population decreased far more slowly. Figure 10.12 shows the decline in each population across time. While the general trend of these graphs is in line with what was anticipated, some of the artifacts are still confusing: for example, why deaths seem to occur in clusters of 3 or 4 at a time.

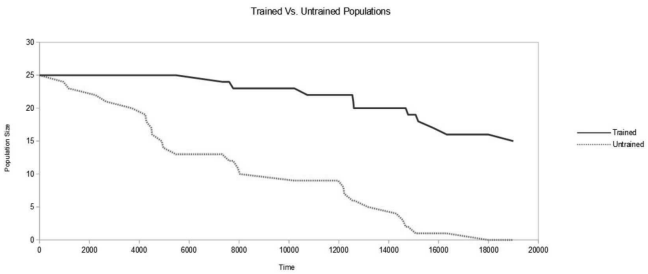


Figure 10.12: Injection of 25 Trained Creatures Into a Population of Random Creatures. As can be seen the trained creatures survive better than the untrained ones. By the time the first trained creature has died (after timestep 7000), nearly half (12 out of 25) of the untrained population has died. At timestep 14680, 80% of the trained population is still alive, while the untrained population is down to 8% (2 survivors out of 25).

10.6 Exercises

1. Run the server (VEco.jar) and control it with a telnet connection.
2. Run the server and main program's .jar files (available here [13]) and evolve a population.
3. Inject a drone into the ecosystem and explore how the population behaves.
4. Vary karma and repeat the above.
5. Change the number of cycles per step and re-evolve.
6. Try working with different sized universes.

7. Change the code to consume energy whether a creature is moving or not.

Part Three - Next Steps

11. Next Steps

The experiments presented in this text represent a single thread of exploration through the space of possible experiments. The genetic approach has been used merely as a stop gap, since the process of designing algorithms for EEXIST is still poorly understood. Many of the design decisions that have been made (e.g. values for Δx, clipping of chemical levels, etc.) are likely enhancing or restricting the richness of the system. As almost all systems explored have varied from each other only in the equations of 10 linear bias gradients, the design space is certainly much larger than what has been explored so far.

While the GA work has yielded some insights into the capabilities of EEXIST, they do not illuminate the *bounds* of the architecture. In particular, only systems that produce very simplified behaviors which score well on the chosen metrics have been studied.

Long-term goals are difficult to speculate on, other than the general notion of "understanding the nature of the sys-

tem's behavior" and looking for natural systems that EEXIST somehow emulates. Shorter-term goals are easier to enumerate, and are described below. This is not an exhaustive collection; it's simply a growing list of questions and ideas that have occurred throughout this work.

11.1 Evolving vs Learning

While the language of evolution and genetic algorithms has been used in these experiments, it's not clear that the development process is actually "genetic." The underlying structure of each individual is identical: all that is changing is the setting of the bias at each location. This is not a fundamental change to e.g. the morphology of the individual; it seems more like a change to the "wiring" (since setting the bias basically changes the center point of an instruction's address space).

Moreover, while a population of individuals has been used to try out different such wirings, there's no reason these variations can't be explored within a single individual. Hence, it may be that the mechanisms being explored in these experiments are more akin to *learning* than to evolution. What is needed is a way for a single individual to retain the results of past runs, and modulate their wiring over time. While this is somewhat a question of viewpoint, a change from working explicitly with a population of individuals to running all the mechanics inside a single individual likely has consequences for both efficiency and flexibility.

Very recent work, performed since the time of the first draft of this text, has explored this question further, and shown that some of the behaviors discussed can be developed within a single individual, i.e., using a GA with a population size of one. This work will need to be further developed, and documented elsewhere.

11.2 Input and Output

The basic model for input and output is only a first attempt at allowing interactions between EEXIST and the outside world. While the current model uses average SRC and DST levels inside a region, there are other possibilities:

- for output, one may require that *all* chemical levels inside a region be above or below a threshold (as opposed to the average value);
- for output, one may compare the difference between SRC and DST to a threshold;
- for input, one might inject chemicals *only once* rather than continually adding or removing chemicals to maintain the desired level.

11.3 Necessity of Bias

Adding bias gradients as the main genome variable seemed to be a key to successful evolution. However, there were 2 other changes made alongside the switch to bias gradients:

1. the system is reset (i.e. initial chemical levels are restored) before each test, as opposed to leaving the system in its state following each test; and
2. experiments switched to testing systems on the entire set of possible inputs, as opposed to (for example) a random subset of tests.

It would be interesting to re-visit evolving using pure chemical levels vs. bias gradients, following implementation of the above two changes. Not only might evolution still be feasible, it's unclear if bias actually adds any new capabilities to the system.

Here again, since the initial draft of this text, work has been done in this area, and perfect digital logic gates have been evolved using only chemical levels as a genetic signature, i.e., with all bias levels set to 0. This too will need to be explored further, and documented elsewhere.

11.4 Uniqueness of Genomes

An open question is how a particular set of bias gradients translates into a particular behavior. As a first step towards understanding this, an XOR gate was evolved 5 times, from 5 different initial (random) populations. After 37 generations, four of the populations had evolved a perfect individual (12800/12800), while the best individual in the 5th population was performing at 99.91% perfection (12788/12800).

Figure 11.1 shows the bias gradients of these 5 individuals. As can be seen, there is no obvious similarity between these configurations.

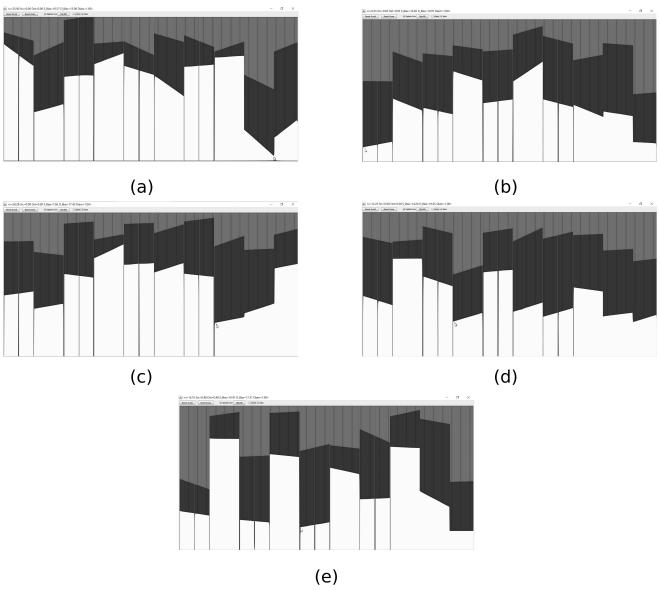


Figure 11.1: Bias Gradients for 5 XOR Configurations. (a)-(d) correspond to perfect individuals; (e) corresponds to an individual performing at 99.91% perfection.

Repeating this exercise with more-complex target behaviors might be illuminating. At the least though, this exercise shows that the mapping between genomes and functions is definitely not injective. It would be interesting to repeat this

over a very large number of runs (say 100 or 1000), and see if clusters of similar bias gradients appear.

11.5 Other Models and Implementations

The chemical model is a first idea about how to represent the enhanced ZBC. Other models are likely possible, and might expose different aspects of EEXIST which are being obscured by the SRC/DST chemical model.

Related to this is the question of implementation: how to build (at least theoretically) a physical version of EEXIST. The chemical model seems unlikely to be directly implementable: using chemical amounts to somehow address distant regions of chemicals seems unnatural. Such addressing in an electrical implementation can perhaps be more easily envisioned: voltage levels can be decreased along a resistive path, and their vanishing can be used to trigger a read or write at a remote location. Triggering at a voltage level near 0 can be used to introduce non-zero karma. The details of such a scheme are a topic of current research.

It is also possible that EEXIST already models some real-world system, and it is an ongoing quest to discover what such a system might be.

11.6 Other Areas to Explore

There are a number of other areas of EEXIST that can be explored:

- All the current work is based on a one-dimensional system (memory is addressed by a single X address). It would be interesting to work with a higher-dimensional system. For a 2D system, one needs 4 chemicals (e.g. SRC_x, SRC_y, DST_x and DST_y). Each location's mix of SRC and DST chemicals thus specifies an (x, y) coordinate for the source and destination of a transfer. Likewise, the bias settings would apply to each of these 4 chemicals. Of course the address space in which these chemicals are placed would also be

two dimensional. All other mechanisms could stay the same. This can obviously be extended to three (or more) dimensions. It's unknown how (or even if) any of these changes would affect the system's capabilities.

- One can change the discretization (Δx and Δt) as well as the maximum and minimum values of x.
- The current work only sets chemical levels with $SRC = DST$: all variation is implemented via bias settings. There are many more possible configurations by allowing SRC and DST to be set independently.
- Most work has been done using *proportional flow*, where the rate at which chemicals move from a source location is proportional to the amount of chemicals present at that source. An alternative is *absolute flow*, where the flow rate is independent of the chemical levels (but still depends on diameter and karma). Recent experiments with absolute flow suggest that it works as well as proportional flow (depending on what the absolute flow rate is set to), which is useful because absolute flow may be easier to implement than proportional flow. More research is required into the differences between these two mechanisms.
- In the tic tac toe work, the address space was broken into 10 regions (to match the locations of the set of bias gradients in the genome). The system requires 9 inputs/outputs (a-i), each of which is allocated to one of the 10 regions as shown in figure 8.1); and the 10^{th} region ($[36,40]$) is used to inject initial chemicals to start the game (since EEXIST is moving first). Since all locations in the address space are used for input and output, it seems as if, in some sense, there's no room for computation: anytime chemicals are moved into most anywhere in the memory (except for $[36,40]$), it contributes directly to a vote for a move in that square. While regions associated with moves that have already been made are in some sense "available," it feels somehow like there isn't enough

space for pure computation. It would be interesting to either:
  - change the genome structure to use smaller bins, and change the input/output regions accordingly, to allow regions of the address space to be essentially unassigned; or
  - increase the address space to cover, say, $[0, 80]$ instead of $[0, 40]$, while also increasing the number of regions in the genome to (in this case) 20.
- Figure 10.12 shows the death of individuals in a mixed population containing both trained and untrained individuals. It would be interesting to explore these dynamics in a population consisting entirely of trained (or untrained) individuals.
- The Gene and Genome class support other genetic structures beyond simple bias gradients. These still remain to be explored. Other modifications in the genetic mechanisms that can be explored include:
  - the number of genes in the genome;
  - alternative mating algorithms (choosing each gene from one parent or the other; selecting collections of genes from one parent or the other; taking a random mix of each parent's genes; and so on); and
  - adjusting the mutation rate.
- Theoretically, one should be able to set negative diameters to effectively reverse the direction of chemical flow. This is relatively simple to explore, but hasn't been studied yet.
- karma is currently fixed for the entire system, across all time. Position-based karma, or karma which changes over time, are additional mechanisms that can be explored. Note that early genetic experiments tended to begin with small karma (for faster simulation), followed by increasing karma for better evolvability. More recent work (including the work reported in this text) has generally set $\kappa = 5$ for the duration of the

runs.

- The address space is currently clipped at 0 and 40, but a wrap-around model may be more natural.
- The current flow model is based on pure flow from SRC to DST ("SD flow"). An alternative is to view an instruction $SRC \rightarrow DST$ as simply connecting two locations, but not specifying a direction for fluid flow. Instead, chemicals flow between connected regions so as to move towards equilibrium, i.e. where the chemical levels at the SRC and DST locations would be the same ("equilibrium flow"). This somehow feels more natural, but is mostly unstudied.

11.7 Other Questions

Some questions about the system that remain unanswered:

- The simulation is based on a discretization of space and time. When the transition is made from finely-discretized to fully continuous, does the behavior change slightly, or is it possible that the behavior changes in some fundamental way?
- Another open question is how many (fundamentally) different behaviors are possible with a given genome structure. In the current set of experiments, the genome consists of 10 pairs of bias gradients, where a bias gradient is basically a pair of real numbers between 0 and 40 (representing, say, the bias value at the start and end of a region). At some level, it seems like such a simple structure would be limited in its possible behaviors. However, if the system exhibits chaos , this limitation could be a non-issue, since even an infinitesimal change in one bias gradient might give significantly different behavior in the resulting system.
- If the instructions in the system are scaled by say $1/10$ (e.g. SRC and DST are each divided by 10); each instruction is relocated from location x to $x/10$; the SRC and DST biases are scaled by $1/10$; κ is divided

by 10; and the diameter of addresses in $(4, 8]$ are set to 0; then the resulting system should behave the same as the original, but will only occupy $1/10^{th}$ the space of the original system. Thus systems can be scaled (and of course the factor of 10 is not special: one could, in theory, scale the system to occupy one millionth of the original space). Similarly, adding an offset to the bias settings and relocating instructions by the same offset allows a system to be moved to different regions of the memory. Combining these, *it seems it's possible to include an infinite number of algorithms in a finite region of address space*. While perhaps limited in practical applications, the theoretical implications of this may be interesting.

Bibliography

If you're reading this text online (e.g. in a PDF), all links and references should be clickable; but if you're reading a hardcopy, rather than typing in a long URL, you can go to http://book.songlinesystems.com and click on the reference number (e.g. [2]).

Index

D

E

F

G

I

J

K

L

M

N

P

R

S

T

U

V

W

X

Z