Infix, Cut and Finite Automata

Clifford A. Reiter
Department of Mathematics
Lafayette College
Easton, PA 18042 USA
reiterc@lafcol.lafayette.edu

Abstract
The behavior of one and two dimensional automata are displayed in two and three dimensions and via animations. Implementations of finite automata in J using "infix" and "cut" to distribute local definitions of finite automata are compared. Abstract automata are considered along with applications to image processing and surface plotting.

Introduction
A *finite automaton* or *cellular automaton* is an infinite array of cells where each cell can assume a value from a finite set along with rules for progressing from one configuration of cell values to another. The set of possible values could just be the set of Boolean values 0 and 1 or it could be a huge set like the set of all allowed RGB-triples used to describe the value of a pixel in a color image. The values in the array are updated in each generation by some local rule. That is, there is a rule that depends only on the values of the cells within some finite neighborhood of each cell and that rule is applied to determine the state of each cell at the next generation. In practice, finite arrays of cells are used and the method selected for handling the boundary cells can have a considerable impact on the observed behavior. For convenience, we will only consider periodic boundary conditions since these are easy to implement and this choice usually has the least impact on the qualitative behavior.

Automata are used to simulate various processes and there is a rich literature of the theory and application of automata, see [4,10]. We will see there is a wonderful variety of qualitative behaviors. Readers familiar with J [2,3] ought to be able to follow all of the details of the examples given herein. Others should be able to get a feeling for the qualitative behavior of automata (and J). We will compare one of the constructions in [8] to alternatives and see the convenience of J for implementing local rules. All examples are given in J Release 2.03 for windows.

One Dimensional Automata
For a first example, consider an automaton that consists of a Boolean vector along with the rule that adds modulo 2 the value of each cell with the value of the cell on its left. Addition modulo 2 is the same as the "not-equals" function, `~:`. The function `rotr` rotates each cell in its argument one position to the right. We then implement the automata described above as `auto1`.

```
   rotr=._1&|.
   rotr i.8
7 0 1 2 3 4 5 6

   auto1=.~: rotr
   u
0 0 1 0 0 1 1 0 0
   auto1 u
0 0 1 1 0 1 0 1 0

   ]v=.0=i.8
1 0 0 0 0 0 0 0
   auto1 v
1 1 0 0 0 0 0 0
```

We can view several iterates of this automaton on the input which has a single initial cell lit.

```
   auto1^:(i.8) v
1 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0
1 0 1 0 0 0 0 0
1 1 1 1 0 0 0 0
1 0 0 0 1 0 0 0
1 1 0 0 1 1 0 0
1 0 1 0 1 0 1 0
1 1 1 1 1 1 1 1
```

The result is a finite version of the Sierpinski triangle. We can easily increase the size of this experiment and view the results.

```
z=.auto1^:(i.128) 0=i.128
pal=.255,:0 0 0
(pal;z) writebmp8 'auto.bmp'
viewbmp 'auto.bmp'
```

There we defined a black and white two color palette: `pal`. We also used two utilities defined in the script *isigraph\bmp.js*. The first is `writebmp8` which is used to create a windows bitmap file called *auto.bmp*. That file is then viewed with `viewbmp`. The result is shown in Figure 1. This makes the self-similarity of this finite array more apparent.

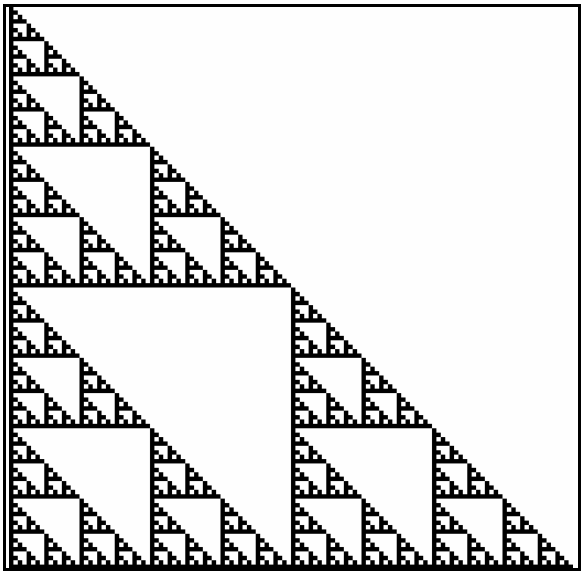

Figure 1. Rule 60 on a single input

Next, the same construction is used on random input.

```
   8 8{.z=.auto1^:(i.128) ?128#2
0 0 1 0 1 1 0 1
0 0 1 1 1 0 1 1
0 0 1 0 0 1 1 0
0 0 1 1 0 1 0 1
1 0 1 0 1 1 1 1
1 1 1 1 1 0 0 0
1 0 0 0 0 1 0 0
1 1 0 0 0 1 1 0
```

As before, this can be saved and viewed; the resulting image is shown in Figure 2. The image has a random feel but there are triangles appearing opening southwest throughout. Also notice near the bottom of the image there is some synchronization occurring.

We next look at more general automata. In particular, we will consider Boolean valued automata that depend only on the cell to the left of the given cell, the given cell, and the cell to the right of the given cell. Since this depends on just three boolean values there are just eight possible configurations and each needs to have a defined result. Here is a table representing the definition of the automaton `auto1`.

L	C	R	Output
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Since the part of the table beneath `L`, `C` and `R` will be the same for each automaton, each automaton can be associated with the column `Output` which specifies a Boolean length 8 vector. Here `0 0 1 1 1 1 0 0`. Below, the function `auto` implements these general automata. First, the input vector is extended at both ends by one element to maintain the periodic boundary conditions. The key idea is to identify which of the rows of the table above applies to each group of three cells. This is done with length 3 infixes.

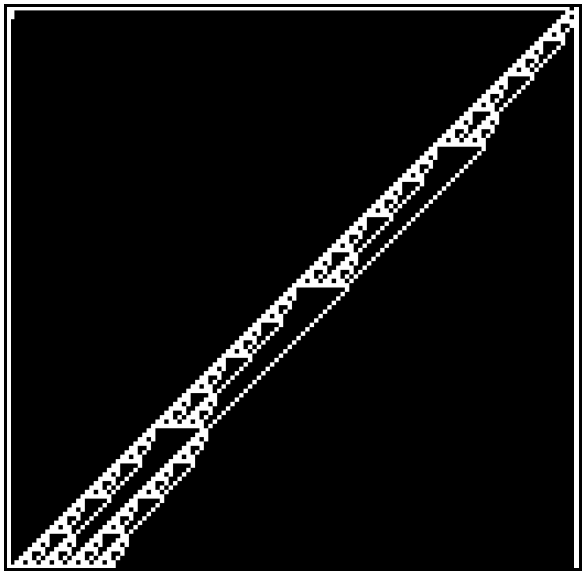

Figure 3. Rule 149 on isolated input.

```
perext=.{: , ] , {.
perext i.8
```

```
7 0 1 2 3 4 5 6 7 0
   u
0 0 1 0 0 1 1 0 0
   3 <\ u
```

```
┌─────┬─────┬─────┬─────┬─────┬─────┬─────┐
│0 0 1│0 1 0│1 0 0│0 0 1│0 1 1│1 1 0│1 0 0│
└─────┴─────┴─────┴─────┴─────┴─────┴─────┘
```

```
      3 #.\u
1 2 4 1 3 6 4
      v=.0 0 1 1 1 1 0 0
      v {~ 3 #.\ perext u
0 0 1 1 0 1 0 1 0
      auto1 u
0 0 1 1 0 1 0 1 0
   auto=.{~ 3&(#.\)@perext
   v&auto^:(i.8) 0=i.8
1 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0
1 0 1 0 0 0 0 0
1 1 1 1 0 0 0 0
1 0 0 0 1 0 0 0
1 1 0 0 1 1 0 0
1 0 1 0 1 0 1 0
1 1 1 1 1 1 1 1
```

It is bulky to list each automata with 8 binary digits, so we can refer to the automata by reinterpreting the 8 digit binary as a number. Thus, some output of rules and their numbers are:

```
Output                  Rule Number
0 0 0 0 0 0 0 0         0
0 0 0 0 0 1 0 1         5
0 0 1 1 1 1 0 0         60
```

Thus, Figures 1 and 2 showed the results of Rule 60 on isolated input and random input. The result of Rule 149 is shown in Figures 3 and 4 on isolated input and random input. These figures show a behavior that clearly contains considerable repetition yet there is still a chaotic or jumbled feel to the figures.

Trying out various rules on a random input gives a feel for the qualitative variations possible. For examples, try Rules 0, 1, 5, 26, 60, 74, 90, 135, 149, 179, 251 and 255. Of course, in Rule 0 everything dies, in Rule 255, everything lives, but for certain automata in between there is a real and wonderful tension between those behaviors.

One dimensional automata can be generalized by considering larger neighborhoods and by allowing more states. Both of these changes can add to the richness of the observed behaviors. For example, Figure 5 shows a 3 state automata on random input.

Here there is an underlying pattern of white surrounding gray triangles - much as was the case for Rule 60, yet there are also black "growths" that persist for a considerable number of iterations but eventually die out.

The Game of Life

Finite automata in two dimensions can be defined in a manner analogous to the one dimensional case. An automaton contains a two dimensional array of states and gives local rules for changing from one generation to the next. Here we will consider the most famous 2-dimensional automata that is often called Conway's game of life [1,6]. This automaton uses 3 by 3 neighborhoods and binary states. A cell is alive at the next stage if at the previous stage either:

• the cell is alive and 2 or 3 of its eight neighbors are alive.

• the cell is dead and exactly 3 of its eight neighbors are alive.

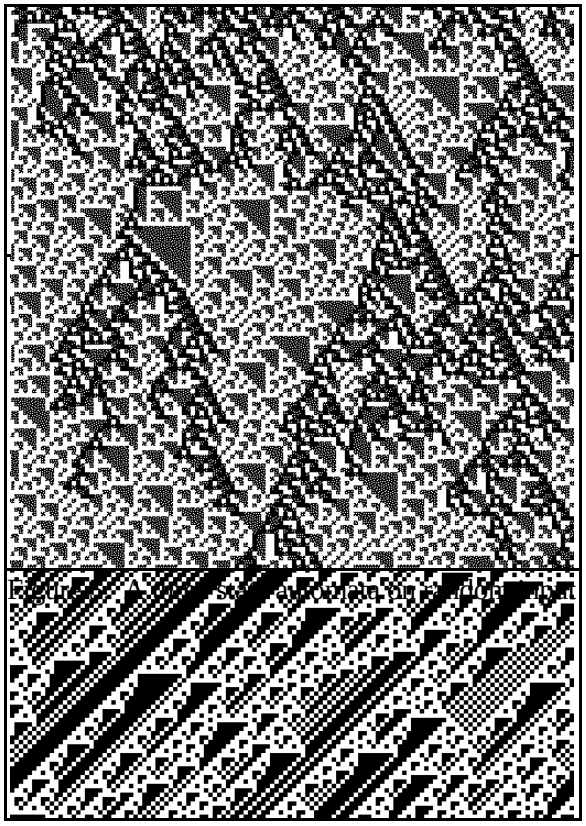

Figure 4. Rule 149 on random input.

All other cells are dead at the next generation.

We describe this game on a 3 by 3 array and then organize work appropriately to apply this to each cell in the array. We need to identify a function, which we will call `filt` (in deference to its image

processing analogue) that applies to 3 by 3 neighborhoods and which describes life. Consider the matrix `L` below and some test neighborhoods.

```
   L          N1         N2         N3
1 1 1      1 0 0      0 0 1      0 1 0
1 9 1      0 1 1      0 0 0      0 1 0
1 1 1      1 0 1      0 1 1      0 0 1

   +/,L*N1
13
   +/,L*N2
3
   +/,L*N3
11
```

Notice that by the rules for life, `N1` should give 0 while `N2` and `N3` should give 1. That is, we only get life if the product of `L` with the 3 by 3 neighborhood has nonzero entries including a nine and 2 or 3 ones or no nine and exactly 3 ones. Thus, the cell will be lit if an 11, 12 or 3 results from the sum of the product of `L` with the neighborhood. Hence, we can define `filt` to test for that.

```
   filt=.e.&3 11 12@(+/)@,@(L&*)
   filt N1
0
   filt N2
1
   filt N3
1
```

We now need to decide on a method for distributing the application of `filt` to all the 3 by 3 neighborhoods.

Cut or Infix

There are two methods that immediately suggest themselves. One is to use cut, `;._3` to directly apply `filt` to the 3 by 3 tessellations, the other is to apply the length 3 infixes as we did in the one dimensional case, but to apply the infixes along two axes. Below are examples of those two strategies. First consider creating the boxed 3 by 3 tessellations on a sample test array `t`.

```
   t
0 0 0 1 0
1 1 1 0 0
0 0 0 1 0
0 1 0 0 1
```

```
0 0 0 0 0
   3 3 <;._3 t
```

0 0 0 1 1 1 0 0 0	0 0 1 1 1 0 0 0 1	0 1 0 1 0 0 0 1 0
1 1 1 0 0 0 0 1 0	1 1 0 0 0 1 1 0 0	1 0 0 0 1 0 0 0 1
0 0 0 0 1 0 0 0 0	0 0 1 1 0 0 0 0 0	0 1 0 0 0 1 0 0 0

```
   3 <\"2 (|:"2) 3 ]\ t
```

0 1 0 0 1 0 0 1 0	0 1 0 0 1 0 1 0 1	0 1 0 1 0 1 0 0 0
1 0 0 1 0 1 1 0 0	1 0 1 1 0 0 0 1 0	1 0 0 0 1 0 0 0 1
0 0 0 0 1 0 0 0 0	0 1 0 0 0 0 1 0 0	0 0 0 1 0 0 0 1 0

Clearly the code for using cut was shorter. Also, the two applications of infix resulted in the transpose of the 3 by 3 neighborhoods - which can be fixed, but the result of `filt` is the same on the transpose, so we can save the time instead of introducing another transpose.

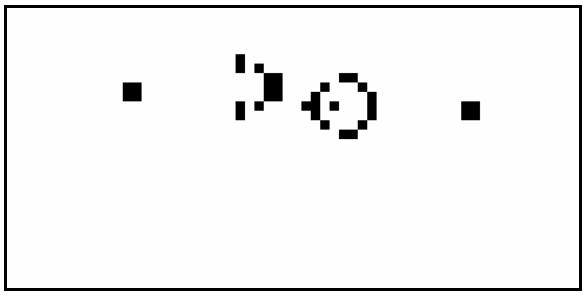

Figure 6. An initial configuration.

The functions `filter1` and `filter2` given below implement the cut and infix based strategies applying `filt` to each 3 by 3 neighborhood instead of box.

```
   filter1=. 3 3&(filt;._3)
```

```
   filter2=. 3&(filt\)@|:"2@(3&(]\))
   filter1 t
1 1 1
0 0 1
0 0 0
   filter2 t
1 1 1
0 0 1
0 0 0
```

The following table gives the time and space required for applying these filters on sample n by n arrays.

n	filter1 time	filter1 space	filter2 time	filter2 space
5	0.32	8344	0.12	2212
10	1.66	45684	0.75	2812
20	8.34	266248	3.57	4564
40	44.1	1679530	15.65	12972

Notice that `filter2` is more efficient in time and dramatically better in space. In fact, if we have as a goal of applying these filters to arrays that are images, we want to be able to deal with arrays that have several hundred entries in each direction. Thus, `filter1`, while very succinct, isn't yet practical for very large arrays.

In order to implement life, we need to apply both the filter above and the periodic extension of boundary conditions. Of course, we need to extend those boundary conditions in two dimensions. Thus we get:

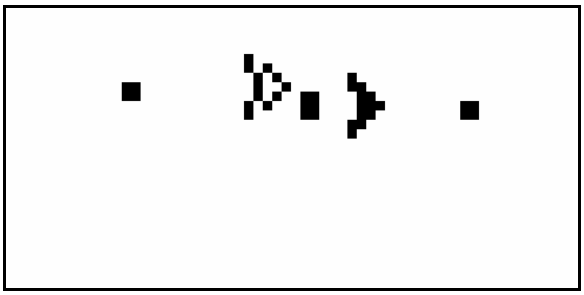

Figure 7. One step of life.

```
   life=.filter2@perext@:(perext"1)
   life t
0 1 1 0 0
0 1 1 1 1
0 0 0 1 1
0 0 0 0 0
0 0 0 0 0
```

Now we look at one step of life on a larger array. Consider the initial configuration in Figure 6. The result of applying life is shown in Figure 7. Notice the stable 2 by 2 blocks on the left and right. These static images don't give a very good feeling for the dynamic behavior of life. Thus, we turn to showing steps of life via an animation.

Animated Life

A nice experiment to try is to create a random array and to iterate life on the array. This can done with the following.

```
   a=.?32 32$2
   aa=.life^:(i.32) a
   animate2 aa
```

Where the function `animate2` is defined by the following.

```
animate2=.3 : 0
256 256 1 animate2 y.
:
pal=.255,:0 0 0
a=.y.
NB. open window for picture
wc=.'pc Animation;xywh 0 0 ',": 0.4*|.2{.x.
wc=.wc,';cc g isipicture;pas 0 0;pscale;'
wd wc,'pcloseok;'
k=.0            NB. frame counter
r=.{:x.         NB. rep's each picture
n=.(#a)*r       NB. total number of frames
while. k<n do.
  (pal;(<.k%r){a) writebmp8 'temp9999.bmp'
  wd 'ctext temp9999.bmp;pshow;'
  k=.>:k
end.
)
```

You can specify the window size and show each frame more than once if the animation runs too quickly. For example `512 512 2 animate2 aa` will create a window about 512 by 512 pixels and show each frame twice.

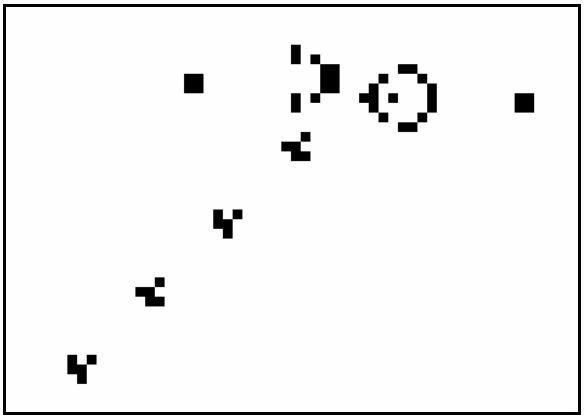

Figure 8. The glider gun configuration after 120 iterations.

The animation on random input shows features that are called "blocks", "blinkers" and sometimes "gliders" appear. See [1]. The configuration shown in Figure 6 is quite special because it replicates itself while generating a sequence of gliders. Figure 8 shows the result of 120 iterations of life on the configuration in Figure 6. Four gliders have been produced at this point. Thus, the configuration in Figure 6 is known as a "glider gun".

The Glider Gun in 3-D

Animating the glider gun is nicest way to view the dynamics involved, but we can think of the iterates of a 2-dimensional automaton as forming a 3-dimensional Boolean array. Then the cells that are lit can be plotted in 3-D. The appendix gives a complete J script for generating the glider gun, its iterates, finding the indices of the lit positions in three space and writing the positions of to a file in a format that can be utilized by the POVRAY raytracing program [9]. The result is shown in black and white in Figure 9.

In this figure, time is shown in the upward direction. The 2 by 2 blocks persisting from state to state appear as guide rails but they do interfere with the growth every 30 iterations. A glider is produced every thirty iterations and they can be seen moving upward and toward the left in the image.

Image Processing as an Automaton

We begin with a grayscale image (for simplicity and printing) that associates with each pixel a graylevel between 0 and 255. Black corresponds to 0 and white to 255. Figure 10 shows a (contrast enhanced) digital grayscale image of the author's family and two research students. The first automaton we will consider is a simple averaging scheme:

Consider the filter mask:

```
   M
0.1 0.1 0.1
0.1 0.2 0.1
0.1 0.1 0.1
```

Since the entries add to 1, multiplying a 3 by 3 neighborhood by `M` and summing the results will give a weighted average of the values at the 9 positions. Thus, if `b` is the array containing the 695 by 798 bitmap in Figure 10, then we can apply this scheme as follows.

```
round=.<.@(0.5&+)
filt=.round@(+/)@,@(M&*)
smooth=.3&(filt\)@|:"2@(3&([\))
s=.smooth b
```

Figure 11 shows the bitmap `s`. Notice that the detail is averaged - there is a bit less contrast and the image looks somewhat blurred. How many iterates of `smooth` would it take to result in a gray blur with no objects visible?

A kind of opposite processing goal is to attempt to highlight edges rather than smooth things. Simple differencing strategies can be used [7], but we suggest here a method that is not direction dependent. This is know as the Sobel edge detector which is a nonlinear detector given by $\sqrt{\Delta x^2 + \Delta y^2}\,1$ where $\Delta x\,2$ and $\Delta y\,3$ are given by the linear filters corresponding to the matrices `dx` and `dy` shown below.

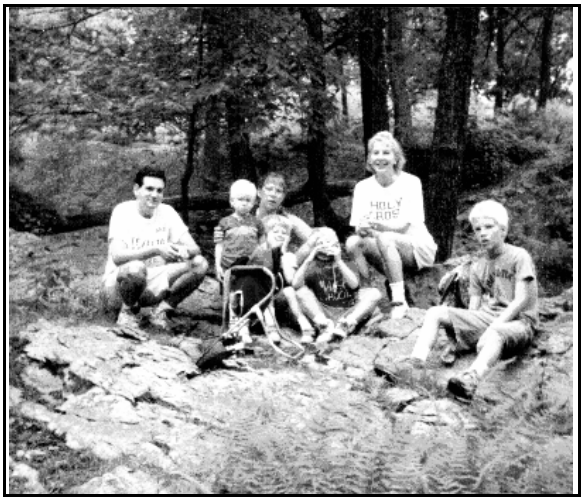

Figure 10. A grayscale image.

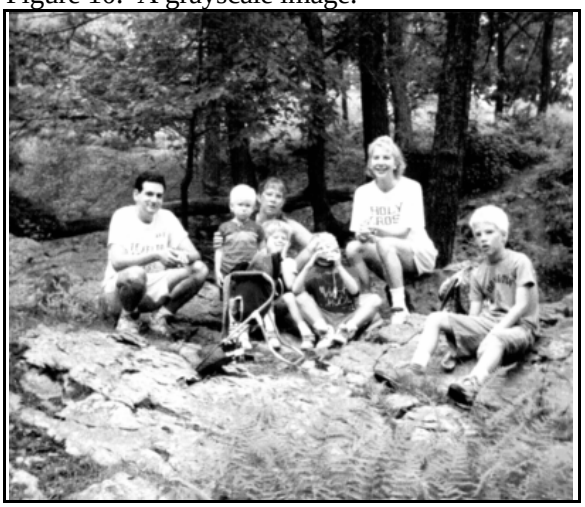

Figure 11. The image with smoothing.

Figure 12. Image after Sobel edge detection.

```
   dx                dy
1 0 _1            1  2  1
2 0 _2            0  0  0
1 0 _1           _1 _2 _1
```

This can be implemented as shown. We create two linear filters `DX` and `DY`. These are put together into `filt` that processes 3 by 3 neighborhoods. The result `sobel` applies `filt` to all the 3 by 3 neighborhoods - in fact, it is identical to `filter2` used for `life` and `smooth`.

```
DX=.+/@,@(dx&*)
DY=.+/@,@(dy&*)
filt=.DX  +&.*: DY
sobel=.3&(filt\)@|:"2@(3&([\))
e=.sobel b
```

Now the result of the Sobel edge detector on the bitmap in Figure 10 is not an integer matrix. The square root gives floating point results. We could rescale and round to show the results - but we will take advantage of a fairly simple utility, `cile`, that classifies the entries into discrete groups by the order of the elements by their size. Thus, `255 cile e` will be a matrix of entries from `i.256` where the smallest entries will be marked with a 0. Equal numbers of 0s, 1s, ... 255s will appear. Finally, we can produce our Sobel edge detected image.

```
cile=.$@]$((/:@/:@]<.@*(% #)),)
E=.255-255 cile e
```

Figure 12 shows the result of that edge detection with colors reversed since we usually draw edges with black. Compared to differencing schemes this does quite well at identifying edges regardless of whether the differences appear on a horizontal or vertical edge.

Organizing Polygons for Surface Plots

While it is clear that the kind of local processing we have considered is useful for games, like life, and image processing, we want to make the point that this local processing really arises in many contexts. In particular, we look at the problem of plotting a surface described in 3-D on a computer screen. We will only consider the part of this problem relevant to local rules; for complete examples of surface plotting, see [8].

The idea is that we want to plot a surface that is described by some function of two variables $z = f(x,y)$4. We can create vectors `x` and `y` that contain the points we want to sample and we want to project the *x-y-z* coordinates of the vertices of quadrilaterals to the screen. That is, suppose

`x=.y=.i.10`, then, if we consider the neighboring *x*-values 3 and 4 and the neighboring *y*-values 7 and 8, this gives a 2 by 2 neighborhood of points in the *x*-*y* plane. Catenating the *z* values, we would get a quadrilateral with vertices:

```
x  y    z
3  7  f(3,7)
3  8  f(3,8)
4  8  f(4,8)
4  7  f(4,7)
```

With a suitable projection, this quadrilateral can be plotted on the computer screen. Since we want to project these *x*-*y*-*z* triples, it is convenient to create an array of points with shape `(#x),(#y),3`. Then 2 by 2 neighborhoods in this array will correspond to quadrilaterals.

```
   x=.y=.i.4
   sin=.1&o.
   f=.sin@[ + sin@]   NB. test func.
   $xyz=.x ([ , ] , f)"0/ y
4 4 3
   {.xyz              NB. points with
x=0
0 0         0
0 1  0.841471
0 2 0.9092974
0 3   0.14112
   filt=.{. , |.@{:
   q=.(2&(filt\)"3)@(1
3&|:)@(2&(]\))
```

The function `filt` takes a 2 by 2 by 3 array and turns it into the desired quadrilateral. The function `q` distributes `filt` over all 2 by 2 neighborhoods. Notice we don't need to worry about the transposes since we don't care whether the quadrilaterals are listed clockwise or counterclockwise. After projection to screen coordinates, we get an image of the surface. Figure 13 shows this surface on a 16 by 16 mesh.

```
   <"2 q xyz
┌─────────────┬─────────────┬─────────────┐
│0 0         0│0 1  0.841471│0 2 0.9092974│
│1 0 0.841471 │1 1   1.68294│1 2   1.75077│
│1 1  1.68294 │1 2   1.75077│1 3  0.982591│
│0 1 0.841471 │0 2 0.9092974│0 3   0.14112│
├─────────────┼─────────────┼─────────────┤
│1 0  0.841471│1 1 1.68294  │1 2  1.75077 │
│2 0 0.9092974│2 1 1.75077  │2 2  1.81859 │
│2 1   1.75077│2 2 1.81859  │2 3  1.05042 │
├─────────────┼─────────────┼─────────────┤
│1 1   1.68294│1 2 1.75077  │1 3 0.982591 │
├─────────────┼─────────────┼─────────────┤
│2 0 0.9092974│2 1  1.75077 │2 2 1.81859  │
│3 0   0.14112│3 1 0.982591 │3 2 1.05042  │
│3 1  0.982591│3 2  1.05042 │3 3 0.28224  │
│2 1   1.75077│2 2  1.81859 │2 3 1.05042  │
└─────────────┴─────────────┴─────────────┘
```

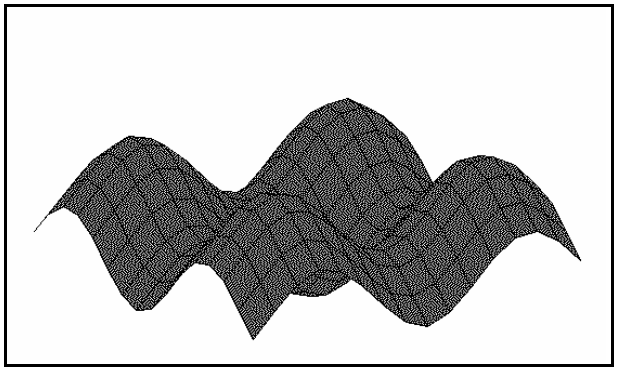

Figure 13. A surface created from quadrilaterals.

References

[1]E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways for Your Mathematical Plays*, Academic Press, New York, 1982.

[2]C. Burke, *J, User Manual*, Iverson Software Inc., Toronto, 1994.

[3]K. E. Iverson, *J, Introduction and Dictionary*, Iverson Software Inc., Toronto, Canada, 1994.

[4]Y. Kayama, M. Tabuse, H. Nishimura, and T. Horiguchi, Characteristic Parameters and Classification of One-dimensional Cellular Automata, *Chaos, Solitons & Fractals*, **3** 6 (1993), 651-665.

[5]C. Linton, From Pentagons and Plasma Clouds to Cliff Dwellers (Part 1) *Gimme Arrays!* **1** 6 (1994), 1,7-10, (Part II) **1** 7 (1994), 1,7-8,16.

[6]D. Peak, M. Frame, *Chaos Under Control*, Freeman, New York, 1994.

[7]C. A. Reiter, Fractals RYIJ, *Vector*, [add ref] (1994).

[8]C. A. Reiter, *Fractals, Visualization and J*, (expected 1995).

[9]T. Wegner, *Image Lab*, Waite Group Press, Mill Valley, CA, 1992.

[10]S. Wolfram, *Theory and Applications of Cellular Automata*, World Scientific, Singapore, 1986.

Appendix. J Script for Creating the Glider Gun in 3D

```
NB. Initial configuration for Glider Gun

GG=.40 _60{.".;._2] 0 : 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
)

NB. Iterates of the Glider Gun - uses life defined in text
GGG=.life^:(i.121) GG

NB. Indices of lit positions
i=.($GGG)#:(,GGG)#i.*/$GGG

NB. Output Utilities
outfile=.'gg3d.pov'          NB. Name output file
write=.[ 1!:2 <@]
append=.[ 1!:3 <@]
eol=.13 10{a.                NB. DOS end of line marker
'' write outfile             NB. Create empty output file
output=.append&outfile@,@(,&eol"1)

NB. Create Povray header
povheader=.0 : 0
camera{
  location <200,200,30>
  direction <0,0,1.6>
  up <0,1,0>
  right <1,0,0>
  look_at<20,75,30>}
#declare white=color rgb<1,1,1>
object{light_source{<200,200,200> color white}}
object{light_source{<-10,200,200> color white}}
object{light_source{<200,200,-10> color white}}
background {color rgb<1,1,1>}
)
NB. end of povray header

output povheader

NB. formatting utilities
chsw=.3 : (': ',:'(a.i.y.){({:x.) (a.i. {.x.)}a.')   NB. character switch
fmtvec=.,&'>'@('<'&,)@((2 2$' _,-')&chsw)@":  NB. vector format '_' to '-' and
' ' to ','
fmtbox=. 3 : 0
z=.1 0 2{y.
```

```
z=.'object{box{',(fmtvec z),',',(fmtvec 1+z)
z=.z,'}pigment{rgb<1,0,1>}}'
)

NB. Output the formatted lit positions
#output fmtbox"1 i
```

Once the J script has been run, the raytracing program POVRAY can be used to change the resulting file *gg3d.pov* into an image. For more information on POVRAY, see [5,8,9]. POVRAY is available by anonymous ftp from *alfred.ccs.carleton.ca*.