SIMP, a Laboratory for Cellular Automata and Lattice-Gas Experiments

Ted Bach and Tommaso Toffoli
Boston University ECE Dept.
tbach@bu.edu, tt@bu.edu

We introduce SIMP, a programming environment for cellular automata and lattice-gases, and demonstrate its use as a laboratory for studying n-dimensional spatially-distributed systems in which complex macroscopic phenomena arise as the aggregate behavior of some simple, local, spatially-invariant, microscopic dynamics. By way of example, we present methods of specifying, visualizing, analyzing, scripting, and interactively controlling SIMP experiments. The latter are expressed as Python scripts and implemented by our fast STEP software engine. (STEP is a multi-platform runtime architecture also suitable for CA/LG hardware.) SIMP is available as free, open-source software from `http://pm.bu.edu`. This paper addresses SIMP version 0.6 and above.

1.1 Introduction

Complex systems often arise as the macroscopic aggregate of the interaction of many simple, local, spatially-distributed microscopic parts. Cellular automata (CA) and lattice gases (LG) can be used as a modeling paradigm for such systems.

The state of a CA or LG is defined on a discrete grid (regular lattice) composed of a large number identical state-variables, and evolves in discrete time steps according to a simple, local dynamics. Due to the simplicity and uniformity of updates, CA models can be implemented very efficiently on a wide variety of computer architectures, allowing the experimenter to deploy the large space-time swatches necessary for studying complex macroscopic phenomena.

The simplicity and uniformity of CA and LG models also makes them relatively easy to implement on a personal computer. A web search for a popular rule like Conway's Game of Life yields hundreds of implementations. However, most are *ad hoc*, supporting only one rule or a small parameterized family of them and providing only rudimentary, canned facilities for auxiliary tasks such as rendering, gathering statistics, and initializing data.

The researcher—as opposed to the hobbyist towards whom most CA software is directed—requires flexible tools that endow her with the freedom to define her

own CA rules and experiments. And, although the `C` programming language is such a tool, she would rather concentrate on conceptual issues than be burdened with low-level issues—optimization, visualizaton, boundary condition handling, *et cetera*—that writing a direct implementation would entail. She will instead desire a CA programming environment that abstracts away accidental details and concentrates on high-level modeling aspects.

Such environments exist, but they vary in their simplicity, efficiency, flexibility, availability, and portability. For example, NetLogo [14] has a relatively simple programming environment and is flexible in that it can handle a wide variety of distributed systems other than CA; however, it is not efficient. Mathematica is flexible [15], however, it is not simple or freely available. Of the many environments—note that we make no attempt to provide a comprehensive survey of them here (see [16] and [8] instead)—JCASim [2] probably comes the closest to the balance of simplicity, efficiency, flexibility, availability, and portability that we seek in SIMP; however, its Java-based implementation and syntax, currently, are not entirely efficient or simple.

Based on collective past experience in programming and efficiently implementing CA experiments on various cellular automata machines [12, 6, 9, 7, 10], we have developed a programming environment called SIMP. With SIMP we aim to externalize the abstractions and methods that have accrued in the CAM community and provide a nice mix of simplicity, efficiency, generality, availability, and portability. Besides, more than the hardware-oriented CAM projects of yore, SIMP is decoupled from implementation-specific particulars.

SIMP supports multiple abstract user-level CA and LG programming interfaces to a low-level set of underlying space-time event processing (STEP) primitives. When a SIMP program runs, the STEP runtime system marshals available computational resources to implement the STEP primitives it invokes in an efficient, effective way. This allows the user to work at a high level of abstraction, and the software to be more portable and widely accessible. Indeed, the current STEP runtime system targets the resources of a regular PC and runs efficiently on Windows, Mac OS X, and Linux systems.

This paper presents several typical CA and LG experiments and demonstrates how they may be programmed using SIMP constructs. Although it aspires to be self-contained, the paper is by no means a complete introduction to or an overview of CA and LG modeling techniques. For this, we instead refer the interested reader to [12, 13, 5].

Similar to [13], we'll first cover a simple CA excitable medium. By programming it in depth we'll reveal the general anatomy of a SIMP program. Next, we'll extend the program in order to make a stochastic excitable medium CA. After that we'll implement the HPP lattice gas via a novel symmetry-preserving construct for building multi-phase LG space-time crystals using a single-phase rule. We'll conclude with some remarks on SIMP, STEP, and future directions.

1.2 A basic excitable-medium CA

The dynamics is defined on a two-dimensional square grid having sites in one of three possible states—*resting*, *ready*, or *firing*—and is summarized as follows:

> *Fire* if *ready* and a neighbor is *firing*; *rest* after *firing*; and become *ready* after *resting*.

This dynamics is called the Greenberg–Hastings rule and is presented graphically in Fig. 1.1. Macroscopically, the CA exhibits propagating *firing*-state waves that—due to the inhibitory effect of *resting*—move forward but not backward.

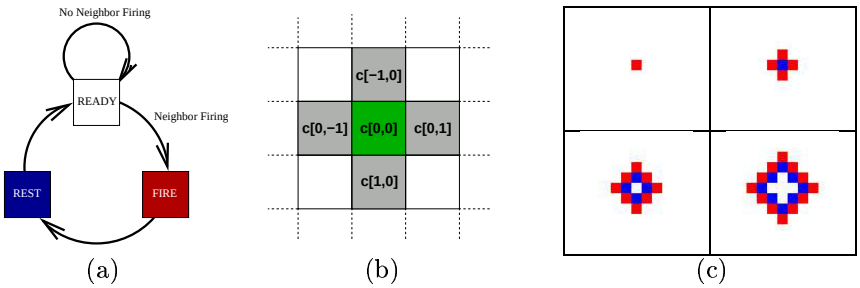


Figure 1.1: **Greenberg–Hastings: A Basic CA Excitable Medium** We show the state transition graph (a), the (von Neumann) neighborhood and state variable offsets that this CA uses (b), and four consecutive snapshots of the dynamics evolving from a plane in the *ready* state with a single *firing* point in the middle (c).

We'll now present the full Python script, `greenberg_hastings.py`, used to obtain the images in Fig. 1.1. Just glance over it—a full explanation will follow. (Although we make no attempt to provide a full introduction to the Python language—for that, see the excellent tutorials at `python.org`—its clear syntax and semantics should be intuitively understandable to even the moderately experienced programmer.)

```
#---------------------------------------------------- HEADER
from simp import *                      # Import simp and helpers
context.switch(simp())                  # Instantiate a simp environment and switch to it
# ---------------------------------------------------- GEOMETRY AND STATE VARIABLES
geometry(size=[11,11])                  # Declare an 11x11 square grid
signals(c=int_range(3))                 # State variable (signal) declaration
READY=0; FIRE=1; REST=2;                # Mnemonics for state interpretations
# ---------------------------------------------------- DYNAMICS
def gh():                               # Function describing the transition rule
  if c==READY:
   if (c[-1,0]==FIRE or c[0, 1]==FIRE or # If north, east, south, or west firing...
       c[ 1,0]==FIRE or c[0,-1]==FIRE):  # (Subscripts indicate neighbor coordinates)
      c._ = FIRE                        # Transition to FIRE
  elif c==FIRE: c._ = REST              # If firing transition to REST
  elif c==REST: c._ = READY             # If resting transition to READY

gh_step = step(rule(gh))                # Package gh into a rule object, then a step object
# ---------------------------------------------------- RENDERING
def tricolor():                         # Function describing an appropriate color map
  if    c==READY: gray._  = 1           #   READY => white
```

```
  if    c==FIRE:  red._   = 1          #   FIRE  => red
  elif  c==REST:  blue._  = 1          #   REST  => blue
rend = render(tricolor)                # Package tricolor into a rendering object
# ------------------------------------------------ RUN
c[:,:]   = READY                       # Initialize all sites to READY
c[5,5]   = FIRE                        # Set site in the center to FIRE

rend.to_file("gh_0.bmp")               # Render init state to an output file
for i in range(1,4):                   # Loop for capturing the views
  gh_step()                            # Do a step of the gh dynamics
  rend.to_file("gh_%i.bmp" % i)        # Render the current state to an output file
```

The script is divided into five sections: header, geometry-and-state, dynamics, rendering, and run. The header imports the `simp` module, creates a new `simp` environment object, and switches the rest of the script to to its context[1].

The geometry-and-state section sets up the spatial grid and attaches state variables to the sites. First it uses the `geometry` declaration method to create a two-dimensional 11×11 square-grid geometry. It does this by setting the value of the function's predefined `size` argument to a list giving the grid size in terms of Y and X. (In general, `size` will be much larger and may declare a grid with any number of dimensions.) Next it uses the `signals` method to allocate the signal `c`, an integer-type state variable capable of taking on values in the set $\{0, 1, 2\}$. ('*Signal*' is SIMP parlance for a site state variable.) Unlike the `geometry` arguments, the `signals` arguments are not predefined: they introduce the named, typed signal objects to be used by the model. Finally, mnemonic names—`READY`, `FIRE`, and `REST`—are given for `c`'s possible values.

The dynamics section gives a Python function—the indented code block starting with "`def gh()`"—that defines the Greenberg–Hastings rule. The statement "`gh_step = step(rule(gh))`" first packages `gh` into a `rule` object and then this into a `step` object called `gh_step`. By packaging `gh` into a `rule` object we obtain a parallel-update primitive that, when called, applies `gh` to all sites. Under the hood, the STEP runtime prepares implementation-dependent structures—such as compiled code loops and lookup tables—for executing the primitive. A `step` object packages a primitive or sequence of primitives that constitute a single step of the dynamics. Packaging a `step` allows the STEP runtime to perform further implementation-dependent optimizations over sequences of primitives that are likely to be invoked over and over again. Calling a `step` not only issues the sequence of primitives that it wraps, but also increments the `simp` environment's `clock`—a user accessible object that gives number of steps of the dynamics that have been run.

The code of `gh` sets the next state of `c`—denoted `c._`—as a function of its present state and that of its neighbors. When `gh_step()` is called—`gh` is applied to all sites simultaneously. `c` denotes the current state of a signal while `c._` denotes the new state after the step. In more detail, the first `if`-statement specifies that the next-state will be *firing* if the present-state is *ready* and a neighbor is *firing*, otherwise it will remain *ready* (since by default `c._ = c`).

[1] The `context` is a helper module that allows one to access the methods and attributes of an object—in this case, the SIMP environment object—without qualification.

The remaining `elif` statements handle the transition from *firing* to *resting* and from *resting* to *ready*.

Within a rule, subscripts are relative to the site being updated. Our rule references the *von Neumann neighborhood* of a site—the site itself and its neighbors at an offset of ± 1 in the Y and X directions as in Fig. 1.1 (b). (Note that SIMP subscripts are listed from the most significant to the least; therefore the subscript in the higher dimension, Y, comes before that of the lower dimension, X. Note also that—in accordance with the conventions of computer graphics—Y grows downwards while X grows rightwards.)

Similar to the dynamics section, the rendering section defines a parallel operation by passing a local function to a constructor. This time, the parallel operation is `render` and it is used to construct the `rend` object. The `tricolor` function indicates how signal values should be rendered to the `red`, `green`, and `blue` color channels of a pixel. Color channel values range from no color at 0.0 (the default) to saturation at 1.0. A special color channel called `gray` sets the value of `red`, `green` and `blue` simultaneously. For three-dimensional rendering, the `alpha` channel determines opacity.

The run section is a relatively straightforward script that initializes the state and runs several updates, rendering each one to an image. The first initialization statement uses Python slice indexing to set all sites to *ready*, while the second sets a single site in the center to firing. Here the subscripts indicate *absolute* coordinates, not offsets. Finally, a small loop uses `rend.to_file` to render the state to image files over the course of several steps invoked via `gh_step()`.

As an alternative to doing batch computations, one will often instantiate a `console` object to perform on-screen rendering and provide an interactive user-interface:

```
ui = console()                          # Instantiate a console called "ui"
ui.bind(gh_step,STEP)                   # Bind gh_step to the console's STEP event
ui.bind(rend,RENDER)                    # Bind rend to the console's RENDER event
ui.start()                              # Start the interactive interface
```

The first line creates the `console` object and brings up an on-screen viewer. The next line binds `gh_step` to `STEP` events generated by the user (pressing SPACE generates one step, pressing SPACE with a numeric argument generates a sequence of them, and pressing ENTER generates steps continuously). The line after that binds `rend` to the `RENDER` events triggered after `STEP` events. Finally, calling `ui.start()` starts the interactive interface. Pressing "`q`" quits causing `ui.start()` to return.

The `bind` method can also be used to bind custom commands to keypress events. For example, to re-initialize the CA state when "`S`" is pressed one could would add the following code,

[1] As of version 0.6 this convention replaces the prior least-significant-first convention inherited from physics and linear algebra. Although the change was precipitated by the adoption of the `numarray` package for handling multidimensional arrays in Python, the most-significant-first convention is more natural when subscripts are interpreted as a generalization of positional notation for numbers—subscripts, like digits, are ordered from the most significant to the least.

```
def seed():
    "Initialize all sites to READY with a FIRE site in the center"
    c[:,:]    = READY          # Set all sites to READY
    c[5,5]    = FIRE           # Set the centered site to FIRE
ui.bind(seed,"S")              # Bind procedure seed to key "S"
```

Lower case keys are reserved for predefined commands; all user-defined commands should be bound to upper case keys. Brief documentation derived from the documentation strings of bound commands is printed when the user presses the help key, "`h`".

1.3 A stochastic excitable-medium CA

Let's now consider a stochastic version of the Greenberg–Hastings rule where state transitions occur probabilistically. The new rule statement is:

> Transition to *firing* with probability p if *ready* and a neighbor is *firing*; transition to *resting* with probability q if *firing*; and transition to *ready* with probability r if *resting*.

We can interpret this dynamics as a model of a prairie fire—a *ready* site 'has grass', a *firing* site is one that's 'on fire', and a *resting* site is 'burned out'. The transition probabilities p, q, and r give the flammability, burn rate, and regrowth rate. The Poisson statistics of the transitions have an average ignition, burning, and regrowth time of $1/p$, $1/q$, and $1/r$. By modulating p, q, and r one arrives at different dynamics as discussed and demonstrated in Fig. 1.2.

Implementing the new dynamics in SIMP amounts to adding stochastic transitions to the basic rule. We'll create three signals—`P`, `Q`, and `R`—and make them into binary random variables having the desired statistics—$\Pr(\texttt{P}=1)=p$, $\Pr(\texttt{Q}=1)=q$, and $\Pr(\texttt{R}=1)=r$. Then we'll use the signals as 'unfair coin tosses' when deciding whether to take a transition—if a signal's value is `1` the transition will be taken, otherwise it will not. The new signal declarations are

```
binary = int_range(2)                  # A binary 'coin' random variable
signals(P=binary,Q=binary,R=binary)     # Declare flammability, burn rate, regrowth rate
```

The modified rule will be

```
def stochastic_gh():
   if   c==READY and P==1:
     if (c[-1,0]==FIRE or c[0, 1]==FIRE or
         c[ 1,0]==FIRE or c[0,-1]==FIRE):
       c._ = FIRE                          # Stochastic transition to FIRE
   elif c==FIRE and Q==1: c._ = REST       # Stochastic transition to REST
   elif c==REST and R==1: c._ = READY      # Stochastic transition to READY
```

We have not explained yet how `P`, `Q`, and `R` implement the desired random variables. Probably the most direct way to do it is by using a random number generator to assign random values to the signals fulfilling the desired distribution. SIMP's randomized value assignment can do exactly this. For example,

```
P[:,:]  = {0:1-p, 1:p}
```

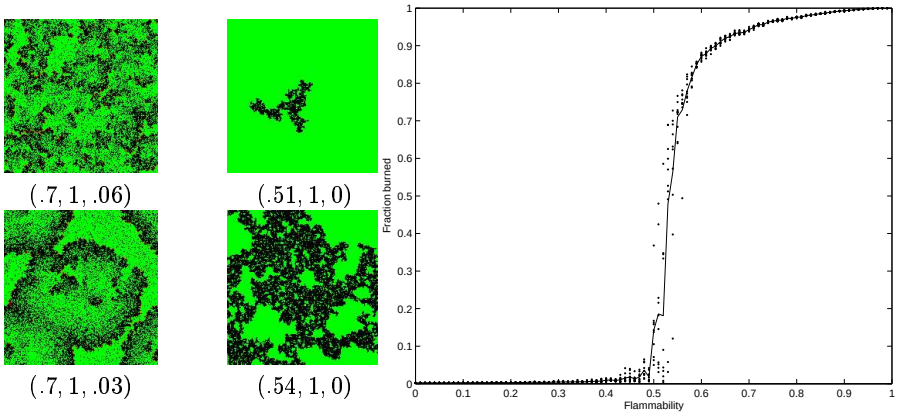


Figure 1.2: **Stochastic Greenberg–Hastings Parameter Space** On the left we show several snapshots of the system taken at different points in the (p, q, r) parameter space. When the flammability p is high and the regrowth rate r is low as in $(.7, 1, .05)$ and $(.7, 1, .1)$ sustained waves form and endlessly propagate over the torroidal space—in a way it is similar to the Belousov–Zhabotinsky chemical reaction. Notice that the regrowth rate affects the wavelength. (The default boundary conditions under SIMP are periodic, wrapping around to form a torus.) When the regrowth rate is zero as in $(.51, 1, 0)$ and $(.54, 1, 0)$ a 'forest-fire' situation arises where the vegetation does not have time to grow back. The amount of forest burned depends critically upon the flammability parameter as evidenced by the amount burned in the two figures. On the right, we have used SIMP to perform a kind of percolation experiment in which we show the fraction of forest burned by a single spark over ten trials as a function of the flammability coefficient. Dots indicate the fraction burned on each trial, and the solid line marks the average over the trials.

sets the values of all `P` signals independently to 0 with a probability of $(1 - p)$ and to 1 with a probability of p. (Distributions need not be normalized to one—for example, the construct `{0:3,1:4}` could be used to set a 3 to 4 ratio of zeros to ones.)

Filling signals with random values works, but requires an expensive call to a random number generator to be made *at each site*. However, refilling sites before *each step* would be terribly expensive. Fortunately, there is a less expensive way. Because our rule can not 'see' long-range correlations—local information tends not to travel too far before 'diffusing'—we can *regenerate* the random variables by *stirring* them. By rearranging the same data in a nonlocal way—say, by shifting it by a random amount[2]—we can cheaply recharge the patch of randomness that a locale sees. It's like a shell game in which a sequence of small patches from a much larger space are revealed randomly, and unless the dynamics is an especially 'smart' adversary tuned to our game it will not be able to tell that the patches it is shown come from our cheaper source of stirred

[2]This is the policy that SIMP actually employs under the hood.

randomness.

To force the distribution to be stirred before each update, we use the `stir` primitive. Like a `rule`, it is a parallel primitive that can be used in making up a step. The code

```
stochastic_gh_step = step(stir(P,Q,R),
                          rule(stochastic_gh))
```

creates a two-stage `step` that first stirs the random signals and then applies the rule.

Finally, we outline the methods used to obtain data plotted in Fig. 1.2. The data was gathered by running an outer loop that iterated over the p values in increments of .01 from 0 to 1. For each value, ten trials were run. At the beginning of each set of trials, a randomized assignment was used to load a new distribution into `P` for the new value of p. At the beginning of an individual trial, `c` was initialized to all *ready* except for a single *firing* spark in the center. Next, an inner loop repeatedly invoked `stochastic_gh_step()`, checking periodically to determine whether 'the fire had burned out' by examing the distribution returned by

```
dist = c[:,:].distribution()
```

The list `dist` returned by the call is a histogram containing a listing of the number of cells in each of the three possible states. Once it was determined the fire had burned out, the fraction burned was computed from `dist`.

1.4 HPP, a simple lattice gas

We now present and program the HPP lattice gas[3]. It is a simple particle-level model of a gas dynamics under which particles move at unit velocity on one of four 'tracks' between lattice sites and scatter at right angles when they collide. The dynamics of HPP is detailed in Fig. 1.3.

The rule in the figure is straightforward; but, on close inspection, one comes to the realization that it results in two interlaced but independent sublattices. By moving only left, right, up, or down a particle will alternate in time between the solid and hollow marked sublattice sites shown in Fig. 1.3. Particles that started on a hollow site will never interact with those that started on a solid site. The whole dynamics is contained in each of the sublattices, we prefer to model just one of them. A straightforward way to do this is to rotate the lattice by 45°, yielding the one depicted in Fig. 1.4. Particles visit hollow and filled locations at alternating time steps, giving the (body-centered cubic) space-time crystal shown.

The program for HPP is similar to those of the CAs we've already considered. First, binary signals (`p0`, `p1` `p2`, and `p3`) on which particles will travel are allocated—`1` represents a particle and `0` a 'hole'. The rotated lattice diagram in the upper-right of Fig. 1.4 uses dashed arrows to show the path signals take in

[3] HPP is named after Hardy, de Pazzis and Pomeau who first presented it in [4].

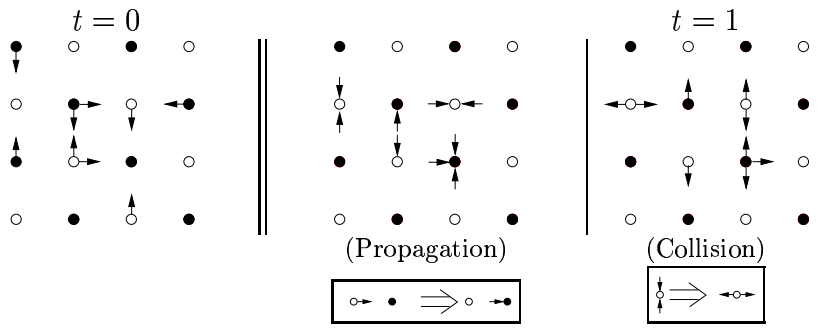


Figure 1.3: **HPP Dynamics** The HPP lattice gas is defined on a two-dimensional lattice. Up to four particles—one ready to move in each of four directions as indicated by the arrows in our example—may occupy each site. As shown in the middle and right, an update occurs in two phases—*propagation* (data-transport) during which the state variables move to adjacent sites and *collision* (data-interaction) during which the collision rule is applied.

the propagation stage—`p0`, `p1` `p2`, and `p3` move by $(1/2., 1/2.)$, $(1/2., -1/2.)$, $(-1/2., -1/2.)$, and $(-1/2., 1/2.)$. Given this data-transport scheme, the data-interaction for the collisions can be written:

```
def hpp():  # scatter at right angles on collision
    if ( (p0==p2) and (p1==p3) ):
      p0._ = p3;    p1._ = p2  # swap horizontally
      p3._ = p0;    p2._ = p1
```

The `if`-statement triggers when two particles collide head-on, four particles collide head-on, or there are no particles. The first case triggers the appropriate scattering while the remaining two are unaffected by permutations.

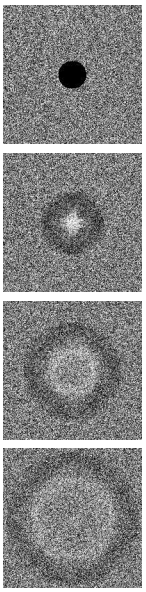
The most significant way that HPP differs from the CAs we've already seen is that it has a signal-propagation stage in each step. This is represented by a `kick` object as in the following

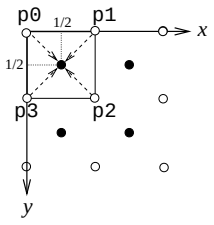
```
hpp_step = step(
       kick(p0=[ 1/2., 1/2.],p1=[ 1/2.,-1/2.],
            p2=[-1/2., 1/2.],p3=[-1/2.,-1/2.]),
       rule(hpp))
```

A `kick` object is constructed from signal and kick vector keyword/value argument pairs that specify the amount by which the signals move when the kick is applied. As previously mentioned, the sequencing of arguments to `step` determines their order of application. In the `hpp_step`—the `kick` comes before the `rule` so that *propagation* precedes *collision*.

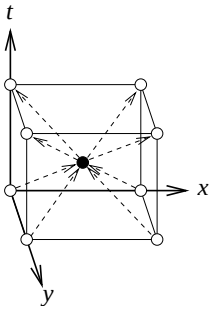
As opposed to the CA's offsets—which indicate where a value is to be *gotten*—the kick vectors specify where a signal is to be *sent*. SIMP uses rational kick vectors in order to avoid floating point roundoff errors and ensure that regular space-time crystal will be obtained. Unfortunately, Python syntax does not support rationals directly, so the `kick` converts floating point kick

```
#-------------------------------- HEADER
from simp import *
context.switch(simp())
#-------------------------------- GOMETRY AND STATE
geometry(size=[200,200])
binary = int_range(2)
signals(p0 = binary, p1 = binary,
        p2 = binary, p3 = binary)
#-------------------------------- COLLISION
def hpp():  # scatter at right angles on collision
    if ( (p0==p2) and (p1==p3) ):
      p0._ = p3;    p1._ = p2  # swap horizontally
      p3._ = p0;    p2._ = p1
#-------------------------------- PROPAGATION
hpp_step = step(
        kick(p0=[ 1/2., 1/2.],p1=[ 1/2.,-1/2.],
             p3=[-1/2., 1/2.],p2=[-1/2.,-1/2.]),
        rule(hpp))
#-------------------------------- RENDERING
def intensity():   # grayscale sum of particles
    gray._ = (p0+p1+p2+p3)/4.
rend  = render(intensity)
#-------------------------------- INITIALIZATION
# Randomize and set a 'vacuum' in the center
sigs = (p0,p1,p2,p3)
for sig in sigs:
    sig[:,:]   =  { 0:.5, 1:.5 }
    sig[ellipse([80,80],[120,120])]   = 0
```





rotated lattice



space-time crystal

Figure 1.4: **Implementing the HPP lattice-gas** The code on the left is a full program for implementing HPP. The four snapshots show the evolution of a localized disturbance after 0, 50, 100 and 150 steps. The two figures on the right give the geometry and space-time crystal used for encoding the rule. The lattice has been rotated by 45° with respect to that of Fig. 1.3 and only one of the two non-interacting sublattices is implemented.

vector elements—such as the ones generated by expressions like "`1/2.`" and "`-1/2.`" to rationals.

With the exception of a few novel constructs, the rest of the program is rather straightforward. The color map renders a gray-scale intensity based on the number of particles in a site. The initialization uses two loops over the signals to randomize them then make a circular vacuum in the center. This last part is accomplished with the help of an ellipsoidal region used as a subscript to select a slice of signals. For example, "`p1[ellipse([80,80],[120,120])]`" would select the sites of `p1` that fall within ellipse circumscribed by the rectangle from $(80, 80)$ to $(120, 120)$.

Programming a LG in the novel way presented here more directly expresses the crystalline structure of the dynamics than do older, perhaps-more-familiar strategies such as the block-partitioned cellular automata in which the symmetry of the 'hollow' and 'solid' updates was broken by the need to specify integer indexes. Because SIMP can handle *rational* indexes, it need only break the symmetry internally when low-level integer indexes are required to construct

operations. These concepts are explained in more detail in [1].

1.5 Final remarks

We have introduced SIMP's core constructs for programming CA and LG experiments; but, there are many more constructs for things like managing interactive user-interface, making parameterized experiments, controlling rendering and viewing, and so on. Although we have focused on two-dimensional systems, SIMP can support as many dimensions as one can reasonably use. For one-dimensional rules, space-time diagrams can easily be rendered. For three dimensional rules, volume rendering is supported via the VTK package. We refer those now ready to start writing SIMP programs to the further examples, documentation, and code available at `http://pm.bu.edu`.

Future versions SIMP will be expanded to support rules written with respect to lattices having non-orthogonal geometries. In particular, a 2-D hexagonal lattice is useful for specifying the FHP lattice gas. (Unlike HPP, FHP does not exhibit spurious symmetries that are not present in a real gas and instead correctly yields the Navier–Stokes equation in the macroscopic limit[3].) Furthermore, direct support for block-partitioned cellular automata—in which blocks of sites separated by a partitioning that changes with time are updated independently—will be added. A past paper [11] motivates some of the constructs that we are implementing. An upcoming paper and expanded tutorial will discuss the new abstractions and how they are represented at the STEP layer.

The primary goal of the STEP interface and runtime system is to make the same SIMP program portable across various machines, architectures, and implementation strategies. The goals, rationale, and implementation strategies of the STEP framework are discussed in [1]. Although SIMP programs are written in the Python programming language—a scripting language—their performance is similar to that of a compiled programs. This is due to the fact that the STEP runtime system translates the high-level constructs and function calls into lookup tables and efficient `C` code. Currently, on an entry-level 400MHz Mobile Intel II Pentium, the PC STEP runtime distributed with version 0.4 can perform HPP's updates at a rate of 5 million sites-per-second (80 CPU cycles-per-site) and on a 2500 MHz Pentium 4 it achieves 41 million sites-per-second (60 CPU cycles-per-site). Memory requirements scale with the size of the space, and interactive rendering with a modern video card typically only slows updates by a factor of two.

Currently, SIMP only supports signals defined over small state sets and update functions are limited in the number of signals that they may use as inputs. This is because STEP converts the update functions into lookup tables (LUTs) and the size of a LUT is exponential in the number of input signals. One can get around this constraint by implementing large updates with a set of smaller, independent updates having smaller LUTs. We are now implementing methods of

automating the compilation of LUTs and considering adding support for mixed integer and floating point types.

We thank Silvio Capobianco for his insightful comments and DOE for supporting this work under grant 4097-5.

Bibliography

[1] T. Bach and T. Toffoli. SIMP/STEP: A rational framework for 'crystalline computing'. In *SCI 2003 Proceedings*, volume XIV, pages 312–320, 2003.

[2] U. Freiwald and J. R. Weimar. The Java based cellular automata simulation system - JCASim. *Future Generation Computing Systems*, 18:995–1004, 2002.

[3] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice gas automata for the Navier-Stokes equation. *Phys. Rev. Let.*, 56:1505–1508, 1986.

[4] J. Hardy, O. D. Pazzis, and Y. Pomeau. Molecular dynamics of a classical lattice gas: Transport properties and time correlation functions.

[5] A. Ilachinski. *Cellular Automata: A Discrete Universe*. World Scientific, 2001.

[6] N. Margolus. *Pattern Formation and Lattice-Gas Automata*, chapter CAM-8: A Computer Architecture Based on Cellular Automata. AMA, 1994.

[7] M. Smith. *Cellular Automata Methods in Mathematical Physics*. PhD thesis, Massachusetts Institute of Technology, 1994.

[8] D. Talia. Cellular processing tools for high-performance simulation. *IEEE Computer*, pages 44–52, September 2000.

[9] T. Toffoli. A fine-grained parallel supercomputer. Technical report, Philips Laboratory, Hanscomb AFB, Department of Defense, November 1994.

[10] T. Toffoli. Programmable matter methods. *Future Generation Computer Systems*, 16(2):187–201, 1999.

[11] T. Toffoli and T. Bach. A common language for 'programmable matter' all that). *Bull. Italian Assoc. for AI*, (2):23–31, June 2001.

[12] T. Toffoli and N. Margolus. *Cellular Automata Machines*. MIT Press, 1987.

[13] J. R. Weimar. *Simulation with Cellular Automata*. Logos-Verlag, 1997.

[14] U. Wilensky. NetLogo. `http://ccl.northwestern.edu/netlogo/`, 1999. Cntr. for Connected Learning and Comp. Based Modeling, Northwestern Univ.

[15] S. Wolfram. *A New Kind of Science*. Wolfram Media, Inc., 2002.

[16] T. Worsch. Programming environments for cellular automata. In *Second Conference on CA and Industry*. ACRI, 1996.