The SIMP/STEP Manual

A Python Environment for Cellular and Lattice-Gas Automata

Release 0.7

Ted Bach

January 8, 2005

Boston University Programmable Matter Group
Email: tbach@pm.bu.edu
http://pm.bu.edu/

Abstract

This reference manual documents SIMP/STEP, a Python environment for cellular and lattice gas automata. SIMP (SIMP Interface to Matter Programming) provides user programming abstractions that target the essence of 'programmable matter' via cellular and lattice gas automata. STEP (Space-Time Event Processor) is an abstract interface to computational machinery (software, hardware etc.) for running the fine-grained, parallel dynamics specified by SIMP programs.

The first chapter is a tutorial that uses examples of cellular automata, lattice gases and partitioning cellular automata to explain how to program with `simp` . This includes methods and classes for declaring and initializing parallel state variables, defining a local dynamics on them (ie. cellular automaton rules), rendering, viewing, creating user interfaces, scripting, and gathering statistics. The second is the reference manual that documents the SIMP programming constructs made accessible to the user through the `simp` module. The third chapter documents the STEP interface, the STEP implementations currently distributed with the software and software engineering and package structuring issues relevant to the developer.

See http://pm.bu.edu/ for current versions of this document.

We thank Silvio Capobianco and our research advisor Tom Toffoli for insightful comments and the DOE for supporting this work under grant 4097-5.

CONTENTS

CHAPTER

ONE

Tutorial

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 swaths 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, and portability. For example, NetLogo (Wilensky, 1999) 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 (Wolfram, 2002), 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 (Worsch, 1996) and (Talia, 2000) instead)—JCASim (Freiwald and Weimar, 2002) 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 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, 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 tutorial 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 tutorial 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 (Toffoli and Margolus, 1987; Weimar, 1997; Ilachinski, 2001).

Similar to (Weimar, 1997), 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 move on to rendering techniques for one-dimensional CA's and see how to program lattice gases and partitioning cellular automata.

See Section 1.11 for information on how to obtain the code for the examples presented in this tutoral as well as further examples.

1.2 Greenberg-Hastings: A basic CA

The dynamics is defined on a two-dimensional square grid having sites in one of three possible states—*resting*, *ready*, and *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. To do an update, the cellular automaton applies the local rule (a) using the neighborhood (b) to all cells simultaneously to get their next state values. In general, CA updates are applied in lock-step parallel. (c) shows three consecutive updates starting from a single firing point. This CA is an excitable medium that exhibits propagating *firing*-state waves that—due to the inhibitory effect of *resting*—move forward but not backward.

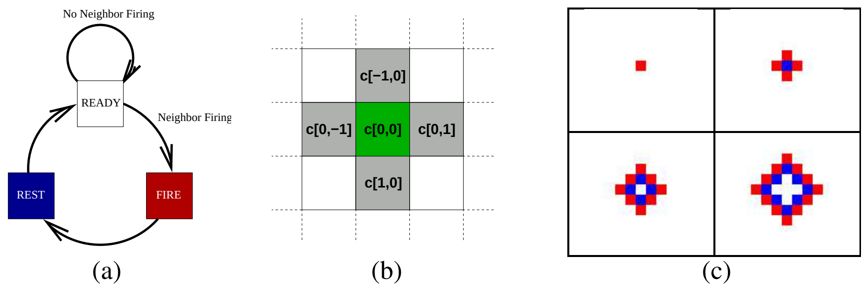


Figure 1.1: **Greenberg–Hastings: A Basic CA Excitable Medium** We show the state transition graph (a), the (von Neumann) neighborhood and state variable offsets that this CA uses (b), and four consecutive snapshots of the dynamics evolving from a plane in the *ready* state with a single *firing* point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

To familiarize the reader with SIMP scripts we present the full code for the Greenberg-Hastings automaton and will disect it in detail. Just glance over the code—a full explanation follows. (Although we make no attempt to provide a full introduction to the Python language—for that, see the excellent tutorials at http://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
Y,X  = 200,200
initialize(size=[Y,X])             # Declare  an  YxX  square  grid
# ------------------------------------          ------ ----  STATE  DECLARATION
c = Signal(SmallUInt(3))           # State  variable  (signal)  declaration
READY=0;  FIRE=1;  REST=2;         # Mnemonics  for  state  interpretations
# ------------------------------------          ------ ----  DYNAMICS
def  gh():                         # Transition  function  definition
  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  neigh  coord)
      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_rule  = Rule(gh)                # Creates  a  Rule  object  for  the  transition  function.
# ------------------------------------          ------ ----  STATE  INITIALIZATION
c[:,:]  = READY                     # Initialize  all  sites  to  READY
c[X/2,Y/2]  = FIRE                  # Set  site  in  the  center  to  FIRE
# ------------------------------------          ------ ----  RENDERING
declarecolors()                    # Declares  the  color  output  signals
def  tricolor():                   # Function  describing  an  appropriate  color  map
  if    c==READY:
      red._=green._=blue._    = 255  #   READY  =>  white
  if    c==FIRE:
      red._   = 255                #   FIRE  =>  red
  elif  c==REST:
      blue._  = 255                #   REST  =>  blue

# Package  tricolor  into  a  rendering  rule  with  red,green,blue   as  outputs
tricolor_rend   = Renderer(Rule(tricolor),rgb)
# ------------------------------------          ------ ----  USER  INTERFACE
ui  = Console(tricolor_rend)        # Instantiate  a  console  called  "ui"
                                    #   and  initialize  its  renderer
ui.bind("STEP",gh_rule)             # Bind  the  console's  STEP  event  to  the  update  rule
ui.start()                          # Start  the  interactive  interface
```

Many SIMP programs follow the same pattern as ‘greenberg_hastings.py’. The script is divided into six sections: header, state declaration, dynamics, initialization, rendering, and user interface. The header imports the SIMP programming environment from the `simp` module and sets up the topology in which the CA will be defined. Together, the state declaration and dynamics sections define the CA's state variables and transition function. The state initialization section sets up the inital state. The rendering section describes how the state is visulalized. The user interface section declares a `Console` object for interactively running the dynamics and viewing rendering results on-screen. We'll now address each section in detail.

Header

The first line in the header section is a Python statement that imports `simp` and all of its definitions[1]. `simp` contains various constructs—methods, functions, and constructors—that the rest of the program uses. Additionally, it maintains global variables and defaults used by these constructs.

Calling `initialize` initializes SIMP's global variables. In particular, *size* parameter declares the size of the two dimensional grid where state variables will be allocated. In the example, the size is 200×200 and is stored two

[1] Although SIMP is designed to be used this way rather than as a module imported with ‘`import simp`’, the latter will work, but all `simp` definitions must be qualified as in `simp.initialize`. When multiple `simp` module instances are required, see `import _locally` in Section 2.9.

variables–`Y` and `X`–for later use. In general, the value of each element declares the size of the grid in that dimension while the number of elements in the size vector specifies the number of dimensions. While *size* is the only parameter that SIMP needs for an ordinary CA, `initialize` also has optional parameters. Among other things, they specify the default lattice generator matrix and the runtime space-time event processor (STEP) implemetation to be used [XXX cross reference].

The grid

The grid defines the geometry at the finest granularity. In two dimensions, one may think of it as a piece of graph paper in which line intersections are coordinatized sites as shown in Fig. 1.2 (a). The grid is bounded by the *size* vector beyond which it wraps around[2] as a torus (Fig. 1.2 (b,c)). Other numbers of dimensions generalize directly.

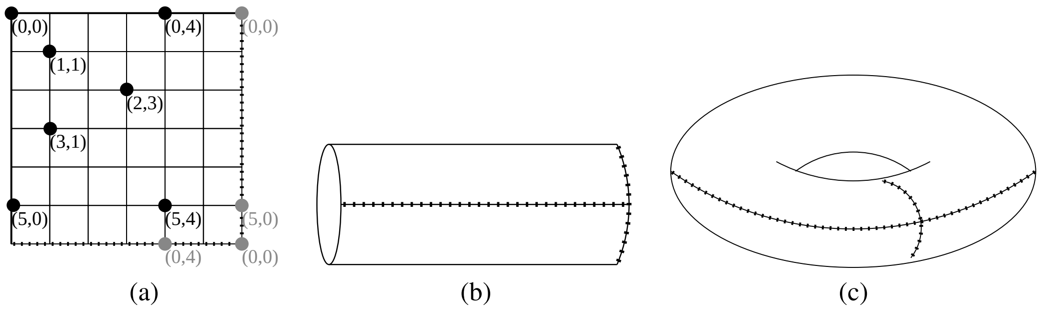


Figure 1.2: **The grid and wrap-around** (a) shows a grid with *size* (6,6) marks some of its sites (line intersections) with circles and gives their coordinates. To emulate a boundaryless space, the coordinates wrap-around with the bottom wrapping to the top (b) and the left wrapping to the right (c) yielding a torus. Coordinates in (a) that wrap around at the boundary are shaded gray.

Note that SIMP coordinate vectors are given in order from the most significant dimension to the least (ie. (y, x) in two dimensions and (z, y, x) in three). This reflects the positional notation implicit in Arabic numbers where the number one hundred and twenty three is written from left to right in decreasing order of significance as 123—more significant digits are appended on the left. It also reflects the storage conventions of `C` and `numarray` where indices of lower significance are stored closer together. When `C` and `numarray` map a multidimensional array to a 1D memory array, the unit-stride dimension—the one with the highest degree of locality—is the least significant, right-most one. When there is a choice, the programmer should align neighbor access with lower dimensions, because, depending on the STEP implementation, doing so increases data locality and may make the computation more efficient. In accordance with computer graphics and typographic conventions, for rendering and display purposes, X goes to the right, Y goes down and Z goes 'behind'.

State Declaration

SIMP state variables are called *signals* and are `Signal` objects. The line '`c = Signal(SmallUInt(3))`' allocates a signal with a ternary integer state set $\{0, 1, 2\}$. *c* has a ternary value at at every point on grid—it's basically a 200×200 array. For convenience, the code also creates some mnemonic names—`READY`, `FIRE`, and `REST`—for each of *c*'s possible values.

Dynamics

The dynamics of a CA is defined locally by a transition function that is evaluated everywhere in parallel. In the dynamics section, the code

[2]Currently, SIMP only supports boundaries that wrap around, however it is possible to emulate other types of boundaries such as fixed boundaries (XXX give reference). In the future, SIMP may directly support other types of boundaries.

```
def  gh():                           # Transition    function   definition
  if  c==READY:
   if  (c[-1,0]==FIRE     or  c[0,  1]==FIRE    or  # If  north,   east,   south,
       c[  1,0]==FIRE    or  c[0,-1]==FIRE):      #   or  west   is  firing.
      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
```

defines the transition function for the Greenberg–Hastings rule. The statement '`gh_step = Rule(gh)`' uses the transition function `gh`—which is just an ordinary Python function—to create a parallel CA `Rule` object that, when called (as in `gh_step()`), applies the transition function to all sites on the grid in parallel.

A transition function locally maps current-state input values to next-state output values. Inside a transition function, signals are accessed locally, therefore, rather of referring to the entire parallel data allocation, as is normally the case, accessing a signal name inside a transition function references its value *at the site being updated*. For example, to check whether *c* at the site being updated is currently in the `READY` state, the transition function uses `c==READY`. To write the output value of a signal, a transition function assigns values to a signal's output attribute—the underscore attribute. For example, to set the the next state value of *c* at the site being updated to *firing* the transition function makes the assignment '`c._ = FIRE`'.

`gh` looks at the *von Neumann neighborhood* of *c*—the site itself and its neighbors at an offset of ±1 in the Y and X directions as shown in Fig. 1.1 (b). The neighbor value of *c* to the right is `c[0,1]`, to the left is `c[0,-1]`, above is `c[-1,0]`, and below is `c[1,0]`. SIMP subscripts are listed from the most significant to the least[3]; therefore the subscript in the higher dimension, Y, comes before that of the lower dimension, X. In accordance with the conventions of computer graphics—Y grows downwards while X grows rightwards (Z grows away from the viewer). **Note:** Within a transition function, neighbor values are referenced by relative subscripts. Outside of transition functions, subscripts are absolute coordinates.

Some comments and restrictions on rules and transition functions

`gh` does not always assign an output value of *c*. For example, if the current state is *ready* and no neighbors are firing, the rule does not write a next state value. What, then is the next state value of *c*? There are two options—*c* is set to some default constant value like 0, or *c* passes through unchanged (identity). In this function, either option would yield correct behavior, but what is actually happening is that the default value is the input value of *c*. Thus the rule specifies that if no neighbors are *firing* and *c* is *ready* then *c* remains *ready*.

Note: In general, the default value for a signal is its input value *if and only if* that signal appears as an input. Otherwise, the default output value is zero. This behavior ensures that extra inputs to the rule are not generated in the case that a signal is only written. If this were not the default behavior, all outputs would also be required to be made into inputs, which would be bad since the cost of evaluating a transition function is exponential in the number of inputs when a lookup table is used to implement the function. (This behavior applies only to signals that appear as outputs. The values of signals that do not actually appear as outputs of a rule are not changed.)

When a `Rule` object is constructed, the transition function is *strobed* during which all global names (names assigned outside the transition function) are bound to their values *at that time*. Because of the binding, unlike a normal Python function, if a global value is subsequently changed, the `Rule` will not notice. Strobing can be used to make parameterized sets of rules by changing global values before constructing each rule. [XXX reference an example demonstrating this]

Neighbor coordinates must be constant within a transition function—they can not be modified as a function of the input signals. SIMP does not allow neighbor coordinates to be computed on-the-fly as a function of the rule inputs because

[3]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.

the doing so would mean that under different conditions, different neighbor values would need to be made available. The input set would be dynamic. Such a situation would make it much more difficult to gather the inputs needed for the transition function in a homogenous, optimized way. However, one can declare named neighbors outside of a rule as in '`nw = c[-1,1]`' and such neighbors can be used freely within a rule as in '`if nw==1: ...`'.

All references to signal objects *m*ust be contained within the top level of the transition function and a rule will ignore any signal reads or writes carried out within sub-functions called by the transition function. This is because, when a constructing a `Rule` object, SIMP only analyzes the code of the transition function itself and not the code of any sub-functions that it may call. Therefore, accessing signals in sub-functions will yield incorrect behavior. (In the future more recursive code analysis may allow signals to be accessed from sub-functions)

When accessing signals *the transition function can only access signals using global names.* This is because the code analysis also does not currently perform the dynamic type analysis necessary to follow the assignment of signals to alternate local names. Therefore, a statement like '`a = c; a._=1`' inside of a transition function would fail.

Initialization

This section uses signal subscripts to assign initial values to the signals. As previously mentioned, the subscripts here are global.

```
c[:,:] = READY              # Initialize all sites to READY
c[X/2,Y/2] = FIRE           # Set site in the center to FIRE
```

The first statement uses Python/`numarray`-style multi-dimensional slicing to assign all states to *ready*. The second sets a single site in the center to firing.

One may also read out signal values using subscripts. For example,

```
a = c[0,5].scalar()
```

reads out the scalar integer value at coordinate $(0,5)$. Slices read out an array of values—for example, to read out the 5×5 array of values from $(0,0)$ up through, but not including $(5,5)$ one would use

```
values = c[0:5,0:5].array()
```

The array returned is a `NumArray` object. One can also assign slices using array values

```
c[15:20,10:15] = c[0:5,0:5].array()
```

To read the entire array, one can a statement like

```
c_arr = c[:,:].array()
```

SIMP usees the `numarray` module extensively for representing and manipulating `Signal` data. The `numarray` module and classes provide full Python support for multidimensional arrays. This support includes utilities for generating arrays, saving them to files and performing all kinds of transformations and analyses.

For example, a `NumArray` object can be saved to a file in a variety of ways. One way is using the Python `pickle` module.

```
import pickle
pickle.dump(c_arr,open("state_file","wb")) # dump the array to "state_file"
```

To read the file back, one can use

```
c_arr = pickle.load(open("state_file"))
```

Another option is to use the `tofile` method of the array. Certainly, there are many other ways as well. While we mention some of the useful calls in this tutorial, one should see the Numarray documentation available at http://www.stsci.edu/resources/software_hardware/numarray for full details. SIMP also defines some additional helper numarray functions such as `makedistribution`, `getdistribution`, `array_to_ppm`, `magnify2d`, see Section 2.3.3 and Section 2.3.5 for details.

Rendering

The rendering section declares how to convert state information into images that can be displayed. Like the dynamics, rendering behavior is defined by a `Rule`. Unlike the dynamics, however, the `Rule` does not output to oridnary signals. Instead it writes to special `OutputSignal` objects—they are just like `Signal` objects, except that they are write-only. SIMP also supplies special `Renderer` classes to manage the conversion and output of rendering data to image arrays. The `Renderer` provides an interface that's suitable for on-screen display by a `Console` user–interface.

The output of a rendering rule is an array containing color information. Color channels are encoded with 8 bit color `OutputSignal` objects of type `UInt8` (8 bit unsigned integer). Although one may directly declare output signal objects with the `OutputSignal` class, SIMP provides a special function called `declarecolors` which automatically declares the color channels—*red*,*green*,*blue*,*white*, and *alpha*—in the global namespace. (*white* is for grayscale rendering and *alpha* is for opacity in three–dimensional rendering.) So instead of having to write a statement like '`red = OutputSignal(UInt8)`' for each color channel, one can instead simply call `declarecolors`.

The RGB rendering function was defined as follows

```
declarecolors()                 # Declares the color output signals

def tricolor():                  # Function giving a color mapping
  if   c==READY:
      red._=green._=blue._   = 255  #  READY => white
  if   c==FIRE:
      red._   = 255             #  FIRE  => red
  elif  c==REST:
      blue._  = 255             #  REST  => blue
```

The default value for an output signal is 0. This is also the minimum intensity for a color. Because the colors are `UInt8`, the maximum value is 255 (this corresponds to 24 bit RGB color image).

After declaring the color map function the program creates a `Renderer` object as follows

```
rend = Renderer(Rule(tricolor),rgb)
```

The first parameter is the rendering rule, the second is the set of outputs that are rendered. The name `rgb` was declared by `declarecolors`. It is a tuple consisting of the output signals `(red,green,blue)`. Notice that in this code, rather than using the *white* channel to write a white output when '`c==READY`', the function assigns the RGB values. This is because *white* is reserved for *grayscale* mode. For grayscale rendering, `declarecolors` also declares `grayscale` as `(white,)` so that the renderer may render grayscale images from rules that only set the *white* intensity value. For an example of grayscale rendering, see Section 1.4.

User Interface

All of the example SIMP programs instantiate a `Console` object to perform on-screen rendering and provide an interactive user-interface. In the current example, the code for declaring and initializing a `Cosole` is

```
ui = Console(tricolor_rend)          # Instantiate  a console  called  "ui"
                                     #   and  initialize  its  renderer
ui.bind("STEP",gh_rule)              # Bind  the  console  console's  STEP  event  to  the  update  rule
ui.start()                           # Start  the  interactive  interface
```

The first line creates a `Console` object called `ui` and brings up an on-screen viewer using `tricolor _rend`, the renderer previously defined. Because `tricolor _rend` implements the `Renderer` interface, the the `Console` knows how to use it to generate rendered image arrays and control the region that is rendered.

The next line binds `gh _rule` to the `'STEP'` events. The console generates a '`STEP`' event and will call `gh _rule` when it wants to update the dynamics. After a `'STEP'` it calls the renderer and displays the result on-screen. '`STEP`' events are generated when the user presses `Space` to do a single step or `Enter` to run them continuously. Finally, calling `ui.start()` starts the interactive interface, and does not return until the user quits by pressing `q`.

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,

```
def  seed():
     "Initialize   all  sites  to  READY  with  a  FIREING  site  in  the  center"
     c[:,:]      = READY          # Set  all  sites  to  READY
     c[Y/2,X/2]    = FIRE         # Set  the  center  site  to  FIRE
ui.bind("S",seed)                 # Bind  'seed'  to  key  "S"
```

Lower case keys are reserved for predefined commands so 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`. For this reason, it is a good idea to define documentation string as is done above.

Scripting

One need not define an interactive user interface if the program is meant to run in script mode. For example, the following code runs several iterations and generates a sequence of portable pixmap ('`.ppm`') images like the ones shown in Fig. 1.1 (c)

```
for  i  in  range(4):
     img_arr  = tricolor_rend()    # get  an  array  containing  the  current  state
     open("gh%i.ppm"   % i,"wb").write(array_to_ppm(img_arr))      # save  to  file
     gh_rule()   # do  a  step  of  the  dynamics
```

Calling a renderer object returns the contents of the renderer's current view (of course, the renderer has methods for changing the view see Section 2.7). Arrays returned by the renderers are three-dimensional numarrays indexed by Y,X, and the color outputs. The SIMP helper function `array _to _ppm` converts such arrays to '.ppm' formatted strings, which is a handy because the format is easy to read and write and can be converted to many other formats using Jef Poskanzer's widely available `netpbm` library and command-line tools.

One may also drive the `Console` from a script. For example, to display a sequence of four updates on-screen before starting the console, one could use the code,

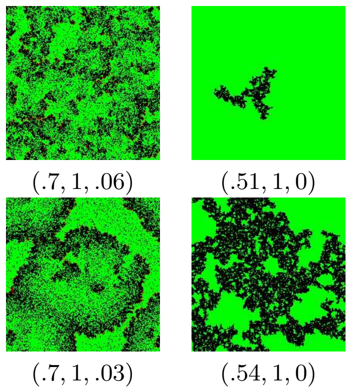
```
for i in range(4):
    ui.render()   # render the current state
    gh_rule()   # do a step of the dynamics
ui.start()   # begin running the console
```

1.3 A stochastic excitable-medium CA

Now that we have seen how to program a simple CA and have a basic idea of how the SIMP environment works, let's now consider a randomized (stochastic) version of the Greenberg–Hastings rule where state transitions occur probabilistically. We'll show how to program it efficiently with SIMP and how to gather statistics. 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. 1.3.



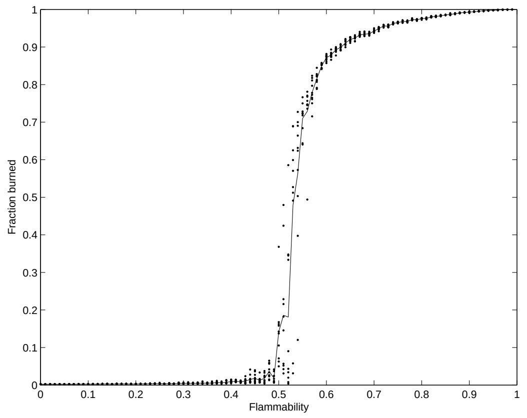


Figure 1.3: **Stochastic Greenberg–Hastings Parameter Space** On the left we show several snapshots of the system taken at different points in the (p, q, r) parameter space. When the flammability p is high and the regrowth rate r is low as in $(.7, 1, .05)$ and $(.7, 1, .1)$ sustained waves form and 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. 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.

Implementing the new dynamics in SIMP amounts to adding stochastic transitions to the basic rule. We'll create three signals—P, `Q`, and `R`—as binary random variables with the desired distribution—Pr(`P`=1)=p, Pr(`Q`=1)=q, and Pr(`R`=1)=r. We'll use these 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[4]

[4]Mastering Python helps one to master SIMP. For example, a more succinct way to declare the three signals is with the line '`P,Q,R = map(Signal,[binary]*3)`' The code `[binary]*3` constructs a list with three references to the binary type, `map` is a builtin Python function

```
binary = SmallUInt(2)        # Binary state variable type with values in 0,1
P = Signal(binary)           # Declare the three signals
Q = Signal(binary)
R = Signal(binary)
```

The modified rule is

```
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 distributions. The most direct way is to use a random number generator to assign random values to the signals so as to fulfill the desired distribution. SIMP provides makedistribution to do this. The function takes two arguments—the shape of the output array and the distribution of values. For example,

```
P[:,:] = makedistribution(P.shape,[1-p,   p])
```

assigns the values of P to a newly generated random array with the same shape as P and having values of 0 and 1 distributed independently probability $(1-p)$ and p. (Distributions need not be normalized to one—for example, a distribution parameter of [3,4] would create an array with a 3 to 4 ratio of zeros to ones.)

Although this strategy works, requires an expensive call to a random generator *for each array element*. Refilling the distribution sites before *each step* slows the computation considerably. 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 local random variables by *stirring* them. By rearranging the same data in a non-local way—say, by shifting it by a random amount[5]—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 randomness.

To stir the distribution before each update, we use the Stir operation and create an update Sequence that stirs the random signals before calling the stochastic rule.

```
stochastic_gh_step = Sequence([Stir([P,Q,R]),
                               Rule(stochastic_gh)])
```

We have introduced two new STEP operations—Stir, which we just explained and Sequence. A Sequence is not really an operation in and of itself, but instead packages an ordered sequence of STEP operations. In addition to being a useful programming construct, a Sequence informs the STEP about sequences of operations that will be called together. A STEP may then optimize such a sequence.

Finally, we outline the methods used to obtain data plotted in Fig. 1.3. 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

that, in this example, calls the Signal constructor on each list element and returns a list with three new binary signals. Finally, Python list comprehension handles the assignment of P,Q,R to the three signals returned by map.

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

beginning of each set of trials, a randomized assignment was used to load a new distribution corresponding to the new value of p into 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 every tenth step to determine whether 'the fire had burned out' by examing the distribution of states returned by 'dist = getdistribution(c[:,:],0,3) '. getdistribution is a SIMP helper function that—given an array and a range of integer values—returns a histogram vector with the number of elements having each value in the range (values between 0 and 3). Once it was determined that the fire had burned out, the fraction burned was computed from dist .

1.4 A one-dimensional CA and space-time renderer

One-dimensional systems are usally rendered using space-time diagrams in which the history of several states is displayed as a two dimensional image with the horizontal axis representing space and the vertical representing time. The XTRenderer provides special support for space-time rendering. We present a simple CA called PARITY as an example. We define it on a one-dimensional lattice with binary signals. The transition function adds the left, right, and center values and transitions to 1 if the sum is odd or 0 if the sum is even.

1.4.1 The program

The code for PARITY's dynamics is

```
from simp import *
X=50
initialize(size=[X])         # 1D grid
# --------------------------------          SIGNAL DECLARATION
c = Signal(SmallUInt(2))       # binary state
# --------------------------------          TRANSITION FUNCTION
l,r = c[-1],c[1]     # declare the neighbor directions
def parity():
    c._ = l^c^r    # ^ denotes 'xor', sets bit if 'l+c+r' is odd.
parity = Rule(parity)
```

The code declares l and r to represent c[-1] and c[1] (left and right)—although c[-1] and c[1] could instead have been used in the transition function we wanted to demonstrate the use of named SignalRegion objects. The code below declares a space-time renderer, initializes the state to zero with a single one point in the middle, and creates a console.

```
# --------------------------------          RENDERING
declarecolors()
def bw():
    if not c: white._ = 255
bw_xt = XTRenderer(Rule(bw),grayscale,time=X/2          )
# --------------------------------          INITIALIZE
c[X/2]=1    # point seed in the center.
# --------------------------------          CONSOLE
ui = Console(bw_xt,mag=8)      # set the renderer and the default magnification
ui.bind('STEP',parity)      # Specifies the 1D renderer.
ui.start()
```

Rather than employing the usual renderer, we employ a XTRenderer to capture the space-time history of the CA. An XTRenderer object defines a special method called record for capturing the space-time diagram over a sequence of time steps. Every time the Console does a STEP, it looks for and automatically calls a renderer's record method if it's defined. The image in Fig. 1.4 depicts a space-time diagram generated by the console. Space runs horizontally while time runs vertically. The window of time recorded by the XTRenderer is set with the *time* parameter. In this diagram, time increases downwards. By using a negative value for *time* one can make time increase going upwards.

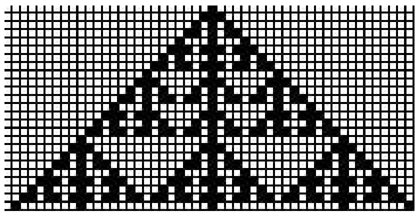


Figure 1.4: **1D Parity CA space-time diagram** Initalized with single point in the center and updated 24 times.

In the example, the magnification was set to 8 using the *mag* parameter of the Console. Grid lines were drawn because the magnification was high. (To turn grid lines off, use the *showgrid* parameter.) The image was collected interactively using the CaptureView command (c), but could have ben captured using the script discussed below.

1.4.2 Using a script to record the history

Instead of using the console, one might capture the image of Fig. 1.4 using a script

```
bw_xt.record()       # Record the initial state
for i in xrange(24):    # Do 24 updates
   parity()   # call the rule (does a step of the dynamics)
   bw_xt.record()    # record the state
arr = bw_xt()    # get the output array
rescaled_arr    = magnify2d(arr,scaling=8,grid=1)        # magnify with grid lines
open("out.ppm","wb").write(array_to_ppm(          rescal ed_arr )) # output image file
```

First it records the initial state, then it runs the dynamics 24 times, calling the record method of the XTRenderer after each update. Next, using magnify2d, it gets the output array, magnifies it, and adds grid lines. Finally, it converts the array to a '.ppm' string and saves it to the file, 'out.ppm'.

1.5 A simple 1D lattice-gas

Using a simple 1D example, this section introduces the concept a lattice-gas and related SIMP programming constructs. To motivate this discussion, suppose we want a one-dimensional dynamics that supports particles moving with unit inertia. At each time step, we want particles to move left and right with unit velocity depending on their inertia.

One way to program such a dynamics is as a CA. One could declare a Signal to encode the presence of a particle moving to the left, to the right, and the absence of particles. To do updates, one might use a Rule that looks to the left and right to decide whether a particle moves into its cell. (If there is a right-moving particle on the left, it sets its next-state value to indicate the presence of a right-moving particle and so on.) To handle the case where there is both a left moving and right moving pariticle in the cell, Signal needs a total of four states.

This CA would have a Signal with four states and Rule that looks at the left and right neighbors with a total of six bits of inputs. Because the number and size of inputs has an effect on the performance of a Rule, we are motivated to do better if possible. It turns out that we can write a simpler, more efficient program by expressing the dynamics as a lattice-gas automaton. In a lattice-gas data movement is expressed directly with a special data transport operation that shifts data without the need to evaluate a transition function. Instead of packing particle state into a single signal, under the lattice-gas approach we make two signals—one to encode the presence of a particle moving in each of the two directions—and transport them to neighboring sites by shifting them left and right independently. Assuming the one-dimensional *size* was already initialized, the following code snipet completely specifies this dynamics.

```
l = Signal(SmallUInt(2))
r = Signal(SmallUInt(2))
dynamics   = Shift(kvdict(l=[-1],r=[1]))        # make  l move  left  and  r move  right
```

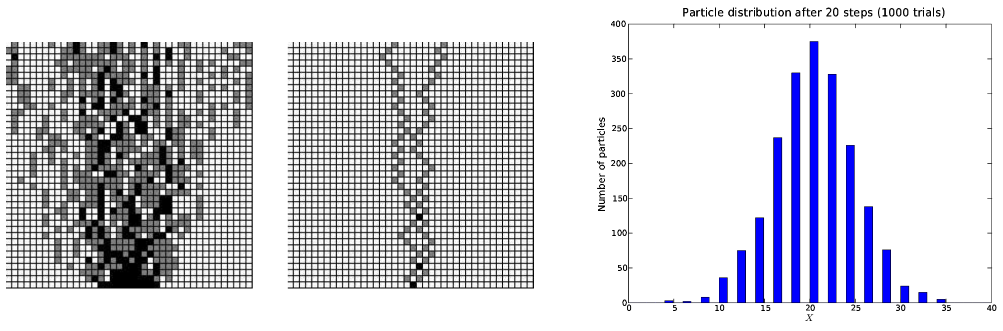
In SIMP, data transport is described by a `Shift` object. The `Shift` constructor expects a dictionary mapping `Signal` objects to vectors giving shift amounts. `kvdict` is a SIMP helper function for constructing dictionaries that are keyed on values (hence the '`kv`' prefix). An equivalent declaration without `kvdict` is '`dynamics = Shift({L:[-1],R:[1]})`'. Expressed in this way, the program need only operate on two (rather than six) bits and no transition rule is required—data is only shifted. The shift is a data-blind operation that—in its simplest implementation—requires only updating registers holding the current offsets of the signals.

This dynamics is not very interesting in and of itself. We can make it more interesting by introducting an interaction rule—in the parlance of LG an interaction rule affects particle states. A simple dynamics is random-swap based diffusion. Particles move with unit velocity, but may change velocity randomly. Assuming the random variable `coin` is declared, initialized and stirred using the method discussed in Section 1.3, the code for such a dynamics is

```
def  swap_randomly():     # change  inertia  if 'coin'  is 1
   if  coin:  l._  = r;  r._  = l
dynamics   = Sequence([Shift(kvdict(l=[-1],r=[1])),Sti          r([coi  n]),
                       Rule(swap_randomly)])
```

In Fig. 1.5 (a) and (b) we render this dynamics with the time axis going up (this axis direction is specified by passing a negative *time* parameter to the `XTRenderer`). Fig. 1.5 (c) shows the histogram of final particle positions after 20 iterations when (b) is repeated 1000 times. The histogram is a decent appoximation of a Gaussian, which is what one would expect for a diffusive random walk.

Note: We used the `matplotlib` (http://matplotlib.sourceforge.net/) in a SIMP script to generate Fig. 1.5 (c). The source code for generating it is included in XXX. Basically, after initializing with two centered, isolated particles, a loop iterates the dynamics a specified number of times and then `numarray.nonzero` is called on `l.array()` and `r.array()` get the positions of the non-zero (particle bearing) elements and `matplotlib.pylab.hist` is called to create a histogram plot from a combined list of the particle positions gathered over the trials.



(a) (b) (c)

Figure 1.5: **Simple 1D Diffusion LG** (a) shows a space-time diagram of a block of particles strobed every fourth iteration and (b) shows two particles doing a random walk. In these diagrams space is horizontal, time increases upwards, and signals are rendered as the grayscale of the number of particles at a site—white means no particles, gray means one and black means two. (c) shows a histogram of the final positions of a two-particle random walk after 20 steps when the experiment is repeated 1000 times.

1.5.1 Interleaved, non-communicating sublattices

Notice, that there are gaps in the histogram of Fig. 1.5 (c). This is because odd positions never have any particles. Why? The reason is simple, but not obvious[6]. Particles move left and right in units of 1 and alternate between odd and even sites as they move. A particle that started on an even site will be on even sites at even times and odd sites at odd times and *vice-versa*. This can be seen by closely examining the walks of the two isolated particles of Fig. 1.5 (b). The situation is shown more clearly in Fig. 1.6.

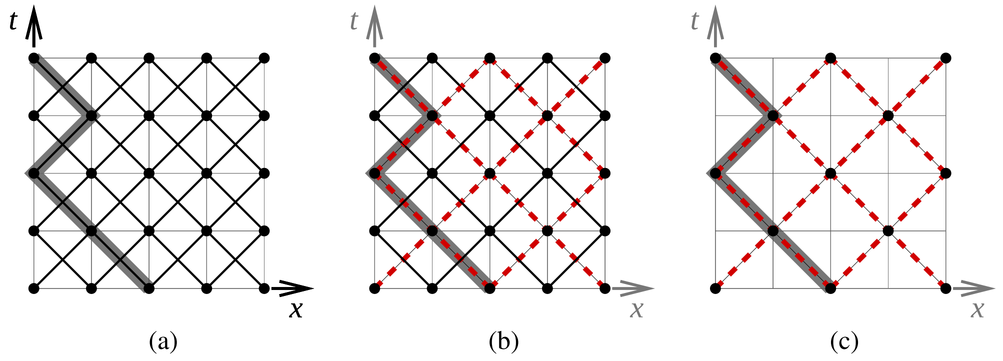


Figure 1.6: **The diffusion LG has two non-communicating sublattices** (a) shows the space-time signal transport and interaction structure (it's a space-time crystal) of the basic diffusion LG. The trajectory of a sample particle that moves left twice, switches tracks to move right and then switches tracks again to move left is shown in gray. As time increases, particles move left and right along the black arcs that indicate the trajectories of `l` and `r` and land on grid sites (intersections of gray lines) at successive time steps. The black nodes mark the positions where the `swap _randomly` rule is applied and particles may change trajectories. Using thick red dashed lines and thin black lines (b) shows the two interleaved, non-communicating sublattices of the dynamics. A particle that starts on one of these sublattices can't move onto the other. (c) shows the red highlighted sublattice by itself. Notice that (c) is essentially a scaled version (a) in which there are interaction nodes every time that signals cross paths rather than every-other time. (`coin` is not shown on these diagrams or those that follow because it is not a true state variable—it merely serves to randomize the transition function.)

Rather than making a dynamics with two non-interacting lattices one can use SIMP to program a dynamics on the sublattice of Fig. 1.6 (c). We cover this in depth in the next section.

1.6 Defining the 1D diffusion lattice-gas on a sublattice

In SIMP one can define `Signal` and `Rule` objects on sublattices of the integer grid and thereby construct systems whose space-time crystal is a sublattice of the grid. As an example, we show how to do this for 1D diffusion on the space-time crystal of Fig. 1.6 (c). First we'll show how to declare `Signal` and `Rule` objects on a sublattice. Then we'll discuss the implications that using a sublattice has for SIMP's indexing, slicing, and shifting conventions. Finally, we'll examine a few options for rendering.

`Signal` and `Rule` objects are both defined on lattices. The lattice of a `Signal` defines the set of points of where its data elements are allocated. The lattice of a `Rule` defines the points where the local transition function is evaluated. In terms of the software engineering, both the `Signal` and `Rule` classes inherit from the base `LatticeArray` class. By default `LatticeArray` objects define lattices that coincide with all points of the the grid. To override this default behavior one specifies an alternate lattice by giving its *generator matrix*. The generator matrix is an ordered set of vector strides that, in each dimension, generates the lattice points through repeated additions and subtractions.

In the present example we need a 1D lattice that has strides of 2 in X. The corresponding generator matrix is `[[2]]` . In higher dimensions, the generator matrices have more structure, but we postpone this discussion until Section 1.7.

[6]Indeed, it took some time before the non-communicating sublattices of the HPP lattice gas were discovered.

We recode the previous example, specifying the generator matrix using the *generator* parameter

```
l = Signal(SmallUInt(2),generator=[[2]])
r = Signal(SmallUInt(2),generator=[[2]])
coin  = Signal(SmallUInt(2),generator=[[2]])
def  swap_randomly():     # change   inertia   if  'coin'   is  1
   if  coin:   l._  = r;  r._  = l
dynamics   = Sequence([Shift(kvdict(l=[-1],r=[1])),Sti          r([coi  n]),
                     Rule(swap_randomly,generator=[[2]])])
```

Each of the `Signal` and `Rule` declarations now have a generator matrix of `[[2]]` . The same generator argument is used four times in the code above. To eliminate such redundancy in the common case that many objects have the same generator, `initialize` has its own *generator* argument. Setting it overrides the default generator for `LatticeArray` objects. For example, '`initialize(size=[40],generator=[[2 ]]`' sets the default generator to `[[2]]` so that it need not be passed to every `LatticeArray` constructor[7].

1.6.1 Implications: Lattice positions, coordinates, sizes and rounding up

We have defined the dynamics for the sublattice version of 1D diffusion, however there are a number of issues relating to the sublattice yet to consider. First we consider the starting position of a lattice.

Lattice starting positions

The STEP keeps track of the starting positions of `LatticeArray` objects. The starting positon is the position of the lattice site that's closest to the origin. As shown in Fig. 1.7 (a) the starting position of the signals varies with time. At time 0, the lattice starts at position `[0]` , at time 1, it starts at `[1]` and at time 2 it returns to `[0]` —the starting position is always smaller than 2—the lattice spacing.

The current position may be queried by calling a `LatticeArray` object's `getposition` method and modified using `setposition` . The `Shift` operation shifts a lattice and may change its starting position, one can use the `SetPosition` operation to explicitly set the starting position. As explained below, the starting position plays an important rule in subscripting signals.

Coordinates round up to the nearest lattice site

Having sparse sublattices raises the question of what to do when a coordinate subscript of a signal does not actually hit a lattice site. In our example, at time 0 the lattice of the signal `l` starts at `[0]` . It is clear that making the assignment '`l[0]=1` ' sets the first element in the lattice, but what happens if one writes '`l[1]=1` '? The answer is simple. Coordinates round up to the nearest lattice site. Therefore, the value at `l[2]` is the one that's actually written. Fig. 1.7 (a) demonstrates this behavior for the coordinate `[2]` .

In SIMP, the policy is to always *round up* to the nearest lattice site. Let's discuss how the program declared above. In the specification of the dynamics, the signals move left and right as shown in Fig. 1.7 (b). However, the rule is actually applied at the sites marked by circles in (c); this is because, unlike the signals, the rule is not shifted between steps and therefore remains stationary. Nevertheless, the result is equivalent to (b), because, when the transition function accesses a signal, its coordinates are automatically rounded up to the nearest lattice. That the two versions are equivalent can be seen in Fig. 1.7 (d) which uses rounded boxes to group grid coordinates that round up to the same site of the lattices of *l* and *r*. Because the rule accesses the signals at an offset of zero[8], the boxes also group the function nodes with the signals that they affect. One can easily verify that both the shifted and unshifted rule lattices

[7]Setting the generator also gives the STEP implementation a hint about the preferred lattice. The STEP may employ this information when making data allocation decisions.

[7]Recall that the default offset for a signal is zero.

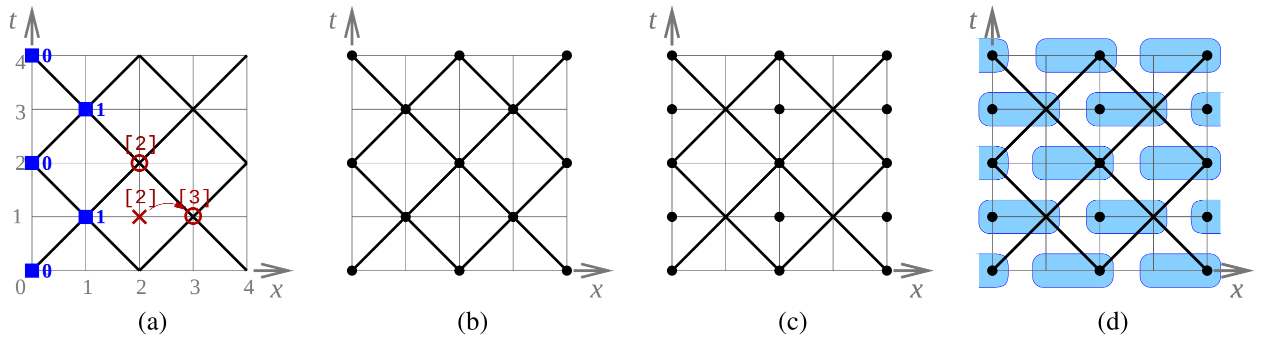


Figure 1.7: **Coordinates and idexing on the sublattice** Figure (a) labels the alternating starting positions of the signals (blue squares) and shows the site that the coordinate `[2]` maps to at two different times when the lattice starts at different positions. (b) shows the desired space-time crystal while (c) shows the one described by our code. In it, the position of the rule's lattice where `swap _randomly` is applied (marked by dots) does not shift. Half of the time, the rule's lattice does not coencide with the signal lattices. However, because coordinates are rounded up, the result is the same as (b). This can be seen in (d) where rounded boxes enclose the groups of coordinates that round up to the same signal lattice site. The rule—which accesses signals at an offset of 0—operates on the same sets of signals whether it is shifted or not.

affect the same set of signals. Although the rule's position does not matter in this particular case, it does matter in certain situations as we'll see when we discuss rendering in Section 1.6.2

Subscripting with slices

SIMP also has some special conventions when it comes slicing a `LatticeArray` . Clearly, `r[1:5].array()` returns an array of 2 elements because that's how many elements fall within the range. At even times it returns the elements at `r[2]` and `r[4]` and on odd those at `r[1]` and `r[3]` . In general, one should make sure that the extent of a slice is a multiple of the lattice spacing in order to avoid the ambiguity and unpredictablity of a call like `r[1:6]` which, taken at face value would return different numbers of elements depending on when it's called—it would return two elements (`r[2]` ,`r[4]`) on even times and three elements (`r[1]` ,`r[3]` ,`r[5]`) on odd times. To eliminate this ambiguity, SIMP takes the policy of always rounding the extent up if it is not a multiple of the lattice spacing, therefore `r[1:6]` would automatically be rounded up to `r[1:7]`

Wrap-around compatible sizes

When a `LatticeArray` object is declared, SIMP also checks to ensure that it wraps around properly on the grid. One can not, for example, have a grid with an odd size, say 11, and then declare signals with a generator of `[[2]]` because there would be a remainder of 1 mod 2—if the starting position were 0, 12 would wrap around to 1 and *begin a new interleaved lattice* on the odd sites. When one trys to declare a `LatticeArray` that does not wrap properly, `simp` raises a `StepError` with a suggestion for a *size* that is compatible with the lattice generator.

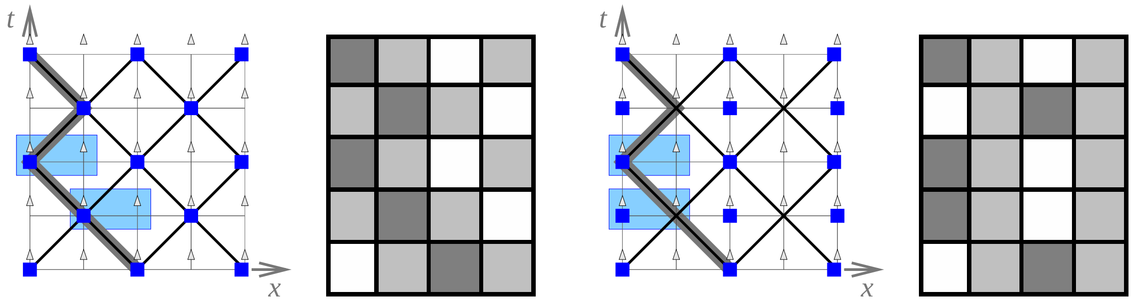
1.6.2 Rendering

There are a few different ways to render a sparse sublattice signal. The first decision is what kind of lattice to render onto. One controls this with the choice of lattice generator for the color `OutputSignal` objects. This can be declared using the `generator` argument of an `OutputSignal` or the same argument of `declarecolors` . First, we'll declare the colors on the grid using a generator of `[[1]]` and a rendering function that maps the sublattice signals to two grayscale outputs—`white[0]` and `white[1]` .

To make the rendered result reflect the alternating odd/even sublattice that the signals are on we'll be making a rendering rule that shades the pixels on the unused sublattice light gray and shades the pixels on the used sublattice white if there are no particles, gray if there is one, and black if there are two. To do this we need a rendering rule that has a spacing of 2. (In general, SIMP requires that a rule's lattice be a sublattice of all of the signals it acts upon.) Based upon the number of particles present, the rendering rule will set the value of `white[0]` (the site on the used sublattice) to white, gray or black. The value of `white[1]` (the site on the unused sublattice) is always set to light gray. To make the rendering rule's lattice coencide with alternating locations of the signals we'll shift the rule. The space-time diagram and results of this rendering stratetgy are shown in Fig. 1.8 (a) and (b) and the code appears below.

```
declarecolors(generator=[[1]])          # declare colors on the sublattice
def bw():   # rendering function
    white[0]._  = (2-(l+r))*127     # render the value where the rule is evaluated
    white[1]._  = 192  # shade the site to the right light gray
bw_rule  = Rule(bw,generator=[[2]])        # render on same sublattice as the signals
bw_rend  = XTRenderer(bw_rule,grayscale,time=-5      ) # space-time  rendering,  T going up
dynamics  = Sequence([dynamics,Shift({bw_rule:[1]})])         # render where signals are
```

The statement '`Sequence([dynamics,Shift(bw_rule:[1])])`' augments the dynamics so that it shifts the rendering rule in the same way that the signals are shifted. If we didn't do this, the rendering would appear like Fig. 1.8 (c),(d).



(a) (b) (c) (d)

Figure 1.8: **Rendering space-time diagrams and results** (a) and (c) show two different space-time structures for rendering and (b) and (c) show the their results. In (a) and (c) triangles denote the `white` output signal, the blue rectangles denote the locations where the rendering rule is applied, and wide gray line shows an example trajectory of a single particle. The size of the grid is [4] (the coordinates are 0-3; 4 wraps so it is the same as 0) and we keep a history of 5 states. (a) and (b) show the space-time structure and results using a rendering rule that sets the grayscale value of `white[0]` based upon the number of particles present and always shades the value of `white[1]` gray. (c) and (d) show the same rendering rule and the unsatisfactory result when the rule's lattice (the rectangles) is not shifted with the signals (diagonal lines).

Instead of rendering in grayscale, one could render in color. One may also use the `OutputSignal` offsets in different ways. In the example below (Fig. 1.9 (a) (b)) we use `red[0]` and `blue[1]` to denote `l` and `r`.

```
def rgb_block():
    if l: red[0]._=255;    # pixel [0] => red
    else: red[0]._  = green[0]._  = blue[0]._  = 255  # pixel [0] => white
    if r: blue[1]._  = 255;  # pixel [1] => blue
    else: red[1]._  = green[1]._  = blue[1]._  = 230  # pixel [1] => white
```

Finally, one need not declare the colors on the grid. One could, for example, declare them on the same sparse lattice as `l` and `r` and make a rendering rule that does not render sites on the unused sublattice at all. Fig. 1.9 (c) and (d) show the space-time diagram and the results for the code below.

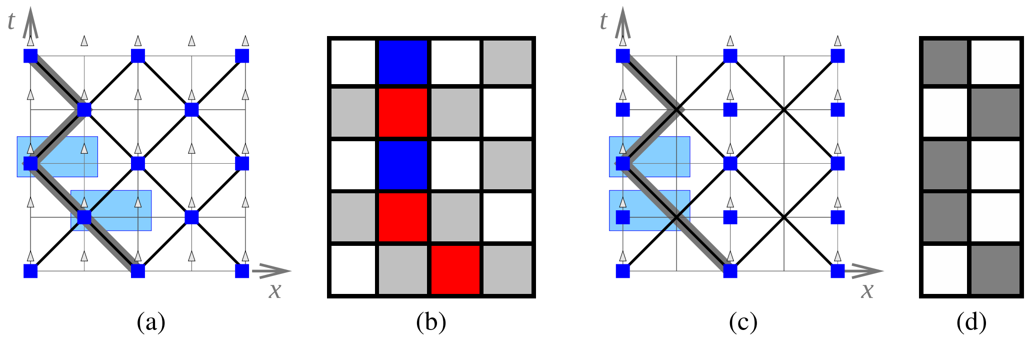


Figure 1.9: **Two more rendering options for 1D diffusion** This figure is similar to Fig. 1.8, but depicts two more rendering options. (a) and (b) show the space-time diagram and results for a rule called `rgb _block` that renders `l` and `r` to `red[0]` and `blue[1]` . (c) and (d) show grayscale rendering with output signals that are on the same sparse sublattice (spacing 2) of the grid that the signals are on.

```
declarecolors(generator=[[2]])          # declare  colors  on  the  sublattice
def  bw():
    white[0]._   = (2-(l+r))*127
bw_rule   = Rule(bw,generator=[[2]])
bw_rend   = XTRenderer(bw_rule,grayscale,time=-5          )
```

This strategy presents a compressed view that does not take the position of the signals into account, but is useful when one would like to economize the number of pixels in the display.

1.7 HPP, a 2D Lattice Gas

We now present and program the HPP lattice gas[9]. It is a simple particle-level model of a gas dynamics in which particles move at unit velocity on one of four 'tracks' between lattice sites and scatter at right angles when they collide head-on. The dynamics of HPP is detailed in Fig. 1.10.

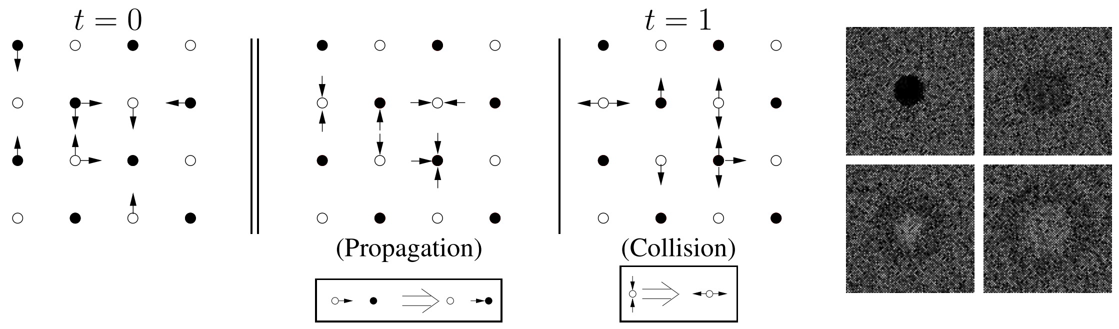


Figure 1.10: **HPP Dynamics** The HPP lattice gas is defined on a two-dimensional lattice. Up to four particles—one ready to move in each of four directions as indicated by the arrows in our example—may occupy each site. As labeled in the middle and right diagrams, an update occurs in two phases—*propagation* (data-transport) during which the state variables move to adjacent sites and *collision* (data-interaction) during which the collision rule is applied. On the right, we show the evolution of the dynamics starting with particles moving in random directions and a 'vacuum' with no particles in the center.

The dynamics in the figure is straightforward; but, on close inspection, one comes to the realization that it results

[9]HPP is named after Hardy, de Pazzis and Pomeau who first presented it in (Hardy et al.,).

in two interleaved, 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. 1.10. 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.

1.7.1 HPP sublattice generator matrix

To program HPP we will use a sublattice generator for the signals and the rule. The appropriate sublattice is one that has a generator vector of $\mathsf{g}_2 = (0, 2)$ in X and thus skips every other element, and a generator vector $g_1 = (1, 1)$ in Y and thus has a skew of 1 in the X direction. The generator vectors are shown graphically in Fig. 1.11(a).

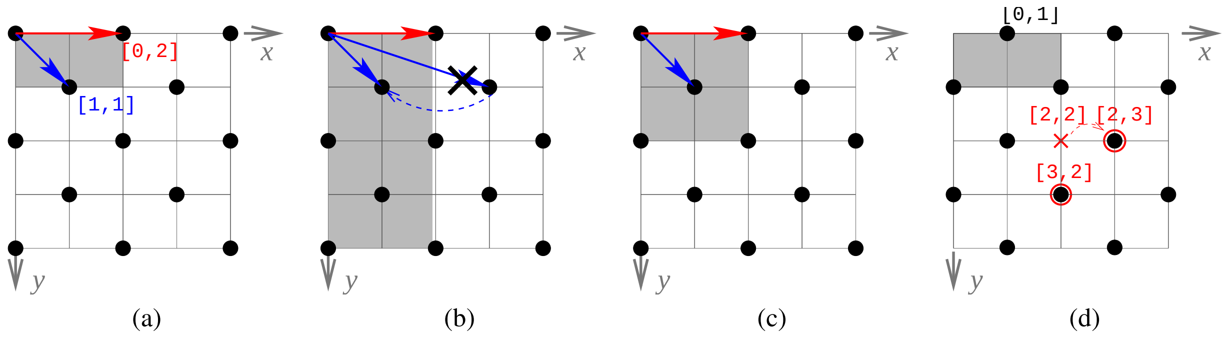


Figure 1.11: **HPP Sublattice** We allocate the HPP rule and its signals on a sublattice of the grid with generators $\mathsf{g}_1 = (1, 1)$, $\mathsf{g}_2 = (0, 2)$. The generators are shown with labeled arrows in (a) and the rectangular unit cell, given by the spacing of the lattice $(1, 2)$, is shaded. (b) gives an equivalent generator $g_1 = (1, 1) \equiv (1, 3)$. Although this generator is equivalent, its skew, 3, is larger than the X dimension of g_2 (the X generator) therefore it can not be used in SIMP. (c) shades the lattice's least common rectangle . The least-common rectangle is the smallest rectangle that's a multiple of the rectangular unit cell and has sites at each of its corners. If the size is a multiple of the least-commmon rectangle, the lattice wraps around to itself, otherwise it does not and the size/lattice combination is invalid. (d) shows the lattice when it's starting position is $(0, 1)$. The starting position of a lattice is the coordinate of the lattice site inside of the rectangular unit cell. (d) also demonstrates how the coordinates $(2, 2)$ and $(3, 2)$ round up on this lattice.

The two generator vectors form the rows of the generator matrix. The generator matrix for the lattice is $\mathsf{G} = \begin{bmatrix} \mathsf{g}_1 \\ \mathsf{g}_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}$. We use a matrix in this form for technical reasons. Mathematically, an n dimensional lattice Λ is defined as the set of points generated by multiplying the generator vectors g_i by integers a_i,

$$\Lambda = \{\sum_{i=1}^{n} a_i \mathsf{g}_i : a_i \in \mathbb{Z}\},$$

or in the matrix formulation by multiplying the generator matrix G by an integer (row) vector a,

$$\Lambda = \{\mathsf{aG} : \mathsf{a} \in \mathbb{Z}^n\}.$$

Notice that we have the given generator matrix as an upper triangular matrix. Mathematically speaking, a generator matrix need not be upper-triangular, however, for SIMP it must be. An advantage of the upper triangular form is that the diagonal of generator matrix gives the *spacing* of the lattice in the Y and X dimensions while the off diagonal upper elements give the skew. In our example, the spacing is $(1, 2)$ and the skew is 1.

One reason that a lattice's spacing is important is that it defines the lattice's *rectangular unit cell*. The signal slice extents are rounded up to the next multiple of the spacing. For example, with a HPP's spacing, $(1, 2)$, `sig[0:3,0:5]` becomes `sig[0:3,0:6]` . Coordinate indexes are also rounded up in the same way—when the lattice starts at `[0,0]` , `sig[1,2]` references the site `sig[1,3]` . Fig. 1.11 gives more examples.

SIMP also requires that all generator matrix elements be positive integers and that the skew elements be less than the spacing of the dimension they skew. For example, although a Y generator of $\mathtt{g}_1 = (1, 3)$ would generate HPP's lattice, it does not satisfy the skew constraint because the skew in X is 3, which is greater than the X spacing, 2. However, an equivalent choice that satisfies our constraints is $1 \equiv 3 \pmod 2$. Fig. 1.11 (b) shows this graphically.

In all, SIMP restricts the generator matrix to an integer upper-triangular form having strictly positive diagonal (spacing) elements and positive upper (skew) elements that are smaller than the diagonal element of their column (smaller than the spacing of the dimension they skew). SIMP requires this form because it is mathematically convenient (computations on upper triangular matrices are simpler), expressive (the spacing and skew in an arbitrary generator matrix are not obvious), and yields lattices that, other than their skew, map naturally into multidimensional memory arrays.

This restriction does not limit the types of integer lattices that may be expresed. Mathematically speaking, SIMP requires that the generator matrix be in an integer Hermite Normal Form (iHNF). It is a theorem of linear algebra that any integer generator matrix can be converted to iHNF using unimodular transformations and a uniform scaling. Geometrically, this corresponds to successively rotating the lattice until its generators are in alignment with the Cartesian coordinate axes (X, then Y, then Z ...), selecting generator vectors having skew elements smaller than the dimension they skew, and applying a uniform scaling so the generators hit integer grid sites. iHNF is not only general, but is a natural way to express a lattice's generators.

For a given generator matrix, there is also a restriction on the allowable sizes. This is necessary for the lattice to wrap-around properly. In particular, the size must be a multiple of what we call the lattice's *least common rectangle*. The *least-common rectangle* is the smallest rectangle (in two or higher dimensions) that has a lattice point on all of its corners. If the size were not a multiple of the least-common rectangle, points of the lattice would not wrap-around to themselves. In our example, the least common rectangle is $(2, 2)$ and is shaded in Fig. 1.11 (c). For it, a grid size of $(4, 4)$ works, but a size of $(4, 3)$ does not. `simp` raises a `StepError` with a new suggestion for *size* if one tries to declare a generator that has a least-common rectangle of which the size is not a multiple.

In our examples we have seen that in going from one dimension to two, expressing a lattice becomes more complicated in that one must use the integer HNF for the generator matrix, the lattice may have skew components, and the size must be a multiple of the least-common rectangle. In three and higher dimensions, the concepts from two dimensions generalize directly.

[XXX perhaps show the other HNF generator matrix examples here]

1.7.2 Programming HPP

Having defined the proper generator matrix for HPP—`[[1,1],[0,2]]`—programming it is now relatively straightforward. The code below sets the default generator using `initialize` and declares four signals—one for each direction as shown in Fig. 1.13 (a).

```
from simp import *
initialize(size=[200,200],generator=[[1,        1],[0, 2]])
p0,p1,p2,p3  = map(Signal,[SmallUInt(2)]*4)      # make 4 binary  signals  for particles
```

In the HPP dynamics, particles continue with unit velocity unless two collide head-on, in which case particle (and hole) trajectories are rotated by $90°$ as shown in Fig. 1.10.

```
#--------------------------------          COLLISION
def  hpp():    # scatter   at  right   angles   on  collision
    if  (  (p0==p2)   and  (p1==p3)   ):
        p0._  = p3;     p1._  = p0        # rotate   by  90  degrees
        p3._  = p2;     p2._  = p1

dynamics   = Sequence([
          Shift(kvdict(p0=[      1,  0],
              p1=[  0,  1],        p3=[  0,-1],
                     p2=[-1,   0])),
        Rule(hpp)])
```

We render to the grid and color the unused lattice sites black and set the white intensity of sites that could have particles to a value that denotes the number of particles present (maximum occupation of 4). In the code below, we declare the `white` output signal directly rather than using `declarecolors`.

```
white   = OutputSignal(UInt8,generator=[1,1])
def  intensity():
    white[0,0]._    = (p0+p1+p2+p3)*255/4
    white[0,1]._    = 0
render_rule    = Rule(intensity)
rend  = Renderer(render_rule,outputs=(white,))
dynamics   = Sequence(dynamics,Shift({render_rule:[0,1           ]}))
```

The offsets $(0,0)$ and $(0,1)$ are two different *cosets* of `white` with respect to the rendering rule's lattice. A coset is the position of the offsets modulo the rendering rule's lattice. [XXX note the restriction on writing values out to cosets]

Rather than coloring the unused coset position `white[0,1]` black, one could instead output the particle count as in

```
def  intensity():
    white[0,0]._    = (p0+p1+p2+p3)*255/4
    white[0,1]._    = (p0+p1+p2+p3)*255/4
```

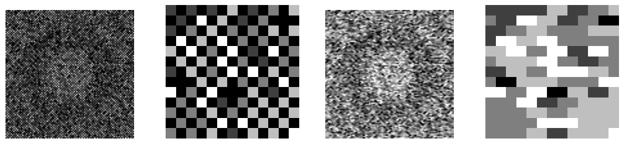
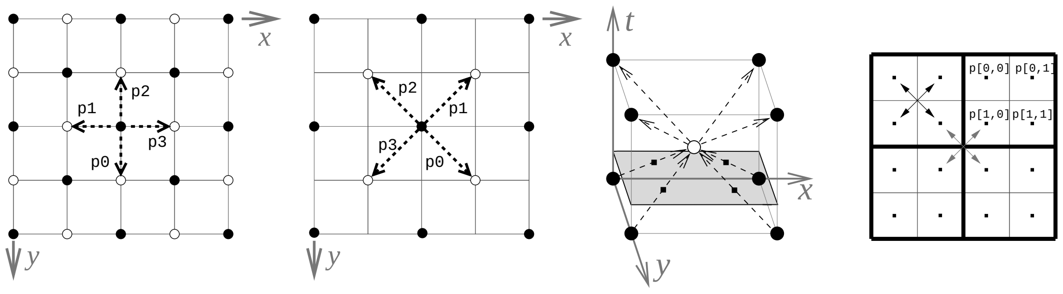


Figure 1.12: **Two HPP rendering options** (a) shows a HPP configuration when we only render the state to one coset (`white[0,0]`) and color the other coset `white[1,0]` black. (b) shows a closeup zoom of the center of the image. (c) shows the same configuration when we render the state to both cosets. (d) shows a closeup zoom of the center of the image.

Again, we shift the rendering rule so that it's sublattice starts at the same place as the signals. To set the initial conditions, we randomize each signal and then use an ellipsoidal mask created with *ellipsemask* to clear the values and make a circle in the center of the space.

```
for  sig  in  [p0,p1,p2,p3]:      # for  each  signal
       sig[:,:]   = makedistribution(sig.shape,[1,1])         # randomize   the  entire   state
       ellipse_region     = sig[200*3/8:200*5/8,200*3/8:200*5/8]          # select   ellipse   rect
       arr  = ellipse_region.array()       # get  the  current   value  for  the  ellipse
       numarray.putmask(arr,ellipsemask(ellipse          _regio  n),0)  # clear   an  ellipse
       ellipse_region._     = arr  # set  the  values   in  the  region
```



(a) (b) (c) (d)

Figure 1.13: **Mapping HPP from a LGA to a BPCA** (a) depicts the lattice and signal transport vectors for our original formulation of HPP while (b) shows the same lattice rotated by 45° and scaled by $\sqrt{2}$. As shown by (c) and (d), the lattice-gas geometry of (b) is the basis for programming HPP as a Margolus neighborhood block partitioning cellular automaton (BPCA). (c) depicts the unit cell of HPP's 3D space-time crystal when it's programmed using the geometry of (b). The shaded plane intersects the signals half-way between interaction nodes. We can use these intersection points to give the signals a spatial representation. (d) uses small rectangles to show a splayed spatial representation of the signals at the points that they intersect plane. In this representation, the original signals `p0`, `p1`, `p2`, and `p3` are now represented by cosets `p[0,0]` , `p[0,1]` , `p[1,1]` , and `p[1,0]` of a single signal `p` that's allocated on the grid. The thick and thin lines partition the signals into blocks of 4 that are updated together by the lattice gas. In the BPCA version, the blocks of four are updated together using a rule that has a lattice spacing of $(2, 2)$. On odd phases, blocks partitioned by the thick lines are updated and on even blocks partitioned by the thin lines are updated. The rule moves particles horizontally and vertically by swapping diagonally as shown by the arrows.

1.8 HPP programmed as a Margolus neighborhood block partitioning CA

We'll program the same dynamics using the block partitioning CA (BPCA) approach. A block partitioning cellular automaton is like an ordinary cellular automaton except that it updates blocks of state concurrently. In SIMP one programs a BPCA by giving the lattice that defines the block and writing a rule that updates the block.

Probably the most well-known way to program the HPP lattice gas is as a BPCA. The strategy is due to Margolus (Toffoli and Margolus, 1987). To arrive at this formulation imagine rotating the HPP lattice gas by 45° and scaling it's lattice to a grid with a lattice of $(2, 2)$. At this point, one may program HPP as above on a lattice whose generator is $\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$ and with `p0`, `p1`, `p2`, and `p3` shifted by $(1, 1)$, $(1, -1)$, $(-1, -1)$, $(-1, 1)$ as shown in Fig. 1.13 (b). The essential modifications to the program are

```
initialize(size=[200,200],generator=[[2,        0],[0,  2]])
p0,p1,p2,p3    = map(Signal,[SmallUInt(2)]*4)
....
dynamics   = Sequence([
             Shift(kvdict(p0=[     1,  1],  p1=[  1,-1],
                          p3=[-1,   1],  p2=[-1,-1])),
             Rule(hpp)])
```

This form of HPP can also be programmed as a BPCA. The transformation from the LG to a BPCA is described in Fig. 1.13. Rather than having four signals that shift, the BPCA approach uses a single signal that's updated in blocks of four using the *Margolus neighborhood*. Between updates, the blocks alternate between the two block partitionings shown by the thick and thin lines in Fig. 1.13 (d). Fig. 1.14 (a) and (b) shows the two phases of the BPCA version of the HPP rule from the standpoint of the signals and the rule's lattice. The code for the BPCA version of HPP appears below.

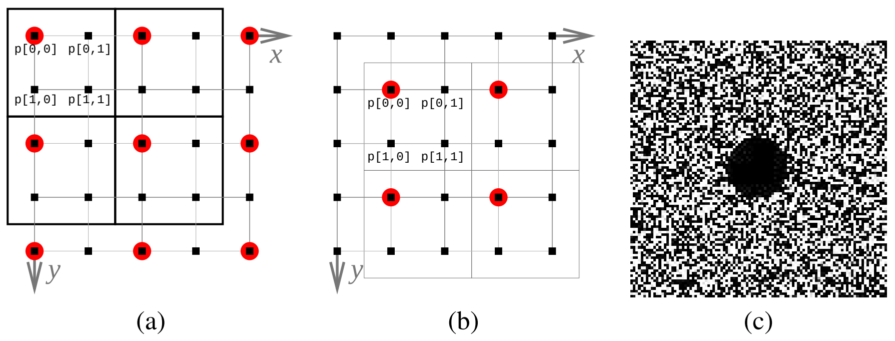


Figure 1.14: **The Margolus neighborhood HPP BPCA** (a) and (b) depict the lattices employed in the BPCA version of HPP and the partitionings imposed by the rule's lattice. The signal lattice is allocated on the grid and depicted using small squares while the rule's lattice has a spacing of $(2, 2)$ and is depicted using large red circles. (c) shows the rendering results when the signal cosets are rendered directly to an image.

```
from  simp  import  *
initialize(size=[Y,X])            # grid  size
p  = Signal(SmallUInt(2))         # signal  allocated  on the  grid
hpp_mesh   = [2,2]       # define  processing  on a  coarser  mesh---lattice   with
                         # spacing  of  (2,2)
def  hpp():
    if  (  (p[0,0]==p[1,1])    and  (p[0,1]==p[1,0])    ):   # head-on  collision
        p[0,0]._   = p[0,1];   p[0,1]._   = p[1,1]   # particles  rotate  90  degrees
        p[1,0]._   = p[0,0];   p[1,1]._   = p[1,0]
    else:                                            # keep  moving
        p[0,0]._   = p[1,1];   p[0,1]._   = p[1,0]   # particles  swap  diagonally
        p[1,0]._   = p[0,1];   p[1,1]._   = p[0,0]

hpp_rule   = Rule(hpp,generator=hpp_mesh)            # Declare  the  hpp_rule  on the  mesh
dynamics   = Sequence(hpp_rule,                      # Apply  the  rule  and
                  Shift({hpp_rule:[1,1]})      )     # shift  it to  next  block
```

Rendering becomes simpler using the partitioning CA. One can render at the level of particles by allocating the color output signals and the rendering rule directly on the same grid. The code used to render Fig. 1.14 (c) appears below.

```
white   = OutputSignal(UInt8)
def  bw():
  white._   = p*255
renderer    = Renderer(Rule(bw),outputs=(white,))
```

1.9 FHP Lattice Gas—a hexagonal lattice gas

HPP exibits some spurious symmetries that are not present in a real gas. The FHP lattice gas, which is defined on a hexagonal lattice correctly yields the Navier–Stokes equation in the macroscopic limit(Frisch et al., 1986). In this section we show how one may approximate a hexagonal lattice with SIMP. SIMP can not implement a true hexagonal lattice directly because a hexagonal lattice has rationals in it's generator matrix and SIMP requires the generator matrix elements to be rationals.

In a true hexagonal lattice, the generators are such that lattice sites are spaced at a distance of one from each-other and each site has six sites at a distance of one from it. The generator matrix is

$$\mathsf{G} = \begin{bmatrix} \frac{\sqrt{3}}{2} & \frac{1}{2} \\ 0 & 1 \end{bmatrix}$$

[XXX perhaps put a diagram here]

The best we can do is to approximate the matrix. One approximation is

$$\mathsf{G} \approx \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

but this approximation is poor because it incorrectly modifies both the skew and the spacing of the Y generator.

To get a better approximation, we can scale the generator by 2. With this scaling we have

$$2\mathsf{G} = \begin{bmatrix} \sqrt{3} & 1 \\ 0 & 2 \end{bmatrix}$$

which, when we round to the nearest integer is

$$2\mathsf{G} \approx \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}$$

This approximation expands Y by 13 percent, however the ratio of the skew to the X generator is correct. For rendering this is more important since the eye will more readily pick up skew than a slight scaling. To render onto the grid, $2 \times 2 = 4$ pixels are required.

Another approximation that preserves the ratio of the skew to its generator is

$$2\mathsf{G} \approx \begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}$$

This time Y skrinks by 73 percent, but only two pixels are needed.

The next best approximation with the skew remaining exact is not until 6 with $6\mathsf{G} \approx \begin{bmatrix} 5 & 3 \\ 0 & 6 \end{bmatrix}$. Y only shrinks by 4 percent, however $5 \times 6 = 30$ pixels are needed for rendering the approximation.

For simplicity, we program the dynamics here with the 'brick wall' approximation $2\mathsf{G} \approx \begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}$.

[XXX put code here[

1.10 A BPCA model of polymers undergoing thermal relaxation

We presented this model in a past (Toffoli and Bach, 2001) and using the BPCA capabilities of SIMP can now program it in a simple fashion.

1.11 Obtaining and using SIMP example code

The example code from this tutorial is available at http://pm.bu.edu. You may download it there. For convenience, we list the files included with the examples and offer a brief description of each.

- 'greenberg_hastings.py' 2D excitable medium
- 'stochastic_greenberg_hastings.py' A randomized (stochastic) excitable medium.
- 'fhp.py' The basic version of FHP described in Section 1.9.
- 'hpp.py' The basic version of HPP described in Section 1.7.
- 'hpp_pca.py' A partitioning cellular automaton version of HPP described in Section 1.8.
- 'hpp_rotated.py' A verion of HPP programmed as a lattice–gas that's rotated by $45°$
- 'parity1d.py' One dimensional version of parity.
- 'parity.py' Two dimensional version of parity.
- 'parity1d.py' One dimensional lattice gas that implements diffusing particles.
- 'parity.py' Two dimensional lattice gas that implements diffusing particles using the dimension splitting technique.
- 'parity.py' Two dimensional
- 'polymer.py' A simple model of a polymer chains undergoing thermal relaxation.
- 'life.py' Conway's game of life.
- 'ising.py' The Ising model of a spin system.
- 'difference.py' An parameterized rule example in which we make two copies of the same dynamics and run it different signals whose initial state varies slightly and compare results.

1.12 A parameterized rule

When a `Rule` is constructed, the values stobed into its transition function can be *parameterized* by passing a *namespace* dictionary overrides the bindings of global names. This is useful for making sets of rules that apply the same transition function to multiple signals.

1.13 Remarks

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 (Bach and Toffoli, 2003). 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 and Tom Toffoli for their insightful comments and DOE for supporting this work under grant 4097-5.

CHAPTER

TWO

Reference

`simp` is a user-level programming module that simplifies writing programmable matter experiments, instantiating run-time STEPs (space-time event processors) to implement them, and making and calling the primitive STEP operations (ops) that serve as a STEP's instructions. In this chapter we document the user-level functions and classes defined by `simp` and its sub-modules.

2.1 Importing and initializing SIMP

The `simp` module is meant to be imported directly with

```
from simp import *
```

Doing this imports the basic SIMP and STEP definitions. After importing `simp`, one must call first `initialize`. It creates and initializes a STEP and sets up defaults used for further SIMP constructs. Typically, one will only need to pass the *size* parameter to `initialize`. This parameter declares the number of dimensions and the extent of the coordinates. *size* is the only parameter without a default value.

When writing rules with the usual orthogonal (square grid) geometries, one need not worry about the optional geometric parameters. For lattice-gas, partitioned cellular automata, and non-orthogonal geometries (such as hexagonal lattices) one will need them.

The generator matrix is a $n \times n$ (n is the number of dimensions) upper triangular integer matrix with strictly positive integers on the diagonal and smaller, non-negative integer column elements specifying the default lattice for `a`nd signal objects. May also be given as a rectangular size vector (diagonal of a HNF) as described in **??**.

`initialize` (*size,generator=None, stepname=None,stepargs=None, verbose=1*)

The SIMP module initialization function sets up the module's global parameters.

Calling initialize instantiates a STEP. Therefore, initialize must be called before any `step` objects are instantiated or `step` methods are called. initialize can only be called once per module, raises an `Exception` if called twice.

Note: The parameters are stored in the `simp` module as private variables (eg. `size` is `simp.__size__`) and uses them first to load a STEP and later to construct default parameters for STEP ops and data types.

`size`

Vector giving the size of the grid. The length of the *size* vector sets the number of dimensions, n.

`generator`

The default generator (an n dimensional matrix or vector) for `LatticeArray` objects (eg. `Rule` and `Signal` objects). Defaults to identity—the generator used for an ordinary CA.

`stepname`

String giving the name of the step implementation module. Defaults to the default module for the installation or set the name set in the `simp` configuration file (see Section 2.1.1). Examples include `"reference"` and `"pc"`.

The module must either exist in `simp.stepmodules` or be in the Python system path (*sys.path*) and define the STEP interface as per Section 3.3. `initialize` raises an `ImportError` if the module can not be found or does not export the STEP interface.

stepargs
A keyword dictionary containing arguments for initializing the STEP. The nature of the arguments depends on the `step` module used. Defaults to no arguments.

verbose
Integer controling how much information `simp` and the step implementation module print. 0 prints nothing. 1 prints the standard information, informing the user when tables are being compiled *et cetera*. 2 prints more detailed information and suggestions. 3 and above print debugging information. Defaults to 1.

2.1.1 The simp confi guration fi le

You can create a configuration file that the `simp` module will read when it is imported. On UNIX systems it's '/home/username/.simp' and on modern Windows systems it's 'C:/Documents and Settings/username/.simp' or 'C:/.simp' if that path doesn't exist.

The configuration file is just an ordinary Python file. Think of it as extra code sourced at the beginning of `simp` module. Code in '.simp' can't override `simp` function definitions, however it may provide new definitions for your scripts. But, the primary use is setting initialization defaults. A full description of the initialization parameters appears in 2.1. Defaults that can be overridden include,

stepname
The name of the STEP implementation module.

verbose
Indicates what SIMP should print.

stepargs
Dictionary of optional arguments intended for the STEP implementation.

2.2 What the SIMP module does

`simp` initializes the STEP implementation, provides default parameters to the STEP operations, and provides the operations with a reference to the STEP implementation. The reference is needed so that the objects may register themselves with the STEP. These objects need a way to get a reference to the `simp` module. Usually they get this constructor arguments (through the `__simp__` attribute), however, a few of the objects require that `simp` be passed as an explicit parameter. In particular, `Signal` and `OutputSignal` require a `simp` instance as a constructor parameter in order to get default values. Because of this, `simp` provides wrapper functions for declaring `Signal` and `OutputSignal` objects.

Warning: `simp.Signal` and `simp.OutputSignal` are not classes. They are wrapper functions—the actual classes are `simp.step.Signal` and `simp.step.OutputSignal`. Therefore calling `isinstance(ob,Signal)` raises an error. Use `isinstance(ob,simp.step.Signal)` instead.

2.3 SIMP helper functions

The user might also use this function to get a reference to the module for accessing private variables when `simp` is imported with '`from simp import *`'.

simpmodule ()
Returns the current simp module.

2.3.1 dictionary functions

kwdict (***kwargs*)
Returns a keyword dictionary keyed on the keywords as strings. Allows one to construct string keyed dictionaries with statements like

```
>>> kwdict(a=1,b=3)
{"a":1,"b":2}
```

kvdict (***kwargs*)
Returns a dictionary keyed on objects and values.

Writing `kvdict(a=1,b=3)` is basically equivalent to `{a:1,b:2}` .

```
>>> a = 5
>>> b = "test"
>>> kvdict(a=1,b=3)
{5:1,"test":3}
```

The key values must be in the namespace, otherwise a `NameError` is raised. A `TypeError` is raised if an object is not a hashable dictionary key.

2.3.2 Constructing subscripts

The subscript (`subscr`) object is a syntax convenience object for constructing Python subscripts. Rather than using the `slice(start,stop,step)` constructor, as in

```
sl = [slice(1,4),slice[4,4]]
```

with the `subscr` object, one may write

```
sl = subscr[1:4,1:4]
```

This is useful for building slices objects for `simp` constructs that expect them.

2.3.3 Image array helpers

SIMP provides some helper functions for manipulating images in by `NumArray` objects.

The expected image format for a 2D image array is a `NumArray` of type `UInt8` . 0 is the lowest intensity and 255 is the highest. If the array is two dimensional, it represents a Y,X grayscale image. If the array is three dimensional, the least significant index is for the color chanel.

The simp helper function `array _to _ppm` can be used to create portable pixmap image strings (‘ppm’). One may save such a string to a file and thereby create a ‘ppm’ file.

array _to _ppm (*array*)
Return a ‘ppm’ image string from an array.

The function is suitable for rendering images to a file, as in

```
arr = rend()
ppm = array_to_ppm(arr)
open("out.ppm","wb").write(ppm)
```

magnify2d (*arr,magnification,grid=0,out=None*)

This function is used to magnify the two most significant dimensions of an image array, *arr*.

magnification is a non-zero integer scaling applied applied in the two most significant dimensions of *arr*. *grid* is the size of the grid lines separating blocks. The *out* array will receive the result. By default a new output array is created.

If *magnification* is negative, the output array is 'decimated' and only elements at strides of $-$*magnification* are kept.

If the shape of the input array is $(200, 100)$ and *magnification* is 3, and *grid* is 0, the output array will be of size $(600, 300)$ and each pixel in the original array will be expanded to 3×3 blocks 9 new pixels. If *grid* is 1, the blocks will be 2×2, there will be a spacing of 1 between each block and the output array size is $(601, 301)$. The extra space is for the grid line at the edge.

Normally the grid lines have an intensity of zero. To specify a different color for the grid lines, supply an output array that's already filled with the desired color.

2.3.4 Declaring colors

The outputs of a rendering rule are `UInt8` `OutputSignal` objects.

A user may declare them with something like

```
red,green,blue = map(OutputSignal,[UInt8]*3)
rgb = (red,green,blue)
```

Since this declaration is such a common one, `simp` provides a special method for making these declarations.

declarecolors (*generator=None*)

Convenience function for declaring a commonly used set of color signals as `UInt8` -type `OutputSignal` objects in the global namespace.

The *generator* is the generator matrix for these signals.

Basically, it acts as a macro for the following code which declares all of the commonly used color `OutputSignal` objects:

```
# declare all the color outputs that one might use
red,green,blue,white,alpha = map(OutputSignal,[UInt8,generator]*5)
grayscale = (white,white,white)  # convenience for grayscale RGB images
rgb = (red,green,blue)  # convenience for RGB images
rgba = (red,green,blue,alpha)  # convenience for RGB images with transparency
```

One of the output tuples is usually passed as an argument to a `Renderer` object.

Will raise a `NameError` if `red`, `green`, `blue`, `alpha`, `white`, `rgb`, `rgba`, or `grayscale` is already defined.

2.3.5 `NumArray` helper functions

`makedistribution` (*shape,dist*)

Return a new `numarray` with a given *shape* and distribution of values.

shape is a vector giving the shape of the array, while *dist* gives the ratios of the integer values to be generated.

For example

```
makedistribution((4,4),[1,3,5])
```

returns a 4×4 array in which elements have a $1/9$ chance of being 0, a $3/9$ chance of being 1 and a $5/9$ chance of being 2.

Use `SeedRandom` to seed the random number generator.

`getdistribution` (*arr,min,max*)

Return the distribution of values between min and max of an integer array.

The primary use is to get distributions of signal states.

arr is the array to be examined. *min* is the minimum value for the histogram while *max* is the non-inclusive maximum value of the range.

```
>>> arr = numarray.array([1,2,2,2,1,1,2,3,3,0])
>>> getdistribution(arr,0,3)
[1,3,4]
```

For efficiency, should only be called on small ranges of values.

`ellipsemask` (*shape*)

Return a 2D mask of values indicating the interior of an ellipse.

shape is the desired shape of the mask array.

The major axes of the ellipse are Y-1,X-1 where shape=Y,X. The goodness of the discrete ellipse approximation is dependent on the shape. Odd number sizes are typically more accurate than accurate than even.

The mask is suitable for use with the `numarray` `putmask(array,mask,values)` function.

Note: This method can currently only be used to create 2D ellipses. We hope to extend it to ellipses in arbitrary numbers of dimensions in the future.

2.4 Declaring and using `Signal` objects

In this section, we describe `Signal` constructors and semantics of `Signal` methods and member data. In particular, we describe how signal objects are declared and used, their possible data types, neighbor indexing and slicing (slice objects). `Signal` objects provide an abstract interface to parallel data allocations that ultimately exist and are managaded entirely inside of a STEP.

2.4.1 Declaring `Signal` objects

Declaring a signal entails specifying the lattice it's allocated on and the type of data it holds. Signals are indexed using multidimensional subscripts. Subscripting a signal implicitly calls `__getitem__` which returns a reference to a `SignalRegion` which can be used to read and write values in an absolute, global context and inside `Rule` declarations to access and set signal values in a relative, local context. Special `OutputSignal` objects declare output-only signals whose state information is not maintained in a STEP, but are rather used for reading output values rendered by local rules.

Both `Signal` and `Rule` objects subclass `LatticeArray` which is the base class for representing lattices.

class `LatticeArray` (*generator,size*)

Base class for representing lattice declarations on the grid.

generator is a HNF lattice generator matrix (see XXX) defined on a rectangular integer grid bounded in each dimension by *size*.

Read-only attributes

`generator`

The generator matrix for the lattice.

`spacing`

The orthogonal spacing between cell sites. (Diagonal of the `generator` .)

`nd`

Number of dimensions.

`shape`

shape of the array. Basically, its the `size` divided by the `spacing` .

lattice position functions

The following two functions read and write the starting position registers from the STEP.

`getposition` ()

Return the starting position of the lattice as described in Section **??**. (Wrapper for `GetPosition`)

`setposition` (*position*)

Set the starting *position* vector of the lattice as described in Section **??**. (Wrapper for `SetPosition`)

geometric helper functions

These are more likely to be more useful to STEP than to the user.

`array_index` (*coord,out=None*)

Return the rectangular array index of a grid *coord*.

Write to the vector *out* if it is specified.

`coset_coord` (*coord,out=None*)

Return the rectangular coset coordinate of a coordinate modulo the lattice.

`coord` (*index,out=None*)

Return the grid coordinate associated with the array *index* vector.

class `Signal` (*type,generator=None,simp=None*)

Declares a parallel data allocation of type '*type*' on the lattice declared by the *generator* matrix. If not specified, the generator taken from the *simp* module's default value.

This class extends `LatticeArray` and therefore has the same methods and attributes. (The size parameter of the `LatticeArray` is obtained from `simp`)

Note: The user need not specify the SIMP argument since `simp` provides a wrapper function for the `Signal` constructor that automatically specifies it.

Read-only members

`type`

the type of the signal.

Methods

`array` ()

Return the entire array for the signal as a `numarray` .

When called from a `SignalRegion` , as in '`sig[:,4:5].array()`' it returns the array for the signal at the slice. The type of the array depends on the signal type.

(Wrapper for the STEP `Read` operation)

`__getitem__`(*subscript*)
Returns a *SignalRegion* with a region specified by the *subscript*.

A `SignalRegion` can be used in transition functions and for reading and writing data.

`__setitem__`(*subscript,value*)
Assigns the *value* of a region specified by the *subscript*.

If the *value* is a scalar, all elements in the region are assigned to that value. Otherwise, if it is an array, elements are assigned to array values. Arrays sizes must match the shape of the region. (Wrapper for the STEP `Write` operation)

Back-end functions

These have lesser significance to the user.

`base_signal`()
Return a reference to self.

Only defined so that one can handle `Signal` and `SignalRegion` objects in a homogenous way.

`neighbor_offset`()
Returns an n dimensional zero vector.

Only defined so that one can handle `Signal` and `SignalRegion` objects in a homogenous way.

Signal Types

Currently, the primary type is the `SmallUInt`. This type is parameterized on the number of values it may take on. One must call the `SmallUint` constructor to instantiate the type.

class `SmallUInt`(*n=2*)
The STEP parameterized small unsigned integer type. It can take on n possible values from 0 to `n`.

Currently, the only[1] other type is `numarray.UInt8` and it may only be used for `OutputSignal` objects. It is used for RGB and grayscale rendered color output signals.

2.4.2 Reading and writing values and using slices

At the level of the STEP interface, a signal's data is managed by the STEP itself and can not be directly accessed. Instead, the data is read and written through `Read` and `Write` operations, and updated using `Rule` operations. For convenience, however, `SignalRegions` objects, which may be obtained by subscripting `Signal` objects can be used to represent the data in a region. A `SignalRegion` may also serve as a wrapper for `Read` and `Write` operations.

[XXX Need some slicing examples or to reference some from the tutorial.]

class `SignalRegion`(*signal,subscript*)
Obtained by subscripting a `Signal`.

It references a *signal* at a specific *subscript*. The *subscript* can be a Python subscript or a *Region* object. (see `i_slice` and `Region`).

If the subscript is a single point, the value can be coerced to an `int` or array. Otherwise, it can only coerce to an array.

To write to a *SignalRegion*, one can assign the value of the output attribute, '_' as in

[1] As soon as we have an efficient STEP that can support them, we will add the rest of the `numarray` types. To do this, however, we will need to add a Python to `C` code compiler to SIMP capable of converting transition functions directly to `C` functions. We believe that the `scipy.weave` or `pyinline` package will be a good way to do this.

```
sigregion   = sig[:,4:4]
sigregion._   = 1
```

Doing this calls a STEP `Write` operation.

Member data (read only)

`region`
The `Region` object for this subscript.

`shape`
The shape of the array associated with the `region`.

Methods

Calling `array()` returns the array associated with this region.

`array` `()`
Return the data array associated with this region as a `NumArray`.

If the subscript references a single coordinate, it returns a scalar. (Wrapper for the STEP `Read` operation)

`base_signal` `()`
Return a reference to the `Signal` of which this is a region selection.

`neighbor_offset` `()`
Return a single coordinate specifying the neighbor offset. Raises an error if the region contains more than one site.

A *neighbor* is a `SignalRegion` that references a single coordinate. Neighbor slices are used to write cellular automata rules.

class `Region` (*region*)
A class for representing multidimensional slices.

The constructor expects *region* to be a `Region` object or a multidimensional Python subscript.

`Member data (read-only)`

`start`
start position for the region (vector)

`stop`
stop position for the region (vector)

`size`
The size of the region (stop-start). Returns `None` if the size is undefined because the size of the boundary is not known.

`methods`

`subscript` `()`
Return a Python list representing the region's subscript

class `LatticeArrayRegion` (*region*)
A class for representing multidimensional slices selecting the sites of a `LatticeArray` object.

Extends the `Region` class.

member data (read-only)

`shape`
The shape of the array representing the region.

2.4.3 Output signals

A special kind of signal, called an `OutputSignal` can be used for rendering purposes. One would often like to apply a rule to do a rendering operation. If one used regular signals to hold the output values and then read the values out of these signals, extra storage would be needed—internal storage for the signals and an external array for the output data. In addition, an extra copy would be needed—a copy from the signal to the output array. Two copies must be kept, because, it is assumed that signals will be used again as inputs to future rules.

If one instead declares a special output signal, a STEP can realize that it the output signal will only be written and avoid keeping an extra copy around—and the data can be written directly to the output.

class `OutputSignal` (*type,generator=None,simp=None*)

A signal that is used as an output only.

It is the same as an ordinary signal in all ways except that it is output only.

`OutputSignal` objects are used in rendered `Read` operations.

2.5 STEP operations

STEP operations (ops) are represented by Python classes. They must be declared before they can be used. An op is *declared* by calling its class constructor and *issued* by calling it (as in `op()`) or by passing it as an argument to the STEP `Do` function (as in `Do(op)`).

An op's constructor automatically registers it with the STEP[2]. This is done so that the STEP may raise an exception if the op is somehow invalid or a `StepError` if for some reason it is unable to perform the op. The STEP may also perform some internal compilation in order to be ready to do the op later.

exception `StepError` ( )

Base class for STEP exceptions. STEP will put relevant information in the exception's string.

2.5.1 Rules

Note: Because the current STEPs use lookup tables (LUTs) to implement rules, one should be careful not to make the number of input signals too large.

The rule function must follow some special restrictions. A STEP may not enforce these restrictions, but as the writer of a STEP function, one should be certain that they are upheld, otherwise unexpected results may arise.

In general, a rule should be written as a simple aritmetic and logical function of neighbors using only simple flow control primitives.

- A rule may not 'carry' values. (it may not write a global value and expect to re-read that global in the next iteration).
- No 'time variant' function calls. All external functions called must be strictly deterministic—given a set of input values, it should always return the same set of outputs.
- No exceptions. The rule may not use `try` statements or raise exceptions.

In addition to these rules, a STEP (especially one that generates C code) may enforce the following

- Single type. A local variable may only be assigned to a single type within the rule
- Restricted allowable function calls. Some function calls may be difficult for the STEP to implement or analyze. A STEP might not allow function calls at all or only allow a fixed subset of them.
- Python construct restrictions. Try to use only simple python constructs.
- Limited types. Mutable types such as lists may be prohibited.

[2]Internally it uses the `step.Register(op)` method. `Signal` objects are also registered in the same way.

class Rule (*rule_function,generator=None,namespace={}*)
A STEP operation that performs parallel, local updates.

A `Rule` updates signals in parallel by locally applying *rule_function* at the sites of the lattice specified by its *generator* and overriding global names in the function with names in `namespace` if it is defined.

The `Rule` class extends the `LatticeArray` class and thus all of its attributes and methods.

When a `Rule` is instantiated, the global names in the *rule_function* are *strobed*—replaced with constant values. Strobing is necessary to allow a STEP to perform type analysis on a the Python *rule_function*, determine the `input` and `output` signals, and compile a LUT to represent the function. A new private namespace for the `Rule` is created and mutable objects are copied with a deep copy. Once the globals have been strobed, the internal representation of the *rule_function* is no longer affected by changes made to global variables.

Names in *namespace* dictionary supersede the the *rule_function*'s globals, allowing one to construct parameterized rules.

When a `Rule` is strobed, it constructs the function's `inputs` and `outputs`. All regions referenced inside of the *rule_function* are relative to the site being updated—signal subscripts with a single coordinate specify neighbors.

inputs
The `Signal` and `SignalRegion` objects accessed as inputs by the `rule _function`.

outputs
The `Signal`, `SignalRegion`, `OutputSignal` objects written by the `rule _function`.

class LutRule (*lut,inputs,outputs,generator=None*)
A STEP `Rule` with a transition function specified by a lookup-table.

inputs is an in-order list of the `Signal` /`SignalRegion` inputs to the *lut* and *outputs* is an in-order list of the `lut` outputs. The *lut* itself is an $m + 1$ dimensional array indexed in the upper m dimensions in-order by the inputs and in the lowest dimension by the index of the output. The type of the array must be compatible with all of the output values. (Usually, a `UInt8` array is the right choice.)

The input values must be unsigned.

XXX not yet implemented

2.5.2 Moving the lattice

class Shift (*shifts*)
STEP operation that shifts the position of `LatticeArray` objects.

shifts is a dictionary keyed `LatticeArray` objects and mapping them to shift vectors.

For `Signal` objects, the data in the lattice is moved by by the amount of the shift. (However the movement may simply be an update an internal address register that says where the data is the next time it is needed.) After a shift, the lattice's start position moves to a new location (modulo the lattice generators) as described in Section **??**. Because the sites of their lattice are undifferentiated, this is all that happens for `Rule` objects.

class Stir (*objects*)
STEP operation that randomly shuffles `LatticeArray` object positions.

objects is a list of `LatticeArray` objects to stir.

Stirring `Signal` objects, performs a mild permutation of the array elements and randomizes its starting position. It is usually implemented by a by a random shift. Stirring data is much more efficient than generating high quality randomness, and is often sufficient to achieve the same effect.

A `Rule` does not have data. Stirring a `Rule` just randomizes its starting position.

class GetPosition (*lattice_object*)
When called, returns the starting position of a `LatticeArray` (`Rule` or `Signal`.

A STEP operation is required for this since the STEP maintains the starting position of its `Rule` and `Signal` objects.

Usually, one will use the `getposition()` wrapper method of `Rule`, `Signal`, and `OutputSignal` objects to call `GetPosition`.

`object`
The object whose position is to be read.

class `SetPosition` (*positions*)
Set the starting positions of a set of `LatticeArray` objects.

positions is a dictionary keyed on `LatticeArray` objects mapping each to its new starting position. The starting position is a vector or another `LatticeArray` object, in which case the new position is the same as the position of the object.

A STEP operation is required for this since the STEP maintains the starting position of its `Rule` and `Signal` objects.

2.5.3 Input and output operations

class `Read` (*region,signals,rule=None,samearray=0*)
Read `Signal` and `OutputSignal` data out to an array.

The operation reads data from *signals*—a set of `Signal` or `OutputSignal` objects—within *region*—a Python subscript or `Region` object—to an output array. If `OutputSignal` objects are read, the *rule* for generating them must be defined—otherwise it need not be. If *samearray* is true, all data is read out to a single array, otherwise, data are read out to multiple arrays—one for each signal.

Usually, a `Read` is generated and called by subscripting a signal and calling the `array()` method, as in `sig[1:14,:].array()`. `Renderer` objects also create and call `Read` operations on color `OutputSignal` objects with user specified rendering `Rule` objects. For non-orthonormal lattices, sites are packed into arrays following the site selection conventions of Section **??**.

The output values are read out to an array or list of arrays when the operation is called.

`__call__`(*array=None*)
Return an array (or list) containing the output values.

By default, a new array (or list of arrays) is allocated and returned. If *array* is given, the output is placed there instead. If the *samearray* constructor parameter was true, the results are compacted into a single array, otherwise, they are returned as a list of arrays—one for each signal.

class `Write` (*region,signals,values=None*)
Write the values in *region* of each `Signal` in the *signals* list with the corresponding value from the *values* list.

One usually writes to `Signal` values using subscript indexes. Behind the scenes, this allocates and calls `Write` operations.

There are two flavors of writes. If *values* is specified in the constructor, the write operation is static. Otherwise, it is dynamic. For a dynamic write, values must be specified when the operation is called.

`__call__`(*values=None*)
If the write is dynamic, *values* must be specified, otherwise they need not be.

2.5.4 Composing operations with `Sequence`

class `Sequence` (*operations*)
Class for representing a sequence of STEP *operations* to be performed. The *operations* are a list of STEP operations to be done in order.

Creating `Sequence` objects notifies the STEP that the operations will be done one-after-another. The STEP, in turn, can sequence optimize the sequence.

Because a sequence is itself an operation, a sequence can be nested.

2.6 STEP methods

These are the public methods of a STEP. When one imports `simp`, it provides wrappers for them.

`Do` (*op,parameters=()*)

Tells the STEP to do the *op* with the specified *parameters*

Note: Normally, this function is not needed since it is called behind the scenes when one calls an op, as in `op()`.

The function is a wrapper for the STEP `Do` method. The optional*parameters* are passed to the operation's call method. Returns whatever the operation returns.

`SeedRandom` (*n=None*)

Seed the SIMP and STEP random number generators with the integer *n*.

Seeds the pseudo-random (deterministic) random number generator that STEP employs for `Stir` operations and SIMP employs for `makedistribution`.

The default seed is the current time (an integer cast of `time.time()`). It is set automatically when SIMP initializes.

Note: At this time, there is no method to get the seed because `SeedRandom` seeds multiple random number generators. At any given time the seed is a collection of all of their seeds.

`Flush` ()

Flush any STEP operations that may be pending in the pipeline.

Note: This is primarily a debugging function—operations are flushed automatically by the STEP.

`ClearCache` ()

Often times a STEP will cache code and lookup tables. `ClearCache` clears the current cache that the STEP uses.

Note: This is primarily a debugging method.

2.7 Renderers

Renderers are objects that render signals to images. Usually, this involves rendering a set of state values to a RGB array. They do this by constructing the STEP `Read` operations necessary to complete the task and performing any extra buffering that may be needed (in particular, the space-time renderers buffer state information).

Renderer objects implement a common interface so that they may be accessed in a common way by the `Console`.

Renderers are used to obtain 2-D rendered images from signals. In particular, they implement the `Renderer` interface, as such, they are compatible with `Console` objects, which rely on a renderers for generating the images they display. They also provide handy mechanisms for rendering from scripts.

At its core, rendering is a specialized `Read` op that does a rendered read to a set of color `OutputSignal`s. The rendering function is declared using in a rule object. Given the rendering rule, the renderer automatically constructs the `Read` ops that obtain the output.

2.7.1 Rendering

The `Renderer` defines the basic rendering interface. The `XTRenderer` extends it. Other extensions might later include a `VolumeRenderer`.

Often one will use the SIMP helper `declarecolors` () to declare the `OutputSignal` objects used in rendering.

class `Renderer` (*render_rule,outputs*)

Given a *render_rule* (`Rule` object), that sets the values of the signals in *outputs* the Renderer constructs the approproate STEP `Read` operations for reading the `outputs` .

It is usually used for rendering two–dimensional spaces, but can render spaces with any number of dimensions.

The renderer provides an interface that the Console and scripts can easily use to control rendering and the *view*—the subset of the space—that is rendered.

Initialization parameters

`render _rule`

The `Rule` used for rendering. It must set values of the *outputs*.

`outputs`

List of the `OutputSignal` objects rendered. If there is more than one output, the least significant dimension of the result array indexes the output values in the order specified by *outputs*

The renderer is a callable object. When called it returns the rendered output signal values in a single array. If there is more than one output signal, the values are packed into a single array with n+1 dimensions—the least significant is indexed by the ordering of the output signals.

`__call __` (*out=None*)

Call the renderer and return the output array.

out is the array be rendered to. If not specified, a new array is created and returned.

Rendering is performed on the current view specified by the `shape` ,`center` , and `region` parameters.

View descriptor data (read-only)

The view is the portion of the state rendered. The renderer has the following *read only* state attributes pertaining to the view. Note that the parameters describe the view in terms of both the array that will be rendered (which is what the console cares about) and the underlying coordinate space.

The `Console` will be primarily interested in modifying the shape and center coordinate.

`size`

The grid size of the region to be rendered.

`maxsize`

The maximum size of the region that could be rendered to.

`shape`

The shape of the array to be rendered for the current view.

`maxshape`

The maximum shape that could be rendered if the view were to be expanded as wide as possible.

`center`

The center coordinate of the view to be rendered.

`region`

The `Region` object specifying the rectangular region containing the view.

View methods

`setregion` (*region*)

Set the grid coordinate *region* for rendering.

The default is from 0 to *maxsize*.

`setcenter` (*center=None*)

Set the center coordinate for rendering.

Augments the `region` to reflect the new *center*. If *center* is `None` , it sets the center to be the center of the coordinate space.

setshape (*shape=None*)

Set *shape* of the array to be rendered to.

If it is larger than `maxshape`, it is clipped at maxshape. Modifies the `region` but not the `center`. The default is *maxshape*.

class XTRenderer (*render_rule,outputs,time=None*)

The space-time renderer, `XTRenderer`, records past rendering history and provides an $n+1$ dimensional space-time view. Beyond this, it implements the basic `Renderer` interface.

Except for the addition of *time*, the initialization parameters are the same. Time gives the amount of history to be maintained. By default, it's the same as the size of the lowest dimension.

Normally, time increases moving downwards. To make it increase going upwards, use a negative value for *time*.

Warning: Currently, *setshape*, *setcenter*, and *setregion* don't do anything.

Calling `record` renders a new line in the space-time history. Typically, it will be called by the `Console`, after `"STEP"` events. The `Console` automatically checks to see whether a renderer defines the `record` event.

record ()

Records the rendered value of the current state.

Calling the renderer returns an array containing the rendered history.

2.8 The Console

The `Console` provides a viewer window and an interactive key command interface for running SIMP programs interactively. The `Console` included with SIMP is built on the `pygame` (http://pygame.org) Python interface to the SDL (simple direct media layer).

class Console (*renderers,shape=None,center=None,mag=None, zoom=1,showgrid=1*)

Console user interface.

The *renderers* argument is a list (or single object) specifying the `Renderer` objects that the `Console` uses to get the image arrays that it displays. `shape` is the shape of the screen—defaults to `maxshape` of the first renderer object. `center` is the center position for rendering (defaults to the center of the space) and `mag` is the magnification (defaults to 1). *zoom* gives the zoom in factor for the display. *showgrid* indicates whether grid lines are to be shown when one zooms in.

bind (*event,object*)

Bind a keypress or event to a function or callable `object` that handles it.

generic key events

The console defines lower-case key commands. When introducing new commands, the user should typically use upper-case keys.

special events

- `'STEP'` Function or operation to be called when running a single step.

binding (*event*)

Return the handler bound to *event*.

unbind (*event*)

Unbind the handler bound to *event*.

array ()

Return the image array of the current view.

start ()

Start the control panel. Will not exit until the user quits.

close ()

Close the console—destroys its window.

XXX document the builtin commands.

2.9 Using `import_locally` to make multiple `simp` instances

The `simp` module is designed to be imported once and be imported with `from simp import *`. In some situations, one might like to have multiple instances of `simp` at the same time. The `import_locally` supports this.

Note: Some methods require that either a SIMP instance be passed to them explicitly or that they be initialized in a namespace where all methods from `simp` have been imported. In particular, they look for the function, `simpmodule()`.

This module is used in cross validation. We don't expect it to be used frequently by users since usually a program requires only one `simp` instance. The basic template is

```
import import_locally
simp1 = import_locally.import_copy("simp")
simp2 = import_locally.import_copy("simp")
```

Now, one must explicitly qualify all of the `simp` methods as in `simp1.import_locally(...)` and `a = Signal(...,simp=simp1)`. One must be careful not to mix operations and signals from the different simp modules.

To emulate `from simp import *` on one of the modules, use

```
import_locally.import_all(simp1)
```

To switch the local definitions to the other module, first unimport the first module and load the second,

```
import_locally.unimport_all(simp1)
import_locally.import_all(simp2)
```

`import_copy``(`*module_name*`)`
Import a new copy of a module, but don't add it to the global list of modules. This way, separate imports don't, as is normally the case in Python, access the same module object.

module_name is the string name of the module. Returns the module object.

`import_all``(`*module*`)`
Import all methods and variables from an *module* object into the current local namespace. Emulates '`from module import *`'.

Note: Only static module attributes should be referenced. Don't expect values to change in the local namespace when they change in the module. To change module values, one must reference the module directly. Although all of the attributes are imported, changing the value associated with the attribute in the local namespace does not change the value that module methods use and attributes whose values are modified by the module will not be updated in the module's namespace, but not in the local namespace.

`unimport_all``(`*module*`)`
Removes all references from a module object from the current local namespace. Python does not have an equivalent.

2.10 Testing and cross-validation

Testing and cross-validation facilities are contained in the `test` module.

test ()
: Run all of the test code.

cross _validate ()
: Cross validate the installed STEP implementation modules.

CHAPTER

THREE

Developer Documentation

This chapter documents the architectural and engineering aspects of the software at a high-level. It is meant to serve as a reference for those would like to know more about the software either for curiosity's sake or in order contribute.

3.1 Module layout

XXX

3.2 Project directory layout

The project contains the following directories and files.

- 'README' — Instructions for building the project
- 'TODO' — List of things to be done
- 'setup.py' — Python setup file used to compile code, install, and make binary installers for `simp` . Follows the usual `distutils` conventions.
- 'Lib/' — Library containing all the python files
  - '__init__.py' The simp module and all of its definitions.
  - 'step.py' The step interface–`Signal` ,ops,`step` base class definition.
- 'Src/' — Source C code for extension modules
- 'Doc/' — Documentation
- 'DosUtils/' — compiler utilities for Windows
- 'Misc/' — Code that we are testing or thinking of including
- 'Examples/' — example SIMP code—including code from the tutorial.
  - 'manual/' — (Latex file and tools for generating this manual.)
  - 'notes/' — file with miscellaneous notes about SIMP and future directions.

3.3 STEP interface

3.3.1 The STEP class

The `step` module contains the definitions of all the STEP objects described in Section 2.5 and an abstract STEP class definition called `step_base`.

A STEP is responsible storing signals, implementing STEP operations and keeping track of the starting positions of lattices. A module exporting the STEP interface contains a class called `step` that implements the following class

class `step` (*size,**kwargs*)

The base constructor does not have any default keyword arguments, but, an individual implementation may define some. The constructor should have default values for any arguments that are not specified explicitly.

`size`

vector giving the size of the grid—the coordinate space on which allocations are made.

`name`

The name of the STEP module.

`Do` (*op,parameters=()*)

Do a STEP . The optional *parameters* list is for passing an op's optional call parameters such as the output array for `Read` ops and the input array for `Write` ops.

`Register` (*object*)

All STEP interface objects (primitives and signals) must be registered before they are valid. When an object is registered, the STEP checks to see whether it can implement the operation or allocate the requested `Signal`. If not, it raises a `StepError` indicating the problem.

`SeedRandom` (*n=None*)

Seed the random number generator with the integer *n*. The seed is used for the pseudo-random (deterministic) random number generator that STEP employs for `Randomize` and `Stir` operations.

The default seed is the current time (an integer cast of `time.time()`). It is also set automatically when a STEP is initialized.

`Flush` ()

Flush all pending STEP operations and return when finished.

Note: This is primarily a debugging function.

`ClearCache` ()

Often times a STEP will cache code and lookup tables. `ClearCache` clears the current cache that STEP uses.

Note: This is primarily a debugging method.

3.3.2 Current and suggested STEP implementations

The STEP implementations are contained in `simp.stepmodules`. Currently, we distribute the following,

- `reference` — A slow, pure Python reference implementation written for correctness and clarity first and speed second.
- `pc` — A fast serial PC implementation based upon numarray and C.
- `pc_thread` — Use multiple threads for a multi-processor/multicore PC environment.

Some suggestions for other implementations are

- `pc_sparse` — Optimized for sparse updates.

- `pc_inline` — uses the `pyinline` module to compile C code on the fly. (supports arbitrary C typed signals) This will require more sophisticated function analysis.
- `multispin` — Use bit operations of the PC vector operations. Fast for repeated steps, but slow for reads and writes.
- `mmx` — Uses MMX vectorized instructions when possible.
- `mpi` — Use MPI (the message passing interface) to make fast updates on parallel machines.
- `fpga` — Use a FPGA as a stream co-processor.

3.4 Rolling your own—extending SIMP and STEP

3.4.1 Writing your own renderer

In order to get different rendering behavior, one may wish to create a new renderer. Such a renderer may be constructed manually using `Rule` and `Read` operations.

In order to make a renderer work with the `Console`, one need only implement the interface defined by the methods of the `Renderer` class.

[Add some more here...]

INDEX

BIBLIOGRAPHY

Bach, T. and Toffoli, T. (2003). SIMP/STEP: A rational framework for ‘crystalline computing’. In *SCI 2003 Proceedings*, volume XIV, pages 312–320.

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

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

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

Ilachinski, A. (2001). *Cellular Automata: A Discrete Universe*. World Scientific.

Talia, D. (2000). Cellular processing tools for high-performance simulation. *IEEE Computer*, pages 44–52.

Toffoli, T. and Bach, T. (2001). A common language for ‘programmable matter’. *Bull. Italian Assoc. for AI*, (2):23–31.

Toffoli, T. and Margolus, N. (1987). *Cellular Automata Machines*. MIT Press.

Weimar, J. R. (1997). *Simulation with Cellular Automata*. Logos-Verlag.

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

Wolfram, S. (2002). *A New Kind of Science*. Wolfram Media, Inc.

Worsch, T. (1996). Programming environments for cellular automata. In *Second Conference on CA and Industry*. ACRI.