STL, vector, cython

Automata

Peter Beerli beerli@fsu.edu

Build a C++ code to generate the Wolfram's R30 Automata algorithm

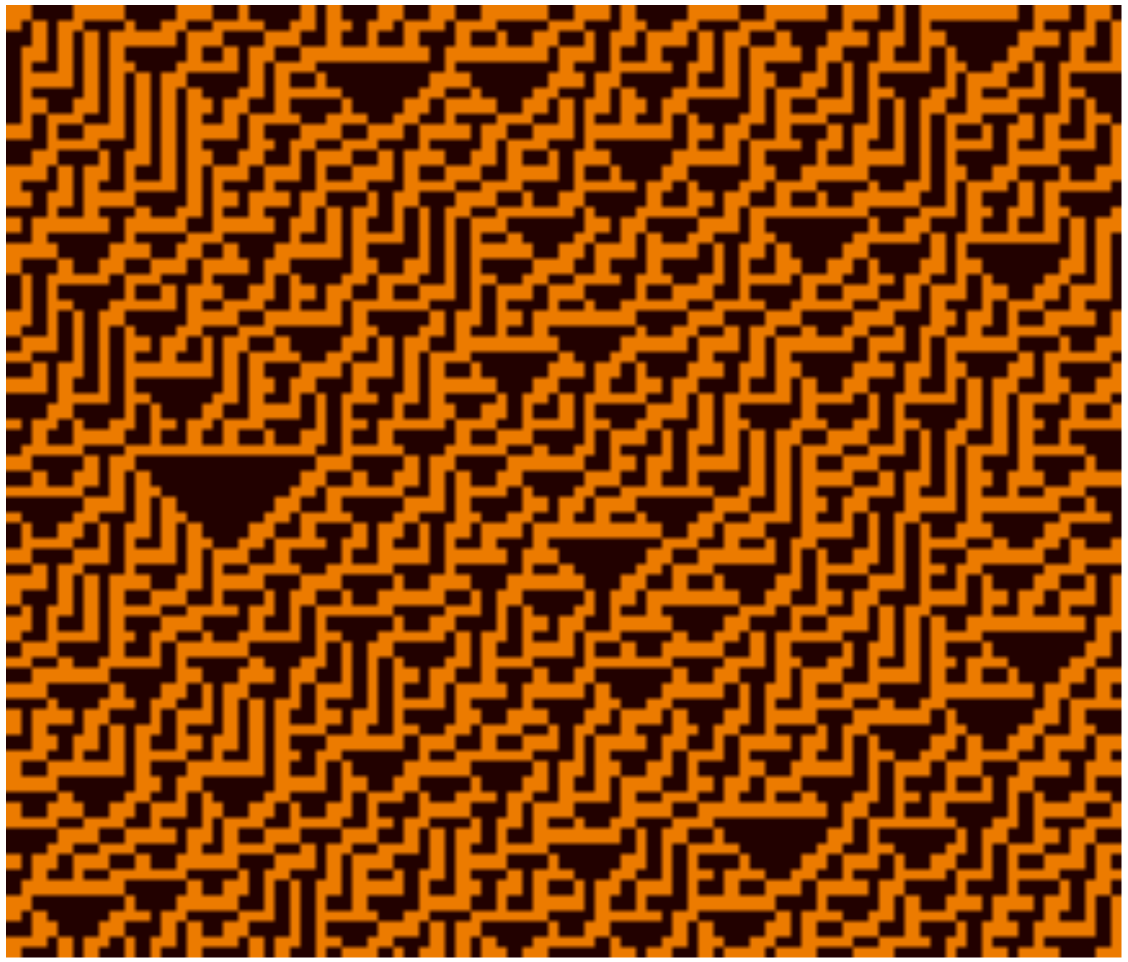
- C++ code
- Check that the code is working
- Generate a cython interface
- Generate a python interface
- Access C++ code through python

R30 algorithm

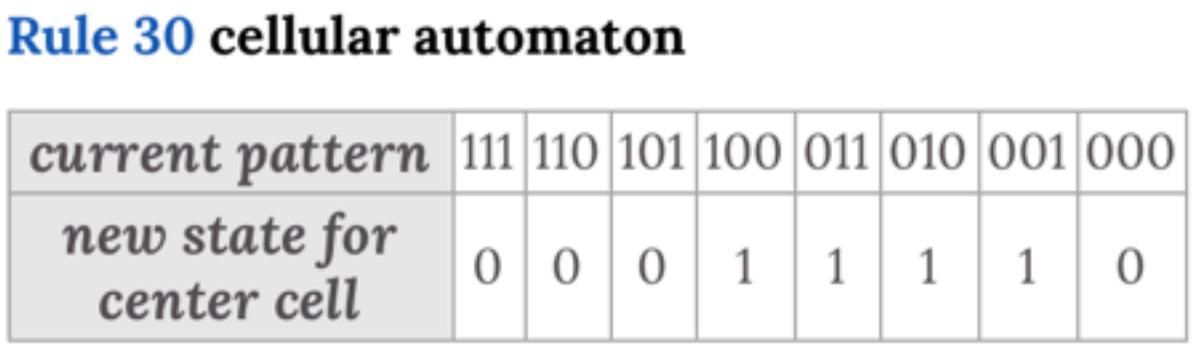
The simplest nontrivial cellular automaton would be one-dimensional, with two possible states per cell, and a cell's neighbors defined as the adjacent cells on either side of it. A cell and its two neighbors form a neighborhood of 3 cells, so there are 23 = 8 possible patterns for a neighborhood. A rule consists of deciding, for each pattern, whether the cell will be a 1 or a 0 in the next generation. There are then 28 = 256 possible rules.[4] These 256 cellular automata are generally referred to by their Wolfram code, a standard naming convention invented by Wolfram that gives each rule a number from 0 to 255. A number of papers have analyzed and compared these 256 cellular automata. The rule 30 and rule 110 cellular automata are particularly interesting.

Rule 30 cellular automaton

current pattern	111	110	101	100	011	010	001	000
new state for center cell	0	0	0	1	1	1	1	0



R30 algorithm

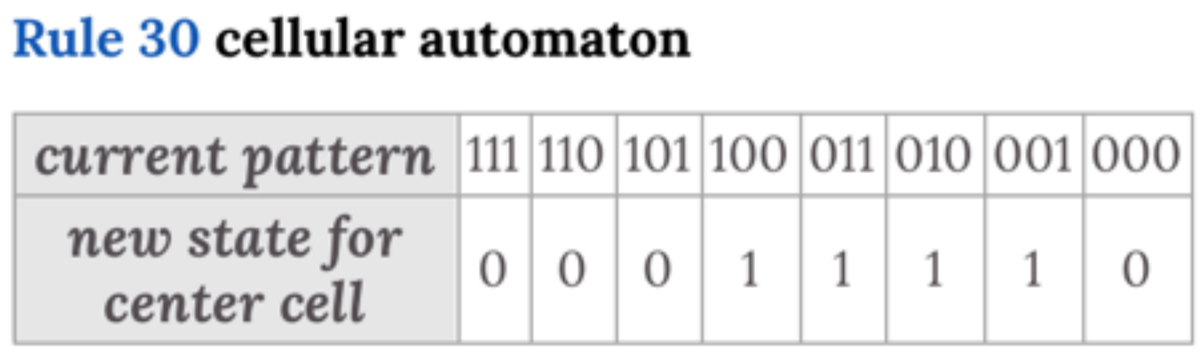


Rule 30 cellular automaton

current pattern	111	110	101	100	011	010	001	000
new state for center cell	0	0	0	1	1	1	1	0

```
00000000000000000000000001000000000000000000000000
00000000000000000000000011100000000000000000000000
00000000000000000000000110010000000000000000000000
00000000000000000000001101111000000000000000000000
00000000000000000000011001000100000000000000000000
00000000000000000000110111101110000000000000000000
00000000000000000001100100001001000000000000000000
00000000000000000011011110011111100000000000000000
00000000000000000110010001110000010000000000000000
00000000000000001101111011001000111000000000000000
00000000000000011001000010111101100100000000000000
00000000000000110111100110100001011110000000000000
00000000000001100100011100110011010001000000000000
00000000000011011110110011101110011011100000000000
```

R30 algorithm



Rule 30 cellular automaton

current pattern	111	110	101	100	011	010	001	000
new state for center cell	0	0	0	1	1	1	1	0

```
00000000000000000000000001000000000000000000000000
00000000000000000000000011100000000000000000000000
00000000000000000000000110010000000000000000000000
00000000000000000000001101111000000000000000000000
00000000000000000000011001000100000000000000000000
00000000000000000000110111101110000000000000000000
00000000000000000001100100001001000000000000000000
00000000000000000011011110011111100000000000000000
00000000000000000110010001110000010000000000000000
00000000000000001101111011001000111000000000000000
00000000000000011001000010111101100100000000000000
00000000000000110111100110100001011110000000000000
00000000000001100100011100110011010001000000000000
00000000000011011110110011101110011011100000000000
```

C++ header file

```cpp
#import <vector>
#import <iostream>

std::vector<int> generate_start_pattern (const int n);
std::vector<int> generate_new_pattern (const std::vector<int> v);
```

```
nagal:L24>g++ standalone_automata.cpp -o automa
nagal:L24>automa
Syntax: automata #numberofcolumns
nagal:L24>automa 50
00000000000000000000000001000000000000000000000000
00000000000000000000000011100000000000000000000000
00000000000000000000000110010000000000000000000000
00000000000000000000001101111000000000000000000000
00000000000000000000011001000100000000000000000000
00000000000000000000110111101110000000000000000000
00000000000000000001100100001001000000000000000000
00000000000000000011011110011111100000000000000000
00000000000000000110010001110000010000000000000000
00000000000000001101111011001000111000000000000000
00000000000000011001000010111101100100000000000000
00000000000000110111100110100001011110000000000000
00000000000001100100011100110011010001000000000000
00000000000011011110110011101110011011100000000000
00000000000110010000101110001001110010010000000000
```

C++   program file part1

```
#import <vector>
#import <iostream>
#import "cautomata.h"
// generate a start pattern
std::vector<int> generate_start_pattern (const int n)
{
  std::vector<int> v(n,0);  // create vector of zeroes
  v[n/2]=1; // set the middle element to 1
  return v;
}

std::vector<int> generate_new_pattern (const std::vector<int> v)
{
  //  current pattern           111 110 101 100 011 010 001 000
  //  new state for center cell 0   0   0   1   1   1   1   0
  std::vector<int>::const_iterator it;
  std::vector<int>::iterator nit;
  std::vector<int> newv(v);
  for(it=v.begin()+1, nit=newv.begin()+1; it < v.end()-1; it++, nit++)
    {
      int value = 100 * *(it-1);
      value  +=  10 * *it;
      value  += *(it+1);
      if (value==100 || value==1 || value==10 || value==11)
   *nit = 1;
      else
   *nit = 0;
    }
  return newv;
}
```

C++ program file part1

```cpp
void print(const std::vector<int> v)
{
  for( std::vector<int>::const_iterator it = v.begin(); it != v.end(); ++it)
    std::cout << *it;
  std::cout << std::endl;
}


int main(int argc,char **argv)
{
  if (argc <= 1)
    {
      std::cout << "Syntax: automata #numberofcolumns" << std::endl;
      return -1;
    }
  int n = atoi(argv[1]);
  std::vector<int> v = generate_start_pattern (n);
  print(v);
  for(int i=0;i<30;++i)
    {
      v = generate_new_pattern (v);
      print(v);
    }
  return 0;
}
```

Road map

- C++ code working
- Create the interface for cython (create a .pxd file)
- Create the interface for python (create a .pyx file)
- Create setup.py to compile the parts
- Create the python testcode
- Improve the python testcode using matplotlib

Interface C++ to Cython (.pxd)

```
# cautomata.pxd
import cython
from libcpp.vector cimport vector

cdef extern from "cautomata.h":
     vector[int] generate_start_pattern (const int n)
     vector[int] generate_new_pattern (const vector[int] v)
```

Interface Cython to Python (.pyx)

```
# automata.pyx
cimport cautomata as ca
from libcpp.vector cimport vector

def gfp(int n): #generate first pattern
    return ca.generate_start_pattern(n)

def gp(const vector[int] v): #take pattern and generate new one
   return ca.generate_new_pattern(v)
```

Interface Cython to Python (.pyx)

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
import numpy
import os
os.environ["CC"] = "clang++"  # on your system use g++
os.environ["CXX"] = "clang++" # on your system use g++
setup( name = 'myautomata',   # module name to call in python code
ext_modules=[Extension("myautomata", # specifies all the files needed
             sources=["automata.pyx","cautomata.cpp"],
             language="c++", # tells cython to use C++ instead of C
             include_dirs=[numpy.get_include(),"."])],
cmdclass = {'build_ext': build_ext},
)
```

build our C++/cython/python hybrid

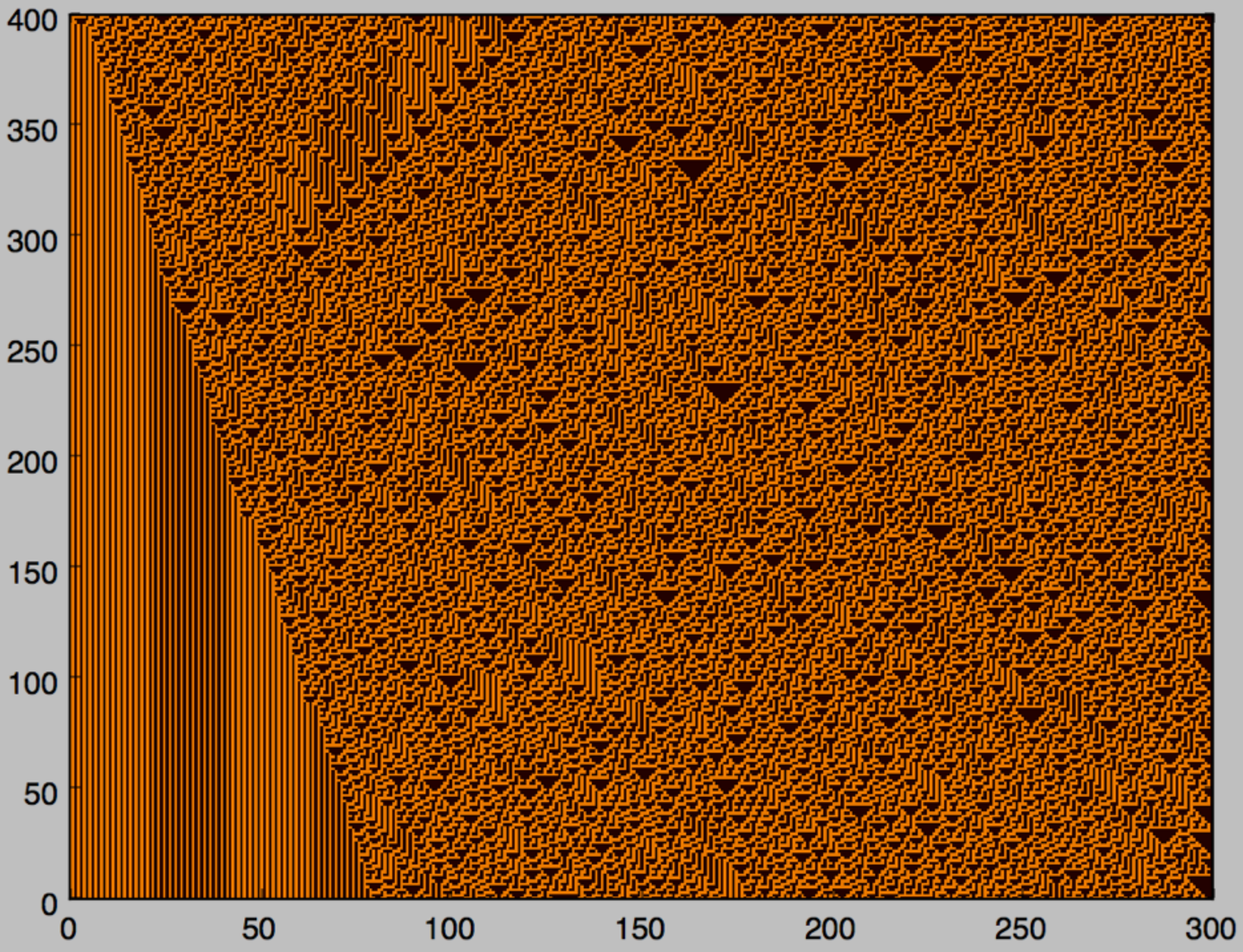
```
nagal:example_automata>python setup.py build_ext --inplace
running build_ext
cythoning automata.pyx to automata.cpp
building 'myautomata' extension
creating build
creating build/temp.macosx-10.10-intel-2.7
clang++ -fno-strict-aliasing -fno-common -dynamic -arch x86_64 -arch i386 -g -Os -pipe -fno-
common -fno-strict-aliasing -fwrapv -DENABLE_DTRACE -DMACOSX -DNDEBUG -Wall -Wstrict-
prototypes -Wshorten-64-to-32 -DNDEBUG -g -fwrapv -Os -Wall -Wstrict-prototypes -
DENABLE_DTRACE -arch x86_64 -arch i386 -pipe -I/usr/local/lib/python2.7/site-packages/numpy/
core/include -I. -I/System/Library/Frameworks/Python.framework/Versions/2.7/include/python2.
-c automata.cpp -o build/temp.macosx-10.10-intel-2.7/automata.o
automata.cpp:1769:28: warning: unused function '__Pyx_PyObject_AsString' [-Wunused-function]
static CYTHON_INLINE char* __Pyx_PyObject_AsString(PyObject* o) {
                           ^

                             ^      ……many more warnings……
automata.cpp:1626:27: warning: function '__Pyx_PyInt_As_long' is not needed and will not be
emitted
      [-Wunneeded-internal-declaration]
static CYTHON_INLINE long __Pyx_PyInt_As_long(PyObject *x) {
                          ^

9 warnings generated.
clang++ -fno-strict-aliasing -fno-common -dynamic -arch x86_64 -arch i386 -g -Os -pipe -fno-
common -fno-strict-aliasing -fwrapv -DENABLE_DTRACE -DMACOSX -DNDEBUG -Wall -Wstrict-
prototypes -Wshorten-64-to-32 -DNDEBUG -g -fwrapv -Os -Wall -Wstrict-prototypes -
DENABLE_DTRACE -arch x86_64 -arch i386 -pipe -I/usr/local/lib/python2.7/site-packages/numpy/
core/include -I. -I/System/Library/Frameworks/Python.framework/Versions/2.7/include/python2.
-c cautomata.cpp -o build/temp.macosx-10.10-intel-2.7/cautomata.o
clang++ -bundle -undefined dynamic_lookup -arch x86_64 -arch i386 -Wl,-F. build/
temp.macosx-10.10-intel-2.7/automata.o build/temp.macosx-10.10-intel-2.7/cautomata.o -o /
Users/beerli/Documents/Work/talks/ISC-4304/lectures/L24/example_automata/myautomata.so
```

Python program: runautomata.py

```
#!/usr/bin/env python
import myautomata
import numpy as np
import matplotlib.pyplot as plt

# takes image matrix an fills a line at index
def fill_line(image,x,index,width):
    for i in xrange(width):
        xx = pick_color(int(x[i]))
        image[index][i] = xx

# fills whole image using width and height, the burnin is allow
# a few cycles of the algorithms before we visualize them
def fill_image(image,width,height,burnin):
    x = myautomata.gfp(width) # start with first pattern
    for i in xrange(burnin):  # burnin (throw a way cycles)
        x = myautomata.gp(x)  # get next new pattern and overwrite the old
line vector
    for i in range(height):   # now collecting results
        x = myautomata.gp(x)  # new line pattern
        fill_line(image,x,i,width) # insert line pattern into image
```

Python program: runautomata.py

```
def pick_color(i):
    # returns x which is a list of
    # RGB color + alpha channel
    # [Red, Green, Blue, Alpha]
    # the values are between 0 and 1
    # this is bright red: [1, 0, 0,1]
    if i==0:
        x = [0.1,0,0,1] # is redish black
    else:
        x = [0.9,0.4,0,1] # this is yellowish orange-brown
    return x

if __name__ == '__main__':
    w=300
    h=400
    image = np.empty((w,h,4),np.float32)
    fill_image(image,h,w,200)
    ii=plt.imshow(image,interpolation='none', extent=[0,w,
0,h],aspect='auto')
    plt.show()
```


400
350
300
250
200
150
100
50
0
0
50
100
150
200
250
300