Running Cython and Vectorization

1. Getting Started with Cython
   - overview
   - hello world with Cython
2. Numerical Integration
   - experimental setup
   - adding type declarations
   - cdef functions & calling external functions
3. Vectorization
   - the game of life of John Conway
   - performance issues
   - vectorizing the neighbor count
   - vectorizing the rules

MCS 275 Lecture 41
Programming Tools and File Management
Jan Verschelde, 21 April 2017

Running Cython and Vectorization

1. Getting Started with Cython
   - overview
   - hello world with Cython
2. Numerical Integration
   - experimental setup
   - adding type declarations
   - cdef functions & calling external functions
3. Vectorization
   - the game of life of John Conway
   - performance issues
   - vectorizing the neighbor count
   - vectorizing the rules

what is Cython?

Cython is a programming language based on Python:

- with static type declarations to achieve the speed of C,
- to write optimized code and to interface with C libraries.

Proceedings of the 8th Python in Science Conference (SciPy 2009):

- S. Behnel, R.W. Bradshaw, D.S. Seljebotn: **Cython tutorial.** In SciPy 2009, pages 4-14, 2009.
- D.S. Seljebotn: **Fast numerical computations with Cython.** In SciPy 2009, pages 15-23, 2009.

Version 0.25.2 (2016-12-08) is available via `cython.org`.

Demonstrations in this lecture were done on a MacBook Pro.
On Windows, Cython works well in cygwin.

compilation and static typing

Python is interpreted and dynamically typed:

- instructions are parsed, translated, and executed one-by-one,
- types of objects are determined during assignment.

Cython code is compiled: a program is first parsed entirely,
then translated into machine executable code, eventually optimized,
before its execution.

Static type declarations allow for

- translations into very efficient C code, and
- direct manipulations of objects in external libraries.

Cython is a Python compiler: it compiles regular Python code.

Running Cython and Vectorization

1. Getting Started with Cython
   - overview
   - hello world with Cython
2. Numerical Integration
   - experimental setup
   - adding type declarations
   - cdef functions & calling external functions
3. Vectorization
   - the game of life of John Conway
   - performance issues
   - vectorizing the neighbor count
   - vectorizing the rules

hello world with Cython

We can compile Cython code in two ways:

1. using `distutils` to build an extension of Python;
2. run the `cython` command-line utility to make a `.c` file and then compile this file.

We illustrate the two ways with a simple `say_hello` method.

Cython code has the extension `.pyx`. The file `hello.pyx`:

```
def say_hello(name):
    """
    Prints hello followed by the name.
    """
    print("hello", name)
```

using `distutils`

The file `hello_setup.py` has content

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

EXT_MODULES = [Extension("hello", ["hello.pyx"])]

setup(
    name = 'hello world' ,
    cmdclass = {'build_ext': build_ext},
    ext_modules = EXT_MODULES
)
```

building a Cython module

At the command prompt we type

```
$ python3 hello_setup.py build_ext --inplace
```

and makes the shared object file `hello.cpython-36m-darwin.so` which we can rename into `hello.so`.

```
$ python3 hello_setup.py build_ext --inplace
running build_ext
cythoning hello.pyx to hello.c
building 'hello' extension
creating build
creating build/temp.macosx-10.6-intel-3.6
/usr/bin/clang -fno-strict-aliasing -Wsign-compare -fno-com
/usr/bin/clang -bundle -undefined dynamic_lookup -arch i386
$
```

testing the Cython module

With the `--inplace` option, the shared object file
is placed in the current directory.

We import the function `say_hello` of the module `hello`:

```
$ python3
Python 3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more i
>>> from hello import say_hello
>>> say_hello("there")
hello there
>>>
```

the command line cython

```
$ which cython
/Library/Frameworks/Python.framework/Versions/3.6/bin/cython
$ cython hello.pyx
$ ls -lt hello.c
-rw-r--r--  1 jan  staff  77731 Apr 20 19:07 hello.c
$
```

The makefile contains the entry

```
hello_cython:
        cython hello.pyx
        /usr/bin/clang -c hello.c -o hello.o \
-fno-strict-aliasing -fno-common -dynamic -arch i386 -arch x86_64 \
-g -O2 -DNDEBUG -g -O3 \
-I/Library/Frameworks/Python.framework/Versions/3.6/include/python3.6m
        /usr/bin/clang -bundle -undefined dynamic_lookup -arch i386 \
-arch x86_64 -g hello.o -o hello.so
```

Typing `hello_cython` executes `cython`, compiles the `hello.c`,
and links the object `hello.o` into the shared object `hello.so`.

Running Cython and Vectorization

1. Getting Started with Cython
   - overview
   - hello world with Cython
2. Numerical Integration
   - experimental setup
   - adding type declarations
   - cdef functions & calling external functions
3. Vectorization
   - the game of life of John Conway
   - performance issues
   - vectorizing the neighbor count
   - vectorizing the rules

approximating π

For a computational intensive, yet simple computation, consider

$$\frac{\pi}{4} = \int_0^1 \sqrt{1 - x^2}\, dx$$

approximated with the composite Trapezoidal rule:

$$\int_0^1 \sqrt{1 - x^2}\, dx \approx \frac{1}{n} \left(\frac{1}{2} + \sum_{i=1}^{n-1} \sqrt{1 - \left(\frac{i}{n} \right)^2} \right).$$

We let $n = 10^7$ and make 10,000,000 square root function calls.

the Python function `integral4pi`

```
from math import sqrt # do not import in circle !!!

def circle(xvl):
    """
    Returns the y corresponding to xvl
    on the upper half of the unit circle.
    """
    return sqrt(1-xvl**2)

def integral4pi(nbvals):
    """
    Approximates Pi with the trapezoidal
    rule with nbvals subintervals of [0,1].
    """
    step = 1.0/nbvals
    result = (circle(0)+circle(1))/2
    for i in range(nbvals):
        result += circle(i*step)
    return 4*result*step
```

timing the execution (script continued)

```
def main():
    """
    Does the timing of integral4pi.
    """
    from time import clock
    start_time = clock()
    approx = integral4pi(10**7)
    stop_time = clock()
    print 'pi =', approx
    elapsed = stop_time - start_time
    print 'elapsed time = %.3f seconds' % elapsed

main()
```

Running this script on a 3.1 GHz Intel Core i7 MacBook Pro:

```
$ python3 integral4pi.py
pi = 3.1415930535527115
elapsed time = 3.464 seconds
$
```

Running Cython and Vectorization

1. Getting Started with Cython
   - overview
   - hello world with Cython
2. Numerical Integration
   - experimental setup
   - adding type declarations
   - cdef functions & calling external functions
3. Vectorization
   - the game of life of John Conway
   - performance issues
   - vectorizing the neighbor count
   - vectorizing the rules

the script integral4pi_typed.pyx

```
from math import sqrt

def circle(double x):
    return sqrt(1-x**2)

def integral4pi(int n):
    cdef int i
    cdef double h, r
    h = 1.0/n
    r = (circle(0)+circle(1))/2
    for i in range(n):
        r += circle(i*h)
    return 4*r*h
```

using `distutils`

We use `distutils` to build the module `integral4pi_typed`.
To build, we define `integral4pi_typed_setup.py`:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext_modules = [Extension("integral4pi_typed",
                         ["integral4pi_typed.pyx"])]

setup(
  name = 'integral approximation for pi' ,
  cmdclass = {'build_ext': build_ext},
  ext_modules = ext_modules
)
```

building the code

```
$ python3 integral4pi_typed_setup.py build_ext --inplace
running build_ext
cythoning integral4pi_typed.pyx to integral4pi_typed.c
building 'integral4pi_typed' extension
/usr/bin/clang -fno-strict-aliasing -Wsign-compare
-fno-common -dynamic -DNDEBUG -g -fwrapv -O3 -Wall
-Wstrict-prototypes -arch i386 -arch x86_64 -g
-I/Library/Frameworks/Python.framework/Versions/3.6/include
-c integral4pi_typed.c
-o build/temp.macosx-10.6-intel-3.6/integral4pi_typed.o
/usr/bin/clang -bundle -undefined dynamic_lookup
-arch i386 -arch x86_64
-g build/temp.macosx-10.6-intel-3.6/integral4pi_typed.o
-o /Users/jan/Courses/MCS275/Spring17/Lec41/
integral4pi_typed.cpython-36m-darwin.so
$
```

The setup tools show us the compilation instructions.

calling `integral4pi` of `integral4pi_typed`

```
from time import clock
from integral4pi_typed import integral4pi

START_TIME = clock()
APPROX = integral4pi(10**7)
STOP_TIME = clock()
print 'pi =', APPROX
ELAPSED = STOP_TIME - START_TIME
print 'elapsed time = %.3f seconds' % ELAPSED
```

Running the script:

```
$ python3 integral4pi_typed_apply.py
pi = 3.1415930535527115
elapsed time = 0.918 seconds
$
```

The code runs more than three times as fast as the original Python version (3.464 seconds).

Running Cython and Vectorization

1. Getting Started with Cython
   - overview
   - hello world with Cython
2. Numerical Integration
   - experimental setup
   - adding type declarations
   - cdef functions & calling external functions
3. Vectorization
   - the game of life of John Conway
   - performance issues
   - vectorizing the neighbor count
   - vectorizing the rules

declaring C-style functions

To avoid the construction of float objects around function calls, we declare a C-style function:

```
from math import sqrt

cdef double circle(double x) except *:
    return sqrt(1-x**2)
```

The rest of the script remains the same.
To compile `integral4pi_cdeffun.pyx`, we define the file `integral4pi_cdeffun_setup.py` and build with

```
$ python3 integral4pi_cdeffun_setup.py build_ext --inplace
```

calling `integral4pi` of `integral4pi_cdeffun`

Similar as with `integral4pi_typed_apply.py`
we define the script `integral4pi_cdeffun_apply.py`.

```
$ python3 integral4pi_cdeffun_apply.py
pi = 3.1415930535527115
elapsed time = 0.583 seconds
$
```

What have we achieved so far is summarized below:

	elapsed seconds	speedup
original Python	3.464	1.00
Cython with cdef	0.918	3.77
cdef function	0.583	5.94

calling external C functions

The main cost is calling `sqrt` 10,000,000 times...

Instead of using the `sqrt` of the Python `math` module,
we can directly use the `sqrt` of the C math library:

```
cdef extern from "math.h":
    double sqrt(double)
```

The rest of the script remains the same.
To compile `integral4pi_extcfun.pyx`, we define the file
`integral4pi_extcfun_setup.py` and build with

```
$ python3 integral4pi_extcfun_setup.py build_ext --inplace
```

calling `integral4pi` of `integral4pi_extcfun`

Similar as with `integral4pi_typed_apply.py`
we define the script `integral4pi_extcfun_apply.py`.

```
$ python3 integral4pi_extcfun_apply.py
pi = 3.141593053552715
elapsed time = 0.041 seconds
$
```

This gives a nice speedup, summarized below:

	elapsed seconds	speedup
original Python	3.464	1.00
Cython with cdef	0.918	3.77
cdef function	0.583	5.94
external C function	0.041	84.49

native C code

```
#include <stdio.h>
#include <math.h>
#include <time.h>

double circle ( double x )
{
   return sqrt(1-x*x);
}

double integral4pi ( int n )
{
   int i;
   double h = 1.0/n;
   double r = (circle(0)+circle(1))/2;

   for(i=0; i<n; i++)
      r += circle(i*h);
   return 4*r*h;
}
```

the main function

```
int main ( void )
{
   int n = 10000000;
   clock_t start_time,stop_time;

   start_time = clock();
   double a = integral4pi(n);
   stop_time = clock();

   printf("pi = %.15f\n",a);
   printf("elapsed time = %.3f seconds\n",
          (double) (stop_time-start_time)/CLOCKS_PER_SEC);

   return 0;
}
```

compiling and running

```
$ /usr/bin/clang -O3 integral4pi_native.c \
                 -o /tmp/integral4pi_native
$ /tmp/integral4pi_native
pi = 3.141593053552711
elapsed time = 0.023 seconds
$
```

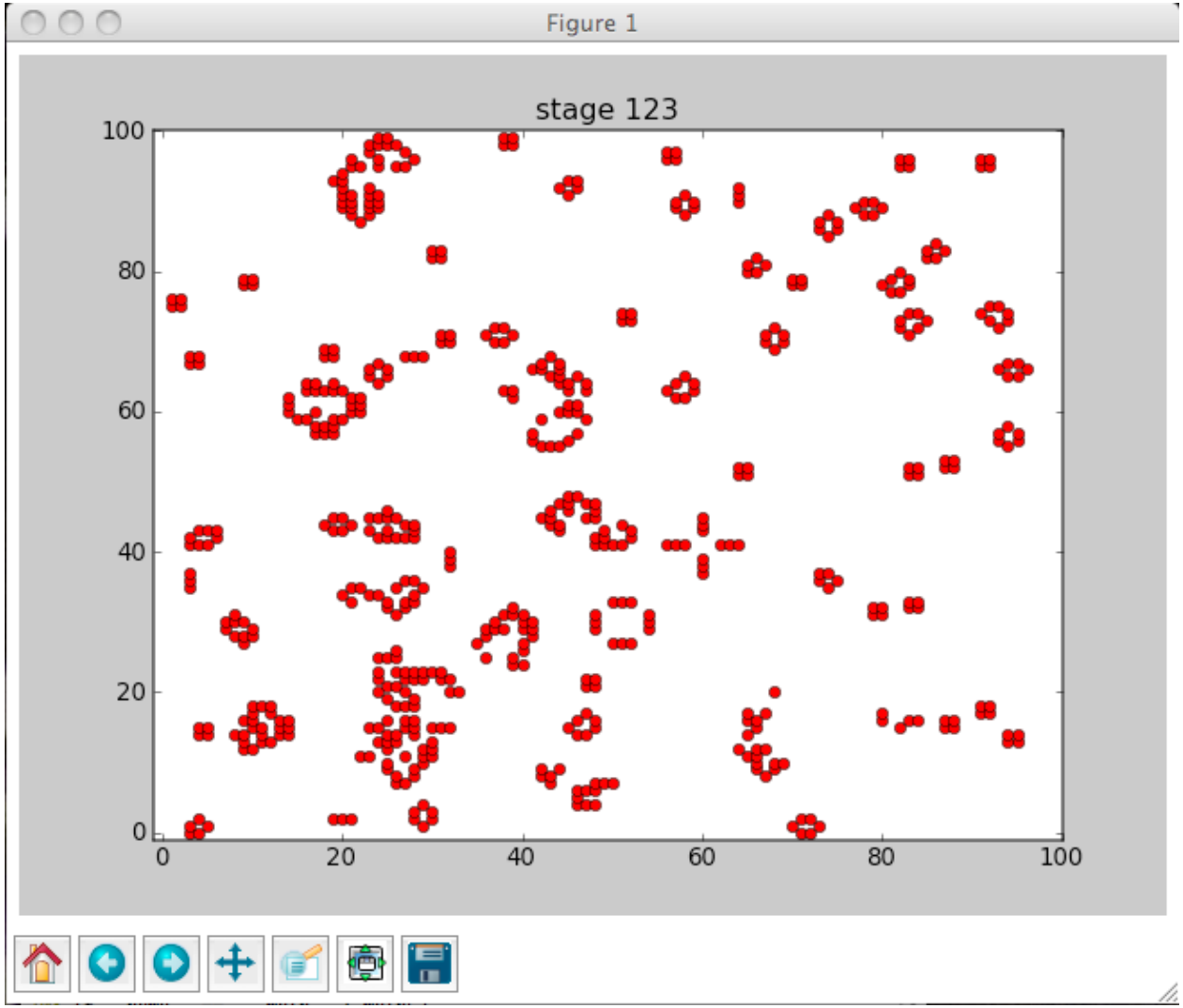
	elapsed seconds	speedup
original Python	3.464	1.00
Cython with cdef	0.918	3.77
cdef function	0.583	5.94
external C function	0.041	84.49
native C code	0.023	150.61

We achieve double digit speedups.
The Cython code which calls the C `sqrt` takes almost double the time of the native C code.

Running Cython and Vectorization

1. Getting Started with Cython
   - overview
   - hello world with Cython
2. Numerical Integration
   - experimental setup
   - adding type declarations
   - cdef functions & calling external functions
3. Vectorization
   - the game of life of John Conway
   - performance issues
   - vectorizing the neighbor count
   - vectorizing the rules

visualizing living cells



simulating cellular growth

The game of life is a discovery of John Conway.

Consider a rectangular grid of cells with rules:

1. An empty cell is born when it has 3 neighbors.
2. A living cell can either die or survive, as follows:
   1. die by loneliness, if the cell has one or no neighbors;
   2. die by overpopulation, if the cell has $\geq$ 4 neighbors;
   3. survive, if the cell has two or three neighbors.

design of the code

Three ingredients:

1. The rectangular grid is represented by a NumPy matrix A
   - of integers: $A_{i,j} \in \{0, 1\}$,
   - $A_{i,j} = 0$: cell (i, j) is dead,
   - $A_{i,j} = 1$: cell (i, j) is alive.
2. We update the matrix applying the rules, running over all pairs of indices (i, j).
3. We use matplotlib for making a spy plot of the nonzeroes.

the main function

```
def main():
    """
    Generates a random matrix and
    applies the rules for Conway's
    game of life.
    """
    ratio = 0.2  # ratio of nonzeroes
    dim = input('give the dimension : ')
    alive = np.random.rand(dim, dim)
    alive = np.matrix(alive < ratio, int)
    for i in xrange(10*dim):
        spm = sparse.coo_matrix(alive)
        plot(spm.row, spm.col, 'r.', \
             axis=[-1, dim, -1, dim], \
             title='stage %d' % i)
        alive = update(alive)
```

applying the rules

```
def update(alive):
    """
    Applies the rules of Conway's game of life.
    """
    result = np.zeros(alive.shape, int)
    for i in range(0, alive.shape[0]):
        for j in range(0, alive.shape[1]):
            nbn = neighbors(alive, i, j)
            if alive[i, j] == 1:
                if((nbn < 2) or (nbn > 3)):
                    result[i, j] = 0
                else:
                    result[i, j] = 1
            else:
                if(nbn == 3):
                    result[i, j] = 1
    return result
```

counting live neighbors

```
def neighbors(alive, i, j):
    """
    Returns the number of cells alive
    next to alive[i, j].
    """
    cnt = 0
    if i > 0:
        if alive[i-1, j]:
            cnt = cnt + 1
        if(j > 0):
            if alive[i-1, j-1]:
                cnt = cnt + 1
        if(j < alive.shape[1]-1):
            if alive[i-1, j+1]:
                cnt = cnt + 1
```

counting live neighbors continued

```
if(i < alive.shape[0]-1):
    if alive[i+1, j]:
        cnt = cnt + 1
    if(j > 0):
        if alive[i+1, j-1]:
            cnt = cnt + 1
    if(j < alive.shape[1]-1):
        if alive[i+1, j+1]:
            cnt = cnt + 1
if(j > 0):
    if alive[i, j-1]:
        cnt = cnt + 1
if(j < alive.shape[1]-1):
    if alive[i, j+1]:
        cnt = cnt + 1
return cnt
```

Running Cython and Vectorization

1. Getting Started with Cython
   - overview
   - hello world with Cython
2. Numerical Integration
   - experimental setup
   - adding type declarations
   - cdef functions & calling external functions
3. Vectorization
   - the game of life of John Conway
   - performance issues
   - vectorizing the neighbor count
   - vectorizing the rules

performance issues

The straightforward code does not work well
on a MacBook when the dimension gets at $n = 300$.

Reasons for the problems with performance:

- Traversals through the matrix with double for loops in Python: incrementing loop counters is expensive.

  The loop counts `i` and `j` are Python objects.

- In counting the neighbors we access not only the (i, j)-th data element, but also $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, and $(i, j+1)$.

  In the row or column oriented storing of the matrix, getting access to respectively $(i-1, j), (i+1, j)$ or $(i, j-1), (i, j+1)$ means accessing n elements before or after the (i, j)-th element.

Vectorization reorganizes the counting of the live neighbors
and the application of the rules.

Running Cython and Vectorization

1. Getting Started with Cython
   - overview
   - hello world with Cython
2. Numerical Integration
   - experimental setup
   - adding type declarations
   - cdef functions & calling external functions
3. Vectorization
   - the game of life of John Conway
   - performance issues
   - vectorizing the neighbor count
   - vectorizing the rules

vectorizing the neighbor count

For every cell (i, j) in the matrix we need to count
the number of neighbors that are alive.

We need to avoid to access at the same time the left, right, upper and lower neighbors because of the way matrices are stored.

Idea: to count the right neighbors of live cells
add to the matrix the same matrix shifted one column.

counting the right live neighbors

```
>>> import numpy as np
>>> A = np.random.rand(5,5)
>>> A = np.matrix(A < 0.3,int); A
matrix([[1, 1, 1, 1, 1],
        [1, 1, 0, 0, 1],
        [0, 0, 0, 0, 0],
        [0, 0, 1, 1, 0],
        [0, 0, 0, 0, 0]])
>>> B = np.copy(A[:,+1:])
>>> B
array([[1, 1, 1, 1],
       [1, 0, 0, 1],
       [0, 0, 0, 0],
       [0, 1, 1, 0],
       [0, 0, 0, 0]])
>>> A[:,:-1] = np.copy(A[:,:-1]) + B
>>> A
matrix([[2, 2, 2, 2, 1],
        [2, 1, 0, 1, 1],
        [0, 0, 0, 0, 0],
        [0, 1, 2, 1, 0],
        [0, 0, 0, 0, 0]])
```

the script game_neighbors.py

```
def neighbor_count_matrix(alive):
    """
    Returns a matrix counting the
    number of live neighbors.
    """
    acopy = np.copy(alive)
    result = np.copy(alive)
    left = np.copy(result[:, :-1])  # omit last column
    right = np.copy(result[:, +1:]) # omit first column
    result[:, +1:] = np.copy(result[:, +1:]) + left
    result[:, :-1] = np.copy(result[:, :-1]) + right
    upper = np.copy(result[:-1, :]) # omit last row
    lower = np.copy(result[+1:, :]) # omit last column
    result[+1:, :] = np.copy(result[+1:, :]) + upper
    result[:-1, :] = np.copy(result[:-1, :]) + lower
    result = result - acopy
    return result
```

the main test

```
def main():
    """
    Test on neighbor count.
    """
    dim = 8
    ratio = 0.2
    alive = np.random.rand(dim, dim)
    alive = np.matrix(alive < ratio, int)
    count = neighbor_count_matrix(alive)
    print 'cells alive\n', alive
    print 'vectorized neighbor count\n', count
    orgcnt = neighbor_count(alive)
    print 'original neighbor count\n', orgcnt
    print 'equality check :'
    print np.equal(count, orgcnt)
    print sum(sum(np.equal(count, orgcnt)))
```

Running Cython and Vectorization

1. Getting Started with Cython
   - overview
   - hello world with Cython
2. Numerical Integration
   - experimental setup
   - adding type declarations
   - cdef functions & calling external functions
3. Vectorization
   - the game of life of John Conway
   - performance issues
   - vectorizing the neighbor count
   - vectorizing the rules

the `where` method

Converting a 0/1 matrix into a Boolean matrix:

```
>>> A = np.random.rand(4,4)
>>> A = np.matrix(A < 0.5, int)
>>> A
matrix([[0, 1, 1, 1],
        [0, 1, 0, 1],
        [1, 1, 1, 0],
        [0, 0, 1, 1]])
>>> B = np.where(A > 0, True, False)
>>> B
matrix([[False,  True,  True,  True],
        [False,  True, False,  True],
        [ True,  True,  True, False],
        [False, False,  True,  True]], dtype=bool)
```

die of loneliness

```
>>> import numpy as np
>>> A = np.random.rand(5,5)
>>> A = np.matrix(A < 0.4,int)
>>> A
matrix([[0, 0, 0, 0, 0],
        [1, 1, 0, 0, 1],
        [1, 0, 0, 1, 0],
        [0, 0, 0, 1, 0],
        [0, 0, 0, 0, 1]])
>>> from game_neighbors import neighbor_count_matrix
>>> B = neighbor_count_matrix(A)
>>> B
array([[2, 2, 1, 1, 1],
       [2, 2, 2, 2, 1],
       [2, 3, 3, 2, 3],
       [1, 1, 2, 2, 3],
       [0, 0, 1, 2, 1]])
>>> lonely = np.where(B < 2,0,A); lonely
array([[0, 0, 0, 0, 0],
       [1, 1, 0, 0, 0],
       [1, 0, 0, 1, 0],
       [0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0]])
```

the script `game_rules.py`

```
import numpy as np
from game_neighbors import neighbor_count_matrix

def update_matrix(alive, count):
    """
    The vectorized version of update.
    """
    starve = np.where(count > 3, 0, alive)
    lonely = np.where(count < 2, 0, starve)
    return np.where(count == 3, 1, lonely)
```

To test, we compare the update methods.

testing the vectorized rules

```
def main():
    """
    Test on the rules, comparing the original
    with the vectorized version of the rules.
    """
    dim = 10
    ratio = 0.3
    alive = np.random.rand(dim, dim)
    alive = np.matrix(alive < ratio, int)
    count = neighbor_count_matrix(alive)
    first_update = update(alive, count)
    print 'live cells :\n', alive
    print 'apply original rules :\n', first_update
    second_update = update_matrix(alive, count)
    print 'apply matrix rules :\n', second_update
    print 'equality check :'
    print np.equal(first_update, second_update)
    print sum(sum(np.equal(first_update, second_update)))
```

the vectorized game

The script `game_vector.py` has a different update:

```
import numpy as np
import matplotlib as plt
from scipy import sparse

from game_neighbors import neighbor_count_matrix
from game_rules import update_matrix

def update(alive):
    """
    Applies the rules of Conway's game of life.
    """
    counts = neighbor_count_matrix(alive)
    return update_matrix(alive, counts)
```

Summary + Exercises

Compiling modified Python code with Cython we obtain significant speedups and performance close to native C code.
Compared to vectorization we can often keep the same logic.
Random walks and cellular automata simulate
and model complicated dynamical systems.
Vectorization is often critical for good performance.

1. Read the Cython tutorial and learn how to extend the function for the composite trapezoidal rule so we may call the function given as argument the integrand function.
2. Use the canvas of Tkinter for a GUI to make the game of life. Allow the user to define configurations of living cells via mouse clicks on the canvas.
3. Apply vectorization to your GUI. Compare the performance of your versions with and without factorization.