Running Cython

1. Getting Started with Cython
   - overview
   - hello world with Cython
2. Numerical Integration
   - experimental setup
   - adding type declarations
   - cdef functions & calling external functions
3. Using Cython with NumPy
   - fast array access
   - adding type declarations

MCS 507 Lecture 30
Mathematical, Statistical and Scientific Software
Jan Verschelde, 4 November 2013

Running Cython

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.

Cython arose from the needs of the Sage project.

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 0.19.2 (2013-10-13) 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

1. Getting Started with Cython
   - overview
   - hello world with Cython
2. Numerical Integration
   - experimental setup
   - adding type declarations
   - cdef functions & calling external functions
3. Using Cython with NumPy
   - fast array access
   - adding type declarations

hello world with Cython

We can compile Cython code in three ways:

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

We illustrate the three 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

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

and this will make the shared object file `hello.so`.

```
$ python 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-2.7
gcc-4.2 -fno-strict-aliasing -fno-common -dynamic -isysroot
gcc-4.2 -bundle -undefined dynamic_lookup -arch i386 -arch
$
```

testing the Cython module

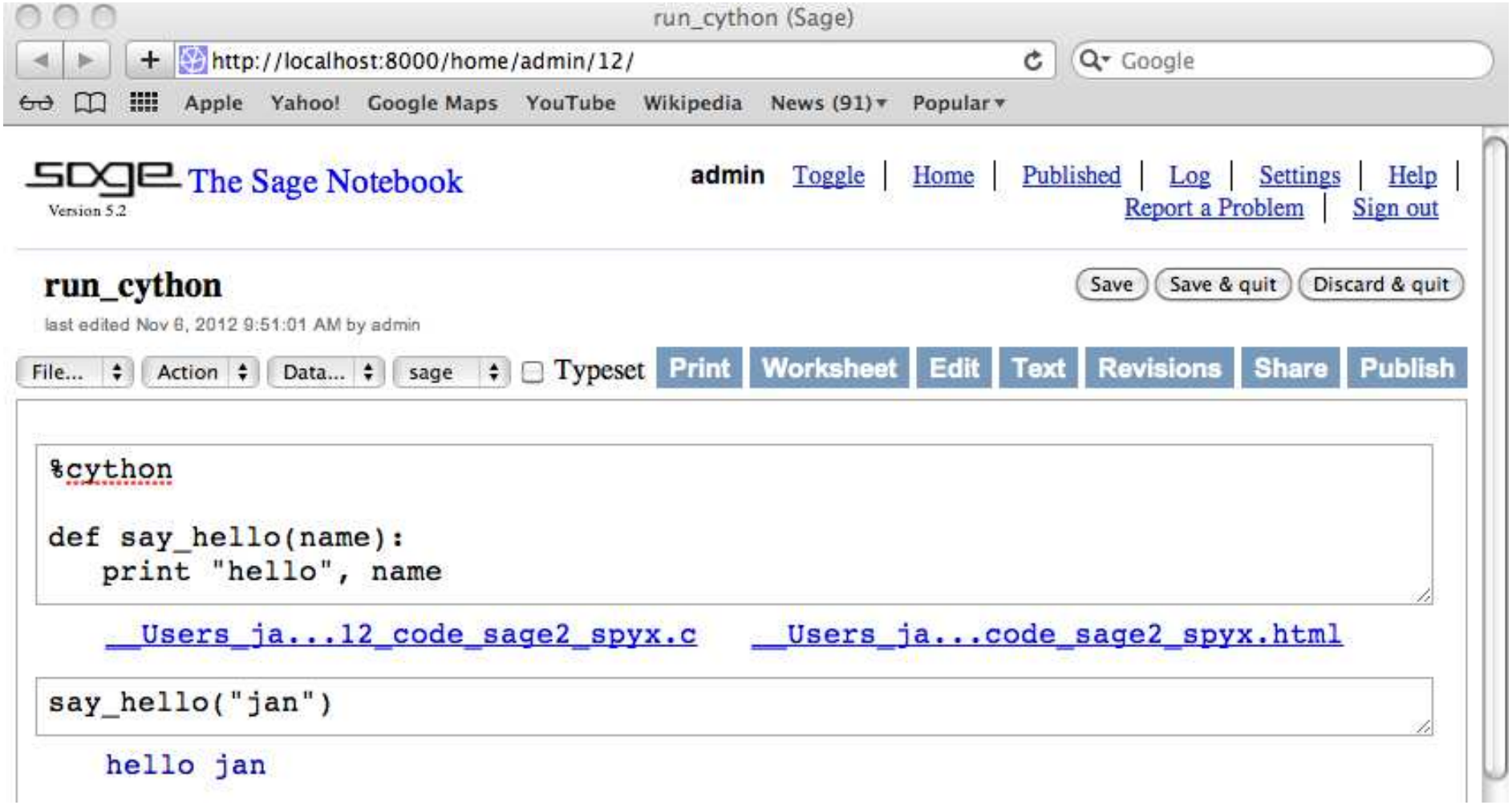
With the `--inplace` option, the shared object file `hello.so` is placed in the current directory.

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

```
$ python
Python 2.7.3 (v2.7.3:70274d53c1dd, Apr  9 2012, 20:52:
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for m
>>> from hello import say_hello
>>> say_hello("jan")
hello jan
>>>
```

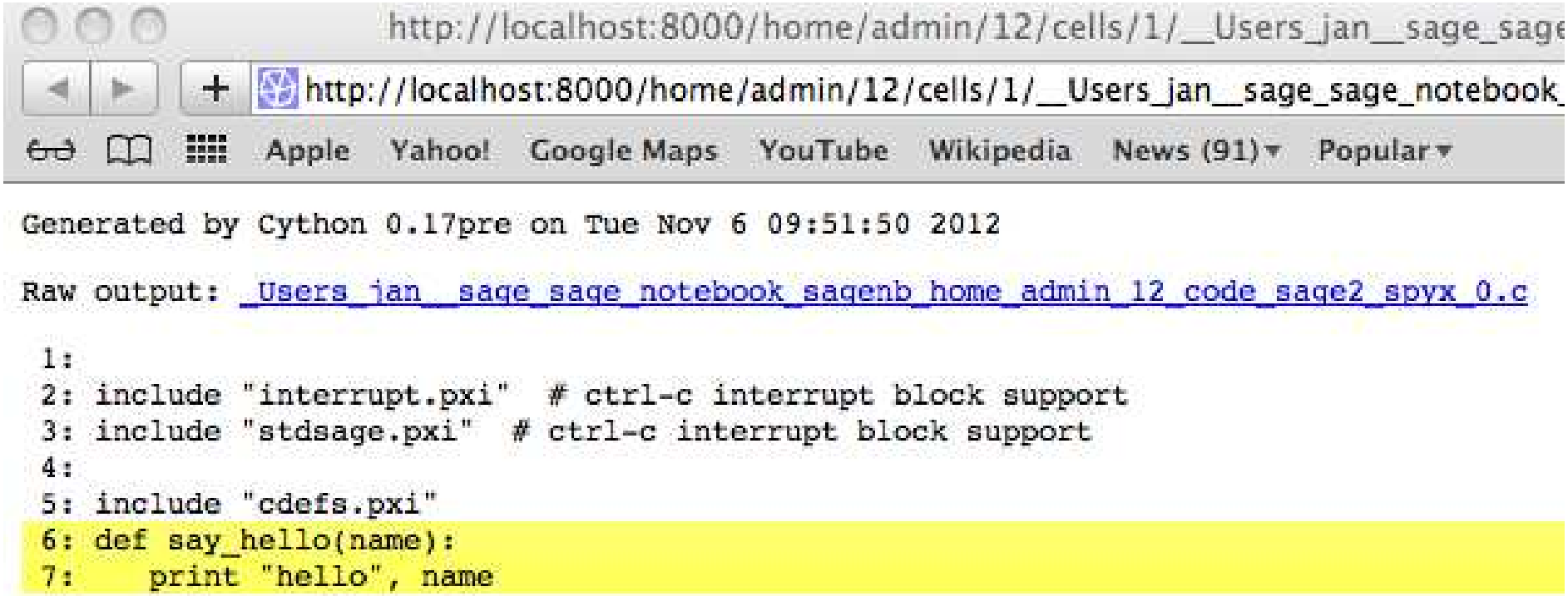
Cython in a Sage notebook

Put `%cython` in front of Cython code and press evaluate:



Clicking on the links shows the generated C code.

the generated code



the command line cython

```
$ which cython
/Library/Frameworks/Python.framework/Versions/2.7/bin/cython
$ cython hello.pyx
$ ls -lt hello.c
-rw-r--r--  1 jan  staff  72904 Nov  2 18:00 hello.c
$
```

The makefile contains the entry

```
hello_cython:
        cython hello.pyx
        gcc-4.2 -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/2.7/include/python2.7
        gcc-4.2 -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

1. Getting Started with Cython
   - overview
   - hello world with Cython
2. Numerical Integration
   - experimental setup
   - adding type declarations
   - cdef functions & calling external functions
3. Using Cython with NumPy
   - fast array access
   - adding type declarations

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 2.5 GHz Intel Core i5 MacBook Pro:

```
$ python integral4pi.py
pi = 3.14159305355
elapsed time = 4.837 seconds
$
```

Running Cython

1. Getting Started with Cython
   - overview
   - hello world with Cython
2. Numerical Integration
   - experimental setup
   - adding type declarations
   - cdef functions & calling external functions
3. Using Cython with NumPy
   - fast array access
   - adding type declarations

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

```
$ python integral4pi_typed_setup.py build_ext --inplace
running build_ext
cythoning integral4pi_typed.pyx to integral4pi_typed.c
building 'integral4pi_typed' extension
gcc-4.2 -fno-strict-aliasing -fno-common -dynamic -arch i38
-arch x86_64 -g -O2 -DNDEBUG -g -O3 -I/Library/Frameworks/
Python.framework/Versions/2.7/include/python2.7
-c integral4pi_typed.c -o build/temp.macosx-10.6-intel-2.7/
integral4pi_typed.o
gcc-4.2 -bundle -undefined dynamic_lookup -arch i386
-arch x86_64 -g build/temp.macosx-10.6-intel-2.7/
integral4pi_typed.o -o /Users/jan/Courses/MCS507/Fall13/
Lec30/integral4pi_typed.so
$
```

Notice the `-O2` and the `-O3` flags in the compilation phase.

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:

```
$ python integral4pi_typed_apply.py
pi = 3.14159305355
elapsed time = 1.479 seconds
$
```

The code runs twice as fast as the original Python version.

Running Cython

1. Getting Started with Cython
   - overview
   - hello world with Cython
2. Numerical Integration
   - experimental setup
   - adding type declarations
   - cdef functions & calling external functions
3. Using Cython with NumPy
   - fast array access
   - adding type declarations

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

```
$ python 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`.

```
$ python integral4pi_cdeffun_apply.py
pi = 3.14159305355
elapsed time = 0.777 seconds
$
```

What have we achieved so far is summarized below:

	elapsed seconds	speedup
original Python	4.837	1.00
Cython with cdef	1.479	3.27
cdef function	0.777	6.23

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

```
$ python 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`.

```
$ python integral4pi_extcfun_apply.py
pi = 3.14159305355
elapsed time = 0.123 seconds
$
```

This gives a nice speedup, summarized below:

	elapsed seconds	speedup
original Python	4.837	1.00
Cython with cdef	1.479	3.27
cdef function	0.777	6.23
external C function	0.123	39.33

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

```
$ gcc -O3 integral4pi_native.c -o /tmp/integral4pi_native
$ /tmp/integral4pi_native
pi = 3.14159053552711
elapsed time = 0.050 seconds
$
```

	elapsed seconds	speedup
original Python	4.837	1.00
Cython with cdef	1.479	3.27
cdef function	0.777	6.23
external C function	0.123	39.33
native C code	0.050	96.74

Despite the double digit speedups,
Cython code is still more than twice as slow as native C code.

Running Cython

1. Getting Started with Cython
   - overview
   - hello world with Cython
2. Numerical Integration
   - experimental setup
   - adding type declarations
   - cdef functions & calling external functions
3. Using Cython with NumPy
   - fast array access
   - adding type declarations

fast array access

Cython has support for fast access to NumPy arrays.

Requirements for fast access:

- Data type and dimensions must be fixed at compile time and declared as of `ndarray` type.
- The indices are declare of type `Py_size_t`.

Recall the game of life of John Conway: to update the matrix, we count the number of neighboring live cells.

Vectorization is effective, but may alter the logic of the script drastically so that its correctness is no longer obvious.

Goal: a faster game of life with ***same*** logic in the code.

experimental setup

We take the original Python code (without vectorization) of lecture 29 and remove the visualization.

On a random matrix of some dimension *n*
we perform the update *n* times and take the running time:

```
$ python run_game_of_life.py
give the dimension : 100
elapsed time 4.743604 seconds
$
```

Also done on a 2.5 GHz Intel Core i5 MacBook Pro.

the `main()` of the script `run_game_of_life.py`

```
def main():
    """
    Generates a random matrix and applies
    the rules for Conway's game of life.
    """
    ratio = 0.2  # ratio of nonzeroes
    # dim = 100  # dimension of the matrix
    dim = input('give the dimension : ')
    alive = np.random.rand(dim, dim)
    alive = np.matrix(alive < ratio, int)
    start_time = clock()
    for i in range(dim):
        alive = update(alive)
    stop_time = clock()
    elapsed = stop_time - start_time
    print 'elapsed time', elapsed, 'seconds'
```

Running Cython

1. Getting Started with Cython
   - overview
   - hello world with Cython
2. Numerical Integration
   - experimental setup
   - adding type declarations
   - cdef functions & calling external functions
3. Using Cython with NumPy
   - fast array access
   - adding type declarations

counting the live neighbors

Code in the script `run_game_better.pyx`:

```
import numpy as np
cimport numpy as np
ctypedef np.uint8_t dtype_t

def neighbors(np.ndarray[dtype_t, ndim=2] alive,
              Py_ssize_t i, Py_ssize_t j):
    """
    Returns number of cells alive next to alive[i, j].
    """
    cdef dtype_t cnt = 0
```

The rest of the function remains the same.

updating the matrix

```
def update(np.ndarray[dtype_t, ndim=2] alive):
    """
    Applies the rules of Conway's game of life.
    """
    cdef Py_ssize_t i, j
    cdef np.ndarray[dtype_t, ndim=2] result
    cdef dtype_t nbn

    result = np.zeros(alive, np.uint8)
```

The rest of the function remains the same.

building with `distutils`

The script `run_game_better_setup.py` has content:

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

import numpy

EXT_MODULES = [Extension("run_game_better", \
    sources=["run_game_better.pyx"], \
    include_dirs=[numpy.get_include()])]

setup(
    name = 'game of life', \
    cmdclass = {'build_ext': build_ext}, \
    ext_modules = EXT_MODULES
)
```

using the `run_game_better` module

```
from time import clock
import numpy as np
from run_game_better import update

def main():
    """
    Generates a random matrix and applies
    the rules for Conway's game of life.
    """
    ratio = 0.2  # ratio of nonzeroes
    # dim = 100  # dimension of the matrix
    dim = input('give the dimension : ')
    alive = np.random.rand(dim, dim)
    alive = np.matrix(alive < ratio, np.uint8)
    start_time = clock()
    for i in range(dim):
        alive = update(alive)
    stop_time = clock()
    elapsed = stop_time - start_time
    print 'elapsed time', elapsed, 'seconds'
```

running run_game_better_apply.py

After running

```
$ python run_game_better_setup.py build_ext --inplace
```

do

```
$ python run_game_better_apply.py
give the dimension : 100
elapsed time 0.508983 seconds
$
```

The speedup: 4.744/0.509 = 9.32 times faster.

a visual check on correctness

```
import numpy as np
from scitools.std import plot
from scipy import sparse
from run_game_better import update

def main():
    """
    Generates a random matrix and applies
    the rules for Conway's game of life.
    """
    ratio = 0.2  # ratio of nonzeroes
    # dim = 100  # dimension of the matrix
    dim = input('give the dimension : ')
    alive = np.random.rand(dim, dim)
    alive = np.matrix(alive < ratio, np.uint8)
    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)
```

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.

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. Compare the performance of the vectorized code of lecture 30 with `run_game_better_apply.py`. Which one is faster? Explain why.
3. Can you improve `run_game_better.pyx`?

Homework Five due Friday 15 November at 9AM:
exercises 2 and 3 of lecture 21; exercise 1 of lecture 22;
exercise 2 of lecture 23; exercises 1 and 2 of lecture 24;
exercises 1 and 2 of lecture 25; exercises 1 and 3 of lecture 26.