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
4. Numba
   - a just-in-time compiler for Python

MCS 507 Lecture 15
Mathematical, Statistical and Scientific Software
Jan Verschelde, 25 September 2023

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
4. Numba
   - a just-in-time compiler for Python

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.

W. Stein: **Sage for Power Users**, 2012.
`wstein.org/books/sagebook/sagebook.pdf`

The web page of the project is at `cython.org`.

The pip3 installs `cython-3.0.2`. It works on Windows.

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
4. Numba
   - a just-in-time compiler for Python

hello world with Cython

We can compile Cython code in two ways:

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

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

```
# cython: language_level=3

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

The line on top indicates we use python3 and not python2.

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
gcc -Wno-unused-result ... -c hello.c ... -o ... /hello.o
gcc -bundle ... /hello.o -o ... .so
$
```

The . . . are omitted compiler flags and directories.
These specifics are useful for the command line use.

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 3.7.3 (default, Mar 27 2019, 16:54:48)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, In
Type "help", "copyright", "credits" or "license" for m
>>> from hello import say_hello
>>> say_hello('me')
hello me
>>>
```

the command line cython

```
$ which cython
/Users/jan/miniconda3/bin/cython
$ cython hello.pyx
```

Windows users: replace `which cython` by `get-command cython`.
The `cython hello.pyx` produces the C code `hello.c`.

The generation of the C code and the compilation
is better automated with a makefile.

Typing `make hello_cython` at the command line then

1. executes `cython`,
2. compiles the `hello.c`, and
3. links the object `hello.o` into the shared object `hello.so`.

an example of a makefile

```
hello_cython:
        cython hello.pyx
        gcc -c hello.c -o hello.o \
            -Wno-unused-result -Wsign-compare \
            -Wunreachable-code -DNDEBUG -g \
            -fwrapv -O3 -Wall -Wstrict-prototypes \
            -I/Users/jan/miniconda3/include -arch x86_64
            -I/Users/jan/miniconda3/include/python3.7m
        gcc -bundle -undefined dynamic_lookup \
            -L/Users/jan/miniconda3/lib \
            -arch x86_64 -L/Users/jan/miniconda3/lib \
            -arch x86_64 -L/usr/local/opt/libffi/lib \
            hello.o -o 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

4 Numba
- a just-in-time compiler for Python

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 $n = 10^8$ and make 100,000,000 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 perf_counter
    start_time = perf_counter()
    approx = integral4pi(10**8)
    stop_time = perf_counter()
    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:

```
$ python integral4pi.py
pi = 3.141592693586875
elapsed time = 28.129 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
4. Numba
   - a just-in-time compiler for Python

the script `integral4pi_typed.pyx`

```
# cython: language_level=3

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 ... -O3 ...
$
```

Notice the `-O3` flag in the compilation phase.

calling `integral4pi` of `integral4pi_typed`

```
from time import perf_counter
from integral4pi_typed import integral4pi

START_TIME = perf_counter()
APPROX = integral4pi(10**8)
STOP_TIME = perf_counter()
print('pi =', APPROX)
ELAPSED = STOP_TIME - START_TIME
print('elapsed time = %.3f seconds' % ELAPSED)
```

Running the script:

```
$ python integral4pi_typed_apply.py
pi = 3.141592693586875
elapsed time = 5.728 seconds
$
```

The previous run took 28.129 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
4. Numba
   - a just-in-time compiler for Python

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 = 3.141592693586875
elapsed time = 2.718 seconds
$
```

What have we achieved so far is summarized below:

	elapsed seconds	speedup
original Python	28.129	1.00
Cython with cdef	5.728	4.91
cdef function	2.718	10.35

calling external C functions

The main cost is calling `sqrt` 100,000,000 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 = 0.222 seconds
$
```

This gives a nice speedup, summarized below:

	elapsed seconds	speedup
original Python	28.129	1.00
Cython with cdef	5.728	4.91
cdef function	2.718	10.35
external C function	0.222	126.71

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 = 3.14159269358687 5
elapsed time = 0.224 seconds
$
```

	elapsed seconds	speedup
original Python	28.129	1.00
Cython with cdef	5.728	4.91
cdef function	2.718	10.35
external C function	0.222	126.71
native C code	0.224	125.58

The final Cython code is as fast as the 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
4. Numba
   - a just-in-time compiler for Python

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.

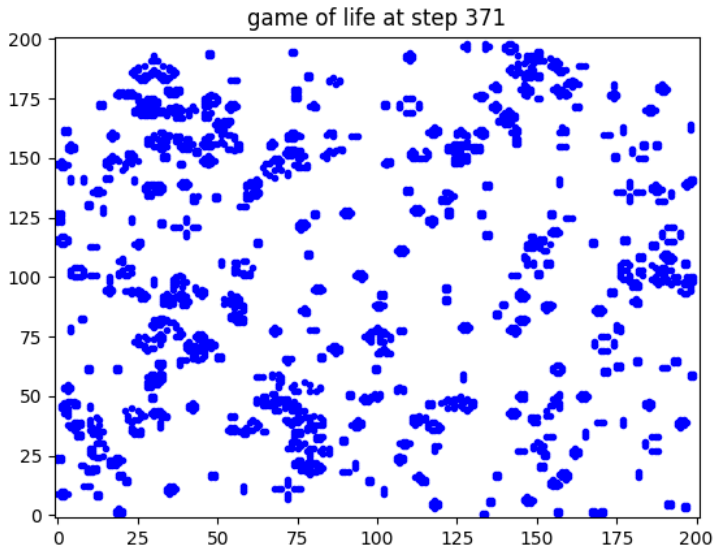
the rules of the game

The rules of the game of life of Conway are as follows.

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.

the animation



experimental setup

We take the original Python code (without vectorization) of lecture 4 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
elapsed time 20.542221478 seconds
$
```

Done on a 3.1 GHz Intel Core i7 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 = 200    # dimension of the matrix
    alive = np.random.rand(dim, dim)
    alive = np.matrix(alive < ratio, int)
    start_time = perf_counter()
    for i in range(dim):
        alive = update(alive)
    stop_time = perf_counter()
    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
4. Numba
   - a just-in-time compiler for Python

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 perf_counter
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 = 200    # dimension of the matrix
    alive = np.random.rand(dim, dim)
    alive = np.matrix(alive < ratio, np.uint8)
    start_time = perf_counter()
    for i in range(dim):
        alive = update(alive)
    stop_time = perf_counter()
    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
elapsed time 4.797007945 seconds
$
```

The speedup: 20.542/4.797 = 4.28 times faster.

visualization with matplotlib

```
import numpy as np
from scipy import sparse
import matplotlib
import matplotlib.pyplot as plt
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 = 200  # dimension of the matrix
    alive = np.random.rand(dim, dim)
    alive = np.matrix(alive < ratio, np.uint8)
```

using scipy sparse matrices

To show the cells that are alive, we construct a sparse matrix, in coordinate format. Each nonzero element of a matrix is stored by a triplet: (row, column, value).

```
plt.ion()
fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_xlim(-1, dim+1)
ax.set_ylim(-1, dim+1)
spm = sparse.coo_matrix(alive)
dots, = ax.plot(spm.row, spm.col, 'b.')
strtitle = 'game of life'
ax.set_title(strtitle)
fig.canvas.draw()
plt.pause(0.00001)
```

the script continued

Then the animation runs in the following loop:

```
for i in range(5*dim):
    spm = sparse.coo_matrix(alive)
    dots.set_xdata(spm.row)
    dots.set_ydata(spm.col)
    strtitle = 'game of life at step %d' % (i+1)
    ax.set_title(strtitle)
    fig.canvas.draw()
    plt.pause(0.00001)
    alive = update(alive)
```

timing the visualization

```
$ time python game_of_life.py

real     2m23.216s
user     2m20.564s
sys      0m2.240s

$ time python run_game_better_show.py

real     1m5.237s
user     1m2.324s
sys      0m2.337s
$
```

The visualization runs more than twice faster.

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
4. Numba
   - a just-in-time compiler for Python

numba

From the user manual of numba 0.51.2:

- Numba is a just-in-time compiler for Python that works best on code that uses NumPy arrays, functions, and loops.
- Numba is used through the addition of decorators applied to functions with instructions to compile them.
- The compiled code then runs at native machine code speed.
- Works with Cython.
- Supports multithreading and GPU acceleration.

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.

Exercises:

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 4 with `run_game_better_apply.py`.
   Which one is faster? Explain why.
3. Can you improve `run_game_better.pyx`?