ICHEC
Irish Centre for High-end Computing

PRACE

Introduction to Cython

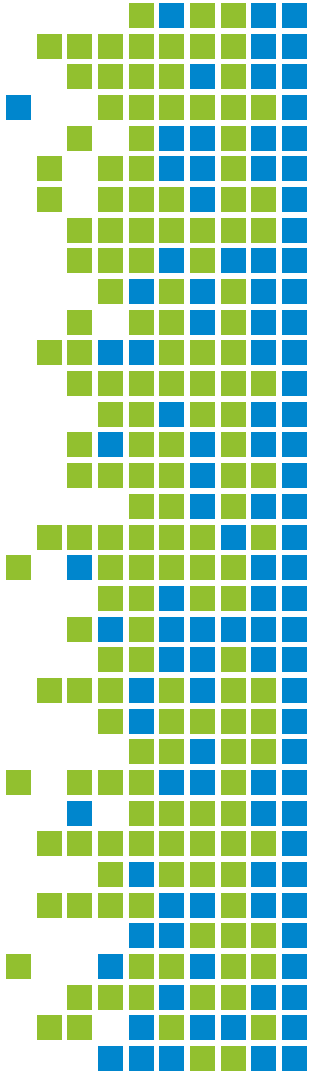
SoHPC 2021

James Nelson

ICHEC
Irish Centre for High-end Computing

Outline

1. Cython Fundamentals
2. Accelerating Cython
3. Putting Everything Together
4. Summary

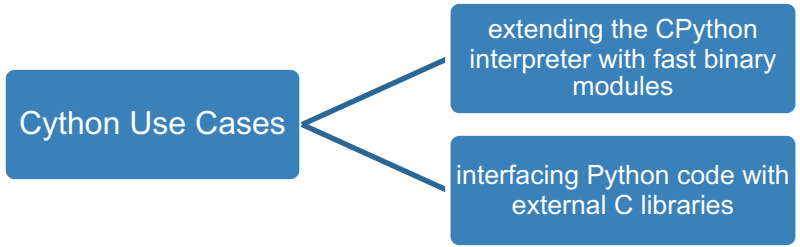


Cython Fundamentals

ICHEC
Irish Centre for High-end Computing

What is Cython?

- **Cython** is a programming language that makes writing C extensions for the Python language as easy as Python itself. The source code gets **translated into optimised C/C++ code** and **compiled as Python extension modules**.
- The code is executed in the CPython runtime environment, but at the speed of compiled C with the ability to call directly into C libraries, whilst keeping the original interface of the Python source code.



- **Cython IS Python**, just with C data types (so some basic C knowledge is recommended)

Typing

Cython supports **static type declarations**, thereby turning readable Python code into plain C performance.

Static Typing:

- Type checking is performed during compile-time
- e.g. `x = 4 + 'e'` would not compile
- Can detect type errors in rarely used code paths
- Must declare data types of variables
- Examples: C, C++, Java

Dynamic Typing:

- Type checking is performed during run-time
- e.g. `x = 4 + 'e'` would result in a runtime type error
- Don’t have to declare data types of variables
- Examples: Python

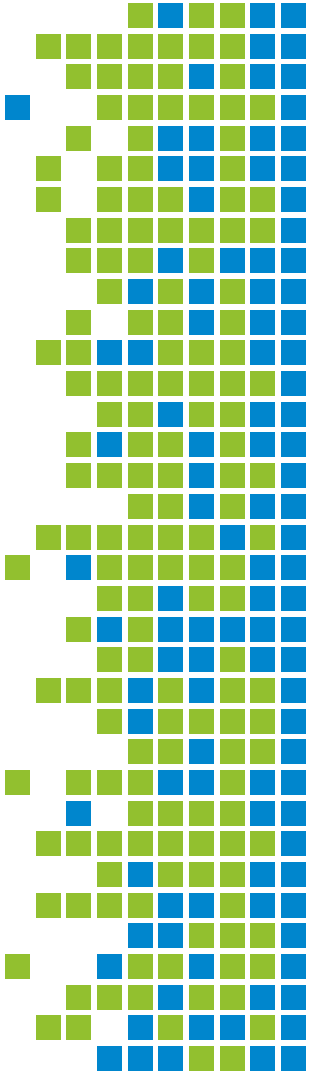
This allows for fast program execution and tight integration with external C libraries.

ICHEC
Irish Centre for High-end Computing

Implementing Cython

Three Steps:

1. Write the python code (which is to be Cythonized) to a .pyx file
2. Create a setup.py file to compile the code to a .so file (shared object)
3. Import the shared object



ICHEC
Irish Centre for High-end Computing

Implementing Cython

```
def fib_func(n):
    # Prints the Fibonacci series up to n.
    a, b = 0, 1
    while b < n:
        print(b)
        a, b = b, a + b
```

This function is written to fib.pyx

Demo 1

ICHEC
Irish Centre for High-end Computing

Implementing Cython

- The next step is to create the setup.py file
- Throughout this tutorial it will always have the same format

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
    ext_modules=cythonize(<file_name>)
    )
```

In this case the file name will be "fib.pyx"

Demo 1

Implementing Cython

- Now we compile the code

```
python setup.py build_ext --inplace
```

We use build_ext --inplace to compile the extension for use in the current directory

- This creates a:
  - .c file, which is then compiled using a C compiler
  - build directory which contains the .o file generated by the compiler
  - .so file. The compiled library file

Demo 1

ICHEC
Irish Centre for High-end Computing

Implementing Cython

- Finally import the Cython file
- Just like importing a python module

```
from fib import fib_func

fib_func(10)
```

Demo 1

Implementing Cython

```
def fib_func(n):
    # Prints the Fibonacci series up to n.
    a, b = 0, 1
    while b < n:
        print(b)
        a, b = b, a + b
```

fib.pyx

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
    ext_modules=cythonize("fib.pyx")
    )
```

setup.py

```
python setup.py build_ext --inplace
```

```
from fib import fib_func

fib_func(10)
```

Demo 1

Accelerating Cython

Cython Features

- Compiling with Cython is fine, but it doesn't speed up our code significantly
- We need to implement the C-features that Cython was designed for:
  - Static type declarations
  - Typing Function Calls
  - NumPy Arrays with Cython
  - Compiler Directives

Static type declarations

- These allow Cython to step out of the dynamic nature of the Python code and generate simpler and faster C code - sometimes faster by orders of magnitude.
- This is often the simplest and quickest way to achieve significant speedup, but the code can become more verbose and less readable
- Types are declared with the cdef keyword

ICHEC
Irish Centre for High-end Computing

Static type declarations

no cdef for function arguments

```
def pi_montecarlo(n=1000):

    in_circle = 0
    for i in range(n):
        x, y = random(), random()
        if x ** 2 + y ** 2 <= 1.0:
            in_circle += 1

    return 4.0 * in_circle / n
```

pi_estimator.py

cdef type

```
def pi_montecarlo(int n = 1000):

    cdef int in_circle = 0, i
    cdef double x, y
    for i in range(n):
        x, y = random(), random()
        if x ** 2 + y ** 2 <= 1.0:
            in_circle += 1

    return 4.0 * in_circle / n
```

pi_estimator_cython.pyx

Demo 2

Typing Function Calls

- As with 'typing' variables, you can also 'type' functions.
- Function calls in Python can be expensive, and can be even more expensive in Cython as one might need to convert to and from Python objects to do the call.
- There are two ways in which to declare C-style functions in Cython:
  - Declaring a C-type function - cdef
  - Creation of a Python wrapper – cpdef
- Both are implemented the same way (same as declaring a variable)
- A side-effect of cdef is that the function is no longer available from Python-space, so Python won't know how to call it
- Using cpdef allows the function to be available from both Python and Cython, although cdef is more efficient

ICHEC
Irish Centre for High-end Computing

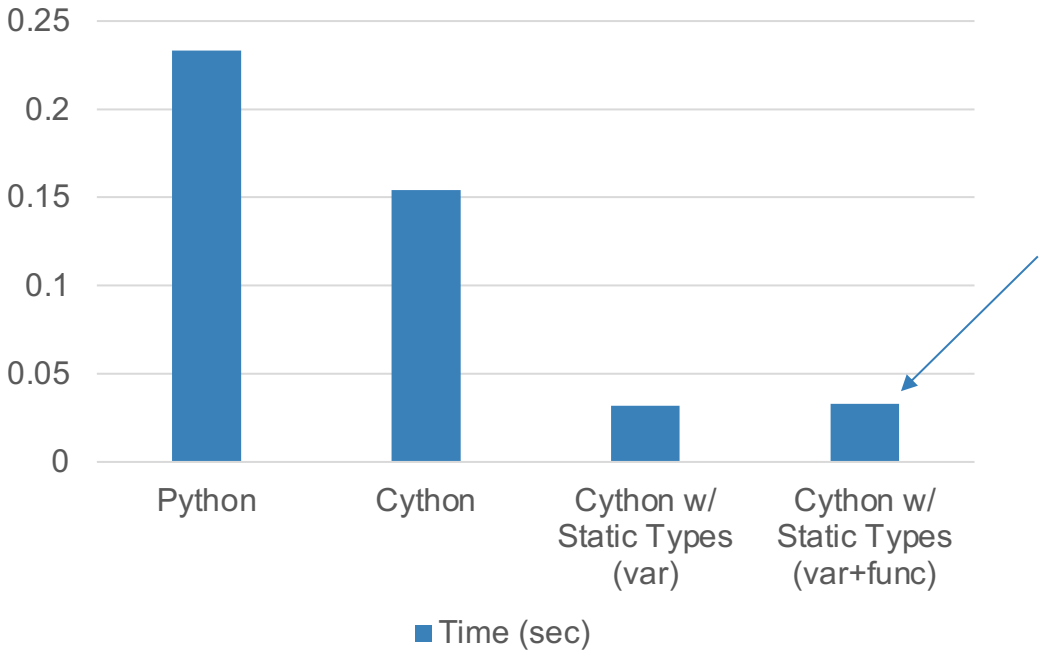
Typing Function Calls

```
cpdef double pi_montecarlo(int n = 1000):

    cdef int in_circle = 0, i
    cdef double x, y
    for i in range(n):
        x, y = random(), random()
        if x ** 2 + y ** 2 <= 1.0:
            in_circle += 1

    return 4.0 * in_circle / n
```

Here we use cpdef as we are calling the function in the python code

Demo 2

Pi Estimation Results (N=1000000)



As the function is not being repeatedly called, typing it doesn't improve the efficiency

When to add static types?

- For those of new to Cython and the concept of declaring types, there is a tendancy to 'type' everything in sight. This reduces readability and flexibility and in certain situations, even slow things down.
- It is also possible to kill performance by forgetting to 'type' a critical loop variable. Tools we can use are **profiling** and **annotation**.
- Profiling is the first step of any optimisation effort and can tell you where the time is being spent. By default, Cython code does not show up in profile produced by cProfile. Profiling can be enabled by including in the first line `# cython: profile=True`
- Cython's annotation creates a HTML report of Cython and generated C code which can tell you why your code is taking so long.
- To create the report pass the `annotate=True` parameter to `cythonize()` in the setup.py file (Note, you may have to delete the c file and compile again to produce the HTML report).

ICHEC
Irish Centre for High-end Computing

Cython annotation

```
Generated by Cython 0.29.21

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: pi_estimator_cython.c

+01: from random import random
 02:
+03: def pi_montecarlo_cyt1(n=1000):
 04:     '''Calculate PI using Monte Carlo method'''
+05:     in_circle = 0
+06:     for i in range(n):
+07:         x, y = random(), random()
+08:         if x ** 2 + y ** 2 <= 1.0:
+09:             in_circle += 1
 10:
+11:     return 4.0 * in_circle / n
 12:
 13:
+14: def pi_montecarlo_cyt2(int n=1000):
 15:     '''Calculate PI using Monte Carlo method'''
+16:     cdef int in_circle = 0, i
 17:     cdef double x, y
+18:     for i in range(n):
+19:         x, y = random(), random()
+20:         if x ** 2 + y ** 2 <= 1.0:
+21:             in_circle += 1
 22:
+23:     return 4.0 * in_circle / n
 24:
 25:
+26: cpdef double pi_montecarlo_cyt3(int n=1000):
 27:     '''Calculate PI using Monte Carlo method'''
+28:     cdef int in_circle = 0, i
 29:     cdef double x, y
+30:     for i in range(n):
+31:         x, y = random(), random()
+32:         if x ** 2 + y ** 2 <= 1.0:
+33:             in_circle += 1
 34:
+35:     return 4.0 * in_circle / n
```

```
from distutils.core import setup
from Cython.Build import cythonize

setup(ext_modules=cythonize("pi_esti
mator_cython.pyx", annotate=True))
```

Demo 3

ICHEC
Irish Centre for High-end Computing

NumPy Arrays with Cython

- If you are dealing with numpy arrays, you can type the contents of the array to speed up the runtime

```
cimport numpy as cnp
cdef cnp.ndarray[cnp.int_t, ndim=2] <array_name>
```

Datatype (for each type in numpy there is a corresponding type with _t)

Number of dimensions

- Note the array still needs to be initialized e.g. <array_name> = np.zeros(2,3)

ICHEC
Irish Centre for High-end Computing

NumPy Arrays with Cython

```
import numpy as np

def powers_array(N):
    data = np.arange(N*N).reshape(N,N)
    for i in range(N):
        for j in range(N):
            data[i,j] = i**j
    return(data[2])
```

```
import numpy as np
cimport numpy as cnp

def powers_array_cy(int N):
    cdef cnp.ndarray[cnp.int_t, ndim=2] data
    data = np.arange(N*N).reshape((N, N))
    for i in range(N):
        for j in range(N):
            data[i,j] = i**j
    return(data[2])
```

Demo 4

ICHEC
Irish Centre for High-end Computing

NumPy Arrays with Cython

- If using Cython-NumPy functionality we must modify the setup.py file

```
from distutils.core import setup
from Cython.Build import cythonize
import numpy


setup(ext_modules=cythonize("pow_cyt.pyx"),
      include_dirs=[numpy.get_include()]
      )
```

Demo 4

Compiler Directives

- These affect the code in a way to get the compiler to ignore things that it would usually look out for.
- boundscheck - If set to False, Cython is free to assume that indexing operations in the code will not cause any IndexErrors to be raised
- wraparound - If set to False, Cython is allowed to neither check for nor correctly handle negative indices. This can cause data corruption or segmentation faults if mishandled.

To implement

```
# cython: boundscheck=False
# cython: wraparound=False
```

Header comment at the top of a .pyx file (must appear before any code)

```
cimport cython
@cython.boundscheck(False) # turns off
@cython.wraparound(False)
```

Locally for specific functions (need cython module imported)

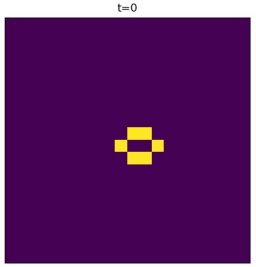
Putting Everything Together

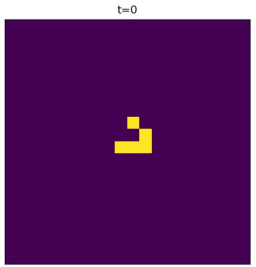
Conway's Game of Life

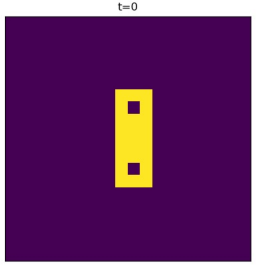
ICHEC
Irish Centre for High-end Computing

Conway's Game of Life

- 2D grid of cells
- Each cell is either alive (1) or dead (0)
- An alive cell will stay alive if it has 2 or 3 neighbours, else it will die
- A dead cell will become alive if it has 3 neighbours, else it stays dead
- The neighbours are the nearest 8 cells

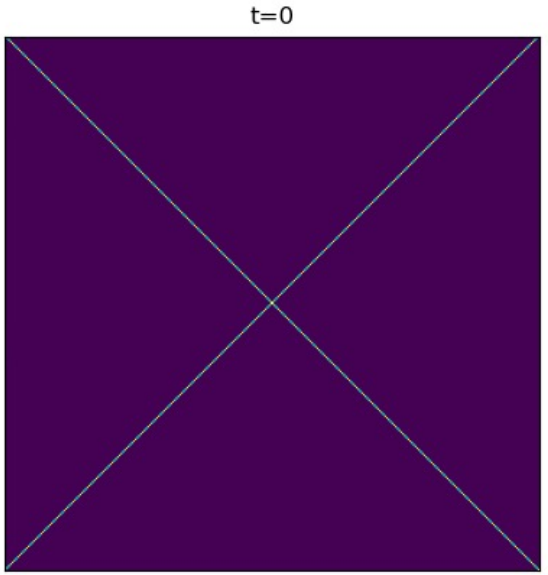






Conway's Game of Life

- Initialize as a cross
- Box size=300
- How long to update the lattice 300 times?



ICHEC
Irish Centre for High-end Computing

Python

```
def update(lattice):
    box_length = len(lattice) - 2
    lattice_new = [[0 for _ in range(box_length + 2)] for _ in range(box_length + 2)]

    for i in range(1, box_length + 1):
        for j in range(1, box_length + 1):
            lattice_new[i][j] = update_rule(lattice, i, j)
    return lattice_new


def update_rule(lattice, i, j):
    n_neigh = lattice[i + 1][j] + lattice[i][j + 1] + lattice[i + 1][j + 1] + \
              lattice[i + 1][j - 1] + lattice[i - 1][j] + lattice[i][j - 1] + \
              lattice[i - 1][j + 1] + lattice[i - 1][j - 1]

    if (lattice[i][j] == 1) and (n_neigh in [2, 3]):
        return 1
    elif lattice[i][j] == 1:
        return 0
    elif (lattice[i][j] == 0) and (n_neigh == 3):
        return 1
    else:
        return 0
```

Demo 5

ICHEC
Irish Centre for High-end Computing

Cython 1

```
def update(lattice):
    box_length = len(lattice) - 2
    lattice_new = [[0 for _ in range(box_length + 2)] for _ in range(box_length + 2)]

    for i in range(1, box_length + 1):
        for j in range(1, box_length + 1):
            lattice_new[i][j] = update_rule(lattice, i, j)
    return lattice_new


def update_rule(lattice, i, j):
    n_neigh = lattice[i + 1][j] + lattice[i][j + 1] + lattice[i + 1][j + 1] + \
              lattice[i + 1][j - 1] + lattice[i - 1][j] + lattice[i][j - 1] + \
              lattice[i - 1][j + 1] + lattice[i - 1][j - 1]

    if (lattice[i][j] == 1) and (n_neigh in [2, 3]):
        return 1
    elif lattice[i][j] == 1:
        return 0
    elif (lattice[i][j] == 0) and (n_neigh == 3):
        return 1
    else:
        return 0
```

Demo 5

ICHEC
Irish Centre for High-end Computing

Cython 2

```
def update(lattice):
    cdef int box_length = len(lattice) – 2
    cdef int i, j
    lattice_new = [[0 for _ in range(box_length + 2)] for _ in range(box_length + 2)]

    for i in range(1, box_length + 1):
        for j in range(1, box_length + 1):
            lattice_new[i][j] = update_rule(lattice, i, j)
    return lattice_new


def update_rule(lattice, int i, int j):
    cdef int n_neigh
    n_neigh = lattice[i + 1][j] + lattice[i][j + 1] + lattice[i + 1][j + 1] + \
              lattice[i + 1][j - 1] + lattice[i - 1][j] + lattice[i][j - 1] + \
              lattice[i - 1][j + 1] + lattice[i - 1][j - 1]

    if (lattice[i][j] == 1) and (n_neigh in [2, 3]):
        return 1
    elif lattice[i][j] == 1:
        return 0
    elif (lattice[i][j] == 0) and (n_neigh == 3):
        return 1
    else:
        return 0
```

Demo 5

ICHEC
Irish Centre for High-end Computing

Cython 3

```
def update(lattice):
    cdef int box_length = len(lattice) – 2
    cdef int i, j
    lattice_new = [[0 for _ in range(box_length + 2)] for _ in range(box_length + 2)]

    for i in range(1, box_length + 1):
        for j in range(1, box_length + 1):
            lattice_new[i][j] = update_rule(lattice, i, j)
    return lattice_new


cdef int update_rule(lattice, int i, int j):
    cdef int n_neigh
    n_neigh = lattice[i + 1][j] + lattice[i][j + 1] + lattice[i + 1][j + 1] + \
              lattice[i + 1][j - 1] + lattice[i - 1][j] + lattice[i][j - 1] + \
              lattice[i - 1][j + 1] + lattice[i - 1][j - 1]

    if (lattice[i][j] == 1) and (n_neigh in [2, 3]):
        return 1
    elif lattice[i][j] == 1:
        return 0
    elif (lattice[i][j] == 0) and (n_neigh == 3):
        return 1
    else:
        return 0
```

Demo 5

ICHEC
Irish Centre for High-end Computing

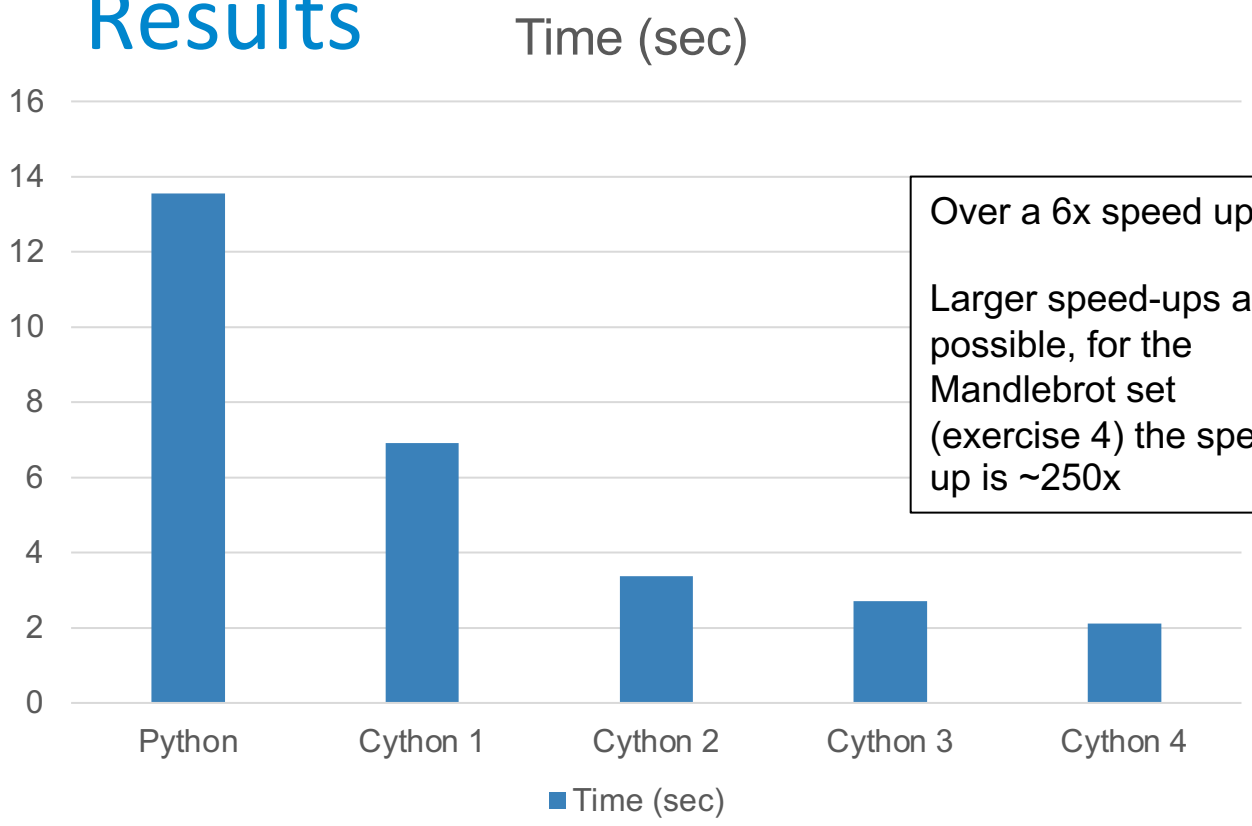
Cython 4

```
# cython: boundscheck=False
# cython: wraparound=False

def update(lattice):
    cdef int box_length = len(lattice) – 2
    cdef int i, j
    lattice_new = [[0 for _ in range(box_length + 2)] for _ in range(box_length + 2)]

    for i in range(1, box_length + 1):
        for j in range(1, box_length + 1):
            lattice_new[i][j] = update_rule(lattice, i, j)
    return lattice_new


cdef int update_rule(lattice, int i, int j):
    cdef int n_neigh
    n_neigh = lattice[i + 1][j] + lattice[i][j + 1] + lattice[i + 1][j + 1] + \
              lattice[i + 1][j - 1] + lattice[i - 1][j] + lattice[i][j - 1] + \
              lattice[i - 1][j + 1] + lattice[i - 1][j - 1]

    if (lattice[i][j] == 1) and (n_neigh in [2, 3]):
        return 1
    elif lattice[i][j] == 1:
        return 0
    elif (lattice[i][j] == 0) and (n_neigh == 3):
        return 1
    else:
        return 0
```

Demo 5

ICHEC
Irish Centre for High-end Computing

Results



Over a 6x speed up

Larger speed-ups are possible, for the Mandlebrot set (exercise 4) the speed up is ~250x

Summary

ICHEC
Irish Centre for High-end Computing

Cython Summary

- To run Cython:
  - Write code in .pyx
  - Compile the .pyx with setup.py
  - python setup.py build_ext –inplace
  - Import module as normal
- To speed up Cython:
  - Type variables
  - Type function calls
  - Use Cython's numpy functionality
  - Use Compiler directives