UNIVERSIDAD POLITÉCNICA DE YUCATÁN

BIS Universities

High-performance Computing

Extemporaneous

Students:

Hector De Jesus Fernandez Zanatta

26/04/2024

Report

Conway's Game of Life is a cellular automaton created by the British mathematician John Horton Conway in 1970. It's a zero-player game, meaning its evolution is determined by its initial state, requiring no further input from human players. It's a popular example of emergent behaviors, complex systems arising from simple rules.

The Game of Life is played on an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states: alive or dead. Each cell interacts with its eight neighbors, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

Birth: A dead cell with exactly three live neighbors becomes a live cell (as if by reproduction).
Survival: A live cell with two or three live neighbors stays alive (continues to live).
Death:
Overpopulation: A live cell with more than three live neighbors dies.
Loneliness: A live cell with fewer than two live neighbors dies as well.
The initial pattern constitutes the seed of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed—births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a tick. Each generation is a pure function of the preceding one, with the rules applied repeatedly to create further generations.

PYTHON

```
def update(lattice):
    # Determines the size of the 'lattice' minus the outer shell.
    box_length = len(lattice) - 2
    # Creates a new lattice with the same size as the input
lattice.
    lattice_new = [[0 for _ in range(box_length + 2)] for _ in
range(box_length + 2)]

    # Loops over each cell inside the lattice, excluding the outer
shell.
    for i in range(1, box_length + 1):
        for j in range(1, box_length + 1):
            # Updates the cell state according to the Game of Life
rules.
            lattice_new[i][j] = update_rule(lattice, i, j)

    # Returns the updated lattice.
    return lattice_new

def update_rule(lattice, i, j):
    # Counts the number of alive neighbors around the cell at
position (i, j).
    n_neigh = lattice[i + 1][j] + lattice[i][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] +
lattice[i - 1][j + 1]

    # Applies the Game of Life rules to determine the next state
of the cell.
    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

def print_lattice(lattice):
    for row in lattice:
```

```
        print(' '.join(['■' if cell else ' ' for cell in row]))

# Example to initialize a small lattice with a "glider" pattern
lattice = [
    [0, 0, 0, 0, 0],
    [0, 1, 0, 0, 0],
    [0, 0, 1, 1, 0],
    [0, 1, 1, 0, 0],
    [0, 0, 0, 0, 0]
]

# Run the game for 5 generations and print each generation
for _ in range(5):
    print_lattice(lattice)
    lattice = update(lattice)
    print("\nNext generation:\n")
```

Lattice Representation:
The lattice is represented as a 2D list (a list of lists) in Python, where each sublist represents a row in the grid, and each element in the sublist is a cell in that row. A cell can be either alive (1) or dead (0).

update Function:
This function takes the current state of the lattice as an argument and generates the next state based on the Game of Life rules. It loops over each cell in the lattice, excluding the border cells since their behavior is not well-defined in a finite grid.

update_rule Function:
This is a helper function called by update, which applies the rules of the Game of Life to an individual cell. It calculates the number of live neighbors for the cell and determines the next state of the cell: survival, death, or birth.

Edge Handling:
The code assumes a "dead border" around the lattice. This means cells on the edge of the lattice do not have neighbors wrapping around the grid. This simplifies the code but limits the space in which patterns can evolve.

Main Logic:
The main part of the script (which you would need to add) is responsible for initializing the lattice, typically with a predefined pattern, running the update function in a loop to simulate the passage of time, and printing the lattice to visualize each generation.

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
```

let's pretend you have a big square mat where you put blocks. Each spot on the mat can either have a block or be empty. Now, you have a special rule for placing a new block or taking one away every time we sing a nursery rhyme. That's kind of what this code is doing with something called a lattice, which is like our mat made up of spots.

The update function is like the nursery rhyme. Every time you sing it, you look at the mat, and for each spot, you decide if you're going to put a new block there or take one away based on the special rule.

The update_rule is that special rule. It says:

If a spot has a block and it has two or three blocks right next to it, it gets to keep its block.
If a spot has a block but doesn't have two or three neighbors, you take the block away.
If a spot doesn't have a block but there are exactly three blocks next to it, you put a new block there.
In all other cases, you do nothing.

This is like a game to see how the blocks will look after each round of singing!

Now, how is this different in Cython? Imagine if you had a helper who could move super fast to put blocks in and take them out while you're singing. That's what Cython is like. It's a way of helping the computer run our block game much faster. Instead of picking up one block at a time, it's like having a superpower to pick up and put down lots of blocks all at once

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
```

Cython: Introduces cdef to declare C static types, which reduces the overhead of dynamic typing. This is because the types of variables are known at compile time, allowing the C compiler to optimize the code more effectively. By declaring loop variables as C integers, the

Cython compiler can convert these loops into pure C loops, eliminating the Python interpreter's overhead in these operations.
Using cdef integer indices, Cython can directly access the elements in the lists (which are now treated more like arrays in C), bypassing much of the dynamic lookup process.
The neighbor sum uses C integers, which means the operations can be carried out at the processor level without the need to instantiate and manipulate Python objects.
The **update_rule** function can be declared with cdef as well, which means it can be compiled to a C function, leading to faster function calls when called from other Cython cdef functions.

CYTHON 3

```
# cython: language_level=3

def update(lattice):
    assert len(lattice) > 2 and all(len(row) == len(lattice[0])
for row in lattice), "Lattice must be at least 3x3 and
rectangular."
    cdef int box_length = len(lattice) - 2
    cdef int i, j
    lattice_new = [[0 for _ in range(len(lattice))] for _ in
range(len(lattice))]
    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


cpdef int update_rule(lattice, int i, int j):
    cdef int n_neigh = 0
    # Get the number of rows and columns
    cdef int num_rows = len(lattice)
    cdef int num_cols = len(lattice[0])

    # Check all eight directions with boundary conditions
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            if di == 0 and dj == 0:
                continue  # skip the cell itself
            ni, nj = i + di, j + dj
            # Check if the neighbor is within bounds
            if 0 <= ni < num_rows and 0 <= nj < num_cols:
                n_neigh += lattice[ni][nj]
```

```
    # Apply the rules based on the count of active neighbors
    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

def test_update_rule(lattice, int i, int j):
    return update_rule(lattice, i, j)
```

This version introduces a more elegant and robust boundary checking method using loop constructs to iterate over neighboring indices. It skips the cell itself and checks if neighboring indices are within bounds before accessing them. This not only improves readability but also makes the code less prone to off-by-one errors and other boundary-related bugs.

The cpdef keyword in this version allows the update_rule function to be called both from Python (using def) and Cython (using cdef). This provides the speed benefits of cdef when called from Cython and the flexibility of def when the function needs to be accessed from Python.

The code ensure that the lattice is of a minimum size and rectangular, While this adds a bit of overhead, it significantly increases the robustness of the code by providing a clear, upfront contract about the input.

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

def test_update_rule(lattice, int i, int j):
    return update_rule(lattice, i, j)
```

The most recent code snippet provided introduces two Cython compiler directives at the top: boundscheck=False and wraparound=False. These are optimizations that can improve the runtime performance of the code under certain conditions.

boundscheck=False: By default, Cython checks that indexing operations are within the bounds of a list or array to prevent segmentation faults. This check, while useful for safety,

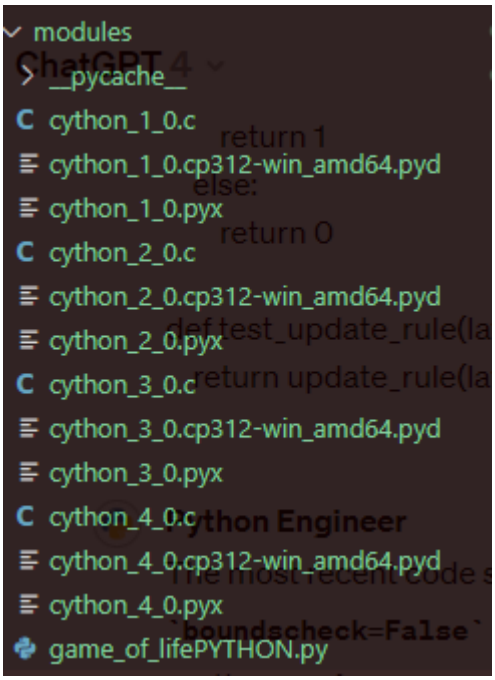
adds overhead to each indexing operation. By setting boundscheck to False, we're telling Cython to skip these checks, which can speed up execution when you're sure that your code will not index outside the bounds of an array.
wraparound=False: Normally, negative indices in Python are interpreted as indexing from the end of the list. For instance, an index of -1 refers to the last item, -2 to the second-to-last item, and so on. This behavior is called 'wraparound'. By setting wraparound to False, we disable this Python feature. When we are certain our code does not rely on this behavior, turning it off can lead to performance gains, as it simplifies the generated C code by eliminating the need to check for and handle negative indices.

To run the code we need to create the setup and it is the next one:

```
from setuptools import setup
from Cython.Build import cythonize

setup(
    ext_modules=cythonize([
        "modules/cython_1_0.pyx",
        "modules/cython_2_0.pyx",
        "modules/cython_3_0.pyx",
        "modules/cython_4_0.pyx"
    ]),
    zip_safe=False
)
```

and with it we create the cythonozed code:



And at final we test it and we have the code to test all at the same time see the time of each one

```
import matplotlib.pyplot as plt
import time

from modules.game_of_lifePYTHON import update as up0
from modules.game_of_lifePYTHON import update_rule as upr0

from modules.cython_1_0 import update as up1
from modules.cython_1_0 import update_rule as upr1

from modules.cython_2_0 import update as up2
from modules.cython_2_0 import update_rule as upr2

from modules.cython_3_0 import update as up3
from modules.cython_3_0 import test_update_rule as upr3

from modules.cython_4_0 import update as up4
from modules.cython_4_0 import test_update_rule as upr4



lattice = [
    [0, 1, 0],
    [0, 1, 0],
    [0, 1, 0]
```

```
]

def benchmark_update(update_func, lattice, iterations):
    start_time = time.time()
    for _ in range(iterations):
        update_func(lattice)
    end_time = time.time()
    return end_time - start_time


times = {
    'Python': benchmark_update(up0, lattice),
    'Cython 1': benchmark_update(up1, lattice),
    'Cython 2': benchmark_update(up2, lattice),
    'Cython 3': benchmark_update(up3, lattice),
    'Cython 4': benchmark_update(up4, lattice),
}

# Plot the results
labels, time_values = zip(*times.items())
plt.bar(labels, time_values, color='blue')
plt.ylabel('Time (sec)')
plt.title('Results')
plt.annotate('Over a 6x speed up\nLarger speed-ups are possible, for
the\nMandelbrot set (exercise 4) the speed\nup is ~250x',
             xy=(2.5, max(time_values)), xytext=(3, max(time_values)+1),
             arrowprops=dict(facecolor='black', shrink=0.05),
             )
plt.show()
```

A lattice, which is a grid representing the state of a cellular automaton, is defined in the script.
A benchmarking function called benchmark_update is created to measure the time taken by different update functions to process the lattice.
The benchmark_update function works as follows:
It first captures the current time and stores it as the start time.
Then, it calls the update function (passed to it as an argument) repeatedly for a specified number of iterations, updating the lattice each time.
After all iterations are complete, it captures the current time again as the end time.
It calculates the elapsed time by subtracting the start time from the end time.
This elapsed time represents how long the update function took to process the lattice for the given number of iterations and is returned by the function.

Wikipedia contributors. (2024, 25 abril). *Conway's Game of Life*. Wikipedia. https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life