CENTRO DE CIENCIAS BÁSICAS

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

AUTÓMATAS II

7° "A"

PROYECTO FINAL

Dr. Francisco Javier Ornelas Zapata

Alumnos:

Almeida Ortega Andrea Melissa

Espinoza Sánchez Joel Alejandro

Flores Fernández Óscar Alonso

Gómez Garza Dariana

González Arenas Fernando Francisco

Orocio García Hiram Efraín

Fecha de Entrega: Aguascalientes, Ags.**, 3** de diciembre de 2021

Índice

Introducción

Un autómata celular (A.C.) es un modelo matemático y computacional para un sistema dinámico que evoluciona en pasos discretos. Es adecuado para modelar sistemas naturales que puedan ser descritos como una colección masiva de objetos simples que interactúen localmente unos con otros.

Son sistemas descubiertos dentro del campo de la física computacional por John von Neumann en la década de 1950. La teoría de los autómatas celulares se inicia con su precursor John von Neumann a finales de la década de 1940 con su libro *Theory of Self-reproducing Automata* (editado y completado por A. W. Burks).

Aunque John von Neumann puso en práctica los AA.CC., estos fueron concebidos en los años 40 por Konrad Zuse y Stanislaw Ulam. Zuse pensó en los "espacios de cómputo" (computing spaces), como modelos discretos de sistemas físicos. Las contribuciones de Ulam vinieron al final de los 40, poco después de haber inventado con Nicholas Metropolis el Método de Montecarlo.

No existe una definición formal y matemática aceptada de autómata celular; sin embargo, se puede describir a un A.C. como una tupla, es decir, un conjunto ordenado de objetos caracterizado por los siguientes componentes:

- Una rejilla o cuadriculado (lattice) de enteros (conjunto $\mathbb{Z}$) infinitamente extendida, y con dimensión $d \in \mathbb{Z}^+$. Cada celda de la cuadrícula se conoce como célula.
- Cada célula puede tomar un valor en $\mathbb{Z}$ a partir de un conjunto finito de estados k.
- Cada célula, además, se caracteriza por su vecindad, un conjunto finito de células en las cercanías de la misma.
- De acuerdo con esto, se aplica a todas las células de la cuadrícula una función de transición (f) que toma como argumentos los valores de la célula en cuestión y los valores de sus vecinos, y regresa el nuevo valor que la célula tendrá en la siguiente etapa de tiempo. Esta función f se aplica, como ya se dijo, de forma homogénea a todas las células, por cada paso discreto de tiempo.

Algunos ejemplos de aplicaciones de los autómatas celulares son:

- Modelado de flujo de tráfico y peatones.
- Modelado de fluidos (gases o líquidos).
- Modelado de la evolución de células o virus como el VIH.
- Modelado de procesos de percolación.

A continuación se modelará un modelado de ondas en el espacio a través de Python como autómata celular.

El Autómata Celular

Primeramente se inicializan algunos valores para el autómata celular:

```
# initial values
import sys

from numpy.core.umath import pi
from numpy.ma import sin

scale = 50  # 1m -> 50 cells
size_x = 6 * scale
size_y = 4 * scale
damping = 0.99
omega = 3 / (2 * pi)

initial_P = 200
vertPos = size_y - 3 * scale
horizPos = 1 * scale
wallTop = size_y - 3 * scale
wall_x_pos = 2 * scale
max_pressure = initial_P / 2
min_presure = -initial_P / 2
```

Posteriormente se crea una clase Simulation para crear la velocidad de las ondas del autómata y presiones de la onda que actuará a partir del epicentro y la dispersión que tomará:

```
class Simulation:
    def __init__(self):
        self.frame = 0
        self.pressure = [[0.0 for x in range(size_x)] for y in range(size_y)]
        # outflow velocities from each cell
        self._velocities = [[[0.0, 0.0, 0.0, 0.0] for x in range(size_x)] for y in range(size_y)]
        self.pressure[vertPos][horizPos] = initial_P
```

Esto conlleva realizar métodos para actualizar los vértices de la onda y los puntos de presión de rebote de las ondas, así como la actualización de pasos del autómata:

```
def updateV(self):
    """Recalcula las velocidades de cada posición basándose en la diferencia de presión de cada vecino"""
    V = self._velocities
    P = self.pressure
    for i in range(size_y):
        for j in range(size_x):
            if wall[i][j] == 1:
                V[i][j][0] = V[i][j][1] = V[i][j][2] = V[i][j][3] = 0.0
                continue
            cell_pressure = P[i][j]
            V[i][j][0] = V[i][j][0] + cell_pressure - P[i - 1][j] if i > 0 else cell_pressure
            V[i][j][1] = V[i][j][1] + cell_pressure - P[i][j + 1] if j < size_x - 1 else cell_pressure
            V[i][j][2] = V[i][j][2] + cell_pressure - P[i + 1][j] if i < size_y - 1 else cell_pressure
            V[i][j][3] = V[i][j][3] + cell_pressure - P[i][j - 1] if j > 0 else cell_pressure
```

```
def updateP(self):
    for i in range(size_y):
        for j in range(size_x):
            self.pressure[i][j] -= 0.5 * damping * sum(self._velocities[i][j])

def step(self):
    self.pressure[vertPos][horizPos] = initial_P * sin(omega * self.frame)
    self.updateV()
    self.updateP()
    self.frame += 1
```

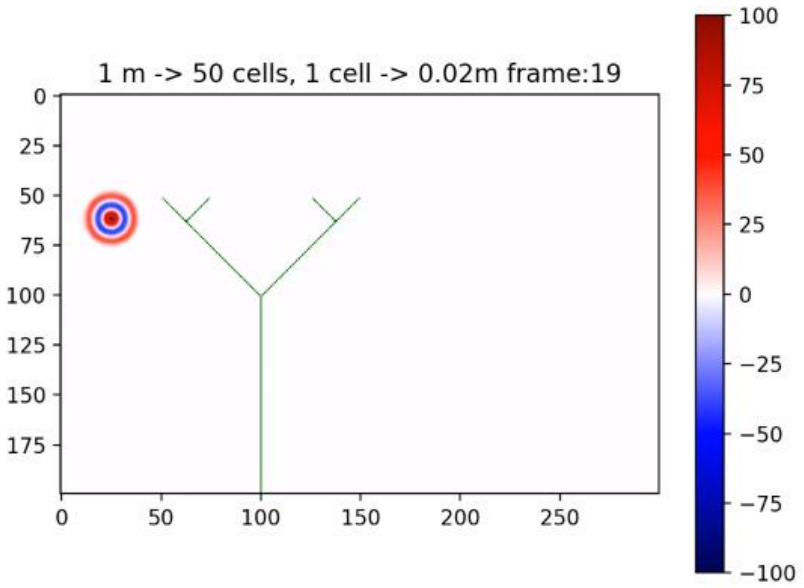
El flujo celular del autómata se irá actualizando con relación de las demás posiciones por medio del siguiente segmento de código:

```
argc = len(sys.argv)
if argc > 1 and sys.argv[1] == '1':
    wall = [[1 if x == wall_x_pos and wallTop < y < size_y else 0
             for x in range(size_x)] for y in range(size_y)]
elif argc > 1 and sys.argv[1] == '2':
    wall = [[1 if (x == wall_x_pos and wallTop + scale < y < size_y) or
                  (wall_x_pos - scale < x < wall_x_pos and
                   x - wall_x_pos == y - wallTop - scale - 1) or
                  (wall_x_pos < x < wall_x_pos + scale and
                   x - wall_x_pos == -y + wallTop + scale + 1)
             else 0
             for x in range(size_x)] for y in range(size_y)]
else:
    wall = [[1 if (x == wall_x_pos and wallTop + scale < y < size_y) or
                  (wall_x_pos - scale < x < wall_x_pos and
                   x - wall_x_pos == y - wallTop - scale - 1) or
                  (wall_x_pos < x < wall_x_pos + scale and
                   x - wall_x_pos == -y + wallTop + scale + 1) or
                  (wall_x_pos - 0.75 * scale < x < wall_x_pos - scale / 2 and
                   x - wall_x_pos == -y + wallTop - scale / 2 + 1) or
                  (wall_x_pos + scale / 2 < x < wall_x_pos + 0.75 * scale and
                   x - wall_x_pos == y - wallTop + scale / 2 - 1)
             else 0
             for x in range(size_x)] for y in range(size_y)]
```

Con ello se tendría en esencia el autómata celular, el resto de código se enfoca en el resultado a mostrar:

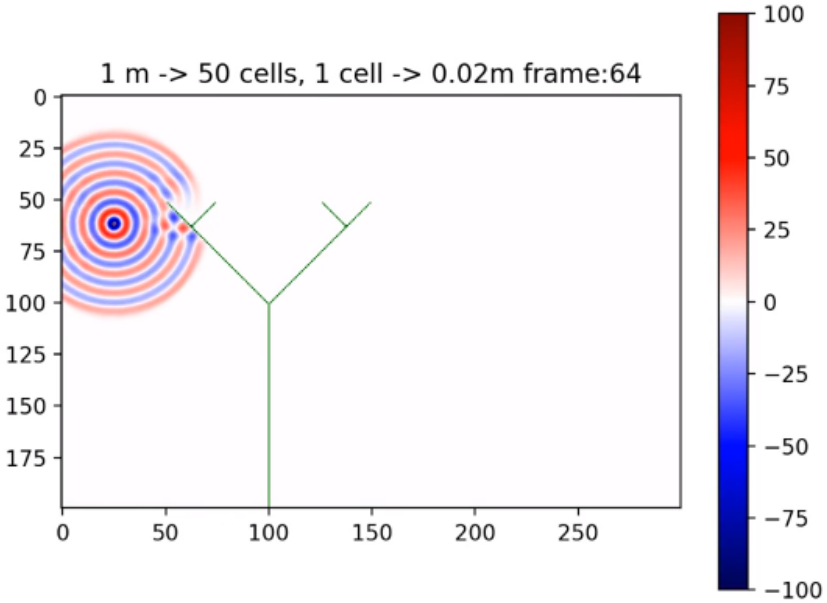
```
matplotlib.use('Qt5Agg')
import matplotlib.pyplot as plt

simulation = Simulation()
figure = plt.figure()
ca_plot = plt.imshow(simulation.pressure, cmap='seismic', interpolation='bilinear', vmin=min_presure, vmax=max_pres
plt.colorbar(ca_plot)
transparent = colorConverter.to_rgba('black', alpha=0)
wall_colormap = LinearSegmentedColormap.from_list('my_colormap', [transparent, 'green'], 2)
plt.imshow(wall, cmap=wall_colormap, interpolation='bilinear', zorder=2)
```

La función de animación muestra el siguiente mapa:

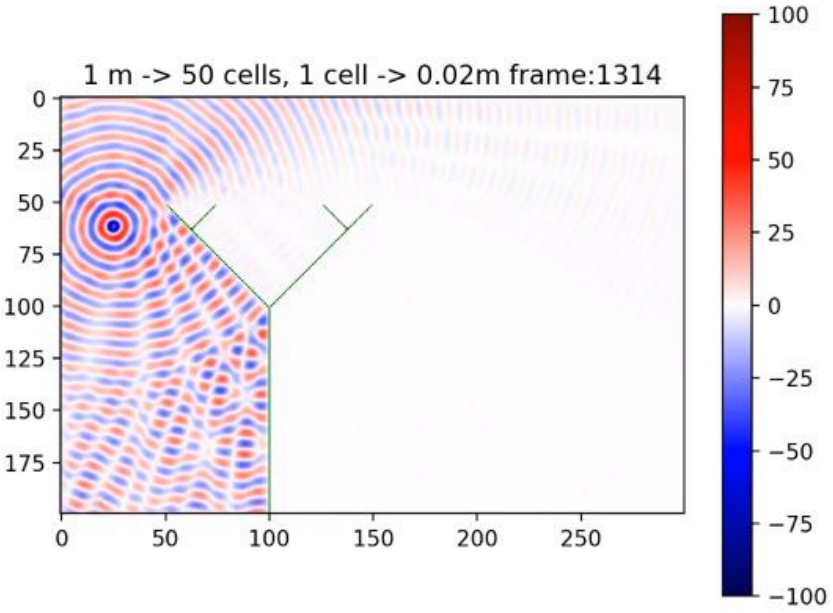


El resultado se encuentra anexo como imagen de animación GIF.

El autómata comienza a evolucionar de manera propia rebotando por las paredes gracias a la diferencia de presión calculada por las coordenadas vecinas:



El modelo evoluciona hasta 2,000 imágenes (frames) con una muestra de cómo se distribuyen las ondas en este mapa:



Conclusiones

Andrea Melissa Almeida Ortega: Con este proyecto, implementamos un autómata célula orientado en el uso y aplicación de autómatas celulares como lo es en este proyecto con base en la dispersión de ondas que puede ser muy bien aplicable como simulador de estos autómatas celulares.

Joel Alejandro Espinoza Sánchez: Gracias a la realización de este proyecto, hemos observado cómo se realiza paso a paso un autómata celular, en el que, debido a que previamente se realizó un autómata, se optó por desarrollar algo novedoso como sería un autómata celular orientado a la distribución de ondas.

Óscar Alonso Flores Fernández: Gracias a este proyecto aprendí la parte de autómatas celulares, tanto la parte teórica como la práctica y funcional de forma más profunda, sobre como funcionan internamente y externamente, lo cual nos servirá a todos en futuros trabajos.

Dariana Gómez Garza: En la realización de esta práctica nos queda aún más claro el concepto de autómata celular. Para esta actividad tratamos de dividirnos el trabajo en partes iguales para hacer el código y la documentación y así todos hacer aportes equitativos.

Fernando Francisco González Arenas: Los autómatas celulares son muy útiles como cualquier autómata, con la diferencia que los autómatas celulares van calculando con relaciones cercanas dando mayor flexibilidad al momento de hacer cambios al programa o de ejecutarlo en diferentes maquinas convirtiéndose en un tipo de simulador.

Hiram Efraín Orocio García: En este proyecto aprendí la teoría, funcionamiento y composición de los autómatas celulares más a fondo, conociendo la forma como funcionan interna y externamente, lo cual me servirá en futuros trabajos ya sea académicos o en el campo laboral.

Fuentes de Consulta

- Anónimo. (2012). *Autómata celular*. Noviembre 29, 2021, de Wikipedia Sitio web: https://es.wikipedia.org/wiki/Aut%C3%B3mata_celular.
- Hopcroft, J. (2007). *Teoría de autómatas, lenguajes y computación*. México: Pearson.
- Kawala, A. (2018). Wave Propagation in Python. Noviembre 29, 2021, de YouTube Sitio web: https://www.youtube.com/watch?v=qeltZU2GFgU.
- Linz, P. (1990). *An Introduction to Formal Languages and Automata*. India: Jones & Barlett.
- Sipser, M. (1997). *Introduction to the Theory of Computation*. México: PWS Publishing.

Anexos

Anexo 1: Archivo `main.py`

```
#%% Portada
'''
                Centro de Ciencias Básicas
         Departamento de Ciencias de la Computación
                       Autómatas II
                          7° "A"

                     Proyecto Final

         Profesor: Francisco Javier Ornelas Zapata

         Alumnos:
                  Almeida Ortega Andrea Melissa
                  Espinoza Sánchez Joel Alejandro
                  Flores Fernández Óscar Alonso
                  Gómez Garza Dariana
                  González Arenas Fernando Francisco
                  Orocio García Hiram Efraín

Fecha de Entrega: Aguascalientes, Ags., 3 de diciembre de 2021
'''


#%% Simulator
# initial values
import sys

from numpy.core.umath import pi
from numpy.ma import sin

scale = 50  # 1m -> 50 cells
size_x = 6 * scale
size_y = 4 * scale
damping = 0.99
omega = 3 / (2 * pi)

initial_P = 200
vertPos = size_y - 3 * scale
horizPos = 1 * scale
wallTop = size_y - 3 * scale
wall_x_pos = 2 * scale
max_pressure = initial_P / 2
```

```
min_presure = -initial_P / 2


class Simulation:
    def __init__(self):
        self.frame = 0
        self.pressure = [[0.0 for x in range(size_x)] for y in
range(size_y)]
        # outflow velocities from each cell
        self._velocities = [[[0.0, 0.0, 0.0, 0.0] for x in
range(size_x)] for y in range(size_y)]
        self.pressure[vertPos][horizPos] = initial_P

    def updateV(self):
        """Recalculate outflow velocities from every cell basing on
preassure difference with each neighbour"""
        V = self._velocities
        P = self.pressure
        for i in range(size_y):
            for j in range(size_x):
                if wall[i][j] == 1:
                    V[i][j][0] = V[i][j][1] = V[i][j][2] =
V[i][j][3] = 0.0
                    continue
                cell_pressure = P[i][j]
                V[i][j][0] = V[i][j][0] + cell_pressure - P[i -
1][j] if i > 0 else cell_pressure
                V[i][j][1] = V[i][j][1] + cell_pressure - P[i][j +
1] if j < size_x - 1 else cell_pressure
                V[i][j][2] = V[i][j][2] + cell_pressure - P[i +
1][j] if i < size_y - 1 else cell_pressure
                V[i][j][3] = V[i][j][3] + cell_pressure - P[i][j -
1] if j > 0 else cell_pressure

    def updateP(self):
        for i in range(size_y):
            for j in range(size_x):
                self.pressure[i][j] -= 0.5 * damping *
sum(self._velocities[i][j])

    def step(self):
        self.pressure[vertPos][horizPos] = initial_P * sin(omega *
self.frame)
        self.updateV()
        self.updateP()
        self.frame += 1
```

```
argc = len(sys.argv)
if argc > 1 and sys.argv[1] == '1':
    wall = [[1 if x == wall_x_pos and wallTop < y < size_y else 0
             for x in range(size_x)] for y in range(size_y)]
elif argc > 1 and sys.argv[1] == '2':
    wall = [[1 if (x == wall_x_pos and wallTop + scale < y < size_y)
or
                  (wall_x_pos - scale < x < wall_x_pos and
                   x - wall_x_pos == y - wallTop - scale - 1) or
                  (wall_x_pos < x < wall_x_pos + scale and
                   x - wall_x_pos == -y + wallTop + scale + 1)
             else 0
             for x in range(size_x)] for y in range(size_y)]
else:
    wall = [[1 if (x == wall_x_pos and wallTop + scale < y < size_y)
or
                  (wall_x_pos - scale < x < wall_x_pos and
                   x - wall_x_pos == y - wallTop - scale - 1) or
                  (wall_x_pos < x < wall_x_pos + scale and
                   x - wall_x_pos == -y + wallTop + scale + 1) or
                  (wall_x_pos - 0.75 * scale < x < wall_x_pos - scale
/ 2 and
                   x - wall_x_pos == -y + wallTop - scale / 2 + 1)
or
                  (wall_x_pos + scale / 2 < x < wall_x_pos + 0.75 *
scale and
                   x - wall_x_pos == y - wallTop + scale / 2 - 1)
             else 0
             for x in range(size_x)] for y in range(size_y)]

#%% Main
from sys import argv

import matplotlib
from matplotlib.animation import FuncAnimation, FFMpegWriter
from     matplotlib.colors     import     LinearSegmentedColormap,
colorConverter

#from simulation.py import Simulation, min_presure, max_pressure,
scale, wall

matplotlib.use('Qt5Agg')
import matplotlib.pyplot as plt
```

```
simulation = Simulation()
figure = plt.figure()
ca_plot    =    plt.imshow(simulation.pressure,    cmap='seismic',
interpolation='bilinear', vmin=min_presure, vmax=max_pressure)
plt.colorbar(ca_plot)
transparent = colorConverter.to_rgba('black', alpha=0)
wall_colormap  =  LinearSegmentedColormap.from_list('my_colormap',
[transparent, 'green'], 2)
plt.imshow(wall,   cmap=wall_colormap,   interpolation='bilinear',
zorder=2)


def animation_func(i):
    simulation.step()
    ca_plot.set_data(simulation.pressure)
    return ca_plot


if len(argv) > 2 and argv[2] == 'save':
    writer = FFMpegWriter(fps=30)
    frames = 100
    with writer.saving(figure, "writer_test.mp4", 200):
        for i in range(frames):
            animation_func(i)
            writer.grab_frame()
            print(f'\rframe: {i}/{frames}', end='')
else:
    animation = FuncAnimation(figure, animation_func, interval=1)
    mng = plt.get_current_fig_manager()
    mng.window.showMaximized()
    plt.title(f"1 m -> {scale} cells, 1 cell -> {1 / scale}m")
    plt.show()
```