MATH 580A Assignment 1

1 Short Exercises

1. In class, we saw an example program used to blur an image via the box blur algorithm. The box blur algorithm is what is known as a *separable filter*. To reduce the complexity from $\mathcal{O}(whr^2)$ down to $\mathcal{O}(whr)$, instead of averaging across a box of $(2r+1) \times (2r+1)$ centered at each pixel, you can instead do two 1-dimensional passes.

   First blur along the x-axis. For each pixel, average it along the horizontal line of length $2r+1$ centered at the pixel. After blurring once, do a second blur pass along the y-axis. For each already blurred pixel, average it along the vertical line of length $2r+1$ centered at the pixel. It turns out that these two blur passes will result in precisely the same value as the original box blur, but the complexity will only be linear in r.

   Modify the existing box blur program to perform two linear blur passes one after another, and observe whether or not it speeds up the blurring process.

2. In class, we also saw an algorithm for sorting a list known as selection sort. The complexity of the selection sort algorithm is $\mathcal{O}(n^2)$ where n is the length of the input list. In this exercise, you will implement a different algorithm known as mergesort, which has time complexity $\mathcal{O}(n \log n)$. This algorithm is most easily implemented using recursion and proceeds as follows.

   Given a list ℓ, divide it into two sublists ℓ_1 and ℓ_2 where ℓ_1 is the first half of ℓ and ℓ_2 is the second half of ℓ. If ℓ has an odd number of elements, let ℓ_1 also contain the middle element. Recursively use the mergesort algorithm to sort ℓ_1 and ℓ_2 respectively.

   Once ℓ_1 and ℓ_2 are sorted, apply a *merge* procedure to produce the final sorted list ℓ'. To merge the lists, repeat the following procedure: look at the first element of ℓ_1 and ℓ_2 and pop the smaller element off and append it to ℓ'. If at any point one of the lists becomes emptied, simply continue to pop elements off the remaining list and append it onto ℓ'. This procedure will terminate once both lists are emptied and we can return the sorted list ℓ'.

After implementing the two short exercises, pick and implement one of either the Lindenmayer System project or the Heat Equation project.

2 Lindenmayer Systems

Introduction

For this program, you will implement and draw some self-similar pictures using Lindenmayer systems (L-systems). L-systems were originally developed by the botanist Aristid Lindenmayer in order to model the development of plants and draw them. An L-system consists of an *initial*

state s in some formal alphabet Ω and a set of *replacement rules* P that tell you how to evolve the initial state. Replacement rules look like $s \to s_1 s_2 \cdots s_\ell$, where s is a single letter in Ω and $s_1 s_2 \cdots s_\ell$ is replacement string with letters in Ω. To evolve the state in an L-system, read the letters in the current state from left to right, and for each letter s that has a replacement rule $s \to s_1 s_2 \cdots s_\ell$, replace it with $s_1 s_2 \cdots s_\ell$ in the evolved state. If a letter has no replacement rule, then keep it as is in the evolved state.

Iterating these replacement rules over and over gives a state that has a lot of self-similarity. By assigning a certain drawing operation to each letter in Ω, we can interpret the resulting state and draw fractal-like or tree-like images. We will use the `turtle` module in the Python standard library to draw these images.

Using the `turtle` library

This section will give a brief introduction to the `turtle` library. For the full documentation, refer to https://docs.python.org/3/library/turtle.html.

Turtle graphics was originally part of the Logo programming language designed in 1967. The implementation in Python is similar. In turtle graphics, there is an object, the turtle, that has an (x, y) position and an angle θ that it is facing. Through calling functions in the `turtle` library, you can tell the turtle to move forwards or backwards a certain distance, or to turn left or right a certain angle (in degrees). The turtle holds a pen that can either be held *up* or held *down*. When the pen is down, the turtle draws a line as it moves. In the Python `turtle` module, there is also the option to change the pen color between turtle movements, allowing for multicolored drawings.

The turtle starts off at $(0, 0)$, facing the positive x-axis and with the pen down in black and width 1. The main functions you will need to make use of in this assignment are:

- `turtle.forward(`*`distance`*`)`: Moves the turtle forwards *distance* units.
- `turtle.backward(`*`distance`*`)`: Moves the turtle backwards *distance* units.
- `turtle.left(`*`angle`*`)`: Turns the turtle *angle* degrees to the left (counter-clockwise).
- `turtle.right(`*`angle`*`)`: Turns the turtle *angle* degrees to the right (clockwise).
- `turtle.setposition(`*`pos`*`)`: Sets the turtle position to *pos*.
- `turtle.setheading(`*`angle`*`)`: Sets the turtle to face *angle* degrees counter-clockwise from the positive x-axis.
- `turtle.penup()`: Sets the pen up.
- `turtle.pendown()`: Sets the pen down.
- `turtle.pencolor(`*`color`*`)`: Sets the pen color to *color* interpreted as a hex string (e.g. `"#FF00FF"` is magenta).
- `turtle.pensize(`*`width`*`)`: Sets the pen width to be *width* pixels wide.
- `turtle.position()`: Returns the turtle's current position.
- `turtle.heading()`: Returns the angle the turtle is currently facing.

- `turtle.hideturtle()`: Hide the turtle itself in the resulting drawing.
- `turtle.done()`: This must be called after all drawing functions to display the result.

Listing 1 is an example program that uses the `turtle` to draw a polygon-like shape with each edge in a different color. The resulting output is in Fig. 1.

```
import turtle

colors = ["#000000","#FF0000","#FFFF00","#00FF00","#00FFFF","#0000FF","#FF00FF"]
n = len(colors)

turtle.pensize(10)

for i in range(n):
  turtle.pencolor(colors[i])
  turtle.pendown()
  turtle.forward(40)
  turtle.left(360/(2*n))
  turtle.penup()
  turtle.forward(40)
  turtle.left(360/(2*n))

turtle.done()
```

Listing 1: Example Program Using the `turtle` Module.

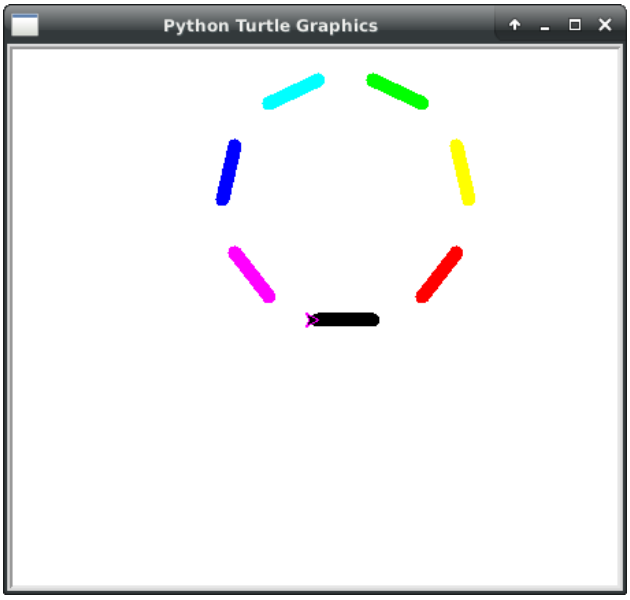

Figure 1: Graphical Output of Listing 1.

There are a few more features of turtle graphics that are not covered here. You can peruse the official documentation to learn about the other functions included in the module.

Drawing a Binary Tree

Let us now take a look at an example of an L-system and a set of drawing instructions that results in a binary tree. Our alphabet will be $\Omega = \{\texttt{X}, \texttt{F}, \texttt{+}, \texttt{-}\}$. The initial state will be `X` and we have two replacement rules `F` $\to$ `FF` and `X` $\to$ `F+X-X`. The drawing operations associated with `X` and `F` are to move forward and draw a line of fixed length. The drawing operations associated with + and - is a little more complicated. For +, append the turtle's current position and angle to a list, then turn 45° left. For -, reset the turtle's position and angle to the most recent entry in the list and pop that entry out. Then turn 45° right. After 6 iterations, this L-system results in the binary tree in Fig. 2.

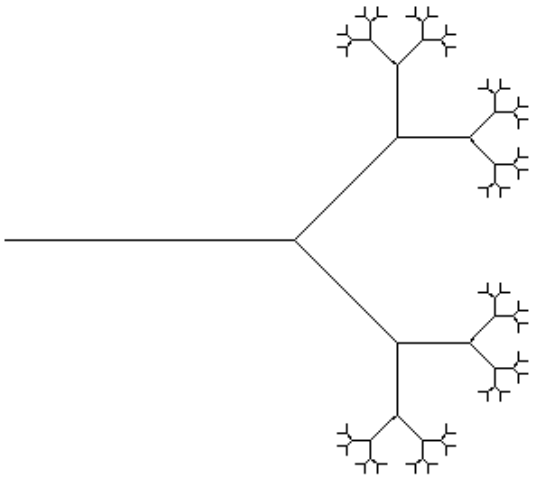

Figure 2: Binary Tree L-System Example

Why does this L-system work? Let us take a look at the first few iterations of the state to discern how it produces a binary tree.

Iteration	State
0	`X`
1	`F+X-X`
2	`FF+F+X-X-F+X-X`
3	`FFFF+FF+F+X-X-F+X-X-FF+F+X-X-F+X-X`

Let w_n denote the state at iteration n. Observe that these replacement rules lead to the recursive relations

$$w_{n+1} = \underbrace{\texttt{F}\cdots\texttt{F}}_{2^{n-1}\text{ times}} \texttt{+}w_n\texttt{-}w_n.$$

Thus, when w_n gets drawn, first the turtle moves forward to draw a stem of length 2^{n-1}. Then, it encounters a + symbol to draw a smaller binary tree corresponding to w_{n-1} starting from the end of the stem and rotated 45° counter-clockwise. Finally, it encounters a -, resetting its position to the end of the stem and facing towards the positive x-axis. It then draws a smaller binary tree corresponding to w_{n-1} starting from the end of the stem and rotated 45° clockwise.

Implementation

We've now seen a full example of how a simple L-system can be used to draw self-similar objects like a binary tree. In this program, you will implement a function `draw_lindenmayer_system()` that takes in 4 parameters `initial_state`, `rules`, `turtle_commands` and `iterations`, and computes the resulting state and executes the turtle commands to visualize the state. See the `lindenmayer.md` file in the repl for more implementation details.

After implementing the generic `draw_lindenmayer_system()` function, use it to draw the following objects:

(a) **Koch Snowflake**: The alphabet is $\Omega = \{\texttt{X}, \texttt{+}, \texttt{-}\}$. The initial state is `X-X-X` and there is a single replacement rule $\texttt{X} \to \texttt{X+X-X+X}$. The drawing operation for `X` is to move forward by a fixed length and draw a line. The drawing operation for `+` is to rotate left by 60° and for `-` is to rotate right by 120°.

(b) **Sierpinski Triangle**: The alphabet is $\Omega = \{\texttt{X}, \texttt{F}, \texttt{+}, \texttt{-}\}$. The initial state is `X-F-F` and the replacement rules are $\texttt{X} \to \texttt{X-F+X+F-X}$ and $\texttt{F} \to \texttt{FF}$. The drawing operations for `X` and `F` are to move forward by a fixed length and draw a line. The drawing operation for `+` is to rotate left by 120° and for `-` is to rotate right by 120°.

(c) **Algorithmic Fern**: By adding drawing operations that append and pop the turtle position and angle, we can draw organic plant-like objects. The alphabet is $\Omega = \{\texttt{X}, \texttt{F}, \texttt{[}, \texttt{]}, \texttt{+}, \texttt{-}\}$. The initial state is `X` and the production rules are $\texttt{X} \to \texttt{F+[[X]-X]-F[-FX]+X}$ and $\texttt{F} \to \texttt{FF}$. The drawing operations for `X` and `F` are to move forward by a fixed length and draw a line. The drawing operations for `+` and `-` are to rotate left and rotate right by 25 degrees respectively. The drawing operation for `[` and `]` is to append and pop the turtle position and angle respectively.

Extensions

The L-systems described thus far have all been *deterministic*. There are also *stochastic* L-systems in which one chooses randomly among a set of replacements for each letter in the alphabet. To implement a *stochastic* L-system, you can use the Python `random` module.

Another small extension is to add drawing operations that set the pen color and pen width, injecting some aesthestics to your program output.

Finally, the theory of modeling plant growth and drawing them is quite well-developed and goes far beyond what we have done here with L-systems. If you are interested in studying more on this subject, take a look at *The Algorithmic Beauty of Plants* by Przemyslaw Prusinkiewicz and Aristid Lindenmayer available at http://algorithmicbotany.org/papers/abop/abop.pdf. Chapter 1 is devoted to modeling using L-systems and has many more examples of fractals and plants one can draw using what we have covered here.

3 Modeling the Heat Equation

Introduction

For this program, you will model the diffusion of heat in two dimensions by using the finite difference method to approximate solutions to the heat equation

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right).$$

There are three different common schemes used: forward time centered space (FTCS), backward time centered space (BTCS), and Crank-Nicolson. Each of them expresses the heat function at the next time step linearly in terms of the heat function at the current time step. You will use a mathematical Python package called NumPy to perform the necessary matrix operations. Starter code relying on a package called Matplotlib will be provided to visualize the heat distribution.

Finite Difference Methods

In order to model heat in a program, we will first discretize the problem. That is, we will assume time advances in discrete steps of Δt and that the spatial domain can be thought of as a discrete grid of points spaced Δx apart. We will use the notation $u^t_{x,y}$ to denote the heat at time t and position (x, y).

Let us first look at the forward time difference and backward time difference approximations. Fixing x and y, the Taylor series of $u^t_{x,y}$ centered at $t = t_0$ can be expressed as

$$u^{t_0+\Delta t}_{x,y} = u^{t_0}_{x,y} + \Delta t \frac{\partial u^t_{x,y}}{\partial t}\bigg|_{t_0} + \frac{(\Delta t)^2}{2!}\frac{\partial^2 u^t_{x,y}}{\partial t^2}\bigg|_{t_0} + \frac{(\Delta t)^3}{3!}\frac{\partial^3 u^t_{x,y}}{\partial t^3}\bigg|_{t_0} + \cdots. \tag{1}$$

Rearranging Eq. (1) and dividing through by $(\Delta t)^2$, we obtain

$$\frac{1}{\Delta t}\left(\frac{u^{t_0+\Delta t}_{x,y} - u^{t_0}_{x,y}}{\Delta t} - \frac{\partial u^t_{x,y}}{\partial t}\bigg|_{t_0} \right) = \frac{1}{2!}\frac{\partial^2 u^t_{x,y}}{\partial t^2}\bigg|_{t_0} + \frac{\Delta t}{3!}\frac{\partial^3 u^t_{x,y}}{\partial t^3}\bigg|_{t_0} + \cdots \tag{2}$$

By the mean-value theorem, there exists some $s \in [t_0, t_0 + \Delta t]$ such that $\frac{u^{t_0+\Delta t}_{x,y} - u^{t_0}_{x,y}}{\Delta t} = \frac{\partial u^t_{x,y}}{\partial t}\Big|_s$. Applying the mean-value theorem once more, there exists $r \in [t_0, s]$ such that $\frac{1}{\Delta t}\left(\frac{\partial u^t_{x,y}}{\partial t}\Big|_s - \frac{\partial u^t_{x,y}}{\partial t}\Big|_{t_0} \right) = \frac{\partial^2 u^t_{x,y}}{\partial t^2}\Big|_r$. Substituting this into Eq. (2), then Eq. (1), and rearranging gives us

$$\frac{\partial u^t_{x,y}}{\partial t}\bigg|_{t_0} = \frac{u^{t_0+\Delta t}_{x,y} - u^{t_0}_{x,y}}{\Delta t} - \Delta t \frac{\partial^2 u^t_{x,y}}{\partial t^2}\bigg|_r. \tag{3}$$

The term $\Delta t \frac{\partial^2 u^t_{x,y}}{\partial t^2}\Big|_r$ is our error term which is dependent on Δt. Thus far, all the equations have

been exact equalities. Dropping the error term gives our *forward difference time approximation*:

$$\left.\frac{\partial u^t_{x,y}}{\partial t}\right|_{t_0} \approx \frac{u^{t_0+\Delta t}_{x,y} - u^{t_0}_{x,y}}{\Delta t}. \tag{4}$$

By using $-\Delta t$ instead of Δt, we obtain the *backward difference time approximation*:

$$\left.\frac{\partial u^t_{x,y}}{\partial t}\right|_{t_0} \approx \frac{u^{t_0 t}_{x,y} - u^{t_0-\Delta t}_{x,y}}{\Delta t}. \tag{5}$$

Next, we will again examine the Taylor series of $u^t_{x,y}$ but for a fixed value of t and y and centered at $x = x_0$. Using both Δx and $-\Delta x$ we have

$$u^t_{x_0+\Delta x,y} = u^t_{x_0,y} + \Delta x \left.\frac{\partial u^t_{x,y}}{\partial x}\right|_{x_0} + \frac{(\Delta x)^2}{2!}\left.\frac{\partial^2 u^t_{x,y}}{\partial x^2}\right|_{x_0} + \frac{(\Delta x)^3}{3!}\left.\frac{\partial^3 u^t_{x,y}}{\partial x^3}\right|_{x_0} + \cdots \tag{6}$$

$$u^t_{x_0+\Delta x,y} = u^t_{x_0,y} - \Delta x \left.\frac{\partial u^t_{x,y}}{\partial x}\right|_{x_0} + \frac{(\Delta x)^2}{2!}\left.\frac{\partial^2 u^t_{x,y}}{\partial x^2}\right|_{x_0} - \frac{(\Delta x)^3}{3!}\left.\frac{\partial^3 u^t_{x,y}}{\partial x^3}\right|_{x_0} + \cdots. \tag{7}$$

Summing Eq. (6) and Eq. (7) and solving for the second partial derivative gives us

$$\left.\frac{\partial^2 u^t_{x,y}}{\partial x^2}\right|_{x_0} = \frac{u^t_{x_0+\Delta x,y} + u^t_{x_0-\Delta x,y} - 2u^t_{x_0,y}}{(\Delta x)^2} + \frac{2(\Delta x)^4}{4!}\left.\frac{\partial^4 u^t_{x,y}}{\partial x^4}\right|_{x_0} + \cdots. \tag{8}$$

A similar mean-value theorem argument allows us to express the higher order terms as an error term dependent on $(\Delta x)^2$. We drop the error term to obtain the *central difference space approximation*:

$$\left.\frac{\partial^2 u^t_{x,y}}{\partial x^2}\right|_{x_0} \approx \frac{u^t_{x_0+\Delta x,y} + u^t_{x_0-\Delta x,y} - 2u^t_{x_0,y}}{(\Delta x)^2}. \tag{9}$$

An analogous approximation is used for y when t, x are fixed. Note that we use a uniform grid size Δx regardless of which spatial variable we are considering.

FTCS, BTCS, and Crank-Nicolson

We are now ready to describe the three different schemes for modeling the heat equation. In each of them, we can express the values $\{u^t_{x,y} \mid t = t_0\}$ as a vector $\boldsymbol{u}_{t_0}$ and compute $\boldsymbol{u}_{t_0+\Delta t}$ as the matrix product $A\boldsymbol{u}_{t_0}$.

The forward time centered space scheme uses the forward time difference equation to approximate $\frac{\partial u^t_{x,y}}{\partial t}$ and the centered space equation to approximate $\frac{\partial^2 u^t_{x,y}}{\partial x^2}$. Substituting these approximations into the heat equation yields,

$$\frac{u^{t+\Delta t}_{x,y} - u^t_{x,y}}{\Delta t} = \frac{u^t_{x+\Delta x,y} + u^t_{x-\Delta x,y} + u^t_{x,y+\Delta y} + u^t_{x,y-\Delta y} - 4u^t_{x,y}}{(\Delta x)^2}. \tag{10}$$

By multiplying both sides by Δt and moving $-u^t_{x,y}$ over to the right hand side, we obtain

$$u^{t+\Delta t}_{x,y} = \alpha \left(u^t_{x+\Delta x,y} + u^t_{x-\Delta x,y} + u^t_{x,y+\Delta y} + u^t_{x,y-\Delta y} - 4u^t_{x,y} \right) + u^t_{x,y}, \tag{11}$$

where $\alpha = \frac{\Delta t}{(\Delta x)^2}$. Since Eq. (11) expresses each coordinate of $\boldsymbol{u}_{t+\Delta t}$ as a linear combination of the coordinates of $\boldsymbol{u}_t$, there exists a matrix F such that $\boldsymbol{u}_{t+\Delta t} = F\boldsymbol{u}_t$. You will need to work out how to compute F when implementing the FTCS scheme. Unlike the other two schemes, the FTCS scheme only results in a stable solution when $\alpha \leq \frac{1}{4}$. When $\alpha > \frac{1}{4}$, you will observe oscillation in the resulting model.

The other two schemes are similar and also result in a matrix equation relating $\boldsymbol{u}_{t+\Delta t}$ and $\boldsymbol{u}_t$, but are unconditionally stable. In order to implement the BTCS scheme, the backwards time difference equation is used, resulting in

$$\frac{u^t_{x,y} - u^{t-\Delta t}_{x,y}}{\Delta t} = \frac{u^t_{x+\Delta x,y} + u^t_{x-\Delta x,y} + u^t_{x,y+\Delta y} + u^t_{x,y-\Delta y} - 4u^t_{x,y}}{(\Delta x)^2}. \tag{12}$$

Observe that Eq. (13) will result in a matrix equation $B\boldsymbol{u}_t = \boldsymbol{u}_{t-\Delta t}$ and by inverting B, we obtain $\boldsymbol{u}_t = B^{-1}\boldsymbol{u}_{t-\Delta t}$.

Finally, in the Crank-Nicolson scheme, the backwards time difference equation is also used, but instead of the central difference space approximation at time t, we use the average of the approximation at times t and $t - \Delta t$. This results in the equation

$$\begin{aligned} \frac{u^t_{x,y} - u^{t-\Delta t}_{x,y}}{\Delta t} = {} & \frac{u^t_{x+\Delta x,y} + u^t_{x-\Delta x,y} + u^t_{x,y+\Delta y} + u^t_{x,y-\Delta y} - 4u^t_{x,y}}{2(\Delta x)^2} \\ & + \frac{u^{t-\Delta t}_{x+\Delta x,y} + u^{t-\Delta t}_{x-\Delta x,y} + u^{t-\Delta t}_{x,y+\Delta y} + u^{t-\Delta t}_{x,y-\Delta y} - 4u^{t-\Delta t}_{x,y}}{2(\Delta x)^2}. \end{aligned} \tag{13}$$

In this case, there will be matrices C and N such that $C\boldsymbol{u}_t = N\boldsymbol{u}_{t-\Delta t}$. Multiply by C^{-1} to then obtain $\boldsymbol{u}_t = C^{-1}N\boldsymbol{u}_{t-\Delta t}$.

Using NumPy

For this program, you will need to manipulate matrices using the NumPy package. A basic overview of NumPy is available at https://numpy.org/doc/stable/user/absolute_beginners.html. For the most part, you can also treat a matrix in NumPy as a 2-dimensional list in Python (i.e. a list of lists) and you can index and modify the matrix values in the same way. More generally, NumPy is often used for scientific computing and provides functions for various mathematical operations in linear algebra, discrete fourier transform, and other areas.

The matrix operations that you need to be aware of for this program is matrix multiplication and matrix inversion. See Listing 2 for a basic example of how to create NumPy arrays, how to multiply two matrices, and how to invert a matrix.

Implementation and Extensions

For this program, starter code has been provided to display the results of the heat equation and update the heat state using the finite difference method. However, it is up to you to implement

```python
import numpy as np

# Create a 2x2 matrix in numpy.
A = numpy.array([
    [1, 2],
    [3, 4],
])
B = numpy.array([
    [5, 6],
    [7, 8]
])

# Use the @ operator to multiply two matrices.
prod = A @ B
print(prod)

# Use np.linalg.inv() to invert square matrices.
inv = np.linalg.inv(prod)
print(inv)
```

Listing 2: Example Program Using the `numpy` Package.

the FTCS, BTCS, and Crank-Nicolson schemes. More specifically, you will need to write code to compute the matrices described in the section on these 3 schemes.

In the provided starter code, the starting heat state is also randomized for you and a color scheme for display has been set. The heat values will be between 0 and 1, where 1 is the highest heat value and 0 is the lowest.

After implementing the 3 schemes, you may wish to try extending the assignment in one of the following ways:

- Modify the boundary conditions to have more complicated shapes. For example, try adding circles in the interior of the current boundary where the temperature is fixed.
- The package used for visualizing the heatmap is known as Matplotlib. Read up on how to plot 3d surfaces at https://matplotlib.org/stable/gallery/mplot3d/surface3d.html and use it to visualize the heat as a 3d surface where the z-coordinate value is the heat at a given coordinate (x, y).