The *Game of Life* on Penrose Tilings: Robinson Triangle

by
Seung Hyeon Mandy Hong

Department of Mathematics
Denison University
December 6, 2022

Research Advisor:
Prof. May Mei

1 Introduction

John Horton Conway's *Game of Life* gained much attention in different areas because of the concept that it generates complexity from simplicity.[1] If we are given three simple rules: survival, death, and birth, and initial configuration, we can play *Game of Life*. Compared to the simple setting of this game, what we get is more complex. Generally, *Game of Life* is played on the regular square tiling, a periodic tiling. However, we implement *Game of Life* on an aperiodic tiling, specifically on the Robinson triangle tiling, a variation of the Penrose tiling. Then, we identify all distinct neighborhoods in the Robinson triangle tiling. In addition, we prove the existence of four-cell configurations that satisfy still life conditions in all distinct neighborhoods.

Sections 2 and 3 define rules, terms, and methods related to *Game of Life* and Penrose tiling. Section 4 discusses the algorithm we use to play *Game of Life* on Robinson triangle tiling. Section 5 shows initially founded still life and oscillators. Section 6 explains nine distinct neighborhood in the Robinson triangle tiling. Sections 7, 8, and 9 describe two algorithms we create to find all possible four-cell still life and how algorithms could be applied to a certain neighborhood. We use an emerging programming language called `Julia`. `Julia` is designed for high-performance and enables us to use features such as animated visualization or numerical and scientific computing.[2] All codes to generate algorithms are uploaded in the GitHub repository. [3]

2 *Game of Life*

Conway's *Game of Life* (also called *life*) resembles a society of living organisms because it can be explained by the rise, fall, and alterations. Initially, Conway used a large checkerboard (assumed to be an infinite plane) and flat square counters of two colors to be placed on the checkerboard. The checkerboard is, in fact, a regular square tiling and we call each tile in the tiling a *cell*. To play *Game of Life*, we start with a simple configuration of live cells on the tiling and observe how it changes as we apply Conway's *genetic laws*.[4] A simple arrangement of live cells needs to meet three criteria. First, there should be no initial pattern that the population can grow without limit. Second, there should be simple initial patterns that do grow without limit. Lastly, there should be simple initial patterns that grow and change before ending by fading away, reaching stable figures, or oscillating by repeating several periods. Meeting these criteria makes the population's behavior unpredictable. After setting the initial stage of this game, we apply genetic laws for births, deaths, and survivals to cells. The rules are the following[4]:

Definition 2.1. *Genetic laws* apply three rules: survival, death, and birth on cells.

1. Survival: Every cell with two or three live neighbors (other cells that share its edge) survives to the next generation.

2. Death: Each cell with four or more live neighbors or one or none die.

3. Birth: A dead cell with three adjacent live neighbors is alive at the next generation.

Births and deaths co-occur, constituting a single generation or move. Using the genetic laws to create subsequent generations, we can find the population undergoing changes: Society could vanish, reach a stable figure, or oscillate forever. There are several significant patterns people are interested in finding when playing *Game of Life*. Our primary interest is to find an oscillator, a pattern that repeats a finite number of configurations and a still life, a pattern that reaches a stable state without changing from generation to generation.[4]

3 Penrose Tiling

A *tiling* can either be *periodic* or *nonperiodic*.[5].

Definition 3.1. A *tiling* of the plane $\mathbb{R}^2$ with *protoset* $\mathcal{P}$ is a collection of $\mathfrak{T}$ of closed sets in the plane, called *tiles* and a finite collection $\mathcal{P}$ of closed sets in the plane, each called a *prototile*, such that the following hold [6]:

1. Each prototile is topologically equivalent to a disk.

2. Each tile is congruent to a prototile.

3. $\mathbb{R}^2 = \bigcup_{T \in \mathfrak{T}} T$ ($\mathfrak{T}$ is a covering.)

4. $\text{int}(T_i) \bigcap \text{int}(T_j) = \emptyset$ for all $i \neq j$ ($\mathfrak{T}$ is a packing.)

Definition 3.2. A *periodic tiling* is one on which you can outline a region that tiles the plane by translation, that is, by shifting the position of the region without rotating or reflecting it. A *nonperiodic* tiling is not periodic.

The checkerboard that Conway used to play *Game of Life* is a periodic tiling because we can outline a square region that tiles the plane by shifting the position of that region without rotating or reflecting it. We investigate how *Game of Life* will proceed if we change underlying square tiling (periodic tiling) to Robinson triangle tiling (aperiodic tiling). Among various tiles that tile aperiodically, one of the most well-known tiles is Penrose tiles[5], discovered by British mathematical physicist Roger Penrose.

Definition 3.3. The *Penrose tiles* are sets of tiles with two different shapes that tile only nonperiodically. The shapes of a pair of Penrose tiles that force nonperiodcity can vary.

Using sets of Penrose tiles as prototiles, it is possible to get different Penrose tilings. The most popular Penrose tilings are Pentagonal Penrose tiling, kite and dart tiling, and Penrose rhomb tiling (Figure 1).

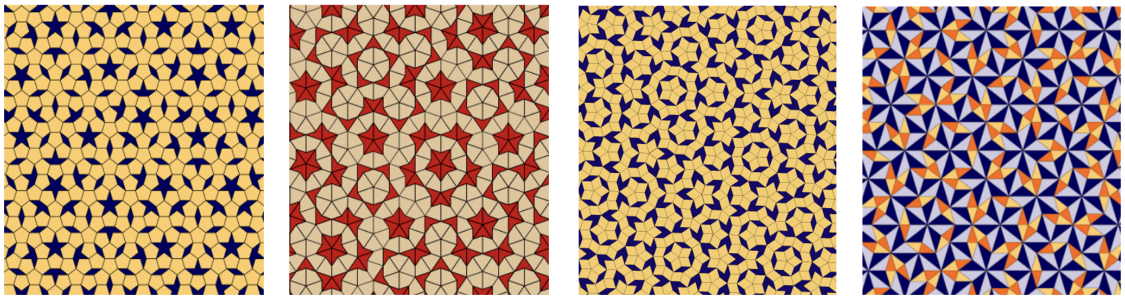


Figure 1: (From left to right) Pentagonal Penrose tiling[7], kite and dart tiling[8], Penrose rhomb tiling[9], and Robinson triangle tiling[10].

We are particularly interested in Robinson triangle tiling (the last figure in Figure 1), the variation of Penrose rhomb tiling suggested by the American mathematician Raphael M. Robinson. If we cut two Penrose rhombs into triangles, we get four prototiles of the Robinson triangle tiling (Figure 2).

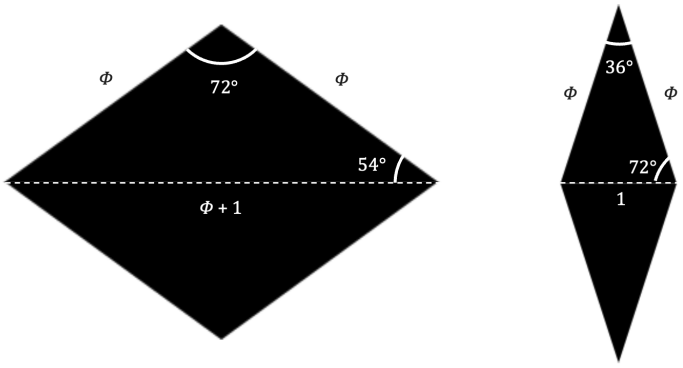


Figure 2: Dividing Two Penrose rhombs into triangles. Four triangles are prototiles of the Robinson triangle tiling.

The Robinson triangle tiling is created using a substitution rule, a method to generate aperiodic tiling.[11] Suppose we are given a finite set of building blocks $T_1, T_2 \cdots T_m$ (the prototiles), an expanding linear map Q (with inflation factor $|Q|$), and a rule, how to dissect each scaled tile QT_i into copies of the original prototiles $T_1, T_2, \cdots T_m$.[11] Then, using the substitution method (inflation and subdivision), we first inflate each prototile by the inflation factor and subdivide each inflated prototile into copies of the original prototiles. For the Robinson triangle tiling, we use the inflation factor of $\phi = \frac{1+\sqrt{5}}{2}$ which

is the golden ratio to inflate each prototile. Then, we subdivide each tile into three or two tiles depending on the tile type (Figure 3). By continued inflation and subdivision, we can generate a tiling of the plane.[6]

Figure 3: Substitution rule for prototiles in Robinson triangle tiling.

4 *Game of Life* on Robinson Triangle Tiling

In order to play *Game of Life* on Robinson triangle tiling, we create several functions that are explained in each subsection.

4.1 Getting Functions to Generate Coordinates and Graphic of Four Prototiles

To plot the Robinson triangle tiling, we first need to find the coordinates of four prototiles. Using information about the angles and lengths of four tiles, we first find three coordinates of the two wider triangles which are shown with labels in Figure 4. Here, we calculate $a = \cos(\frac{\pi}{5}) \cdot \phi$ and $h = \sin(\frac{\pi}{5}) \cdot \phi$ using trigonometry. Next, we find three coordinates of two narrower triangles shown with labels in Figure 4. Here, we calculate $a = \frac{1}{2}$ and $h = \sin(\frac{2}{5}\pi) \cdot \phi$ using trigonometry.

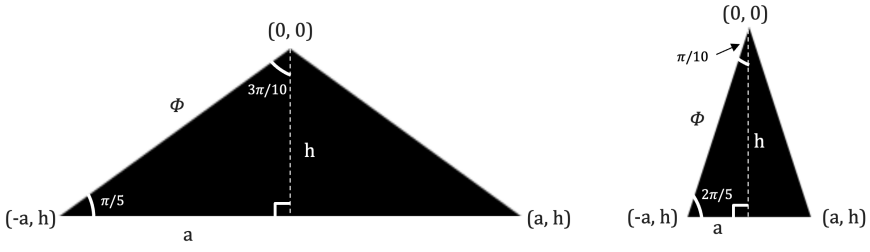


Figure 4: Calculating three coordinates of the Robinson triangle.

Using three coordinates of each triangle, we create four functions `wider_triangle_up`, `wider_triangle_down`, `narrower_triangle_up`, and `narrower_triangle_down`. Four functions use same parameter `s` which is the size/side length of each prototile. Four functions plot a size `s` polygon, filled with colors, using three coordinates we have. Similarly, we get four functions `wider_triangle_up_stroke`, `wider_triangle_down_stroke`, `narrower_triangle_up_stroke`, and `narrower_triangle_down_stroke` which plot stoke of each polygon. These functions are modified version of four functions mentioned above that draw polygons. Using the same parameter `s` four functions plot a size `s` polygon using three coordinates of triangles. However, since we are not filling each polygon with color but getting the stroke of the polygon, we need to add one more coordinate (the first coordinate we initially started drawing the polygon) to close the stroke. We also create four functions `w_triangle_up_coordinate`, `w_triangle_down_coordinate`, `n_triangle_up_coordinate`, and `n_triangle_down_coordinate`. Four functions also use same parameter `s` and return coordinates of each triangle as a list when the size of a tile is `s`.

If triangles are rotated or translated, we need to update coordinates. We create two functions `new_coordinate_r` and `new_coordinate_t` to update coordinates. For the first function `new_coordinate_r`, we get an updated coordinate when list of coordinates (`pL`) are rotated by the given angle, `orientation`. Starting by creating an empty vector to store updated coordinates, we go over all points in the list of coordinates given to calculate new coordinates. When we say original coordinate is (x, y), we can update this using the fact that new x coordinate can be calculated by $x\cos(\texttt{orientation}) - y\sin(\texttt{orientation})$ and new y coordinate can be calculated by $x\sin(\texttt{orientation}) + y\cos(\texttt{orientation})$. After calculating new coordinate, we add that point to the empty vector we initially created. For the second function `new_coordinate_t`, we get an updated coordinate when the origin in the list of coordinates given (`pointL`) is translated to the new location (`translate`). Starting by creating an empty vector to store updated coordinates, we go over all points in the list of coordinates given and calculate new coordinates by adding the new location point (`translate`). After calculating new coordinate, we add that point to the empty vector we initially created. Both functions return list of updated coordinates.

4.2 Generating Robinson Triangle Tiling Using Substitution Method

For substitution, we create two functions, `substitution_robinson_triangle_list` and `plot_Robinson_Triangle_png` which use functions mentioned above. The first function produces a level-n substitution when given three parameters: `upper_level_coordinatesL, nstep, size`. The first parameter, `upper_level_coordinatesL`, is a list of vectors. Each vector has tile types, the origin of a tile, and the orientation of a tile. Set of vectors with three elements compose the initial configuration of tiling, which is also a level-0 substitution. Two other parameters, `nstep` is the level of substitution we want, and `size` is the side length of each tile. As an output, we get a list of vectors with tile type, origin, and orientation after n substitution, which is information of all tiles after substitution.

We repeat same process for `nstep` times. First, we define new size by multiplying `size` by ϕ^{i-1} where i refers to current step. Then we pass a list of vectors given to a new variable (`initial`) and create an empty vector to store information of all tiles after substitution (`v`). Going over all vectors in the list given, we inflate the given tile and subdivide it into three or two tiles depending on the tile type. If the tile type of the given list is a wider upward triangle, we first inflate the tile by ϕ and get new coordinates. Then we find three locations to place three triangles when subdividing inflated tile which are `new_p1`, `new_p2`, and `new_p3`. Using the inflated tile's coordinate, we get three new origins and add them to the coordinate list of the inflated tile. Then we update all six coordinates after rotation and translation for the inflated wider upward triangle.

For the subdivision, the first wider triangle is at the fourth element of the updated coordinate list and rotated by the angle of $4\pi/5$, considering the orientation of the updated inflated triangle is 0. If the tile we are subdividing is rotated by some angle, we add that to $4\pi/5$ and consider this added angle as the orientation of the first wider triangle. The second narrower triangle is at the fifth element of the updated coordinate list and rotated by the angle of $\pi/5$ considering the orientation of the updated inflated triangle is 0. If the tile we are subdividing is rotated by some angle, we add that to $\pi/5$ and consider this added angle as the orientation of the second narrower triangle. The third wider triangle is at the sixth element of the updated coordinate list and rotated by the angle of π considering the orientation of the updated inflated triangle is 0. If the tile we are subdividing is rotated by some angle, we add that to π and consider this added angle as the orientation of the third wider triangle. Then we store the tile type, position, and orientation of three tiles as a vector and append three vectors to `v`.

If the tile type of the given list is a wider, downward triangle, we first inflate the tile by ϕ and get new coordinates. Then we find three locations to place three triangles when subdividing inflated tile which are `new_p1`, `new_p2`, and `new_p3`. Using the inflated tile's coordinate, we get three new origins and add them to the coordinate list of inflated tile. Then we update all six coordinates after rotation and translation for the inflated wider triangle.

For the subdivision, the first wider triangle is at the fourth element of the updated coordinate list and rotated by the angle of $-4\pi/5$, considering the orientation of the updated inflated triangle is 0. If the tile we are subdividing is rotated by some angle, we add that to $-4\pi/5$ and consider this added angle as the orientation of the first wider triangle. The second narrower triangle is at the fifth element of the updated coordinate list and rotated by the angle of $-\pi/5$ considering the orientation of the updated inflated triangle is 0. If the tile we are subdividing is rotated by some angle, we add that to $-\pi/5$ and consider this added angle as the orientation of the second narrower triangle. The third wider triangle is at the sixth element of the updated coordinate list and rotated by the angle of π considering the orientation of the updated inflated triangle is 0. If the tile we are subdividing is rotated by some angle, we add that to π and consider this added angle as the orientation of the third wider triangle. Then we store the tile type, position, and

orientation of three tiles as a vector and append three vectors to `v`.

If the tile type of the given list is a narrower upward triangle, we first inflate the tile by ϕ and get new coordinates. Then we find two locations to place two triangles when subdividing the inflated tile. The second location is already in the coordinate list, and we only need to get one new location which is `new_p1`. Using the inflated tile's coordinate, we get two new origins and add `new_p1` to the coordinate list of inflated tile. Then we update all four coordinates after rotation and translation for the inflated narrower triangle.

For the subdivision, the first wider triangle is at the fourth element of the updated coordinate list and rotated by the angle of $3\pi/5$ considering the orientation of the updated inflated triangle is 0. If the tile we are subdividing is rotated by some angle, we add that to $3\pi/5$ and consider this added angle as the orientation of the first wider triangle. The second narrower triangle is at the third element of the updated coordinate list and rotated by the angle of $-3\pi/5$ considering the orientation of the updated inflated triangle is 0. If the tile we are subdividing is rotated by some angle, we add that to $-3\pi/5$ and consider this added angle as the orientation of the second narrower triangle. Then we store the tile type, position, and orientation of two tiles as a vector and append two vectors to `v`.

If the tile type of the given list is a narrower, downward triangle, we first inflate the tile by ϕ and get new coordinates. Then we find two locations to place two triangles when subdividing the inflated tile. The second location is already in the coordinate list, and we only need to get one new location which is `new_p1`. Using the inflated tile's coordinate, we got two new origins and added `new_p1` to the coordinate list of inflated tile. Then we update all four coordinates after rotation and translation for the inflated narrower triangle.

For the subdivision, the first wider triangle is at the fourth element of the updated coordinate list and rotated by the angle of $-3\pi/5$ considering the orientation of the updated inflated triangle is 0. If the tile we are subdividing is rotated by some angle, we add that to $-3\pi/5$ and consider this added angle as the orientation of the first wider triangle. The second narrower triangle is at the third element of the updated coordinate list and rotated by the angle of $3\pi/5$ considering the orientation of the updated inflated triangle is 0. If the tile we are subdividing is rotated by some angle, we add that to $3\pi/5$ and consider this added angle as the orientation of the second narrower triangle. Then we store the tile type, position, and orientation of two tiles as a vector and append two vectors to `v`.

After each substitution, we get `v` which is the list of vectors for the ith substitution and use this for the next level substitution. After all substitution ends, we get a final list of vectors which is information of all tiles after level-n substitution as an output.

The second function, `plot_Robinson_Triangle_png`, plots Robinson triangle tiling when given three parameters: `nstep`, `list`, and `size`. The first parameter, `nstep` and the third parameter, `size` come from the substitution function, which is what level of substitution we want and the size of each prototile. Second parameter, `list`, is a list of vectors and each vector has tile types, the origin of a tile, and the orientation after level-n substitution. We get `list` as an output of the substitution function. As an output for `plot_Robinson_Triangle_png`, we get graphic of Robinson triangle tiling after level-n sub-

stitution in a `png` format. For each level of substitution, we first update the side length of each tile by multiplying `size` by inflation factor. Going over all vectors in the list given, we first translate to the origin of each tile which is the second element of each vector, and rotate by the given orientation, which is the third element of each vector. Then, depending on the tile type, we plot the corresponding triangle using functions that draw polygons using coordinates.

In addition to plotting Robinson triangle tiling using different colors for four triangles, we create a function that plots tiling without colors, `plot_Robinson_Triangle_stroke_png`. This function uses same three parameters: `nstep`, `list`, and `size` from above function. As an output, we get graphic of Robinson triangle tiling that is not filled with colors, after level-n substitution in `png` format. For each level of substitution, we first update the side length of each tile by multiplying `size` by inflation factor. Going over all vectors in the list given, we first translate to the origin of each tile which is the second element of each vector, and rotate by the given orientation, which is the third element of each vector. Then, depending on the tile type, we plot the corresponding triangle using four functions that draw strokes of polygons using coordinates.

4.3 Gathering Neighboring Tile Information

In order to get neighboring tile information, we create three functions, `coordinate_list`, `round_coordinate`, and `neighbor_robinson_triangle_more_info`. The first function outputs a level-n substitution and gives us tile type and three coordinates of all tile after level-n substitution. This function is same as `substitution_robinson_triangle_list`, the substitution function from Section 4.2, except one additional argument after each subdividing process. In addition to creating an empty vector to store information of all tiles after substitution (`v`), we create one additional empty vector to store information on tile type and three coordinates of all tiles after substitution (`c_l`). After we store tile type, position, and orientation of subdivided tiles as a vector and append two vectors to `v`, we also create a vector with tile type and all three coordinates of subdivided tile. Then we append this vector to `c_l`. After each substitution, we get `c_l` which is the list of vectors with tile type and three coordinates of all tiles after ith substitution and use this for the next level substitution. After all substitution is done, we get a final list of vectors which is information of all tiles after level-n substitution as an output.

The second function, `round_coordinate`, uses `c_list` which is list of vectors with coordinates as a parameter and round each coordinate to ten decimal places. Since the current coordinate list we get as an output from the above function gives us coordinates with 12 decimal places, we round these coordinates to ten decimal places to compare coordinates to get the neighboring tiles. We go over all vectors in the coordinate list and round x and y coordinates of three coordinates of each tile. Then we replace each coordinate using rounded x and y coordinates.

The third function, `neighbor_robinson_triangle_more_info` gives two outputs. The

first output has information on tile type, position, orientation, and neighbor type (corner or edge) of all neighboring tiles of each tile in the tiling. The second output is a vector with a list of vectors that store neighboring tiles' tile numbers. We will frequently use second output when playing *Game of Life*. This function uses two parameters: `substitution_l` and `coordinate_l`. First parameter, `substitution_l` is an output we get after doing level-n substitution using function from Section 4.2 and second parameter, `coordinate_l` is a list of rounded coordinates of all tiles after level-n substitution using two functions above. Inside the function, we first create two empty vectors to store information on adjacent tile(s) of all tiles and information on tile numbers. We go over all tiles, get the current tile's coordinates and create an empty vector to store information about neighboring tiles and tile numbers of the current tile. Then we compare all other tiles' coordinates with the current tile's coordinates. If the current tile's coordinates and comparing tile's coordinates have one shared point, two tiles are corner-to-corner neighbors. Thus, we append comparing tile's information (tile type, position, orientation, neighbor type) to the empty vector that stores all neighboring tiles' information of the current tile. Also, we append neighboring tile numbers to the empty vector. If the current tile's coordinates and comparing tile's coordinates have two shared points, two tiles are edge-to-edge neighbors. Then we append comparing tile's information (tile type, position, orientation, neighbor type) to the empty vector that stores all neighboring tiles' information of the current tile. Also, we append neighboring tile numbers to the empty vector. We repeat the same process for all tiles and return two lists, a list of neighboring tile information and neighboring tile numbers of all tiles.

4.4 Playing *Game of Life*

Using outputs from above three sections, we create functions to play *Game of Life* on Robinson triangle tiling and to plot the results of each generation. In order to play *Game of Life*, we need two functions: `one_move`, and `GoL_Robinson_triangle`.

First function, `one_move` generates a list of live cells after one generation when playing *Game of Life*. This function uses three parameters: `GoL_info`, `neighbor_l_more_info`, and `neighbor_l_tile_num`. The first parameter is the list of live cells' tile number of current generation. The second and third parameters are outputs from the function in Section 4.3. We start by creating an empty state vector that will have the state (0 or 1, live or dead) of all tiles in the tiling. Beginning with an all-zero vector that has the same length as the total number of tiles in the tiling, we go over all tiles and change the state of each tile by checking `GoL_info`. If the tile number is in `GoL_info`, that tile is alive and we change the state from 0 to 1. After going over all tiles, we get the state vector, `state_all_tile`.

Next, we create a vector to store the sum of alive neighbors of each tile. Beginning with an empty vector, we go over all tiles' neighboring tile lists with the tile number (`neighbor_l_tile_num`). Using the neighboring tile's tile number, we access the state vector and find the state of each neighboring tile. If the neighboring tile is alive, we add

one to variable `n` which initially starts with 0. If the neighboring tile is dead, we do not change `n`. After going over each tile's neighboring tiles, we update the empty vector that store the sum of alive neighbors of each tile. We repeat the same process for all tiles and get the final output, `total_neighbor`.

Using `total_neighbor`, we are able to play *Game of Life*. We first create an empty vector to store alive cells' tile number after one move (`GoL_info_one_move`). Going over all tiles, first, we check if each tile is alive or dead by using `GoL_info`. If ith tile is in `GoL_info`, ith tile is alive. Then we check how many alive neighbors that ith tile has. Applying original *Game of Life* rules, if the tile has two or three neighbors, the tile survives after the current move and we add that tile's number to `GoL_info_one_move`. If the tile has less than one neighbor or more than four neighbors, the tile is dead after the current move and we do not add that tile's number to `GoL_info_one_move`. If ith tile is not in `GoL_info`, ith tile is dead. We also check how many alive neighbors this ith tile has. If the tile has three alive neighbors, the tile gain birth after the current move and we add that tile's number to `GoL_info_one_move`. Taking three steps, we get our final output `GoL_info_one_move` which is a list of alive cells after one move.

The second function, `GoL_Robinson_triangle` generates a vector with a list of live cells of all moves when playing *Game of Life*. This function uses four parameters and the first three parameters are the same as the above function. One additional parameter is `max_generation` which indicates the maximum number of generations we will allow when playing *Game of Life*. Beginning with an empty vector to store all lists of alive cells after all moves (`all_step_GoL_info`), we first append a list of live cells of initial configuration to the empty vector. Then we get an updated live cell list after one move. We continuously generate an updated live cell list after one move and compare it with the live cell list from the previous move. If the current live cell list and previous live cell list are not the same, we keep playing the *Game of Life* and append the current live cell list to `all_step_GoL_info`. If two lists are the same, we reach a steady-state (still life or all cells dead) and stop playing *Game of Life*. Then we append the current list to `all_step_GoL_info`. If the number of generation (`g`) is same as `max_generation`, we also stop playing *Game of Life*. As a result of running this function, we get a vector with a list of live cells of all generations.

4.5 Plotting *Game of Life*

To plot the results of each move and get graphic of *Game of Life* in `png` format, we need two functions: `plot_one_move_png`, and `plot_GoL_Robinson_triangle_png`. The first function, `plot_one_move_png` colors all live cells of current move when given four parameters: `nstep`, `list`, `size`, and `GoL_info`. The first three parameters come from the substitution function in Section 4.2, which is the level of the substitution, output of the substitution function (substitution list), and the size of each prototile. The fourth parameter, `GoL_info`, is a list of live cells' tile numbers of current generation. Going over all vectors in the `list` given, we first generate tiling of level-n substitution. Then we color all live cells in `GoL_info`.

After identifying tile number of each live cell, we get the information (tile type, position, and orientation) of that cell using `list` and color live cell.

The second function, `plot_GoL_Robinson_triangle_png` shows graphics of all generations when given four parameters: `nstep`, `list`, `size`, and `all_GoL_info`. All parameters are same as above functions except `all_GoL_info`. Instead of using one list of live cells of one generation, we use a vector with lists of live cells of all generations. First, we start by creating an empty dictionary to store each plot of each move. Going over all moves, we generate each plot and store the plot in a dictionary using i from ith generation as key to the dictionary. Then we display each plot to see the result of all move.

4.6 Generating *Game of Life* Animation

In order to see how the *Game of Life* proceed, we need to generate an animation. Thus, we create a function, `GoL_animation_Robinson_triangle`, to get *Game of Life* animation. This function uses five parameters: `all_GoL_info`, `step`, `size`, `list`, and `gif_name`. The first parameter is a vector with a list of live cells of all moves when playing *Game of Life*. The second and third parameters are the same parameters used to create substitution tiling. The fourth parameter is an output we get after doing level-n substitution and the last parameter is the animation file name. Our animation function has three parts: creating a movie object, generating scenes to store graphics, and getting animation. First, we create a movie object to play animation which has a frame length that is the same as the total number of generations when playing each *Game of Life*. Then we create scenes to make animations. We need two scenes, background which is the level-6 Robinson triangle tiling, and frame which is a *Game of Life* graphic of each generation. Once we have the movie object and scenes, we construct animation using them. Using the internal `animate` function, we put the movie object, the background scene, and frames with *Game of Life* graphic of each generation. As an output, we get *Game of Life* animation that shows all processes.

4.7 Implementing *Game of Life* on Robinson Triangle Tiling

Using all functions we explain from previous sections, in order to start playing *Game of Life*, we take three steps before running our last function, `GoL`.

1. Generate level-6 substitution tiling starting with ten tiles with side length five using `substitution_robinson_triangle_list` function.

2. Generate coordinate list of all tiles in level-6 substitution tiling using `coordinate_list` and `round_coordinate` functions.

3. Generate neighboring tiles' information using `neighbor_robinson_triangle_more_info` function.

After going through three steps, we run our final function `GoL`. When given initial live cells list (initial configuration to play *Game of Life*), maximum generation, and animation file name as parameters, we get the result of *Game of Life* as an animation. Inside the function, using variables created from above three steps, we use `GoL_Robinson_triangle` to generate list of live cells of all move and use that list to generate animation using our animation function, `GoL_animation_Robinson_triangle`.

5 Finding Still Life and Oscilators

As a result of playing *Game of Life* on the Robinson triangle tiling, we discover several still life patterns and oscillators. To find a still life, we run the `GoL_Robinson_triangle` function multiple times with randomly generated live cell list. If we get the output before reaching maximum generation, we use the output to visually check if the configuration is a still life. To find oscillator, we run the `GoL_Robinson_triangle` function multiple times until the function reaches maximum generation. If we get the output that has repeating lists of live cells, we use the output to visually check if there are configurations that repeats after finite number of generations. As a result, we find eight still life patterns (Figure 5) and two oscillators (Figure 6 and Figure 7) by playing *Game of Life* on level-6 substitution tiling.

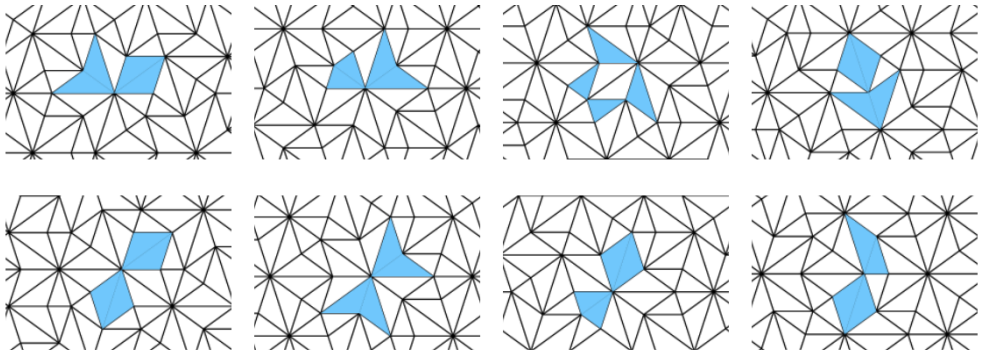


Figure 5: Eight still life patterns.

6 Neighborhoods in the Robinson Triangle Tiling

Interestingly, all still life we have identified using random initial configurations are four-cell configurations. Thus, we explore further to find all four-cell still life in the Robinson triangle tiling. Our approach is to find all distinct neighborhoods and identify valid still

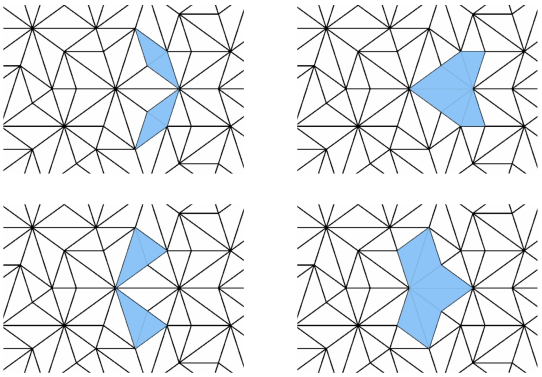


Figure 6: Period-4 oscillator.

life configurations in each neighborhood. To discover all neighborhoods in the Robinson triangle tiling, we first categorize tiles in the level-6 substitution tiling by their prototile types. Going over each tile in four groups, we collect the information on coordinates of each tile's neighboring tiles and store them as a vector in the list. Then, going over all vectors in the list, we center each coordinate in all vectors and compare each vector. As a result, we find distinct vectors with coordinates of tiles, which are neighborhoods. Therefore, we identify nine neighborhoods in the Robinson triangle tiling shown in Figure 8. Since Robinson triangle tiling is derived from the Penrose rhomb tiling, nine neighborhoods are confirmed by comparing neighborhoods on a Penrose rhomb tiling.[12]

7 Identifying All Valid Four-Cell Still Life

Within each neighborhood with n neighbors, we have two scenarios to get four-cell configurations: including and excluding the center cell. For the first scenario, there are total of $\binom{n}{3}$ configurations. For the second scenario, there are total of $\binom{n}{4}$ configurations. Among them, to find configurations that satisfy still life conditions, we go over all configurations and eliminate configurations that has at least one birth or death. Thus, we create two algorithms for two scenarios that could discard such configurations.

The first algorithm that identifies valid four-cell still life with the center cell alive has four layers, and each step finds a configuration with a birth. When the center cell and three cells in the neighborhood are alive, the center cell always has three neighbors. Thus, we do not have to consider eliminating configurations that has at least one death. However, the second algorithm that identifies valid four-cell still life with the center cell dead has five layers, and each step finds a configuration with a birth or death. In our algorithm, we use terms, *inside vertex group* and *outside vertex group*. The inside vertex group is a set of

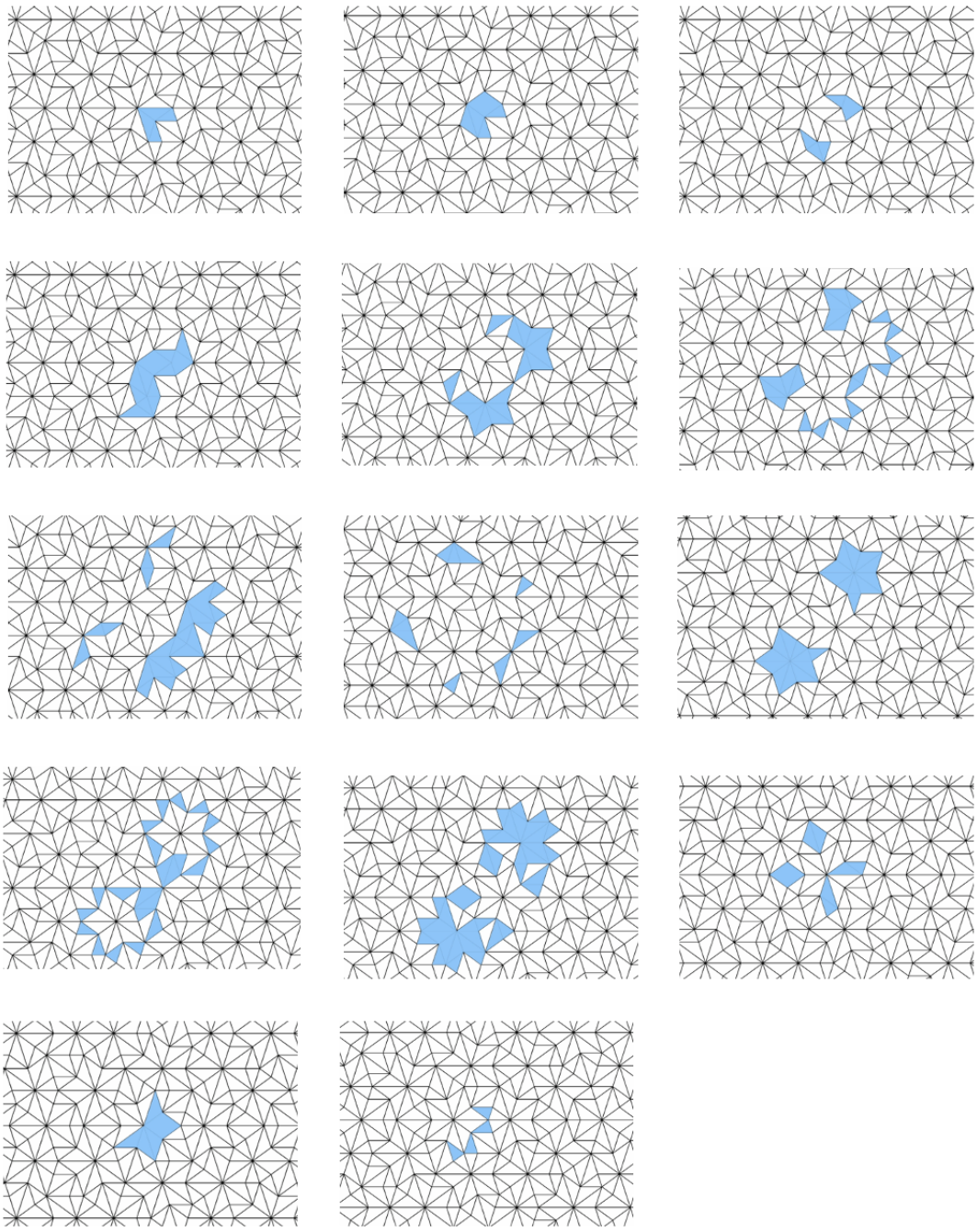


Figure 7: Period-14 oscillator.

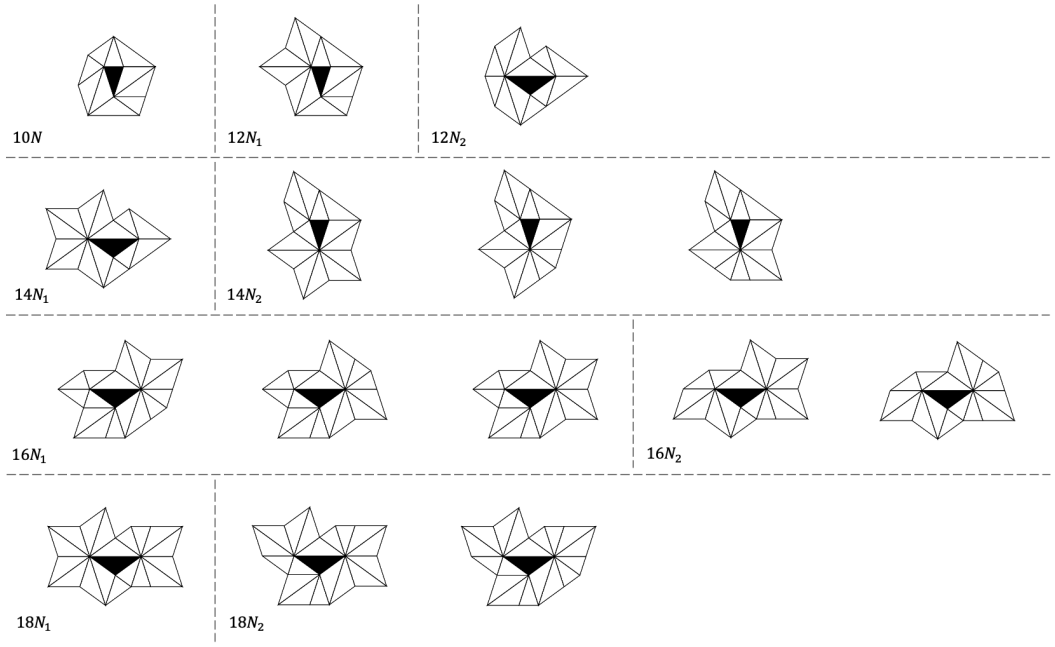


Figure 8: Nine neighborhoods in the Robinson triangle tiling. Numbers in front of N represent the number of neighbors.

tiles that share the same vertex located around the center tile. The outside vertex group is a set of tiles that share the same vertex at the edge of the boundary of the neighborhood.

7.1 Algorithm to Find Still Life Where the Center Cell and Three Neighboring Cells Are Alive

The first algorithm has four steps listed below:

1. Get rid of configurations that share outside neighbors.
2. Get rid of configurations that have a birth in each inside vertex group.
3. Get rid of configurations that have a birth in each outside vertex group.
4. Get rid of configurations that have birth at tiles that belong to two inside vertex groups.

The algorithm needs two preset variables: all possible four-cell configuration and group information. Assume the neighborhood in Robinson triangle tiling has n tiles in the neighborhood without including the center cell. Then we have $\binom{n}{3}$ possible configurations if the center cell and three cells on the neighborhood are alive. To get a list of vectors with four tile numbers representing configurations with four cells, we create a function, `all_configuration` with parameter `n` which is the number of tiles in the neighborhood.

Along with the list of configurations, we need five categories of variables which are explained below:

1. Two lists of tile numbers in the neighborhood (including 0 and not including 0).
2. Three-cell configurations that share outside neighbor(s) as a list of vectors.
3. Inside vertex groups: elements of each inside vertex group and a list containing all inside vertex groups.
4. Outside vertex groups: elements of each outside vertex group and a list containing all outside vertex groups.
5. Tile information: for each tile, we have a list of tiles that share inside or outside vertex groups, that do not share inside or outside vertex groups, and two dictionaries (key: tile number, value: corresponding tile's information).
6. Information on three tiles that belongs to two inside vertex groups: for each tile, we have two lists of tiles corresponding to two different inside vertex groups (not including itself and 0) and a dictionary (key: index, value: list of two inside vertex groups information and list of tiles that do not share inside or outside vertex groups).

7.1.1 Step 1. Get Rid of Configurations That Share Outside Neighbors.

First, we get rid of configurations that share an outside neighbor. If the configuration consists of a center tile with three tiles from the group of tiles that share the same outside neighbors, we have a birth outside the neighborhood. Thus, we create a function `step_one` that eliminates such configurations from all possible configurations. The function returns the remaining configuration as an output using two parameters: `all_combination`, which is an output of `all_configuration` and `outside_num`, which is the list of vectors representing groups of tiles that share different outside neighbors.

We start by generating an empty vector (`get_rid`) to store configurations that share outside neighbors. Then, going over all possible four-cell still life configurations, we compare them with groups of tiles that share different outside neighbors. If two share precisely three elements, we store the corresponding configuration to `get_rid`. In the end, we compute the difference between two sets `all_configuration` and `get_rid`, which is the function's output.

7.1.2 Step 2. Get Rid of Configurations That Have a Birth in Each Inside Vertex Group

Second, we eliminate configurations that have a birth in each inside vertex group. For each inside vertex group, if exactly three tiles from the same group are alive, we have at least one birth at the tile in the same inside vertex group. Thus, we create two functions, `pre_step_two` and `step_two`. The first function, `pre_step_two`, gives us a list of vectors with three tiles, including the center tile from the selected inside vertex group. The function uses one parameter `inside_num`, an inside vertex group. The second function, `step_two`, eliminates configurations that have a birth in each inside vertex group. The function uses three parameters: `all_num_neighbors`, `total_insideG`, and `remaining_vectors`. Three parameters correspond to all tile numbers of the neighborhood, all inside vertex groups, and the remaining potential still life configuration after the previous step.

We start by generating an empty vector (`allL`) to store configurations that has birth in the inside vertex group. The function works in two ways depending on the number of tiles in the inside vertex group. If there are four tiles in the inside vertex group, the function first selects three tiles from the inside vertex group using `pre_step_two`. Assume the unselected tile from the inside vertex group to be x_i. Then, the function selects the last tile that does not share any inside or outside vertex group with x_i. This way, the configuration has birth at x_i, and we store this configuration in `allL`. If we choose the last tile otherwise, x_i has four neighbors, and we do not have a birth.

When there are more than four tiles in the inside vertex group, the function first selects three tiles from the inside vertex group using `pre_step_two`. Then, the function selects the last tile that is not in the same inside vertex group. This way, the configuration has at least one birth at unselected tiles within the inside vertex group, and we store this configuration in `allL`. After going over all inside vertex groups, we get the union of vectors in `allL`

and assign it to `get_rid_step2`. In the end, we compute the difference between two sets `remaining_vectors` and `get_rid_step2`, which is the function's output.

7.1.3 Step 3. Get Rid of Configurations That Have a Birth in Each Outside Vertex Group

Third, we eliminate configurations that have a birth in each outside vertex group. Since we consider cases with center tile alive, the center tile shares at least one inside vertex with tiles in each outside vertex group. Thus, if two tiles from the outside vertex group and one unrelated tile are selected, we have at least one birth at the tile in the same outside vertex group. Thus, we create three functions, `add_zero`, `sort_vector_vector`, and `step_three`.

The first function, `add_zero`, uses `list`, which is a list of vectors as a parameter and appends 0 to each vector inside the list. We use this function to create a configuration with the center cell alive. The second function, `sort_vector_vector`, also use `list`, which is a list of vectors as a parameter and sorts each vector inside the list in ascending order. The third function, `step_three`, eliminates configurations that have a birth in each outside vertex group. The function uses two parameters: `outside_group` and `remaining_vectors`. Two parameters correspond to all outside vertex groups and the remaining potential still life configuration after the previous step.

We start by generating an empty vector (`allL`) to store configurations that has birth in the outside vertex group. Going over all outside vertex groups, the function selects two tiles from each outside vertex group and creates a list of vectors with a two-tiles configuration (`step3`). Assume the unselected tile from the outside vertex group to be x_o. Then, the function generates a list of tiles (`notL`) that does not share any inside or outside vertex group with x_o. Next, we add the center tile to each configuration in `step3` using `add_zero` function, sort each vector, and add the last tile from `notL`. This way, the configuration has a birth at unselected tile, x_o, and we store this configuration in `allL`. After going over all outside vertex groups, we get the union of vectors in `allL` and assign it to `get_rid_step3`. In the end, we compute the difference between two sets `remaining_vectors` and `get_rid_step3`, which is the function's output.

7.1.4 Step 4. Get Rid of Configurations That Have Birth at Tiles That Belong to Two Inside Vertex Groups

Fourth, we eliminate configuration that has birth at tiles that belong to two inside vertex groups. For all neighborhoods in Robinson triangle tiling, three tiles belong to two inside vertex groups. When the center cell is alive, if we want a birth at any of those tiles, we choose two tiles from their two inside vertex groups and one additional tile that is not in the two inside vertex groups.

Thus, we create three functions, `step4_two_combo`, `step4_three_set_combination`, and `step_four`. The first function, `step4_two_combo`, uses `group1` and `group2` which are

first two components of values in each element in the dictionary. The dictionary is given from preset variables and has information on three tiles that belong to two inside vertex groups. Using these parameters, the function creates a list of configurations with two tiles from each group. The second function, `step4_three_set_combination` uses three parameters `two_comboL`, `third_set`, and `third_set_vector_num` and give us a list of configurations with three tiles. Three parameters correspond to an output from the first function, the last components of values in each element in the same dictionary from above, and an index number of tiles in `third_set`. Going over all two-tile configurations, we add a third tile in `third_set` using the index parameter.

The third function, `step_four`, eliminates configurations that have birth at tiles that belong to two inside vertex groups. The function uses two parameters: `step4L` and `remaining_vectors`. Two parameters correspond to the dictionary with information on three tiles that belong to two inside vertex groups and the remaining potential still life configuration after the previous step.

We start by generating an empty vector (`allSet`) to store configurations that has birth at tiles that belong to two inside vertex groups. Going over all three such tiles, we also generate an empty vector (`step4`) to store configurations from each iteration. The function selects two tiles from two inside vertex groups and creates a list of vectors with a two-tiles configuration (`L`). Then, we add the third tile that does not belong to two inside vertex groups, add the center tile to each configuration using `add_zero` function, and sort the configuration. This way, the configuration has a birth at tiles belonging to two inside vertex groups, and we store this configuration in `step4`. Then we store `step4` as a list of vectors in `allSet`. After going over all three tiles, we get the union of vectors in `allSet` and assign it to `get_rid_step4`. In the end, we compute the difference between the two sets of `remaining_vectors` and `get_rid_step4`, which is the function's output.

7.2 Algorithm to Find Still life Where the Center Cell Is Dead and Four Neighboring Cells are Alive

The second algorithm has five steps listed below:

1. Get rid of configurations that share outside neighbors.
2. Get rid of configurations that have a birth in each inside vertex group.
3. Get rid of configurations that have a birth in each outside vertex group.
4. Get rid of configurations that have birth at tiles that belongs to two inside vertex groups.
5. Get rid of configurations that have a death.

The algorithm uses the same second preset variables from the Section 7.1 but now we assume the neighborhood in Robinson triangle tiling has n tiles in the neighborhood without including the center cell. Then we have $\binom{n}{4}$ possible configurations if the center cell is dead and four cells on the neighborhood are alive. To get a list of vectors with four tile numbers representing configurations with four cells, we create a function, `all_configuration4` with parameter `n` which is the number of tiles in the neighborhood.

7.2.1 Step 1. Get Rid of Configurations That Share Outside Neighbors

First, we get rid of configurations that share an outside neighbor. We use the same function from Section 7.1.1. However, for consistency for the rest of the algorithm, we rename the function `step_one` to `step_one4`.

7.2.2 Step 2. Get Rid of Configurations That Have a Birth in Each Inside Vertex Group

Second, we eliminate configurations that have a birth in each inside vertex group. Thus, we create two functions, `pre_step_two4` and `step_two4`. The first function, `pre_step_two4`, gives us a list of vectors with three tiles, which does not include the center tile. The function uses one parameter `inside_num`, an inside vertex group. The second function, `step_two4`, uses two parameters: `total_insideG`, and `remaining_vectors`. Two parameters correspond to all inside vertex groups and the remaining potential still life configuration after the previous step.

We start by generating an empty vector `L` to store configurations that has birth in the inside vertex group. Going over all inside vertex groups, the function first checks how many tiles are in each group. Suppose the inside group has four tiles which always include the center tile. In that case, we cannot generate a configuration that has a birth inside the vertex group. This is because all tiles outside the inside vertex group are connected to the center tile so the last tile we could select is always related to such inside vertex group.

If there are more than four tiles in the inside vertex group, the function first selects three tiles from the inside vertex group using `pre_step_two4`. Then, going over all unselected tiles from the inside vertex group, the function selects the last tile that does not share any inside or outside vertex group with each unselected tile, and we store the configuration in `L`. In the end, we get rid of overlapping configurations in `L`, sort them in ascending order, and assign it to `get_rid_step2`. Finally, we compute the difference between two sets `remaining_vectors` and `get_rid_step2`, which is the function's output.

7.2.3 Step 3. Get Rid of Configurations That Have a Birth in Each Outside Vertex Group

Third, we eliminate configurations that have a birth in each outside vertex group. To get a configuration that we can eliminate, we first choose two tiles from the outside vertex group.

Since all outside vertex group has three tiles, we need to find a configuration that can have a birth at the unselected tile from the outside vertex group, x_o. Thus, one tile should be a tile that shares any inside or outside vertex group with x_o, and one tile should be a tile that does not share any inside or outside vertex group with x_o. Therefore, we create a function, `step_three4` with two parameters: `outside_group` and `remaining_vectors`. Two parameters correspond to all outside vertex groups and the remaining potential still life configuration after the previous step.

We generate an empty vector `final` to store configurations that have a birth in the outside vertex group. Going over all outside vertex groups, the function selects two tiles from each outside vertex group and creates a list of vectors with a two-tiles configuration (`step3`). Then, the function generates a list of tiles (`notL`) that do not share any inside or outside vertex group with x_o. Also, the function generates a list of tiles (`inL`) that share any inside or outside vertex group with x_o. After creating an empty vector `L`, the function goes over all tiles in `notL`, adds one tile to all configurations in `step3`, and store configuration in `L`. Then, we create an empty vector `New`, go over all tiles in `inL`, add one tile to all configurations in `L`, and store configuration in `New`. This way, the configuration has a birth at unselected tiles within the outside vertex group, and we store this configuration in `final`. In the end, we get rid of overlapping configurations in `final`, sort them in ascending order, and assign it to `get_rid_step3`. Finally, we compute the difference between two sets `remaining_vectors` and `get_rid_step3`, which is the function's output.

7.2.4 Step 4. Get Rid of Configurations That Have Birth at Tiles That Belong to Two Inside Vertex Groups

Fourth, we eliminate configuration that has birth at tiles that belong to two inside vertex groups. Thus, we create a function `step_four4` with two parameters: `step4L` and `remaining_vectors`. Two parameters correspond to the dictionary with information on three tiles that belong to two inside vertex groups and the remaining potential still life configuration after the previous step. We start by generating an empty vector `allSet` to store configurations that has birth at tiles that belong to two inside vertex groups.

Going over all three such tiles, we first get the union of the first two components of values in each element in the dictionary. Then, the function generates three-cell configurations from the union set, and we add the last tile that does not belong to the union set. This way, the configuration has a birth at the tile that belongs to two inside vertex groups, and we store this configuration in `allSet`. In the end, we get rid of overlapping configurations in `allSet`, and compute the difference between two sets `remaining_vectors` and `allSet`, which is the function's output.

7.2.5 Step 5. Get Rid of Configurations That Have a Death

Finally, we eliminate the configuration that has a death. If an alive cell has one or no neighbor, the cell dies at the next generation. We consider both cases, one neighbor and no neighbor, for all tiles except the center tile in the neighborhood. If we select tile-n and three additional tiles not in tile-n's vertex groups (inside and outside), it has no neighbor. Thus, the tile dies. If we select tile-n, one tile from tile-n's vertex groups (inside and outside), and two additional tiles not in tile-n's vertex groups , it has one neighbor. Thus, the tile dies.

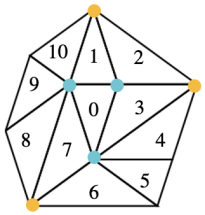
We create a function `step_five4` that follows above procedure. Using three parameters, `numD`, `num_notD` and `remaining_vectors`, the function eliminates configurations that has a death. The first parameter corresponds to the dictionary with tile number as key and vector of tiles that share the same inside or outside vertex groups as value. The second parameter corresponds to the dictionary with tile number as key and vector of tiles that do not share the same inside or outside vertex groups as value. The last parameter is the list of remaining potential still life configurations after the previous step. We start by generating an empty vector `allSet` to store configurations that have a death. Going over all tiles in chosen neighborhood, the function generates three-cell configurations with three tiles that do not share any vertex groups with each tile. Then, creating another empty vector `noNL`, the function appends the current tile to each configuration with three tiles and stores each in `noNL`.

Next, going over all tiles in the chosen neighborhood, the function generates two-cells configurations with two tiles that do not share any vertex groups with each tile. Then, it appends a third tile that shares the same inside or outside vertex group with the current tile. By creating another empty vector `oneNL`, the function adds the current tile to each configuration with three tiles and stores them in `oneNL`. At the end of each iteration, we get the union of `noNL` and `oneNL` and append it to `allSet`. After going over all tiles, we get the union of vectors in `allSet` and assign it to `get_rid_step5`. In the end, we compute the difference between the two sets of `remaining_vectors` and `get_rid_step5`, which is the function's output.

8 Applying Two Algorithms to $10N$

We show how two algorithms can be applied to our neighborhoods using the $10N$ as an example. How inside and outside vertex groups are formed for $10N$ is shown in Figure 9 and configurations that share outside neighbors are shown in Figure 10.

Now we apply the first algorithm and find four-cell still life where the center cell is alive. There are $\binom{10}{3} = 120$ four-cell configurations that can be generated in $10N$. After each step, we eliminate 25, 69, 17, and 5 configurations. Thus, we have four configurations, $\{0, 1, 8, 9\}, \{0, 3, 5, 6\}, \{0, 4, 5, 7\}$, and $\{0, 7, 9, 10\}$, remaining. Checking these configurations graphically, they satisfy still life conditions which are shown in Figure 11.



$10N$

Vertex group	Elements
First inside vertex group (i_1)	{0, 1, 2, 3}
Second inside vertex group (i_2)	{0, 3, 4, 5, 6, 7}
Third inside vertex group (i_3)	{0, 1, 7, 8, 9, 10}
First outside vertex group (o_1)	{1, 2, 10}
Second outside vertex group (o_2)	{2, 3, 4}
Third outside vertex group (o_3)	{6, 7, 8}

Figure 9: Inside and outside vertex groups of $10N$.

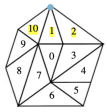
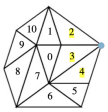
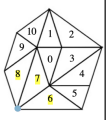
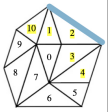
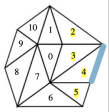
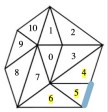
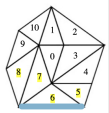
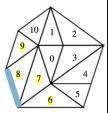
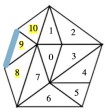
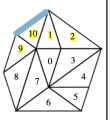
o_1	o_2	o_3	Edge 2	Edge 4	Edge 5	Edge 6	Edge 8	Edge 9	Edge 10
									
{1, 2, 10}	{2, 3, 4}	{6, 7, 8}	{1, 2, 3} {1, 2, 4} {1, 2, 10} {1, 3, 4} {1, 3, 10} {1, 4, 10} {2, 3, 4} {2, 3, 10} {2, 4, 10} {3, 4, 10}	{2, 3, 4} {2, 3, 5} {2, 4, 5} {3, 4, 5}	{4, 5, 6}	{5, 6, 7} {5, 6, 8} {5, 7, 8} {6, 7, 8}	{6, 7, 8} {6, 7, 9} {6, 8, 9} {7, 8, 9}	{8, 9, 10}	{1, 2, 9} {1, 2, 10} {1, 9, 10} {2, 9, 10}

Figure 10: Configurations in $10N$ that share outside neighbors.

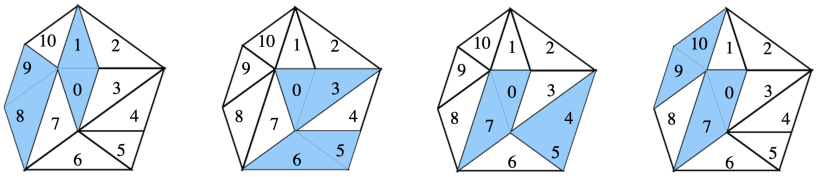


Figure 11: Four configurations that satisfy still life conditions in $10N$ when the center cell is alive.

Next, we apply the second algorithm and find four-cell still life where the center cell is dead. There are $\binom{10}{4} = 210$ four-cell configurations that can be generated in $10N$. After each step, we eliminate 125, 38, 16, 19, and 9 configurations. Thus, we have three configurations, $\{1, 3, 6, 8\}, \{1, 7, 8, 10\}$, and $\{3, 4, 6, 7\}$, remaining. Checking these configurations graphically, they satisfy still life conditions which are shown in Figure 12.

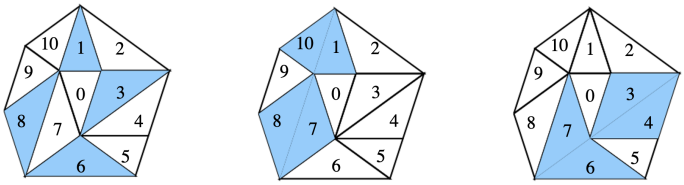


Figure 12: Three configurations that satisfy still life conditions in $10N$ when the center cell is dead.

9 Future Work

We generated an algorithm to play *Game of Life* on Robinson triangle tiling. Also, we created two algorithms to find all four-cell configurations that satisfy still life conditions. Using algorithms to identify still life, we discover all four-cell still life in each neighborhood (Table 1). Thus, our next step is a classification of four-cell still life.

Four-Cell Still Life	
Neighborhood	Total Number
$10N$	7
$12N_1$	42
$12N_2$	42
$14N_1$	154
$14N_2$	154
$16N_1$	159
$16N_2$	189
$18N_1$	301
$18N_2$	194

Table 1: Total number of four-cell still life in each neighborhood.

10 Acknowledgement

The J. Reid and Polly Anderson Endowment funded the first ten weeks of the research, and I thank them for the support of this research. Also, I want to acknowledge and thank Zheng Fang and Raghav Goel for their great feedback and suggestions throughout the process of creating the algorithm to apply *Game of Life* on the Robinson triangle tiling. In addition, I sent my gratitude to Dr. Matthew Neal for reading this thesis as a second reader. Above all, I would like to express my sincere thanks of gratitude to my research advisor, Dr. May Mei, for the supportive mentorship and guidance throughout this year-long research.

References

[1] Siobhan Roberts. The lasting lessons of john conway's game of life. https://www.nytimes.com/2020/12/28/science/math-conway-game-of-life.html?searchResultPosition=1, accessed June 1, 2022.

[2] Julia Computing company. The julia programming language. https://julialang.org/, accessed January 28, 2022.

[3] Mandy S.H. Hong. The game of life on penrose tilings: Robinson triangle. https://github.com/shyeon923/Game-of-Life-on-Penrose-Tiling-Robinson-Triangle, 2022.

[4] Martin Gardner. Mathematical games: The fantastic combinations of john conway's new solitaire game "life". *Scientific American*, 223(4):120–123, 1970.

[5] Martin Gardner. Mathematical games: Extraordinary nonperiodic tiling that enriches the theory of tiles. *Scientific American*, 236(1):110–121, 1977.

[6] Colin Adams. The tiling book: An introduction to the mathematical theory of tilings. unpublished, N.D.

[7] Bielefeld University. Penrose pentagon boat star. https://tilings.math.uni-bielefeld.de/substitution/penrose-pentagon-boat-star/, accessed June 18, 2022.

[8] Bielefeld University. Penrose kite dart. https://tilings.math.uni-bielefeld.de/substitution/penrose-kite-dart/, accessed June 18, 2022.

[9] Bielefeld University. Penrose rhomb. https://tilings.math.uni-bielefeld.de/substitution/penrose-rhomb/, accessed June 18, 2022.

[10] Bielefeld University. Robinson triangle. https://tilings.math.uni-bielefeld.de/substitution/robinson-triangle/, accessed June 18, 2022.

[11] Bielefeld University. Substitution. https://tilings.math.uni-bielefeld.de/glossary/substitution/, accessed June 18, 2022.

[12] Nick Owens and Susan Stepney. *The Game of Life Rules on Penrose Tilings: Still Life and Oscillators*, pages 331–378. Springer London, London, 2010.