Computing Quasiperiodic Patterns in Python

Daniel Gouldsbrough
201324779

May 2021

A Thesis submitted in partial fulfilment of the requirements for the degree of
Bachelor of Science

Under the supervision of Prof. Ronan McGrath and Dr. Hem Raj Sharma

At the
University of Liverpool Department of Physics

Declaration

I hereby declare that this thesis is my own work and effort and that it has not been submitted anywhere for any award. Where other sources of information have been used, they have been acknowledged.

I would like to thank my family, in particular my father for his encouragement. My thanks also go to Dr. Steve Barrett, for his advice on generating FFT images, as well as Sam Coates for aiding me with my implementation of the cut and project method. Finally, I would like to thank both Prof. Ronan McGrath and Dr. Hem Raj Sharma for their regular guidance. In particular Dr. Sharma for his software recommendations and Prof. McGrath for providing many examples of useful literature, including from his own library.

Signature .....

Abstract

This report discusses the generation of images of 2-dimensional quasiperiodic tilings using Python libraries NumPy and Matplotlib. Different methods for generating tilings were investigated, the Fibonacci substitution method, the deflation method, and the cut and project method. Python code was written which produces six different quasiperiodic tilings: the Fibonacci square, Fibonacci cube, Penrose P1, Penrose P2, Penrose P3, and Amman-Beenker A5 tilings. Image processing of the generated quasiperiodic patterns yielded results consistent with published works on quasicrystals [1].

Contents

Declaration i

Abstract i

Contents ii

1 Introduction 1

2 Fibonacci Tiling 2
2.1 The Fibonacci Sequence 2
2.2 The Fibonacci square and cubic tilings 2

3 The Deflation Method 4
3.1 Chair tiling 4
3.2 Penrose tilings 4
3.2.1 Penrose P1 tiling 5
3.2.2 Penrose P2 tiling 5
3.2.3 Penrose P3 tiling 6
3.3 Amman-Beenker tiling 6

4 The Cut and Project Method 8
4.1 The Cut and Project Method in 1D 8
4.2 The Cut and Project Method in Higher Dimensions 8

5 Implementation in Python 9
5.1 Fibonacci Tiling Program 9
5.2 Deflation Method Program 11
5.3 Cut and Project Programs 13

6 Results and Discussion 15

7 Conclusion 21

References 22

Appendices 24

1 Introduction

A tiling is a set of geometric shapes, called tiles, that are arranged in such a way to fill a plane without gaps or overlaps. A tiling can be periodic or non-periodic. Periodic tilings are constructed with a repeating pattern - the tiling has translational symmetry. The different types of tiles that make up a tiling are called the prototiles of that tiling. A tiling may be constructed using one or many prototiles.

The simplest tilings are constructed using regular polygons as the single prototile. There are only three possible regular polygon tilings: made of equilateral triangles, squares or regular hexagons. These have three–, four–, and six–fold rotational symmetry respectively. It is not possible to tile a plane with pentagons, because the internal angle of a pentagon, 108°, is not a divisor of 360°. The same restriction applies to $n > 6$–sided polygons.

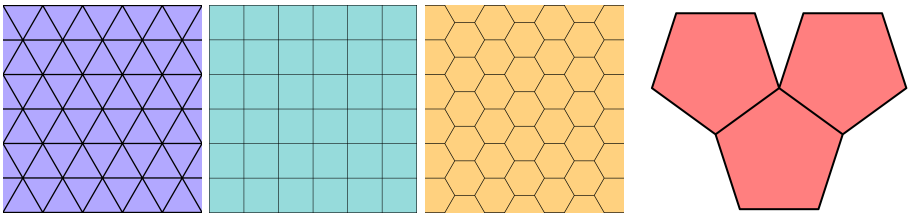


Figure 1: Equilateral triangle tiling, square tiling, and regular hexagon tiling. The 4^{th} image shows the problem that arises when attempting to tile the plane with regular pentagons.

A subset of non-periodic tilings are quasiperiodic tilings. Although these tilings are not periodic, every finite region within the tiling reappears infinitely many times. Some quasiperiodic tilings also exhibit forbidden orders of rotational symmetry.

The concept of quasiperiodic tilings have been of interest to physicists as a mathematical model of quasicrystals (QCs) since their discovery in 1982 by Dan Shechtman [1]. This discovery was made following observations of a rapidly cooled aluminium-manganese alloy through an electron microscope. QCs are structures that possess long-range order, but not translational symmetry. The atomic structures of QCs are associated with quasiperiodic tilings. The QC discovered by Shechtman possesses fivefold symmetry, and other QCs have been found to possess other forbidden symmetries. These materials are typically synthesised in the laboratory. Following analysis conducted in 2009 by Paul Steinhardt on a Siberian meteorite sample [2] the first natural QC, icosahedrite, was found.

The golden ratio, τ, is intrinsically linked to many of the tilings discussed in this report. This irrational number appears when the ratio of two quantities is

equal to the ratio of their sum and the larger value. Expressed algebraically:

$$\tau = \frac{a+b}{a} = \frac{a}{b} = \frac{1+\sqrt{5}}{2}$$

The main objective of this project has been to been to write Python code that reproduces 2-dimensional quasiperiodic tilings, such as are seen in Marjorie Senechal's book "Quasicrystals and geometry" [3]. A further objective has been to adapt this code to produce 3-dimensional models, and to investigate the possibility of 3D printing these structures.

2 Fibonacci Tiling

2.1 The Fibonacci Sequence

The simplest quasiperiodic tiling is the Fibonacci sequence. This is a 1-dimensional set of segments of two possible lengths: short (S) and long (L). The ratio between the length of these two segments is the golden ratio, τ. Generations of this sequence can be created using the substitution rules σ: L→LS, S→L, as seen below:

Generation	Sequence
1^{st}	L
2^{nd}	LS
3^{rd}	LSL
4^{th}	LSLLS
5^{th}	LSLLSLSL
...	...

Table 1: Demonstration of Fibonacci substitution rule.

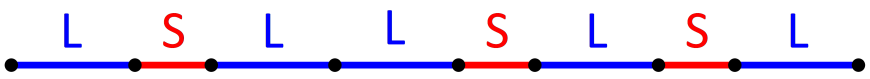


Figure 2: 5^{th} generation Fibonacci sequence. Long segments denoted by blue lines and short segments by red lines. Generated in Python.

2.2 The Fibonacci square and cubic tilings

The 2-dimensional Fibonacci tiling can be obtained by overlapping two 1D Fibonacci sequences orthogonally, and projecting lines through each sequence of points to join the structure together into a grid. This was first shown in a 2002 paper by Ron Lifshitz [4]. As seen in Figure 3, this results in a tiling made up of three prototiles: the L×L square, the S×S square, and the S×L rectangle.

This structure has since been observed experimentally [5]. This was the first example showing quasiperiodicity can also have fourfold symmetry. Before this, it was thought that QCs only possess forbidden symmetries, such as fivefold or tenfold.

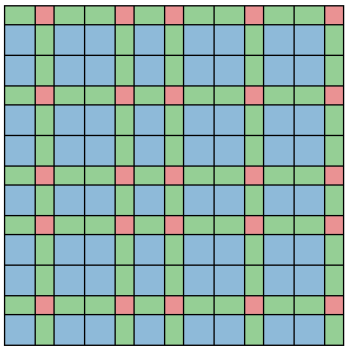


Figure 3: 6^{th} generation 2D Fibonacci square tiling. Generated in Python.

The 3-dimensional cubic Fibonacci tiling can be obtained from the 2D tiling by overlaying a third Fibonacci sequence orthogonal to the 2D plane. This is shown in Figure 4.

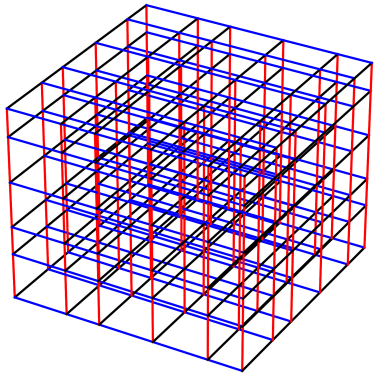


Figure 4: 4^{th} generation 3D Fibonacci cube tiling. Generated in Python.

3 The Deflation Method

The deflation method is the primary method used in this report to generate tilings. This method is similar to the substitution process used to generate Fibonacci sequences, but rather than substituting a value in a sequence, we divide the tile shapes directly. By continuously using the deflation method, an arbitrarily large region of a tiling can be produced. A tiling generated by the deflation method must follow a deflating scheme. This is a process, defined for each prototile, that states exactly how a tile is decomposed into a new set of smaller tiles. This process can then be iterated and applied onto the new tiles produced, to produce arbitrarily large tilings. Deflating schemes may be self-similar or otherwise. Self-similar deflating schemes are unique in that the new tiles generated from each tile occupy the same shape and area as the previous generation. The choice of the initial tile, or set of tiles from which the process starts affects the final structure of later generations of the tiling. For some tilings it may be necessary to start with multiple tiles arranged in a pattern, in order to demonstrate the symmetry of the resulting tiling.

3.1 Chair tiling

The chair tiling is a non-periodic tiling made up of a single prototile [6]. The chair may be rotated by any multiple of 90°. The chair tiling does not possess translational symmetry. This tiling is not quasiperiodic however, in that there are regions of the tiling that appear only once. This is the simplest implementation of the deflation method shown in this report - the tiling is made of one prototile and the deflating scheme is self-similar.

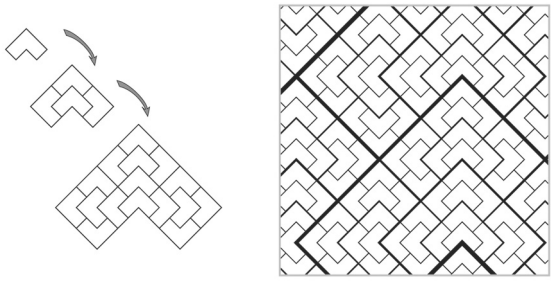


Figure 5: Chair tiling on right, with deflating scheme on left [6].

3.2 Penrose tilings

The Penrose tilings are a set of quasiperiodic tilings first published in 1974 by Roger Penrose [7]. They are made up of three types: P1, P2 and P3. An important feature of these tilings is their fivefold rotational symmetry, which is a trait shared with the first known quasicrystal [1].

3.2.1 Penrose P1 tiling

Penrose's original tiling (P1) is primarily made of pentagons, pentagram, a thin rhombus of 36°, and a section of a pentagram called a 'justice cap'. Although it would appear there are four prototiles, there are in fact six. There are three pentagons in which pentagons must deflate, to distinguish between these we describe them as different prototiles. These three types of pentagon are usually denoted with different colours, although this was not present in Penrose's original paper.

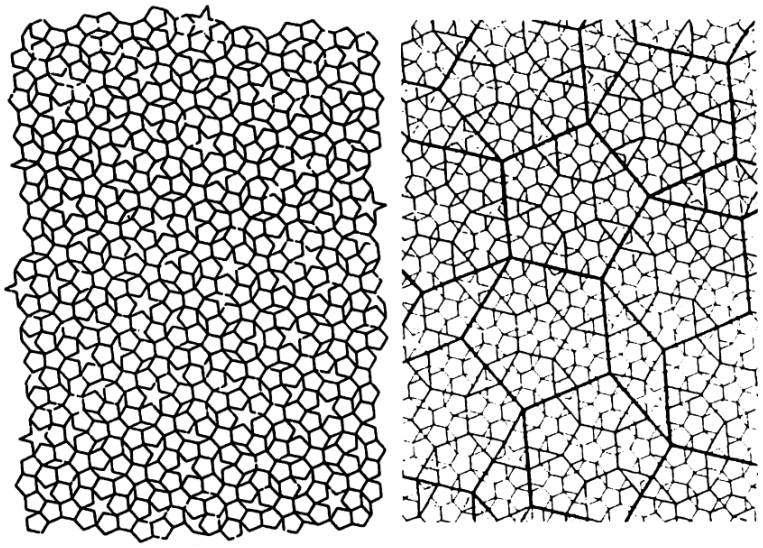


Figure 6: P1 tiling shown on left, overlayed with multiple deflation generations on right [7].

3.2.2 Penrose P2 tiling

The original Penrose tiling can be simplified into a tiling made up of just two tiles. This was first published two years after Penrose's original tiling by Richard Guy [8]. The new tiling formed (P2) is made up of kites and darts. The kite, shown in Figure 7 in light grey, is a quadrilateral with internal angles of 72°, 72°, 72°, and 144°. The dart, in dark grey, is a quadrilateral with internal angles of 36°, 72°, 36°, and 216°. The sides of the prototiles are of two lengths, which are in a ratio of $\tau : 1$. The use of the deflating scheme for kites and darts is shown on the right of Figure 7. Each kite deflates into two kites and two darts, and each dart into two darts and one kite. This deflating scheme produces new tiles smaller in area by τ^2. As the number of tiles in a P2 tiling increases, the ratio between the number of kites to darts tends towards the golden ratio, which is also the ratio between the area of the two tiles.

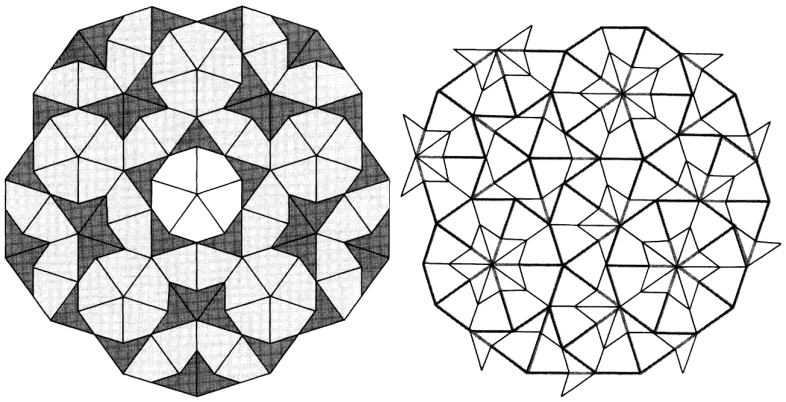


Figure 7: P2 tiling on left, overlayed with next deflation generation on right [9].

3.2.3 Penrose P3 tiling

The final variant of Penrose tilings, the rhombus tiling (P3) is made up of two different rhombuses of equal edge length. Similar to the kite and dart tiling, this can be derived from the original pentagon tiling (P1), also shown in [8]. One has an internal angle of 72°, called the thick or fat rhombus. The other has an internal angle of 36°, called the thin or skinny rhombus. It is possible to join these tiles into a parallelogram. Since this would result in a simple periodic tiling, this joining is forbidden in a P3 tiling. It shares many properties with the kite and dart tiling, such as the frequencies and areas of the two tiles being in a ratio of $\tau : 1$. This type of Penrose tiling is the most commonly seen variant. If lines are extended along the edges of each tile, ten sets of parallel lines will be created, which are separated from each other with distances according to the Fibonacci sequence.

3.3 Amman-Beenker tiling

The Amman-Beenker tilings are a set of tilings discovered by Robert Ammann in the 1970s, and independently by F.P.M. Beenker in 1982 [11]. These are quasiperiodic tilings which exhibit octagonal symmetry. There are five tilings in this set but only A5, the simplest Amman-Beenker tiling, will be shown in this report. Similar to the set of Penrose tilings, this A5 tiling can be derived from the other Amman-Beenker tilings. This tiling is constructed from a square and a 45°rhombus, which share the same edge lengths. The structure of this tiling matches an octagonal quasicrystal discovered in 1987 by Chen and Kuo [12].

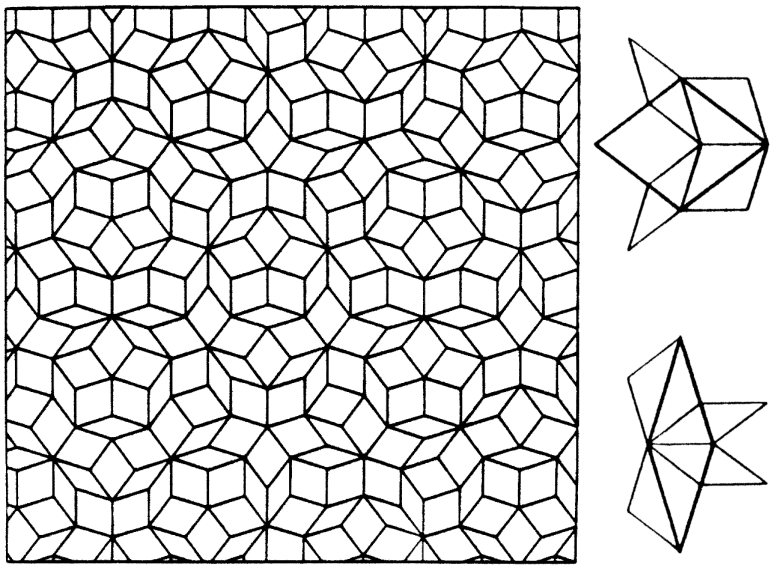


Figure 8: P3 tiling on left, with deflating schemes on right [10].

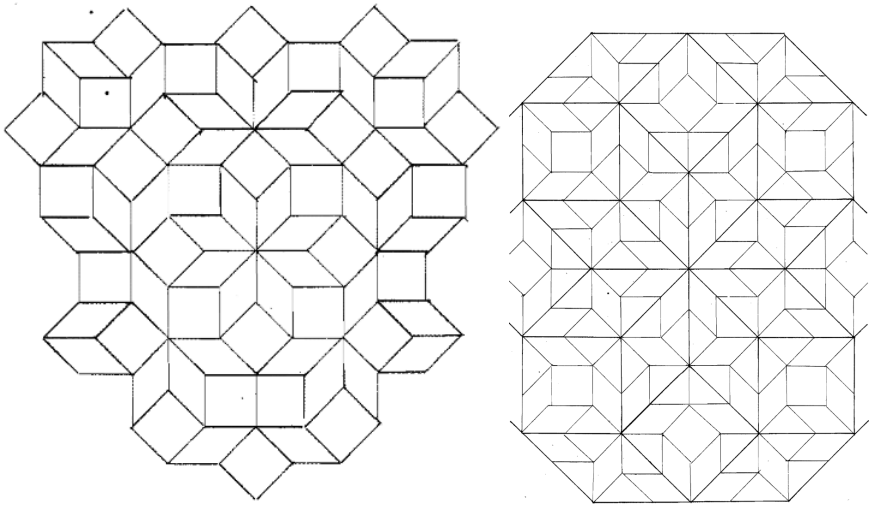


Figure 9: A5 tiling on left, overlayed with next deflation generation on right [11].

4 The Cut and Project Method

The final method for generating tiles discussed in this report is the cut and project method. This method is more versatile than the deflation method and can be extended to generate 3D tilings. It has been shown to be possible to generate the chair tiling [13], the Penrose tilings [14] and the Amman-Beenker tilings [11] through the cut and project method.

4.1 The Cut and Project Method in 1D

In order to generate one-dimensional quasiperiodic sequences, we first define a 2D periodic tiling. A slice is projected through this periodic tiling, at an angle of θ to the x–axis, and of a given thickness Δ. Points that fall within this region are 'accepted' . These accepted points are then projected onto a line parallel to this slice. By then rotating these projected points by the angle θ, these points are now parallel to the x–axis. This process is shown in Figure 10. Provided the angle formed describes an irrational gradient for this region, the sequence produced will exhibit quasiperiodicity. Rational gradients will give periodic sequences. If the thickness and angle of this region are chosen specifically as $\theta = tan^{-1}(\frac{1}{\tau})$ and $\Delta = \tau(sin\theta + cos\theta)$, then the resulting sequence will be a section of the Fibonacci sequence as seen previously. If the thickness is increased by a factor of τ, as seen on the right of Figure 10, the resulting sequence will be another Fibonacci sequence, deflated with respect to the original sequence.

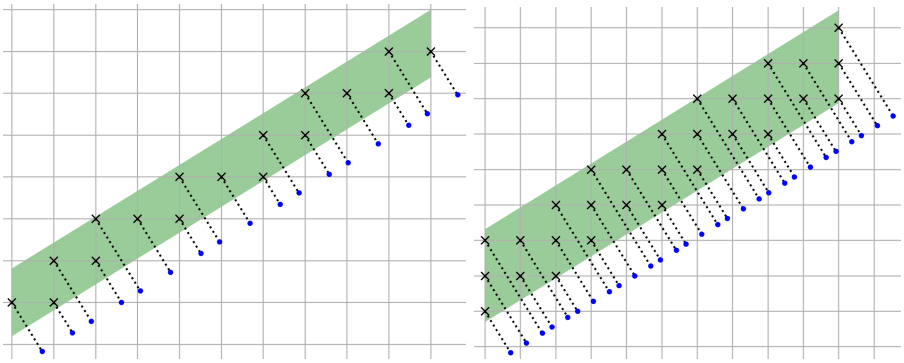


Figure 10: The vertices of a section of the 1D Fibonacci sequence can be created by intersecting a 1D slice at an angle to the x–axis of $\theta = tan^{-1}(\frac{1}{\tau})$ with a 2D lattice. The left figure has a thickness $\Delta = sin\theta + cos\theta$, and the right figure shows a thickness $\Delta = \tau(sin\theta + cos\theta)$.

4.2 The Cut and Project Method in Higher Dimensions

By adding a third dimension to the lattice, it is possible to use the cut and project method to generate 2D tilings. This 3D lattice is intersected by a 2D

window, which is at an angle of θ to both the x–axis and the y–axis. By rotating the accepted points onto the x–y plane, the vertices of the Fibonacci square tiling are formed. Although not achieved by this project, it is also possible to project a window through a 5D lattice in order to generate the vertices of Penrose tilings.

While not achieved during this project, the code that generates Fibonacci square tilings could be extended to higher order lattices to produce 3D (or higher) quasiperiodic patterns. To achieve this it would be necessary to restructure the code into the matrix format as shown in [15].

5 Implementation in Python

This section will outline the programming behind the generation of the images seen in Figures 13-19. There are four programs written for this project: Fibonacci.py, Deflation.py, CutAndProject1D.py, and CutAndProject2D.py. Each of these are written in Python version 3.8, using the following libraries: Math, NumPy, and Matplotlib. These programs are available in full in the appendix.

5.1 Fibonacci Tiling Program

The substitution process for generating Fibonacci sequences is implemented in Python by defining an array of values [0,1] called edgeList. Each generation, 'long' gaps between these values are identified, and a new value inserted into the array to divide this long gap into a long and a short gap. Shown below is an extract from the program displaying this section. Note that N = 4 performs the substitution process four times, to produce a 5^{th} generation sequence.

```
tau = (1 + sqrt(5))/2
edgeList = [0, 1]
longGap = 1
shortGap = 1/tau
N = 4
for x in range(0,N):
    pendingValues= [ ]
    for i in range(0, len(edgeList)-1):
        if isclose(edgeList[i+1]-edgeList[i], long):
            pendingValues.append(edgeList[i] + short)
    edgeList.extend(pendingValues)
    edgeList.sort()
    longGap = longGap/tau
    shortGap = shortGap/tau
```

To create the 1D image in Figure 2, an if statement checks whether the gap between two values is equal to the current value of the variable 'long' or 'short' to determine whether segments should be coloured blue or red. Next, the vertices of the tiling can be displayed onto a Matplotlib figure as dots by

treating the values in the array as y–values along the x–axis. The code handling this is shown below, with some detail omitted:

```
for i in range(len(edgeList)-1):
    if isclose( edgeList[i+1]-edgeList[i], long ):
        plt.plot( [ edgeList[i], edgeList[i+1] ] , [0,0], 'b' )
    if isclose(edgeList[i+1]-edgeList[i], short):
        plt.plot( [ edgeList[i], edgeList[i+1] ] , [0,0], 'r' )
    plt.plot( edgeList[i], 0 , 'ko' )
    plt.plot( edgeList[i+1], 0 , 'ko' )
```

Starting from the array containing the Fibonacci sequence's vertices in the previous section, the 2D Fibonacci grid is generated by plotting a vertical line at each value stored in the array. A set of horizontal lines can be generated by using the same list as y-values rather than x-values. This is a more primitive method than will be used for the rest of the 2D tilings shown in this report, and as such further image processing in an external program is required to add the colour as seen in Figure 3.

```
for i in range(len(edgeList)):
    plt.plot( [ edgeList[i], edgeList[i] ],
              [ edgeList[0], edgeList[-1] ], 'k' )
    plt.plot( [ edgeList[0], edgeList[-1] ],
              [ edgeList[i], edgeList[i] ], 'k' )
```

The 3D Fibonacci square tiling shown in Figure 4 uses the same principle, but with a third orthogonal direction in the z–axis. The code is shown here:

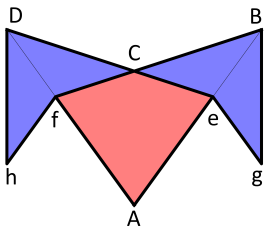
```
for i in range(len(edgeList)):
    for j in range(len(edgeList)):
        ax.plot( [ edgeList[i], edgeList[i] ],
                 [ edgeList[j], edgeList[j] ],
                 [ edgeList[0], edgeList[-1] ], 'r' )
        ax.plot( [ edgeList[i], edgeList[i] ],
                 [ edgeList[0], edgeList[-1] ],
                 [ edgeList[j], edgeList[j] ], 'k' )
        ax.plot( [ edgeList[0], edgeList[-1] ],
                 [ edgeList[i], edgeList[i] ],
                 [ edgeList[j], edgeList[j] ], 'b' )
```

5.2 Deflation Method Program

The code handling the deflation method is the 2nd Python program, Deflation.py attached in the appendix. This program follows the approach outlined in the programming language C in [3], particularly for the chair tiling. The deflating scheme for P1 tiling is implemented based on [16], with the other deflating schemes coming from [17]. Tiles are defined in the code as nested arrays, starting with a string defining the nature of the tile, followed by the 2D coordinates of each vertex in anticlockwise order. For example, a single chair tile is stored as:

```
( 'chair', (2,2), (2,4), (0,4), (0,0), (4,0), (4,2) )
```

The current list of tiles in use are stored in a further array. There are multiple starting tiles stored in the program which are selected based on which tiling the user selects. For some tilings, for example P2, the tiling begins from a set of multiple tiles. For every tiling stored in the code, there is a deflation function defined for each of that tiling's prototiles. For example, the P2 tiling has a deflation function for the kite tiles, and a second function for the dart tiles. A snippet of code showing the latter is shown below as an example:




```
def deflate_dart(dart):
    A = np.array(dart[1])
    B = np.array(dart[2])
    C = np.array(dart[3])
    D = np.array(dart[4])
    e = (B - A)/tau + A
    f = (D - A)/tau + A
    g = A + B - C
    h = A + D - C
    return [ ( 'kite', A, e, C, f ),
             ( 'dart', B, C, e, g ),
             ( 'dart', D, h, f, C ) ]
```

The four vertices of the dart handled by the function are stored as NumPy arrays of length two (A–D). Note that the variable 'tau' is defined previously in the program as the golden ratio. A series of coordinates within and around the tile (e–h) are calculated, and three new tiles are generated using these new coordinates. Note that the name of the function matches the identifying string stored at the beginning of dart tiles. This is used when selecting the correct deflation function for a given tile.

A variable 'N' is requested from the user to determine how many times the deflating schemes are used on the original tiling. This value is 1 less than the tilings generation, for example applying the deflating schemes on a tile 3 times would produce the 4th generation tiling. Each iteration, the previous set of tiles are deleted, and replaced by the new set of deflated tiles. Any duplicates are deleted at this stage by comparing the coordinates of the centres of each tile, although this is only necessary for tilings that follow deflating schemes which are not self-similar.

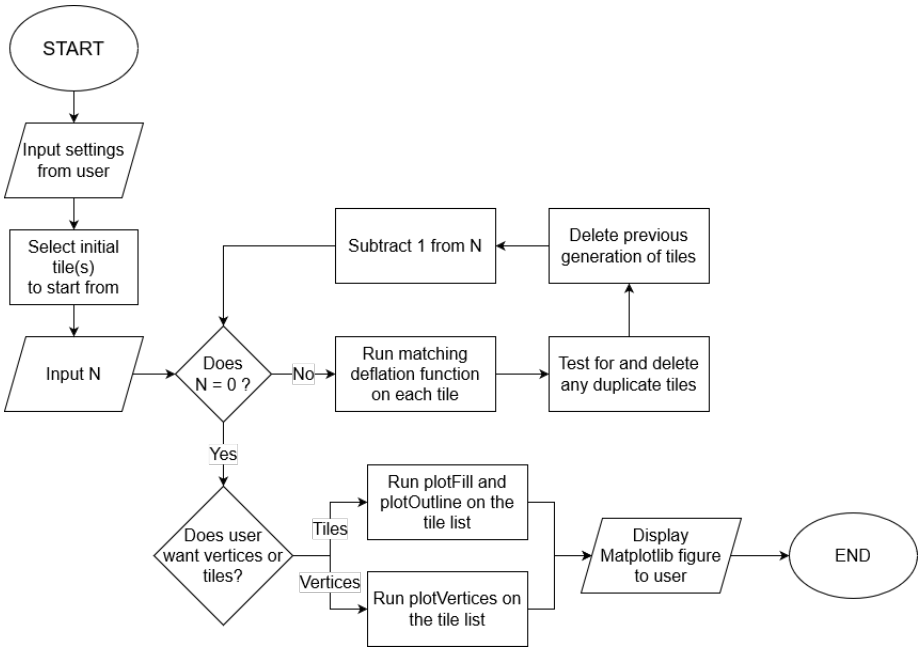


Figure 11: Flowchart describing structure of Deflation.py program.

Once the final array of tiles has been generated from iterations of the deflate functions, there are three plotting functions that the array of tiles may be sent to. The first two of these are plotFill and plotOutline. These fill each tile by it's assigned colour and draw a thin black line around each tile respectively. These two functions together produce the tiling images as seen in Figures 5–10. The other plotting function, plotVertices, creates small black dots at each vertex of each tile. This is shown later in Figure 22, and is intended for use in other programs for further image processing.

```
Choose tiling type: triangle, sierpinski, square, hexagon, chair, P1, P2, P3 or A5: square

How many deflation iterations: 2

Fill tiles or plot vertices? Choose between: fill or vertices: fill
```

Figure 12: The interface that the user is presented with when running Deflation.py, shown in the Spyder integrated development environment. Through this interface the user is able to create any of the tilings written into the program.

5.3 Cut and Project Programs

There are two separate programs explained in this section. The first, CutAndProject1D.py intersects a 2-dimensional lattice with a 1-dimensional slice to produce a 1-dimensional Fibonacci sequence. The second, CutAndProject2D.py creates a 2-dimensional Fibonacci square from a 3-dimensional lattice. Both programs start by creating the figure, and defining the following variables and functions:

```
plt.figure(dpi=1200,figsize=(5,1))
plt.axis('off')
plt.axis('equal')

tau = (1 + sqrt(5))/2
gradient = (1/tau)
theta = atan(gradient)
thickness = ( cos(theta) + sin(theta) )*tau
verticalHeight = thickness * sqrt(gradient**2 + 1)
regionWidth = 5

def maxAccepted(x):
    return gradient*x + verticalHeight/2
def minAccepted(x):
    return gradient*x - verticalHeight/2
```

By changing the gradient and thickness variables here, different tilings will be generated. The maxAccepted and minAccepted functions will be used in the next function, isWithinRegion, to determine whether a given point lies within the accepted region. This function varies slightly between between the 1-dimensional and the 2-dimensional cases. For simplicity, only the 1-dimensional function will be shown for the rest of this section. The 2-dimensional case is very similar, and both full programs are shown in the appendices.

```
def isWithinRegion(x,y):
    if y > minAccepted(x) and y < maxAccepted(x):
        return True
```

Next, the program must decide over which region it will test using the isWithinRegion function. In the 1-dimensional case, the code tests all combinations of x–values between 0 and the variable regionWidth (default 5) and y–values between yMin and yMax. These two variables must be calculated prior to this.

```
yMax = ceil(maxAccepted(regionWidth))
yMin = floor(minAccepted(0))
```

```
x_accepted, y_accepted = [ ], [ ]
for x in range(regionWidth):
    for y in range(yMin,yMax):
            if isWithinRegion(x,y):
                    x_accepted = x_accepted + [x]
                    y_accepted = y_accepted + [y]
```

This returns a list of x–values and their corresponding y–values, which when combined give a list of coordinates within the acceptance region. Next, these points are projected onto the centre of the region, and plotted onto a Matplotlib figure.

```
for i in range(len(x_accepted)):
    x_final = (x_accepted[i]+y_accepted[i]*gradient)/(1+gradient**2)
    plt.plot(x_final, 0, 'ko', markersize=4 )
```

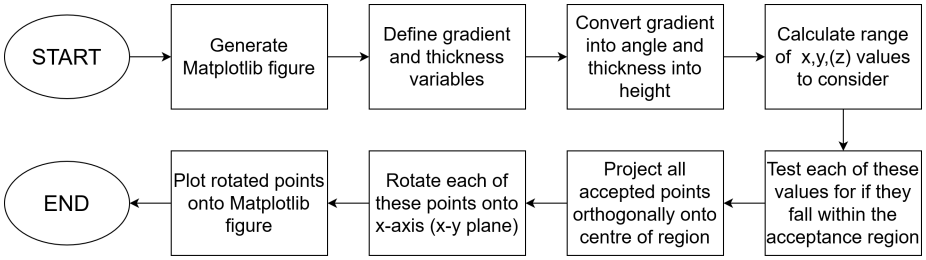


Figure 13: Flowchart describing structure of both cut and project programs.

6 Results and Discussion

Figures 14–20 show some outputs of the deflation program, and Figures 22-23 show an example of their application. Each of the tilings shown can be reproduced from the programs in the appendices. Also shown in Figures 24–25 are the outputs of the cut and project programs.

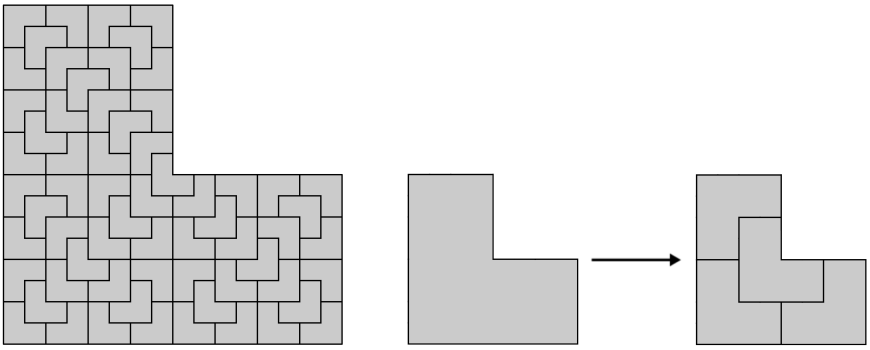


Figure 14: 4^{th} generation chair tiling on left, with deflating scheme on right.

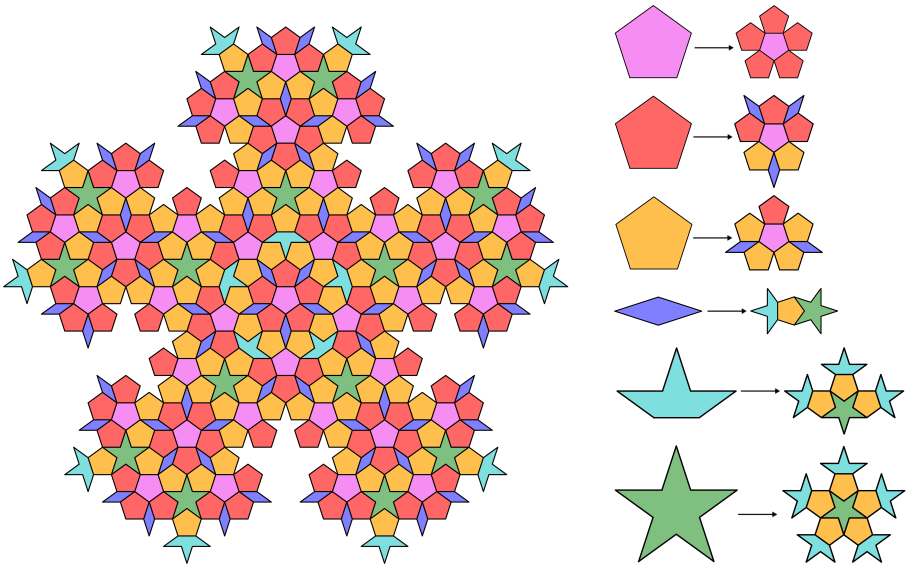


Figure 15: 4^{th} generation P1 tiling shown on left, with deflating schemes on right

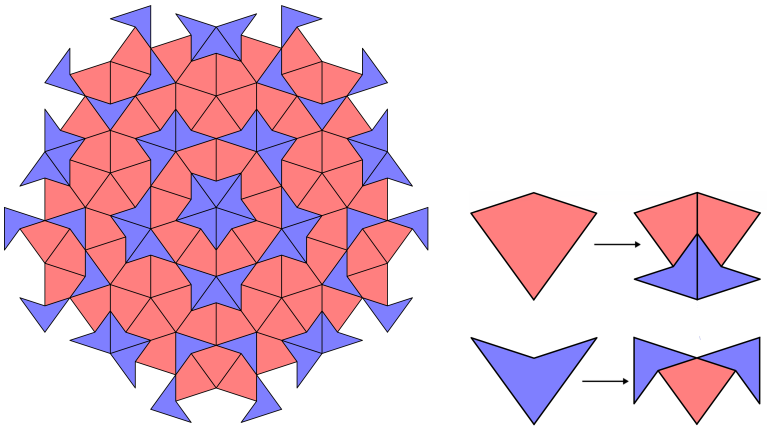


Figure 16: 4^{th} generation P2 tiling, with deflating schemes on right

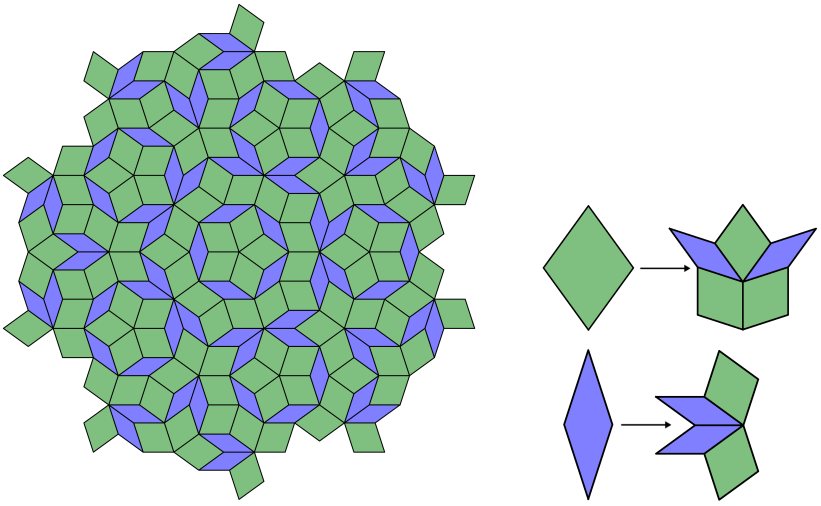


Figure 17: 4^{th} generation P3 tiling, with deflating schemes on right

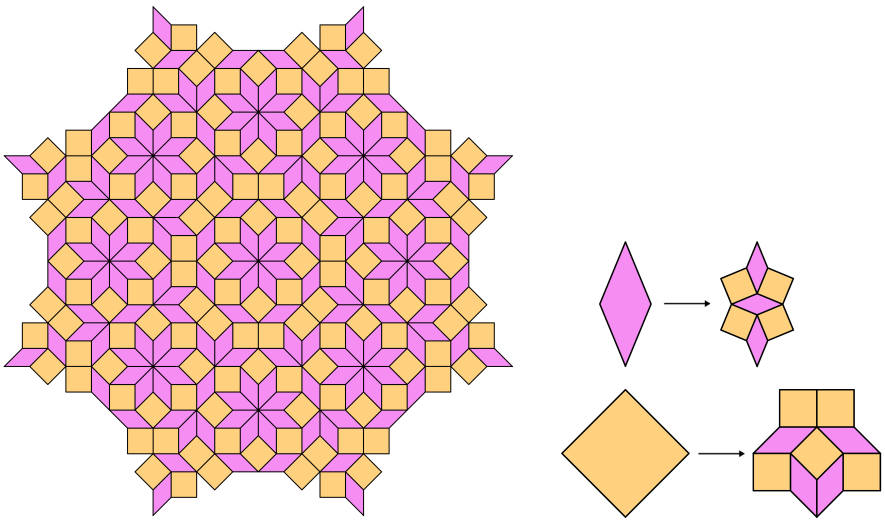


Figure 18: 4^{th} generation A5 tiling, with deflating schemes on right

The Sierpiński triangle is not a tiling because it does not fill the plane without gaps. Nonetheless it is an interesting application of the deflation program written for this report. It is a self-similar fractal first described mathematically by Wacław Sierpiński in 1915 [18], although this pattern has been known since ancient times [19]. It is constructed by dividing an equilateral triangle into three similar triangles, each with a quarter of the area, leaving the central space empty. This process can be implemented recursively, similar to the deflating schemes for the tilings in this report.

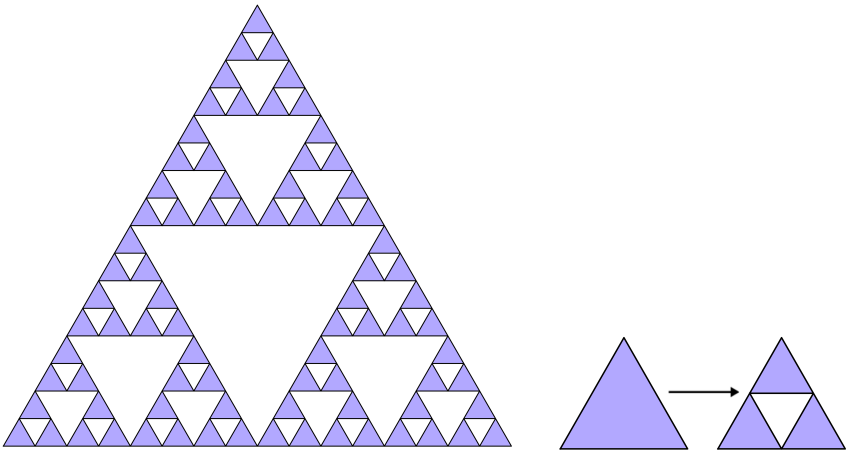


Figure 19: 5^{th} generation Sierpiński fractal, with deflating scheme on right

One limitation of this method for producing tiles is the 'missing' tiles produced at the edge of high generation tilings, as shown in Figure 20. This is intrinsic to some deflating schemes. For simpler gaps of just one tile, external programs can be used to fill in the missing tiles. For more complex tilings, it may be more desirable to produce a higher generation tiling than is needed, and crop this edge effect out. Two approaches to this are shown in Figure 21.

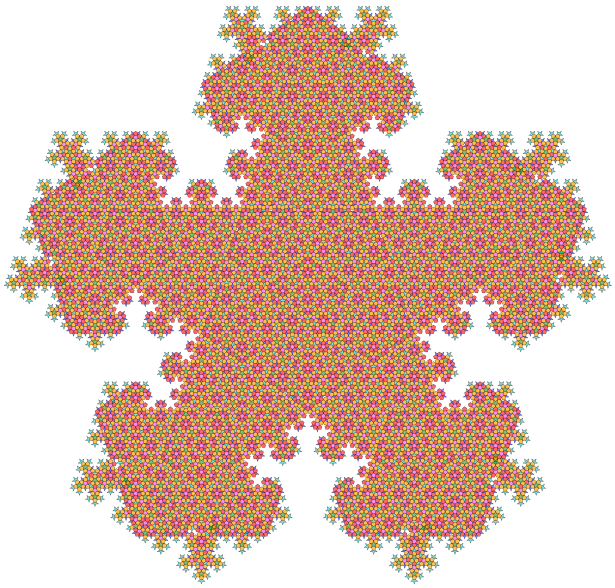


Figure 20: Uncropped 6^{th} generation P1 tiling

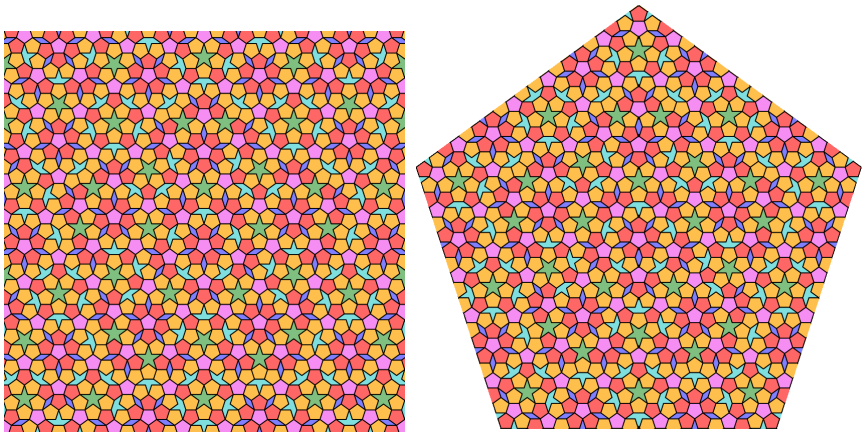


Figure 21: Left: Square–cropped 6^{th} generation P1 tiling
Right: Pentagonally–cropped 6^{th} generation P1 tiling

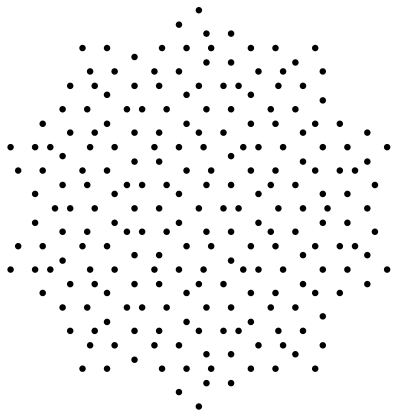


Figure 22: Vertices plot of 4^{th} generation P3 tiling, obtained using the plotVertices option available in the deflation program.

This vertices image shown in Figure 22 can be processed with a 2D autocorrelation, which is a process that reveals the symmetry of a structure by displaying the nearest neighbour to each of the points. This produces the image shown on the left of Figure 23. This has similar symmetry to the image on the right, which is the first evidence obtained of the existence of quasicrystals by Dan Shechtman in 1984. Although there is a variation between the two images in intensity, the positions are equivalent.

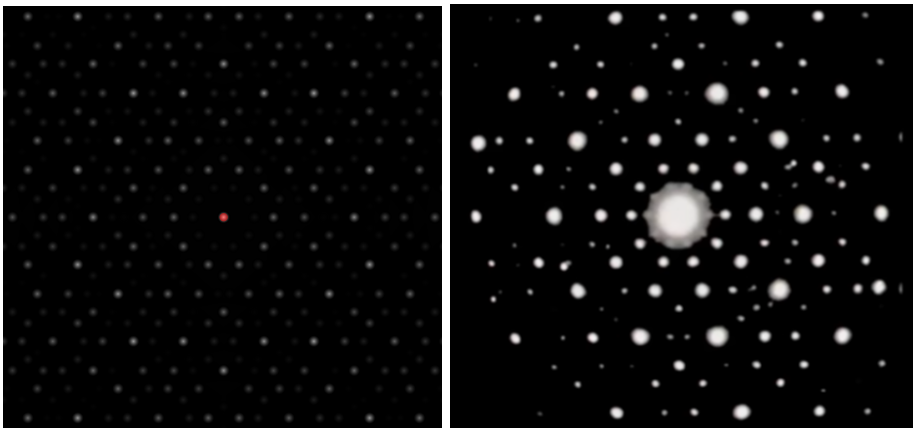


Figure 23: Left: 2D autocorrelation of vertices of 4^{th} generation P3 tiling, generated using Gwyddion.
Right: Electron scattering of rapidly cooled Al_6Mn, reprinted from [1].

Figure 24: The vertices of a section of the 1D Fibonacci sequence, obtained by the cut and project method. A 1D slice at an angle to the x–axis of $\theta = tan^{-1}(\frac{1}{\tau})$ and of thickness $\Delta = \tau(sin\theta + cos\theta)$ is intersected with a 2D lattice.

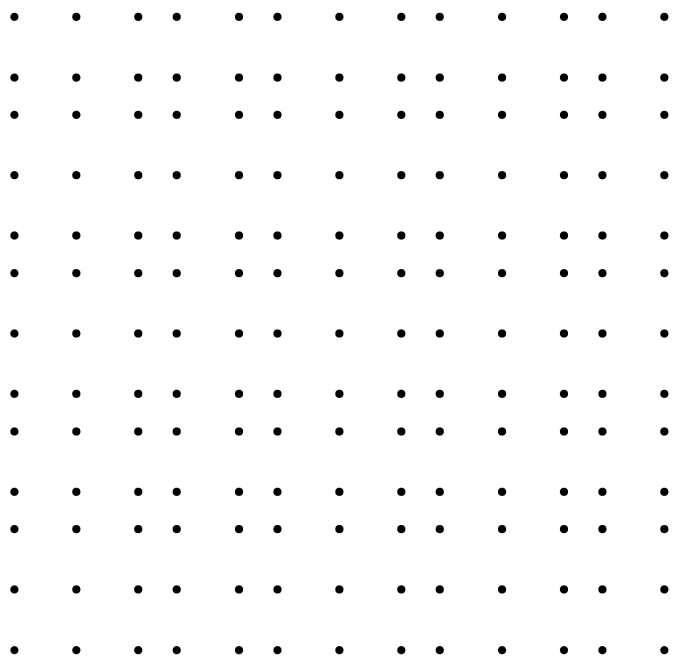


Figure 25: The vertices of a section of the 2D Fibonacci square tiling, obtained by the cut and project method. A 2D window at an angle to the x– and y–axis of $\theta = tan^{-1}(\frac{1}{\tau})$ and of thickness $\Delta = \tau(sin\theta + cos\theta)$ is intersected with a 3D lattice.

7 Conclusion

To summarise, this report has introduced the mathematical concept of tiling, specifically quasiperiodic tiling, as well as explaining the application of quasiperiodic tilings on the modelling of quasicrystals. This report also introduces two main methods for obtaining tilings: the deflation method and the cut and project method. The main objective of this project was to develop Python code that can generate images 2-dimensional quasiperiodic tilings. To achieve this result, the deflation method has been utilised to generate images of P1, P2, P3, and A5 quasiperiodic tilings.

A further objective of this project was to create 3-dimensional quasiperiodic structures that could be 3D printed. Although one 3-dimensional quasiperiodic structure, the cubic Fibonacci tiling, has been generated in Python, further development is needed to develop more complicated 3-dimensional structures, likely by using the cut and project method in higher dimensions. Modifications to the program would also need to be made to produce files in a format for 3D printing. With 3D printing, we can produce quasicrystalline metamaterials with different scales and investigate their physical properties that are important for applications.

References

[1] D. Shechtman, I. Blech, D. Gratias, and J.W. Cahn. Metallic phase with long-range orientational order and no translational symmetry. *Phys. Rev. Lett.*, 53(20):1951–1953, 1984.

[2] L. Bindi, P. Steinhardt, N. Yao, and P. Lu. Natural quasicrystals. *Science (New York, N.Y.)*, 324:1306–9, 2009.

[3] M. Senechal. *Quasicrystals and geometry*. Cambridge University Press, 1995.

[4] R Lifshitz. The square fibonacci tiling. *Journal of Alloys and Compounds*, 342:186–190, 2002.

[5] S. Coates, J. Smerdon, R. McGrath, and H. Sharma. A molecular overlayer with the fibonacci square grid structure. *Nature Communications*, 9, 2018.

[6] C. Goodman-Strauss. *Aperiodic Hierarchical Tilings. In: J.F. Sadoc, N. Rivier (eds) Foams and Emulsions*, pages 481–496. Springer Netherlands, Dordrecht, 1999.

[7] R. Penrose. The role of aesthetics in pure and applied mathematical research. *Bull. Inst. Math. Appl.*, 10:266–271, 1974.

[8] R. K. Guy. The penrose pieces. *Bull. London. Math. Soc.*, 8:9–10, 1976.

[9] M. Gardner. Extraordinary non-periodic tiling that enriches the theory of tiles. *Scientific American*, 236:110–121, 1977.

[10] P.J. Steinhardt and D. Levine. Quasicrystals i. definition and structure. *Physical Review B*, 34:596, 1986.

[11] F.P.M. Beenker. *Algebraic theory of non-periodic tilings of the plane by two simple building blocks : a square and a rhombus*. Eindhoven University of Technology, 1982.

[12] N. Wang, H. Chen, and K. H. Kuo. Two-dimensional quasicrystal with eightfold rotational symmetry. *Phys. Rev. Lett.*, 59:1010, 1987.

[13] M. Baake, R. V. Moody, and M. Schlottmann. Limit-(quasi)periodic point sets as quasicrystals withp-adic internal spaces. *Journal of Physics A: Mathematical and General*, 31(27):5755–5765, 1998.

[14] N. G. de Bruijn. Algebraic theory of penrose's non-periodic tilings of the plane. *Kon. Nederl. Akad. Wetensch. Proc. Ser.*, 43(84):1–7, 1981.

[15] S. Coates and R. Tamura. High dimensional approach to antiferromagnetic aperiodic spin systems. *Materials Transactions*, 62:307–311, 2021.

[16] R. Penrose. Pentaplexity: a class of non-periodic tilings of the plane. *The mathematical intelligencer*, 2(1):32–37, 1979.

[17] B. Grünbaum and G. C. Shephard. *Tilings and patterns*. Freeman, 1987.

[18] W. Sierpiński. Sur une courbe dont tout point est un point de ramification. *C. R. Acad. Sci.*, t. 160:302–305, 1915.

[19] E. Conversano and L Tedeschini Lalli. Sierpinski triangles in stone on medieval floors in rome. *Aplimat Journal of Applied Mathematics*, 4:113–122, 2011.

[20] L. Fibonacci and L. E. Sigler. *Fibonacci's Liber abaci: A translation into modern English of Leonardo Pisano's Book of calculation*. New York: Springer, 2002.

[21] L. Resta. La successione di fibonacci nella fillotassi. Master's thesis, University of Bologna, 2010.

[22] Bravais L. and Bravais A. Essai sur la disposition des feuilles curviseriees. *Ann. Sci. Nat.*, 7:42–348, 1837.

[23] K. F. Schimper. Beschreibung des symphytum zeyheri und seiner zwei deutschen verwandten der s. bulbosum schimper und s. tuberosum jacq. *Magazin für Pharmacie*, 28:3–49, 1830.

[24] I. Adler, D. Barabe, and R.V. Jean. A history of the study of phyllotaxis. *Annals of Botany*, 80:231–244, 1997.

[25] T. Okabe. Biophysical optimality of the golden angle in phyllotaxis. *Scientific Reports*, 5:15358, 2015.

[26] E. Zeckendorf. Représentation des nombres naturels par une somme de nombres de fibonacci ou de nombres de lucas. *Bull. Soc. R. Sci. Liège*, 41:179–182, 1972.

[27] Jean Berstel, Dominique Perrin, and Christophe Reutenauer. *Codes and Automata*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2009.

Appendices

Response to Reviewer Questions

Question 1: It is well known that the Fibonacci sequence is often seen in nature. Can you give an example and explain if there is an underlying physical reason to such natural occurrences?

One of the most well-known examples of the Fibonacci sequence in nature was originally posed by the Italian mathematician Fibonacci in 1202 [20]. We imagine a population of rabbits, made up of adult pairs of rabbits (A) and pairs of kitten rabbits (K). Adult pairs of rabbits give birth each month to a new pair of kittens, which can they themselves mate at the age of one month. The rabbit population in future months resulting from these rules in shown in the following table.

Month	**Rabbit Population**
January	A
February	A, K
March	A, K, A
April	A, K, A, A, K
May	A, K, A, A, K, A, K, A

Table 2: Fibonacci rabbit population growth over five months

This table is analogous to the decomposition of long and short segments shown in Table 1. Table 1 shows the implementation of the substitution rules σ: L$\rightarrow$LS, S$\rightarrow$L, while the monthly change in rabbit population can be similarly described as σ: A$\rightarrow$AK, K$\rightarrow$A. This model is clearly simplified - we assume none of these rabbits die, and that every litter of kittens is comprised of exactly one male and one female rabbit. Nonetheless, this connection shows why the Fibonacci sequence may be observed in the study of population dynamics.

Another natural occurrence of the Fibonacci sequence is in the location of the the growth of leaves on a plant stem [22]. This field of botany is called phyllotaxis [23]. For most plants, the rotational angle between two leaves on a stem is the ratio of one Fibonacci number to its second successor. For example, in pear trees this angle is $\frac{3}{8}$ of a full rotation, and in almond trees this angle is $\frac{5}{13}$ of a full rotation [24]. Similarly, the pattern of seeds within the flower head of a sunflower follow the Fibonacci sequence.

The most likely reason for this occurrence of the Fibonacci sequence is that it is the most effective growth pattern. This arrangement of leaves maximises their exposure to the Sun, allowing for an optimal rate of photosynthesis [25].

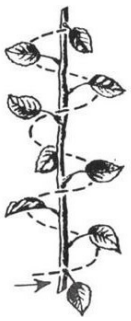


Figure 26: Leaf positions on a pear stem [21]

Question 2: Is there any application of the Fibonacci sequence in coding? And if so, how could it be used?

The Fibonacci sequence has an application in computing called Fibonacci coding. This is a technique for storing positive integers as binary strings. This technique relies on Zeckendorf's theorem, which states that any positive integer can be expressed uniquely as a sum of non-consecutive Fibonacci numbers [26]. Although beyond the scope of this report to formally define, this theorem gives rise to a numeral system called the Zeckendorf representation. This uniquely defines every positive integer as a binary string, although it should not be confused with the binary value of that number. An important property of this system is that the value of 1 never appears twice in a row, due to the non-consecutive requirement of Zeckendorf's theorem. This means that by concatenating a value of 1 to the end of the Zeckendorf representation of a number, the end of a value can be marked with the unique occurrence of '11' .

Integer	**Zeckendorf representation**	**Fibonacci code string**
1	1	11
2	01	011
3	001	0011
4	101	1011
5	0001	00011

Table 3: Zeckendorf representations and Fibonacci code strings for integers 1–5

Denoting the end of a binary string like this is a useful property in computing. This is an example of synchronised code, as defined in [27]. Synchronised code is a more robust method for storing information, since this allows missing bits in data to only affect the byte they are contained within. With most other methods for encoding, the loss of a single bit will corrupt all data following that bit. This Fibonacci coding method therefore allows for more reliable transmission of data and also facilitates more opportunities for data recovery.

Project Proposal

This project is to develop Python code that can produce two-dimensional quasiperiodic tilings. First, the possibility of converting code from Majorie Senechal's book "Quasicrystals and geometry" written in "C" that fulfills a similar purpose will be investigated. This code would then be developed further. To work towards producing quasiperiodic patterns, the first programming objective would be the simplest case, to develop Python code that produces one-dimensional tilings. The next objective is to produce two-dimensional periodic tilings, such as square or hexagonal patterns. Following this, code will finally be developed that produces two-dimensional quasiperiodic patterns.

Provided this can be achieved in the allotted time, further progress could be made in developing quasiperiodic patterns in three dimensions. The Python code could then be modified, or its file outputs converted in existing programs to produce files that can be understood by a 3D printer.

Week 1	Project proposal discussion, Risk assessment, Ethics form
Week 2	Literature search on quasicrystals and metamaterials
Week 3	Set up Python to run from home, read and understand Senechal's C code, look into possibility of C to Python translation
Week 4	Write code for producing 1D quasicrystal pattern (Fibonacci sequence), and code for producing square and hexagonal patterns
Week 5	Write code for producing 2D quasicrystal patterns
Week 6	Write code for producing 3D quasicrystal patterns
Week 7	Learn what file formats 3D printers can use, look into possibility of converting existing patterns into 3D printer format or adapt code to directly produce files in 3D printer format, draft of project presentation
Week 8	Submit video presentation
Week 9–11	Write and submit report

Fibonacci.py

```
from numpy import sqrt, isclose
import matplotlib.pyplot as plt

tau = (1 + sqrt(5))/2   # Golden ratio
W = 1
edgeList = [0, W]
long = W
short = long/tau

N = 4  # Change generation here
# Handles how many times the original will be dissected L -> LS
for x in range(0,N):
    pending=[]
    for i in range(0,len(edgeList)-1):
    # For all values of edgeList except final:
        if isclose(edgeList[i+1]-edgeList[i], long):
        # If the difference between one value and the next is
        # approx. equal to the current value of 'long':
            pending.append(edgeList[i]+short)
            # Adds new value inbetween according to L -> LS rule

    edgeList.extend(pending)
    edgeList.sort()
    # Sorts values of edgeList into size order (since above for loop returns
    # disorded edgeList)
    long = long/tau
    # Reduce the 'long' variable by golden ratio for the next generation
    short = short/tau
    # Reduce the 'short' variable by golden ratio for the next generation

# 1D
plt.figure(dpi=1200,figsize=(5,1))
plt.axis('off')
plt.axis('equal')

for i in range(0,len(edgeList)-1):
    if isclose(edgeList[i+1]-edgeList[i], long):
    # If the difference between one value and the next is
    # approx. equal to the current value of 'long':
        plt.plot( [edgeList[i], edgeList[i+1]] , [0,0], 'b', linewidth=2 )
        # Plots blue line between points
    if isclose(edgeList[i+1]-edgeList[i], short):
    # If the difference between one value and the next is
    # approx. equal to the current value of 'short':
```

```
        plt.plot( [edgeList[i], edgeList[i+1]] , [0,0], 'r', linewidth=2 )
        # Plots red line between points
    plt.plot( edgeList[i], 0 , 'ko', markersize=3 )
    plt.plot( edgeList[i+1], 0 , 'ko', markersize=3 )

# 2D
plt.figure(dpi=1200,figsize=(5,5))
plt.axis('off')
plt.axis('equal')

for i in range(0,len(edgeList)):
    plt.plot( [ edgeList[i], edgeList[i] ],
              [ edgeList[0], edgeList[-1] ], 'k', linewidth=1 )
    plt.plot( [ edgeList[0], edgeList[-1] ],
              [ edgeList[i], edgeList[i] ], 'k', linewidth=1 )

# 3D
fig3d = plt.figure(dpi=600)
ax = fig3d.add_subplot(111, projection='3d')
ax.set_axis_off()

for i in range( 0, len(edgeList) ):
    for j in range( 0, len(edgeList) ):
        ax.plot( [ edgeList[i], edgeList[i] ],
                 [ edgeList[j], edgeList[j] ],
                 [ edgeList[0], edgeList[-1] ], 'r', linewidth=1 )
        ax.plot( [ edgeList[j], edgeList[j] ],
                 [ edgeList[i], edgeList[i] ],
                 [ edgeList[0], edgeList[-1] ], 'r', linewidth=1 )
        ax.plot( [ edgeList[i], edgeList[i] ],
                 [ edgeList[0], edgeList[-1] ],
                 [ edgeList[j], edgeList[j] ], 'k', linewidth=1 )
        ax.plot( [ edgeList[j], edgeList[j] ],
                 [ edgeList[0], edgeList[-1] ],
                 [ edgeList[i], edgeList[i] ], 'k', linewidth=1 )
        ax.plot( [ edgeList[0], edgeList[-1] ],
                 [ edgeList[i], edgeList[i] ],
                 [ edgeList[j], edgeList[j] ], 'b', linewidth=1 )
        ax.plot( [ edgeList[0], edgeList[-1] ],
                 [ edgeList[j], edgeList[j] ],
                 [ edgeList[i], edgeList[i] ], 'b', linewidth=1 )
```

Deflation.py

```
from numpy import sqrt, sin, cos, array, radians
import matplotlib.pyplot as plt

def rotatePoint(x, y, angle):
# Rotates a point (x,y) anticlockwise by angle (in degrees) about (0,0).
# This is used for constructing the initial single tiles
    rotx = x * cos(radians(angle)) - y * sin(radians(angle))
    roty = x * sin(radians(angle)) + y * cos(radians(angle))
    return rotx, roty
def rotateTile(tile, angle):
# Rotates each vertex of any 4 sided tile anticlockwise
# by given angle in degrees about (0,0).
# This is used to generate the initial star which will be deflated.
    return [ tile[0],
             rotatePoint( tile[1][0], tile[1][1], angle ),
             rotatePoint( tile[2][0], tile[2][1], angle ),
             rotatePoint( tile[3][0], tile[3][1], angle ),
             rotatePoint( tile[4][0], tile[4][1], angle ) ]

tau = (1 + sqrt(5)) / 2
# Golden ratio

# Tiles are defined as a list with the 0th element identifying the tile type as
# a string ('kite' or 'dart'), followed by 4 coordinates stored as nested lists
# I.e. in the format: ( 'kite', (x1,y1), (x2,y2), (x3,y3), (x4,y4) )
kite = [ ( 'kite', (0,0), rotatePoint(0,1, -36), (0,1), rotatePoint(0,1, 36) ) ]
dart = [ ( 'dart', (0,0), rotatePoint(0,1, -36), (0,1/tau), rotatePoint(0,1, 36) ) ]
sun = [   # 5 kites at angles of 72 degrees from each other, forming a decagon
          kite[0],
          rotateTile( kite[0], 72 ),
          rotateTile( kite[0], 144 ),
          rotateTile( kite[0], 216 ),
          rotateTile( kite[0], 288 ),
          ]
starP2 = [   # 5 darts at angles of 72 degrees from each other, forming a 5 pointed star
          dart[0],
          rotateTile( dart[0], 72 ),
          rotateTile( dart[0], 144 ),
          rotateTile( dart[0], 216 ),
          rotateTile( dart[0], 288 ),
          ]


fat = [( 'fat', (0,0), (1,0), array(rotatePoint(1,0, 72)) + (1,0),
```

```
         rotatePoint(1,0, 72) )]
thin = [( 'thin', (0,0), (1,0), array(rotatePoint(1, 0, 36)) + (1,0),
          rotatePoint(1, 0, 36) )]
starP3 = [   # 5 fat rhombs at angles of 72 degrees from each other, forming a star
          fat[0],
          rotateTile( fat[0], 72 ),
          rotateTile( fat[0], 144 ),
          rotateTile( fat[0], 216 ),
          rotateTile( fat[0], 288 ),
          ]

square = [( 'square', (0,0), (1,0), (1,1), (0,1) )]

triangle = [('triangle', (0,0), (1,0), rotatePoint(1, 0, 60) )]

sierpinskiTriangle = [('sierpinskiTriangle', (0,0), (1,0), rotatePoint(1, 0, 60) )]

hexagon = [('hexagon', (0,0), (1,0), array(rotatePoint(1, 0, 60)) + (1,0),
           (1,sqrt(3)), (0,sqrt(3)), rotatePoint(1, 0, 120) )]

chair = [('chair', (2,2), (2,4), (0,4), (0,0), (4,0), (4,2)  )]

rhombA5 = [('rhombA5', (0,0), (1,0), array(rotatePoint(1,0,45))+(1,0),
           rotatePoint(1,0,45) )]
starA5 = [
            rhombA5[0],
            rotateTile( rhombA5[0], 45 ),
            rotateTile( rhombA5[0], 90 ),
            rotateTile( rhombA5[0], 135 ),
            rotateTile( rhombA5[0], 180 ),
            rotateTile( rhombA5[0], 225 ),
            rotateTile( rhombA5[0], 270 ),
            rotateTile( rhombA5[0], 315 ) ]

pent1 = [('pent1', (0,0), (1,0), array(rotatePoint(1,0, 72))+(1,0),
         (0.5, sqrt(5+2*sqrt(5))/2), rotatePoint(1,0, 108))]


def plotFill(tiles):   # Fills in each tile by a colour determined by its type
    for tile in tiles:
        if tile[0]=='kite':     # Fills kites in red
            plt.fill( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0] ],
                      [ tile[1][1], tile[2][1], tile[3][1], tile[4][1] ],
                        'r', alpha=0.5 )
        elif tile[0]=='dart':     # Fills darts in blue
            plt.fill( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0] ],
```

```
                [ tile[1][1], tile[2][1], tile[3][1], tile[4][1] ],
                  'b', alpha=0.5 )
    elif tile[0]=='fat':     # Fills fat rhombs in green
        plt.fill( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0] ],
                  [ tile[1][1], tile[2][1], tile[3][1], tile[4][1] ],
                  'g', alpha=0.5 )
    elif tile[0]=='thin':     # Fills thin rhombs in blue
        plt.fill( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0] ],
                  [ tile[1][1], tile[2][1], tile[3][1], tile[4][1] ],
                  'b', alpha=0.5 )
    elif tile[0]=='triangle' or tile[0]=='sierpinskiTriangle':
    # Fills triangle in purple
        plt.fill( [ tile[1][0], tile[2][0], tile[3][0] ],
                  [ tile[1][1], tile[2][1], tile[3][1] ],
                  '#B2A8FF',  alpha=1 )
    elif tile[0]=='square':     # Fills squares in blue colour
        plt.fill( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0] ],
                  [ tile[1][1], tile[2][1], tile[3][1], tile[4][1] ],
                  '#96DBDB',  alpha=1 )
    elif tile[0]=='hexagon':     # Fills hexagons in orange colour
        plt.fill( [ tile[1][0], tile[2][0], tile[3][0],
                  tile[4][0], tile[5][0], tile[6][0] ],
                  [ tile[1][1], tile[2][1], tile[3][1],
                  tile[4][1], tile[5][1], tile[6][1] ],
                  '#FFA500',  alpha=0.5 )
    elif tile[0]=='chair':      # Fills chairs in grey
        plt.fill( [ tile[1][0], tile[2][0], tile[3][0],
                  tile[4][0], tile[5][0], tile[6][0] ],
                  [ tile[1][1], tile[2][1], tile[3][1],
                  tile[4][1], tile[5][1], tile[6][1] ],
                  '#999999', alpha=.5, )
    elif tile[0]=='squareA5':   # Fills squares used for A5 tiling in orange
        plt.fill( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0] ],
                  [ tile[1][1], tile[2][1], tile[3][1], tile[4][1] ],
                  '#FFA500',  alpha=0.5 )
    elif tile[0]=='rhombA5':    # Fills rhombs used for A5 tiling in pink
        plt.fill( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0] ],
                  [ tile[1][1], tile[2][1], tile[3][1], tile[4][1] ],
                  '#ef1de7',  alpha=0.5 )
    elif tile[0]=='pent1':   # Fills type 1 pentagons from the P1 tiling in pink
        plt.fill( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0], tile[5][0] ],
                  [ tile[1][1], tile[2][1], tile[3][1], tile[4][1], tile[5][1] ],
                  '#ef1de7',  alpha=0.5 )
    elif tile[0]=='pent2':   # Fills type 2 pentagons from the P1 tiling in red
        plt.fill( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0], tile[5][0] ],
                  [ tile[1][1], tile[2][1], tile[3][1], tile[4][1], tile[5][1] ],
```

```
                    'r',  alpha=0.6 )
            elif tile[0]=='pent3':   # Fills type 3 pentagons from the P1 tiling in orange
                plt.fill( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0], tile[5][0] ],
                          [ tile[1][1], tile[2][1], tile[3][1], tile[4][1], tile[5][1] ],
                            '#ffa500',  alpha=0.7 )
            elif tile[0]=='diamond':    # Fills diamonds from the P1 tiling in blue
                plt.fill( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0] ],
                          [ tile[1][1], tile[2][1], tile[3][1], tile[4][1] ],
                            'b',  alpha=0.5 )
            elif tile[0]=='boat':       # Fills boats from the P1 tiling in green
                plt.fill( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0],
                            tile[5][0], tile[6][0], tile[7][0] ],
                          [ tile[1][1], tile[2][1], tile[3][1], tile[4][1],
                            tile[5][1], tile[6][1], tile[7][1] ],
                            'c',  alpha=0.5 )
            elif tile[0]=='star':       # Fills stars from the P1 tiling in cyan
                plt.fill( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0],
                            tile[5][0], tile[6][0], tile[7][0], tile[8][0],
                            tile[9][0], tile[10][0] ],
                          [ tile[1][1], tile[2][1], tile[3][1], tile[4][1],
                            tile[5][1], tile[6][1], tile[7][1], tile[8][1],
                            tile[9][1], tile[10][1] ],
                            'g',  alpha=0.5 )


    def plotVertices(tiles):   # Plots only the vertices of each tile, for use in FFT
        for tile in tiles:
            if tile[0]=='triangle' or tile[0]=='sierpinskiTriangle':
                plt.plot( [ tile[1][0], tile[2][0], tile[3][0] ],
                          [ tile[1][1], tile[2][1], tile[3][1] ],
                          'ko', markersize=3 )
            elif tile[0]=='kite' or tile[0]=='dart' or tile[0]=='fat'
                 or tile[0]=='thin' or tile[0]=='square' or tile[0]=='squareA5'
                 or tile[0]=='rhombA5' or tile[0]=='diamond':
                 plt.plot( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0] ],
                           [ tile[1][1], tile[2][1], tile[3][1], tile[4][1]],
                             'ko', markersize=3 )
            elif tile[0]=='pent1' or tile[0]=='pent2' or tile[0]=='pent3':
                plt.plot( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0],
                            tile[5][0] ],
                          [ tile[1][1], tile[2][1], tile[3][1], tile[4][1],
                            tile[5][1] ],
                            'ko', markersize=3 )
            elif tile[0]=='boat':
                plt.plot( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0],
                            tile[5][0], tile[6][0], tile[7][0] ],
```

```
                    [ tile[1][1], tile[2][1], tile[3][1], tile[4][1],
                      tile[5][1], tile[6][1], tile[7][1] ],
                      'ko', markersize=3 )
        elif tile[0]=='star':
            plt.plot( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0],
                      tile[5][0], tile[6][0], tile[7][0], tile[8][0],
                      tile[9][0], tile[10][0] ],
                    [ tile[1][1], tile[2][1], tile[3][1], tile[4][1],
                      tile[5][1], tile[6][1], tile[7][1], tile[8][1],
                      tile[9][1], tile[10][1] ],
                      'ko', markersize=3 )
        elif tile[0]=='hexagon' or tile[0]=='chair':
            plt.plot( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0],
                      tile[5][0], tile[6][0] ],
                    [ tile[1][1], tile[2][1], tile[3][1], tile[4][1],
                      tile[5][1], tile[6][1] ],
                      'ko', markersize=3 )


def plotOutline(tiles):     # Marks a black line around the outline of each tile
    for tile in tiles:
        if tile[0]=='triangle' or tile[0]=='sierpinskiTriangle':
            plt.plot( [ tile[1][0], tile[2][0], tile[3][0], tile[1][0] ],
                    [ tile[1][1], tile[2][1], tile[3][1], tile[1][1] ],
                    'k', linewidth=1 )
        elif tile[0]=='kite' or tile[0]=='dart' or tile[0]=='fat'
             or tile[0]=='thin' or tile[0]=='square' or tile[0]=='squareA5'
             or tile[0]=='rhombA5' or tile[0]=='diamond':
             plt.plot( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0], tile[1][0] ],
                     [ tile[1][1], tile[2][1], tile[3][1], tile[4][1], tile[1][1] ],
                     'k', linewidth=1 )
        elif tile[0]=='pent1' or tile[0]=='pent2' or tile[0]=='pent3':
            plt.plot( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0],
                      tile[5][0], tile[1][0] ],
                    [ tile[1][1], tile[2][1], tile[3][1], tile[4][1],
                      tile[5][1], tile[1][1] ],
                      'k', linewidth=1 )
        elif tile[0]=='boat':
            plt.plot( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0],
                      tile[5][0], tile[6][0], tile[7][0], tile[1][0] ],
                    [ tile[1][1], tile[2][1], tile[3][1], tile[4][1],
                      tile[5][1], tile[6][1], tile[7][1], tile[1][1] ],
                      'k', linewidth=1 )
        elif tile[0]=='star':
            plt.plot( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0],
                      tile[5][0], tile[6][0], tile[7][0], tile[8][0],
```

```
                        tile[9][0], tile[10][0], tile[1][0] ],
                      [ tile[1][1], tile[2][1], tile[3][1], tile[4][1],
                        tile[5][1], tile[6][1], tile[7][1], tile[8][1],
                        tile[9][1], tile[10][1], tile[1][1] ],
                        'k', linewidth=1 )
        elif tile[0]=='hexagon' or tile[0]=='chair':
            plt.plot( [ tile[1][0], tile[2][0], tile[3][0], tile[4][0],
                        tile[5][0], tile[6][0], tile[1][0] ],
                      [ tile[1][1], tile[2][1], tile[3][1], tile[4][1],
                        tile[5][1], tile[6][1], tile[1][1] ],
                        'k', linewidth=1 )

def deflate_kite(kite):
# Deflation process for kite tiles, returns 4 new smaller tiles (2 kites and 2 darts).
    A = array(kite[1])
    B = array(kite[2])
    C = array(kite[3])
    D = array(kite[4])
    e = (C-A)/tau + A
    f = (A-B)/tau + B
    g = (A-D)/tau + D
    h = A + B - e
    i = A + D - e
    return [  ( 'kite', B, C, e, f ),
              ( 'kite', D, g, e, C ),
              ( 'dart', A, e, g, i ),
              ( 'dart', A, h, f, e )  ]

def deflate_dart(dart):
# Deflation process for dart tiles, returns 3 new smaller tiles (1 kite and 2 darts)
    A = array(dart[1])
    B = array(dart[2])
    C = array(dart[3])
    D = array(dart[4])
    e = (B-A)/tau + A
    f = (D-A)/tau + A
    g = A + B - C
    h = A + D - C
    return [  ( 'kite', A, e, C, f ),
              ( 'dart', B, C, e, g ),
              ( 'dart', D, h, f, C )  ]

def deflate_fat(fat):
# Deflation process for fat rhombs, returns 5 new smaller rhombs (2 thin and 3 fat).
    A = array(fat[1])
    B = array(fat[2])
```

```
    C = array(fat[3])
    D = array(fat[4])
    e = (B-A)/tau + A
    f = (C-A)/tau + A
    g = (D-A)/tau + A
    h = B + e - f
    i = B + C - f
    j = C + D - f
    k = D + g - f
    return [  ( 'fat', f, g, A, e ),
              ( 'thin', h, B, f, e ),
              ( 'fat', C, f, B, i ),
              ( 'fat', C, j, D, f ),
              ( 'thin', f, D, k, g )         ]

def deflate_thin(thin):
# Deflation process for thin rhombs, returns 4 new smaller rhombs (2 thin and 2 fat).
    A = array(thin[1])
    B = array(thin[2])
    C = array(thin[3])
    D = array(thin[4])
    e = (D-A)/tau + A
    f = (D-C)/tau + C
    g = D + e - B
    h = A + B - e
    i = D + f - B
    j = B + C - f
    return [  ( 'fat', B, e, A, h ),
              ( 'thin', B, D, g, e ),
              ( 'thin', i, D, B, f ),
              ( 'fat', B, j, C, f )  ]

def deflate_square(square):
# Deflation process for squares, returns 4 new smaller squares
    A = array(square[1])
    B = array(square[2])
    C = array(square[3])
    D = array(square[4])
    e = (A+B)/2
    f = (B+C)/2
    g = (C+D)/2
    h = (D+A)/2
    i = (A+C)/2
    return [  ( 'square', A, e, i, h ),
              ( 'square', e, B, f, i ),
              ( 'square', i, f, C, g ),
```

```
            ( 'square', h, i, g, D )     ]

def deflate_triangle(triangle):
# Deflation process for triangles, returns 4 new smaller squares
    A = array(triangle[1])
    B = array(triangle[2])
    C = array(triangle[3])
    e = (A+B)/2
    f = (B+C)/2
    g = (C+A)/2
    return [  ( 'triangle', A, e, g ),
              ( 'triangle', e, B, f ),
              ( 'triangle', g, f, C ),
              ( 'triangle', e, f, g )     ]

def deflate_sierpinskiTriangle(sierpinskiTriangle):
# Deflation process for squares, returns 4 new smaller squares
    A = array(sierpinskiTriangle[1])
    B = array(sierpinskiTriangle[2])
    C = array(sierpinskiTriangle[3])
    e = (A+B)/2
    f = (B+C)/2
    g = (C+A)/2
    return [  ( 'sierpinskiTriangle', A, e, g ),
              ( 'sierpinskiTriangle', e, B, f ),
              ( 'sierpinskiTriangle', g, f, C )  ]


def deflate_hexagon(hexagon):
# Deflation process for squares, returns 4 new smaller squares
    A = array(hexagon[1])
    B = array(hexagon[2])
    C = array(hexagon[3])
    D = array(hexagon[4])
    E = array(hexagon[5])
    F = array(hexagon[6])
    cen = (A+D)/2
    g = 2*A - cen
    h = 2*B - cen
    i = 2*C - cen
    j = 2*D - cen
    k = 2*E - cen
    l = 2*F - cen
    return [  ( 'hexagon', A, B, C, D, E, F ),
              ( 'hexagon', 2*A-E, 2*B-D, h, B, A, g ),
              ( 'hexagon', h, 2*B-F, 2*C-E, i, C, B ),
```

```
            ( 'hexagon', C, i, 2*C-A, 2*D-F, j, D ),
            ( 'hexagon', E, D, j, 2*D-B, 2*E-A, k ),
            ( 'hexagon', l, F, E, k, 2*E-C, 2*F-B ),
            ( 'hexagon', 2*A-C, g, A, F, l, 2*F-D )   ]


def deflate_chair(chair):
# Deflates given single chair into 4 smaller chairs
    A = array(chair[1])
    B = array(chair[2])
    C = array(chair[3])
    D = array(chair[4])
    E = array(chair[5])
    F = array(chair[6])
    r = (A + B)/2
    s = (C + D)/2
    t = (D + E)/2
    u = (A + F)/2
    v = (3*C + D + 2*A + 2*B)/8
    w = (2*A + 4*D + C + E)/8
    x = (2*A + 2*F + D + 3*E)/8
    y = (2*A + C + D)/4
    z = (2*A + D + E)/4
    print(    ( 'chair', A, r, v, w, x, u, ),
              ( 'chair', v, r, B, C, s, y ),
              ( 'chair', w, y, s, D, t, z ),
              ( 'chair', x, z, t, E, F, u )  )
    return [  ( 'chair', A, r, v, w, x, u, ),
              ( 'chair', v, r, B, C, s, y ),
              ( 'chair', w, y, s, D, t, z ),
              ( 'chair', x, z, t, E, F, u )  ]


silvRatio = 1 + sqrt(2)
# Silver ratio, used in A5 tiling
def deflate_rhombA5(rhombA5):
    A = array(rhombA5[1])
    B = array(rhombA5[2])
    C = array(rhombA5[3])
    D = array(rhombA5[4])
    e = (C-A)/silvRatio + A
    f = (A-C)/silvRatio + C
    g = (B-A)/silvRatio + A
    h = (B-C)/silvRatio + C
    i = (D-C)/silvRatio + C
    j = (D-A)/silvRatio + A
```

```
    k = B + g - e
    l = B + h - f
    m = D + i - f
    n = D + j - e
    return [  ( 'rhombA5', A, g, e, j ),
              ( 'rhombA5', D, e, B, f ),
              ( 'rhombA5', C, i, f, h ),
              ( 'squareA5', B, e, g, k ),
              ( 'squareA5', D, n, j, e ),
              ( 'squareA5', D, f, i, m ),
              ( 'squareA5', B, l, h, f )   ]

def deflate_squareA5(squareA5):
    A = array(squareA5[1])
    B = array(squareA5[2])
    C = array(squareA5[3])
    D = array(squareA5[4])
    e = (C-A)/(silvRatio+1) + A
    f = (D-B)/(silvRatio+1) + B
    g = (A-C)/(silvRatio+1) + C
    h = (B-D)/(silvRatio+1) + D
    i = (B-A)/silvRatio + A
    j = (C-B)/silvRatio + B
    k = (C-D)/silvRatio + D
    l = (D-A)/silvRatio + A
    m = B + i - f
    n = C + j - g
    o = C + k - g
    p = D + l - h
    return [  ( 'squareA5', g, h, e, f ),
              ( 'squareA5', B, m, i, f ),
              ( 'squareA5', C, n, j, g ),
              ( 'squareA5', C, o, k, g ) ,
              ( 'squareA5', D, p, l, h ),
              ( 'rhombA5', A, e, h, l ),
              ( 'rhombA5', A, i, f, e ),
              ( 'rhombA5', B, j, g, f ),
              ( 'rhombA5', D, h, g, k )  ]

def deflate_pent1(pent1):
    A = array(pent1[1])
    B = array(pent1[2])
    C = array(pent1[3])
    D = array(pent1[4])
    E = array(pent1[5])
    f = (C-A)/tau + A
```

```
    g = (D-B)/tau + B
    h = (E-C)/tau + C
    i = (A-D)/tau + D
    j = (B-E)/tau + E
    k = (A-B)/tau + B
    l = (B-A)/tau + A
    m = (B-C)/tau + C
    n = (C-B)/tau + B
    o = (C-D)/tau + D
    p = (D-C)/tau + C
    q = (D-E)/tau + E
    r = (E-D)/tau + D
    s = (E-A)/tau + A
    t = (A-E)/tau + E
    return [  ( 'pent1', f, g, h, i, j ),
              ( 'pent2', f, j, l, B, m ),
              ( 'pent2', g, f, n, C, o ),
              ( 'pent2', h, g, p, D, q ),
              ( 'pent2', i, h, r, E, s ),
              ( 'pent2', j, i, t, A, k )  ]


def deflate_pent2(pent2):
    A = array(pent2[1])
    B = array(pent2[2])
    C = array(pent2[3])
    D = array(pent2[4])
    E = array(pent2[5])
    f = (C-A)/tau + A
    g = (D-B)/tau + B
    h = (E-C)/tau + C
    i = (A-D)/tau + D
    j = (B-E)/tau + E
    k = (A-B)/tau + B
    l = (B-A)/tau + A
    m = (B-C)/tau + C
    n = (C-B)/tau + B
    o = (C-D)/tau + D
    p = (D-C)/tau + C
    q = (D-E)/tau + E
    r = (E-D)/tau + D
    s = (E-A)/tau + A
    t = (A-E)/tau + E
    u = A + B - j
    v = o + p - g
    w = q + r - h
```

```
    return [  ( 'pent1', f, g, h, i, j ),
              ( 'pent3', j, l, B, m, f ),
              ( 'pent2', g, f, n, C, o ),
              ( 'pent2', h, g, p, D, q ),
              ( 'pent2', i, h, r, E, s ),
              ( 'pent3', k, j, i, t, A ),
              ( 'diamond', j, k, u, l ),
              ( 'diamond', g, o, v, p ),
              ( 'diamond', h, q, w, r )  ]


def deflate_pent3(pent3):
    A = array(pent3[1])
    B = array(pent3[2])
    C = array(pent3[3])
    D = array(pent3[4])
    E = array(pent3[5])
    f = (C-A)/tau + A
    g = (D-B)/tau + B
    h = (E-C)/tau + C
    i = (A-D)/tau + D
    j = (B-E)/tau + E
    k = (A-B)/tau + B
    l = (B-A)/tau + A
    m = (B-C)/tau + C
    n = (C-B)/tau + B
    o = (C-D)/tau + D
    p = (D-C)/tau + C
    q = (D-E)/tau + E
    r = (E-D)/tau + D
    s = (E-A)/tau + A
    t = (A-E)/tau + E
    u = B + C - f
    v = A + E - i
    return [  ( 'pent1', g, h, i, j, f ),
              ( 'pent3', m, f, j, l, B ),
              ( 'pent3', f, n, C, o, g ),
              ( 'pent2', h, g, p, D, q ),
              ( 'pent3', s, i, h, r, E ),
              ( 'pent3', i, t, A, k, j ),
              ( 'diamond', f, m, u, n ),
              ( 'diamond', i, s, v, t )  ]


def deflate_diamond(diamond):
    A = array(diamond[1])
```

```
    B = array(diamond[2])
    C = array(diamond[3])
    D = array(diamond[4])
    e = (A-C)/tau + C

    f = (A-B)/tau + B
    g = (B-A)/tau + A
    h = (B-C)/tau + C
    i = (C-B)/tau + B
    j = (C-D)/tau + D
    k = (D-C)/tau + C
    l = (D-A)/tau + A
    m = (A-D)/tau + D

    n = f + g - l
    o = h + i - k
    p = j + k - h
    q = l + m - g

    return [ ( 'pent3', h, k, D, e, B),
             ( 'boat', k, h, o, i, C, j, p ),
             ( 'star', B, e, D, l, q, m, A, f, n, g )  ]

def deflate_boat(boat):
    A = array(boat[1])
    B = array(boat[2])
    C = array(boat[3])
    D = array(boat[4])
    E = array(boat[5])
    F = array(boat[6])
    G = array(boat[7])

    h = (A-B)/tau + B
    i = (B-A)/tau + A
    j = (B-C)/tau + C
    k = (C-B)/tau + B
    l = (C-D)/tau + D
    m = (D-C)/tau + C
    n = (D-E)/tau + E
    o = (E-D)/tau + D
    p = (E-F)/tau + F
    q = (F-E)/tau + E
    r = (F-G)/tau + G
    s = (G-F)/tau + F
    t = (G-A)/tau + A
    u = (A-G)/tau + G
```

```
    v = (A-D)/tau + D
    w = (B-F)/tau + F
    x = (D-A)/tau + A

    y = h + i - x
    z = j + k - m
    z1 = l + m - j
    z2 = n + o - q
    z3 = p + q - n
    z4 = r + s - u
    z5 = t + u - r

    return [ ( 'pent3', j, m, D, w, B ),
             ( 'pent3', n, q, F, x, D ),
             ( 'pent3', r, u, A, v, F ),
             ( 'boat', m, j, z, k, C, l, z1 ),
             ( 'boat', q, n, z2, o, E, p, z3 ),
             ( 'boat', u, r, z4, s, G, t, z5 ),
             ( 'star', D, x, F, v, A, h, y, i, B, w  )  ]

def deflate_star(star):
    A = array(star[1])
    B = array(star[2])
    C = array(star[3])
    D = array(star[4])
    E = array(star[5])
    F = array(star[6])
    G = array(star[7])
    H = array(star[8])
    I = array(star[9])
    J = array(star[10])

    k = (A-B)/tau + B
    l = (B-A)/tau + A
    m = (B-C)/tau + C
    n = (C-B)/tau + B
    o = (C-D)/tau + D
    p = (D-C)/tau + C
    q = (D-E)/tau + E
    r = (E-D)/tau + D
    s = (E-F)/tau + F
    t = (F-E)/tau + E
    u = (F-G)/tau + G
    v = (G-F)/tau + F
    w = (G-H)/tau + H
```

```
    x = (H-G)/tau + G
    y = (H-I)/tau + I
    z = (I-H)/tau + H
    z1 = (I-J)/tau + J
    z2 = (J-I)/tau + I
    z3 = (J-A)/tau + A
    z4 = (A-J)/tau + J

    z5 = (B-H)/tau + H
    z6 = (B-F)/tau + F
    z7 = (F-B)/tau + B
    z8 = (F-J)/tau + J
    z9 = (H-B)/tau + B

    z10 = k + l - z3
    z11 = m + n - p
    z12 = o + p - m
    z13 = q + r - t
    z14 = s + t - q
    z15 = u + v - x
    z16 = w + x - u
    z17 = y + z - z2
    z18 = z1 + z2 - y
    z19 = z3 + z4 - l

    return [ ( 'pent3', z3, l, B, z5, J ),
             ( 'pent3', m, p, D, z6, B ),
             ( 'pent3', q, t, F, z7, D ),
             ( 'pent3', u, x, H, z8, F ),
             ( 'pent3', y, z2, J, z9, H ),

             ( 'boat', l, z3, z19, z4, A, k, z10 ),
             ( 'boat', p, m, z11, n, C, o, z12 ),
             ( 'boat', t, q, z13, r, E, s, z14 ),
             ( 'boat', x, u, z15, v, G, w, z16 ),
             ( 'boat', z2, y, z17, z, I, z1, z18 ),

             ( 'star', F, z8, H, z9, J, z5, B, z6, D, z7 )  ]


def getCentre(tile):
# Returns approximate (9 d.p.) centre of given tile,
# used later for checking for duplicates
    cenx = round((tile[1][0]+tile[3][0])/2,9)
    ceny = round((tile[1][1]+tile[3][1])/2,9)
```

```
    return [cenx,ceny]

def deflateGeneral(tiles, n):
# Takes list of tiles and performs deflation method on each tile n times
    if n > 0:
        for i in range(n):

            nextGenTiles = []
            # Clears information on previous generation of tiles each iteration
            # (to avoid mixing of generations)
            for tile in tiles:
                # Chooses between the different deflation processes by checking
                # first element, returns the next generation of tiles
                function = eval('deflate_'+tile[0])
                nextGenTiles = nextGenTiles + function(tile)

            uniqueCentres, uniqueTiles = [], []
            # Clears information on previous generation of tiles each iteration
            # (to avoid mixing of generations)
            for tile in nextGenTiles:
            # Filters this new list of deflated tiles to remove duplicates
                centre = getCentre(tile)
                # Calls the getCentre function on given tile
                if centre not in uniqueCentres:
                # Only adds tile to tile list if it does not match centres with
                # any already in centre list
                    uniqueTiles.append(tile)
                    # List of unique tiles
                    uniqueCentres.append(centre)
                    # Corresponding list of unique centres

            tiles = uniqueTiles

    return tiles


tilingType = input("Choose tiling type: triangle, sierpinski, square, hexagon,
                    chair, P1, P2, P3 or A5: ")

if tilingType=='triangle':
    initialTile = triangle

elif tilingType=='square':
    initialTile = square

elif tilingType=='hexagon':
```

```
    initialTile = hexagon

elif tilingType=='chair':
    initialTile = chair

elif tilingType=='P1':
    initialTile = pent1

elif tilingType=='P2':
    initialTile = eval(input("Choose intitial tile: kite, dart, starP2 or sun: "))

elif tilingType=='P3':
    initialTile = eval(input("Choose initial tile: fat, thin, starP3 or decagon: "))

elif tilingType=='A5':
    initialTile = starA5

elif tilingType=='sierpinski':
    initialTile = sierpinskiTriangle

# How many times the deflation process will be applied to the initial set of tiles
N = int(input("How many deflation iterations: "))

fillType = input("Fill tiles or plot vertices? Choose between: fill or vertices: ")

plt.figure(dpi=1200,figsize=(5,5))
# Constructs the matplotlib figure as a high definition, square plot
plt.axis('off')
# Disables axis plots on image
plt.axis('equal')
# Fixes aspect ratio issue

if fillType=='fill':
    plotFill(deflateGeneral(initialTile, N))
    plotOutline(deflateGeneral(initialTile, N))

elif fillType=='vertices':
    plotVertices(deflateGeneral(initialTile, N))
```

CutAndProject1D.py

```
import matplotlib.pyplot as plt
from math import ceil,floor,sin,cos,atan,sqrt

plt.figure(dpi=1200,figsize=(5,1))
# Constructs the matplotlib figure as a high definition, square plot
plt.axis('off')     # Disables axis plots on image
plt.axis('equal')

tau = (1 + sqrt(5))/2  # Golden ratio

gradient = (1/tau)  # Gradient of shaded region in green
theta = atan(gradient)  # Angle formed between shaded region and x-axis (in radians)

thickness = ( cos(theta) + sin(theta) )*tau
# Distance between upper limit and lower limit of region
verticalHeight = thickness * sqrt(gradient**2 + 1)
# Distance between two parallel lines inverse

regionWidth = 5  # How many x values are considered

def maxAccepted(x):
    return gradient*x + verticalHeight/2
def minAccepted(x):
    return gradient*x - verticalHeight/2

def isWithinRegion(x,y):
    if y > minAccepted(x) and y < maxAccepted(x):
        return True

yMax = ceil(maxAccepted(regionWidth))
yMin = floor(minAccepted(0))

x_accepted, y_accepted = [ ], [ ]

for x in range(regionWidth):
    for y in range(yMin,yMax):
            if isWithinRegion(x,y):
                    x_accepted = x_accepted + [x]
                    y_accepted = y_accepted + [y]


for i in range(len(x_accepted)):
    x_final = (x_accepted[i]+y_accepted[i]*gradient)/(1+gradient**2)
    plt.plot(x_final, 0, 'ko', markersize=4 )
```

CutAndProject2D.py

```
import matplotlib.pyplot as plt
from math import ceil,floor,sin,cos,atan,sqrt

plt.figure(dpi=1200,figsize=(5,1))
# Constructs the matplotlib figure as a high definition, square plot
plt.axis('off')     # Disables axis plots on image
plt.axis('equal')

tau = (1 + sqrt(5))/2  # Golden ratio

gradient = (1/tau)  # gradient of shaded region in green
theta = atan(gradient)  # Angle formed between shaded region and x-axis (in radians)
thickness = ( cos(theta) + sin(theta) )*tau
# Distance between upper limit and lower limit of region
verticalHeight = thickness * sqrt(gradient**2 + 1)
# Distance between two parallel lines inverse
regionWidth = 5  # How many x values are considered

def maxAccepted(x):
    return gradient*x + verticalHeight/2
def minAccepted(x):
    return gradient*x - verticalHeight/2

def isWithinRegion(x,y,z):
    if z > minAccepted(x) and z < maxAccepted(x):
        if z > minAccepted(y) and z < maxAccepted(y):
            return True

zMax = ceil(maxAccepted(regionWidth))
zMin = floor(minAccepted(0))
x_accepted, y_accepted, z_accepted = [ ], [ ], [ ]
for x in range(regionWidth):
    for y in range(regionWidth):
            for z in range(zMin,zMax):
                    if isWithinRegion(x,y,z):
                            x_accepted = x_accepted + [x]
                            y_accepted = y_accepted + [y]
                            z_accepted = z_accepted + [z]

for i in range(len(x_accepted)):
    for j in range(len(y_accepted)):
        x_final = (x_accepted[i]+z_accepted[i]*gradient)/(1+gradient**2)
        y_final = (y_accepted[j]+z_accepted[j]*gradient)/(1+gradient**2)
        plt.plot(x_final, y_final, 'ko', markersize=4 )
```