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 )
```