Please execute the following cell at first

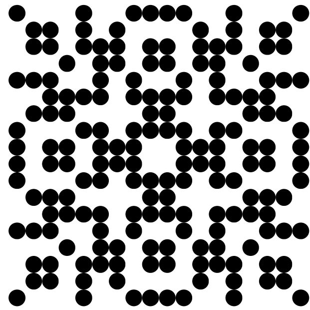
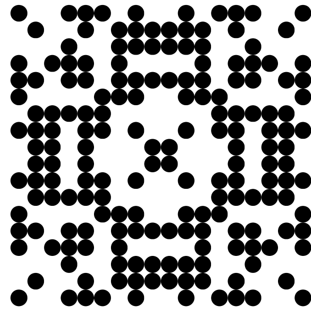
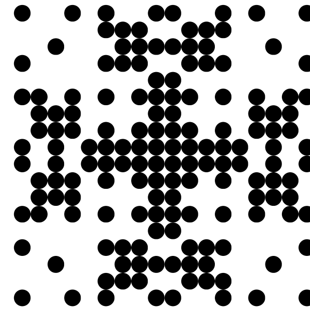
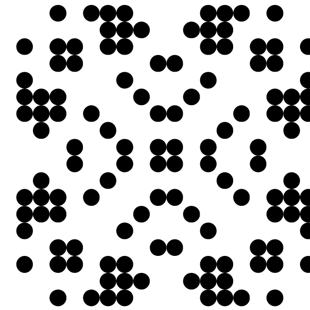
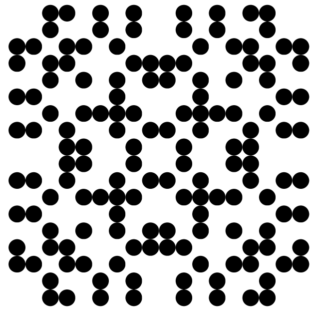
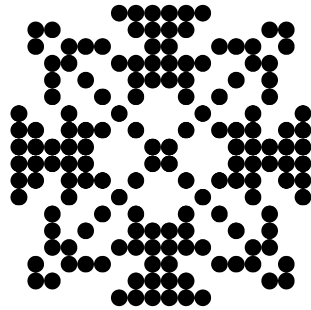
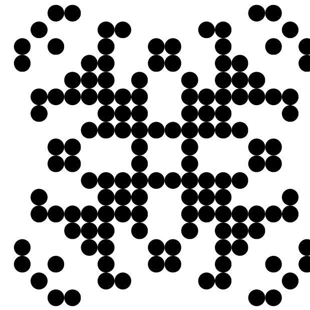
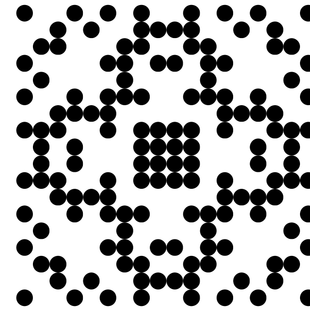
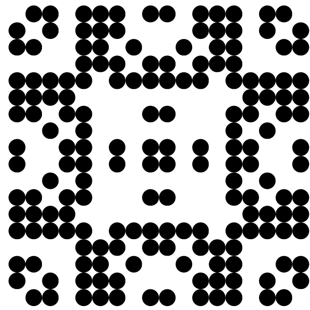
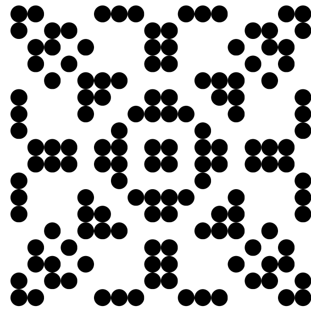
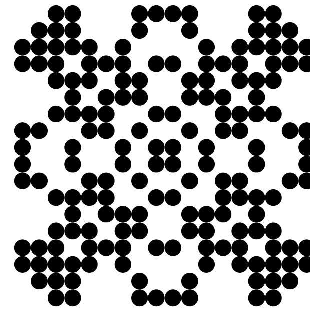
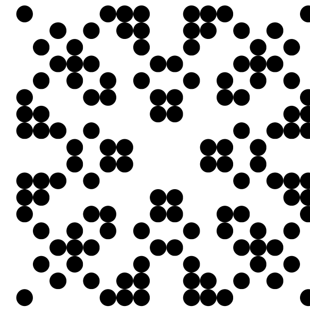
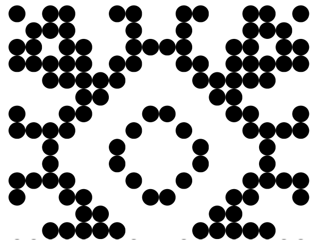
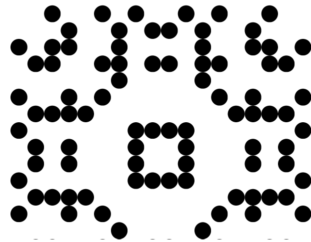
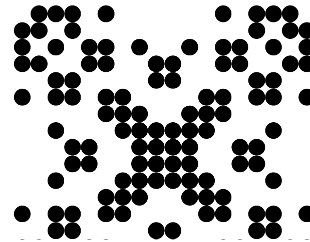
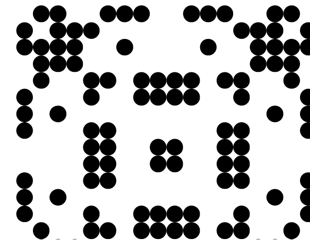
Entrée [3]:
```
from IPython.display import SVG
```

Circles tiles

Entrée [2]:
```
import random, sys
rand_colors = [1,  0]
colors = {0: 'black',
           1: 'white'}

svgdata = ""

def addsvgdata(data):
    global svgdata
    svgdata += data

def draw_circle(x, y, width, height, color='black'):
    r = height / 2
    cx = x + r
    cy = y + r
    return f'<circle cx="{cx}" cy="{cy}" r="{r}" fill="{color}" stroke-width="0"/>'

def create_tile(size):
    mosaic = []
    #only do the first n//2 rows (treating the first n//2 cols first)
    for r in range(size // 2 + 1):
        temp_row = []
        #needed to flip black and white circles
        was_black = 0
        for c in range(size // 2 + 1):
            #if on the horizontal or vertical centre or on the diagonal
            #try not to make it too heavy on the black
            if was_black > 2:
                temp_row.append(1)
                was_black = 0
                continue
            #otherwise
            if r == c or c == size // 2 + 1 or r == size // 2 + 1 or r < c:
                col = random.choice(rand_colors)
                temp_row.append(col)
                if col == 0:
```

```python
                    was_black += 1
                else:
                    was_black = 0
            #bottom of diag
            elif r>c:
                temp_row.append(mosaic[c][r])
        for c in range(size // 2, -1, -1):
            temp_row.append(temp_row[c])
        mosaic.append(temp_row)
    for r in range(size // 2, -1, -1):
        mosaic.append(mosaic[r])
    return mosaic

def make_circle_tiles(size, tiles ,image_size):
    #size is number of sections per tile, tiles is number of tiles, image_size is
    #how big the image is in pixels
    header = f'<svg viewBox="0 0 {image_size} {image_size}" xmlns="http://www.w3.org/2000/svg">\
    addsvgdata(header)
    tile_size = image_size / tiles
    padding = 40
    for x in range(tiles):
        for y in range(tiles):
            top_left_x = x * tile_size + padding / 2
            top_left_y = y * tile_size + padding / 2
            bottom_right_x = top_left_x + tile_size - padding
            bottom_right_y = top_left_y + tile_size - padding
            square_size = (bottom_right_x - top_left_x) / size
            tile = create_tile(size)
            for r in range(len(tile)):
                for c in range(len(tile)):
                    col = tile[r][c]
                    top_left_x_tile = c * square_size + top_left_x
                    top_left_y_tile = r * square_size + top_left_y
                    bottom_right_x_tile = top_left_x_tile + square_size
                    bottom_right_y_tile = top_left_y_tile + square_size

                    line = draw_circle(
                        top_left_x_tile,
                        top_left_y_tile,
                        square_size,
                        square_size,
                        colors[col])

                    addsvgdata(line)
    footer = f'</svg>'
    addsvgdata(footer)

make_circle_tiles(17 , 4 ,500)
```

```
85  display(SVG(data=svgdata))
86  with open("circle tiles.svg","w") as fo:
```

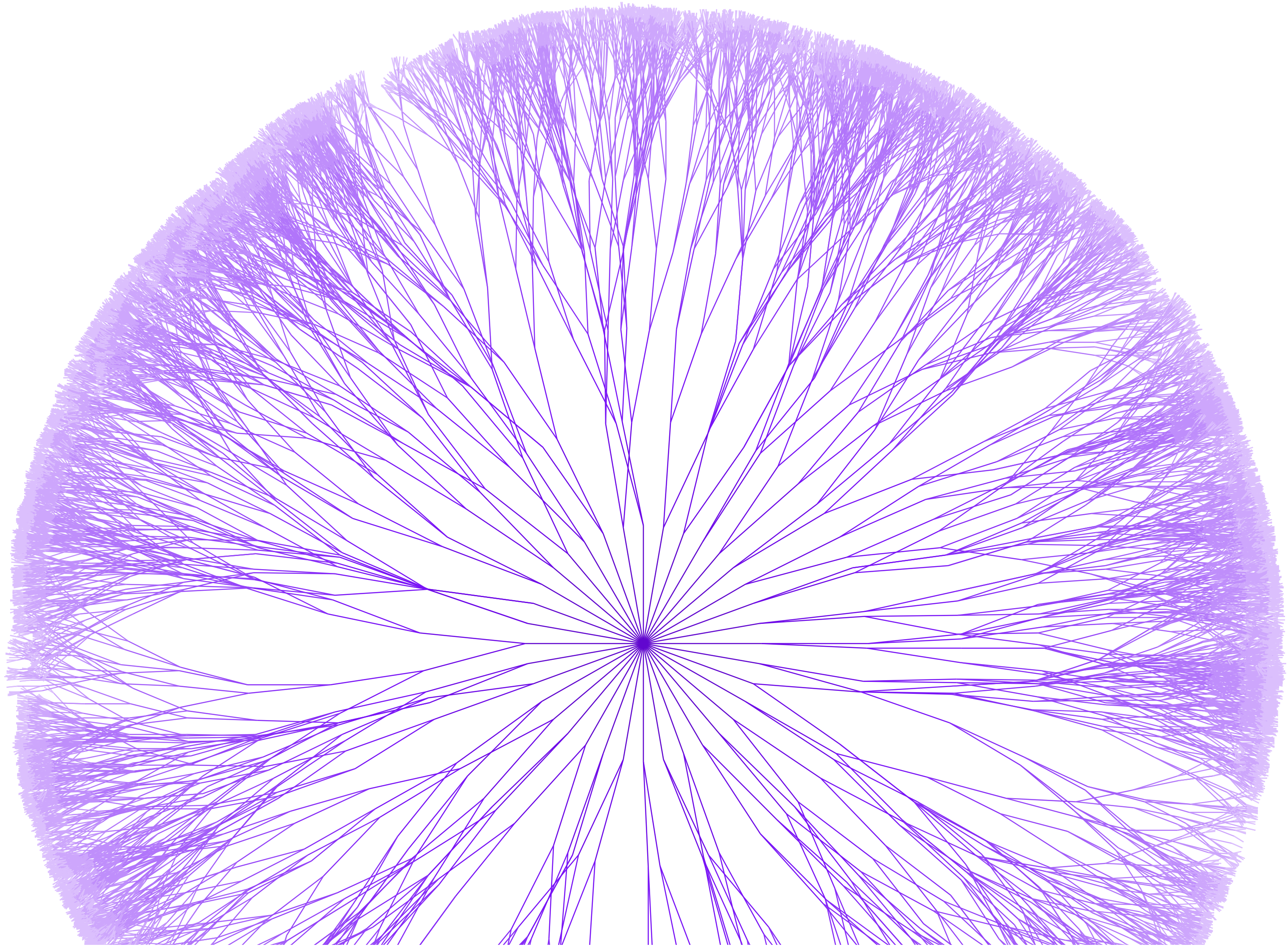

















Tree

Entrée [3]:

```
import math, random

svgdata = ""

def addsvgdata(data):
    global svgdata
    svgdata += data

def draw_line(x, y, x2,  y2, color=(0, 100, 0), sw=1):
    return f'<line x1="{x}" y1="{y}" x2="{x2}" y2="{y2}" \
    style="stroke:hsl({str(color[0])+","+str(color[1])+"%,"+str(color[2])+"%"});\
    stroke-width:{sw}" />'

def draw_tree(x1, y1, angle, depth, col):
    '''
    Makes a recursive randomised tree
    Params:
    x1 - the starting x
    y1 - the starting y
    angle - the direction from the vertial - allows for rotation around an origin
    depth - how many iterations
    col - colour
    '''

    if depth:
        x2 = x1 + int(math.cos(math.radians(angle)) * depth * 10.0)
        y2 = y1 + int(math.sin(math.radians(angle)) * depth * 10.0)
        addsvgdata(draw_line(x1, y1, x2, y2, color=(col)))
        diffA = random.randint(-5, 5) * 3
        diffB = random.randint(-5, 5) * 3
        draw_tree(x2, y2, angle - diffA, depth - 1, (col[0], col[1], col[2] + 5))
        draw_tree(x2, y2, angle + diffB, depth - 1, (col[0], col[1], col[2] + 5))

def make_tree_circle(width, color):
    '''Makes a circle full of randomised recursively drawn trees in a particular colour

    Params:
    width - the diameter fo the resulting circle
    color: the hsl values of the centre colour as a tuple
    '''
    global svgdata
    header = f'<svg viewBox="0 0 {width} {width}" xmlns="http://www.w3.org/2000/svg">\n'
```

```
43     addsvgdata(header)
44
45     for angle in range(0, 360, 10):
46
47         #draw_tree(600, 600, angle, 10, (212,100,30))
48         draw_tree(width // 2, width // 2, angle, 10, color)
49     footer = f'</svg>'
50     addsvgdata(footer)
51
52 make_tree_circle(1200, (267, 93, 42))
53 # Warning: long wait before display
54 display(SVG(data=svgdata))
55 with open("tree.svg","w") as fo:
```



A bit Mondrian

Entrée [4]:

```
import random

svgdata = ""

def addsvgdata(data):
    global svgdata
    svgdata += data

def draw_rect(x, y, width, height, color='black'):
    return f'<rect x="{x}" y="{y}" width="{width}" height="{height}" fill="{color}" stroke-width

def draw_mondrian(width, height, color_nums, step=10):
    '''draw_mondrian draws a bunch of squares.
    Params:
      width: Width in pixels
      height: Height in pixels
      colornums: a dictionary built from the analysis of an existing mondrian image. Pixels were
    header = f'<svg viewBox="0 0 {width} {height}" xmlns="http://www.w3.org/2000/svg">\n'
    addsvgdata(header)
    colors = list(color_nums.keys())
    for y in range(0, height, step):
      x = 0
      while x < width:
        col = random.choice(colors)
        numsq = random.choice(color_nums[col])
        w = step * numsq
        if x + w >= width:
            w = width - x
        addsvgdata(draw_rect(x, y, w, step, col))
        x += w
    footer = f'</svg>'
    addsvgdata(footer)

#markov chain like lookup based on mondrian image analysis
#where did these magical numbers come from?
COLORNUMS = {'#f7352f': [1, 2, 2, 1, 1, 1, 14, 7, 4, 2, 2, 1, 1, 2, 1, 1, 15, 1, 4, 11, 3, 1, 1,
'#017bfc': [1, 1, 1, 1, 1, 2, 1, 1, 1, 8, 2, 8, 1, 13, 31, 1, 2, 28, 21, 1, 2, 11, 1, 23, 2, 4,
'#fad000': [1, 1, 1, 1, 1, 1, 2, 7, 3, 3, 2, 1, 2, 7, 8, 2, 4, 8, 1, 7, 2, 1, 4, 3, 1, 14],
'black': [9, 12, 9, 7, 4, 10, 2, 5, 6, 4, 11, 3, 23, 4, 4, 2, 19, 3, 2, 9, 9, 2, 4, 3, 3, 6, 10,
'white': [1, 3, 2, 1, 3, 2, 17, 2, 17, 1, 1, 1, 1, 2, 1, 9, 19, 1, 6, 1, 1, 4, 1, 5, 4, 1, 23, 2

draw_mondrian(500, 500, COLORNUMS)
```

```
43 display(SVG(data=svgdata))
44 with open("abitmondrian.svg","w") as fo:
45     fo.write(svgdata)
46
```

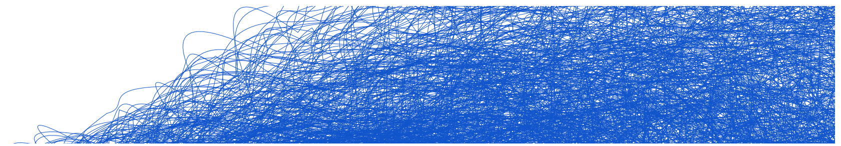
Circle wiggle

Entrée [5]:

```
import math
import random

svgdata = ""

def addsvgdata(data):
    global svgdata
    svgdata += data

def find_points(radius, num_points = 12):
    ''' Finds 12 (or how many are entered) points on the path of a circle and returns their coo
    points = []
    r = radius
    for index in range(num_points):
        points.append((r * math.cos((index * 2 * math.pi) / num_points), r * math.sin((index * 
    return points

def two_points(m, point, length):
    #returns two points for handles given one point
    px,py = point
    if m is None:
        x= px
        x1= px
        y= py - length
        y1= py + length
    else:
        x = px + length * math.sqrt(1 / (1 + m ** 2))
        x1 = px - length * math.sqrt(1 / (1 + m ** 2))
        y = py + m * length * math.sqrt(1 / (1 + m ** 2))
        y1 = py - m * length * math.sqrt(1/ (1 + m ** 2))
    return ((x, y), (x1, y1))

def inv_grad(p1, p2 = (0, 0)):
    #Find the perpendicular gradient
    x1, y1 = p1
    x2, y2 = p2
    m = (y2 - y1) / (x2 - x1)
    if m == 0:
        return None
    return -1 / m

def make_handles(flip_flag, points, length):
```

```
43     #this function gets passed the points and returns the handles
44     handles = []
45     for i, point in enumerate(points):
46         #C format past anchor, current anchor, current point
47         m = inv_grad(point)
48         a, b = two_points(m, point, length)
49         if flip_flag:
50             #handling an exception I don't understand the maths for, sorry purists
51             if i == 9:
52                 handles.append((a, b))
53             else:
54                 handles.append((b, a))
55         else:
56             handles.append((a, b))
57         if m is None or m >80:
58             flip_flag = not flip_flag
59     return handles, flip_flag
60 
61 def wiggle(px, py, length):
62     #this function adjusts the point given by a small random amount
63     #sets to -1 or 1 based on x direction from origin
64     x_shift = (px < 0) * -2 + 1
65     y_shift = (py < 0) * -2 + 1
66 
67     px += x_shift * random.random() * (length * 2)
68     py += y_shift * random.random() * (length * 2)
69     return px, py
70 
71 
72 
73 def make_wiggly_circles(radius, size, moves, color='#000000'):
74     length = radius/6
75 
76     #used to get the order of the handles correct
77     flip_flag  = False
78 
79     addsvgdata(f'<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xl
80     addsvgdata(f'<style type="text/css"> .blk{{fill:none;stroke:{color};stroke-width:0.25;}}</s
81 
82     points = find_points(radius)
83     #rotate points 4 times
84     for _ in range(3):
85         point = points.pop(0)
86         points.append(point)
87 
88     handles, flip_flag = make_handles(flip_flag, points, length)
89 
90     for move in range(moves):
```

```
 91         for _ in range(moves*2):
 92             x_trans = move * random.random() * radius / 10 + move * 10
 93             y_trans = move * random.random() * random.choice((-1, 1)) * radius / 20
 94             p = f'<path transform="translate({x_trans} {y_trans})" class="blk" d="'
 95 
 96             first_point = points[-1] #need the circle to loop
 97             fx, fy = first_point
 98             fx, fy = wiggle(fx, fy, length)
 99             p+=f'M {fx} {fy} '
100 
101             #slightly adjust all the points in the circle
102 
103             for i, point in enumerate(points[:-1]):
104                 px, py = point
105                 handle1 = handles[i-1][1]
106                 h1x, h1y = handle1
107                 h1x, h1y = wiggle(h1x, h1y, length)
108                 handle2 = handles[i][0]
109                 h2x, h2y = handle2
110                 h2x, h2y = wiggle(h1x, h1y, length)
111                 px, py = wiggle(px, py, length)
112                 p+=f'C {round(h1x,2)} {round(h1y,2)} {round(h2x,2)} {round(h2y,2)} {round(px,2)
113 
114             #last point
115             px, py = fx, fy
116             handle1 = handles[-2][1]
117             h1x, h1y = handle1
118             h1x, h1y = wiggle(h1x, h1y, length)
119             handle2 = handles[-1][0]
120             h2x, h2y = handle2
121             p+=f'C {round(h1x,2)} {round(h1y,2)} {round(h2x,2)} {round(h2y,2)} {round(px,2)} {r
122             #this fixes the stupid spiky join stupid
123 
124             p+=' Z"/>\n'
125 
126             addsvgdata(p)
127 
128 
129     addsvgdata('</svg>')
130 
131 make_wiggly_circles(150, 300, 40, color='rgb(17, 85, 204)')
132 display(SVG(data=svgdata))
133 with open("wiggle.svg","w") as fo:
134     fo.write(svgdata)
```



Circles

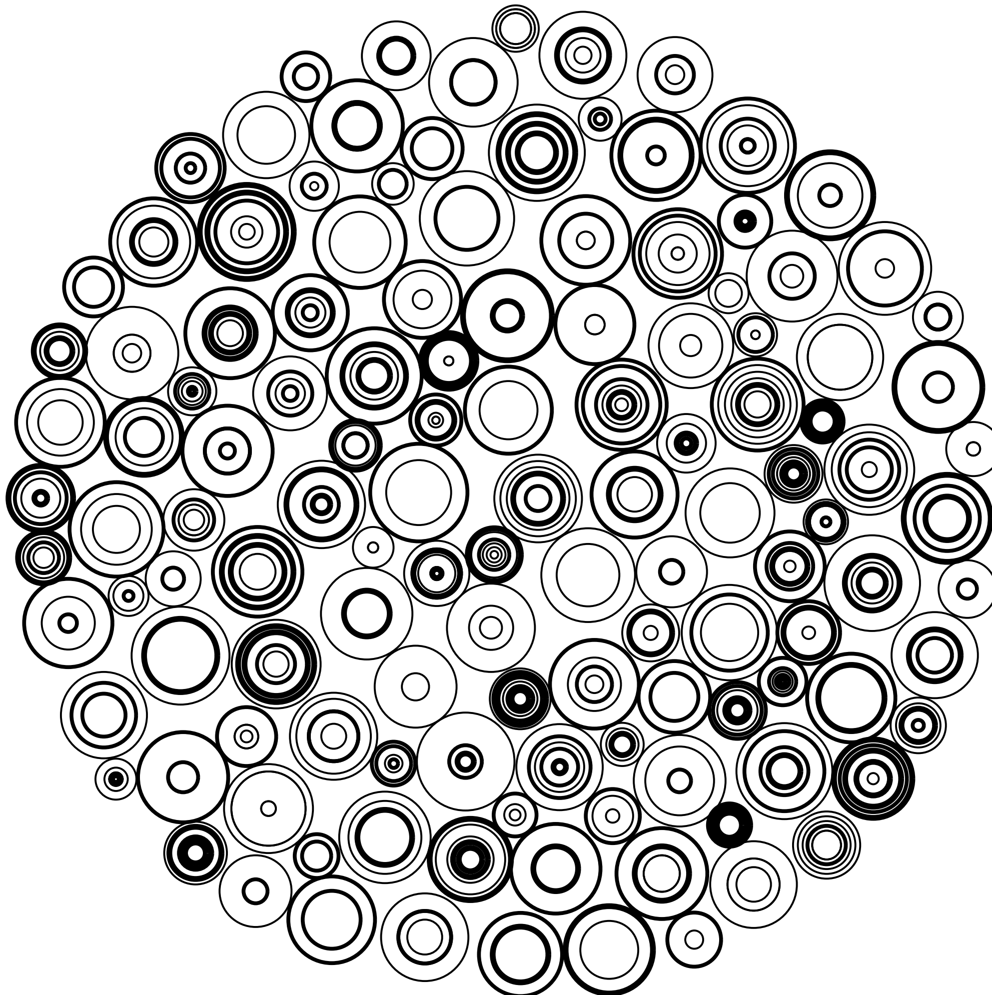
Entrée [6]:

```
import numpy as np
import random

svgdata = ""

def addsvgdata(data):
    global svgdata
    svgdata += data

#original source https://scipython.com/blog/packing-circles-in-a-circle/

class Circle:
    """A little class representing an SVG circle."""

    def __init__(self, cx, cy, r, maxrings=6, icolour=None):
        """Initialize the circle with its centre, (cx,cy) and radius, r.

        icolour is the index of the circle's colour.

        """
        self.cx, self.cy, self.r = cx, cy, r
        self.rings = random.randint(2,maxrings)
        self.icolour = icolour

    def overlap_with(self, cx, cy, r):
        """Does the circle overlap with another of radius r at (cx, cy)?"""

        d = np.hypot(cx-self.cx, cy-self.cy)
        return d < r + self.r

    def draw_circle(self):
        """Write the circle's SVG to the output stream"""
        for _ in range(self.rings-1):
            num = random.randint(1,self.rings)
            gap = self.r/(self.rings+1)
            rad = gap*num
            sw = random.choice([1,1,1,2,2,3])
            addsvgdata(f'<circle cx="{self.cx}" cy="{self.cy}" r="{rad}" stroke-width="{sw}" cla
        sw = random.choice([1,1,1,2,2,3])
        addsvgdata(f'<circle cx="{self.cx}" cy="{self.cy}" r="{self.r}" stroke-width="{sw}" cla

class Circles:
```

```
43     """A class for drawing circles-inside-a-circle."""
44
45     def __init__(self, width=600, height=600, R=250, n=600, rho_min=0.04,
46                  rho_max=0.1, colours=None):
47         """Initialize the Circles object.
48
49         width, height are the SVG canvas dimensions
50         R is the radius of the large circle within which the small circles are
51         to fit.
52         n is the maximum number of circles to pack inside the large circle.
53         rho_min is rmin/R, giving the minimum packing circle radius.
54         rho_max is rmax/R, giving the maximum packing circle radius.
55         colours is a list of SVG fill colour specifiers to be referenced by
56             the class identifiers c<i>. If None, a default palette is set.
57
58         """
59
60         self.width, self.height = width, height
61         self.R, self.n = R, n
62         # The centre of the canvas
63         self.CX, self.CY = self.width // 2, self.height // 2
64         self.rmin, self.rmax = R * rho_min, R * rho_max
65         self.colours = colours or ['none']
66
67     def preamble(self):
68         """The usual SVG preamble, including the image size."""
69
70         addsvgdata('<?xml version="1.0" encoding="utf-8"?>\n\n')
71         addsvgdata(f'<svg xmlns="http://www.w3.org/2000/svg"\n xmlns:xlink="http://www.w3.org/1
72
73     def defs_decorator(func):
74         """For convenience, wrap the CSS styles with the needed SVG tags."""
75
76         def wrapper(self):
77             addsvgdata("""
78             <defs>
79             <style type="text/css"><![CDATA[""")
80
81             func(self)
82
83             addsvgdata("""]]></style>
84             </defs>""")
85         return wrapper
86
87     @defs_decorator
88     def svg_styles(self):
89         """Set the SVG styles: circles are coloured with no border."""
90
```

```
91          addsvgdata('circle {stroke: black;}')
92          for i, c in enumerate(self.colours):
93              addsvgdata(f'.c{i} {{fill: {c}; }}')
94
95      def make_svg(self, *args, **kwargs):
96          """Create the image as an SVG file with name filename."""
97
98          self.preamble()
99          self.svg_styles()
100         for circle in self.circles:
101             circle.draw_circle()
102         addsvgdata('</svg>')
103
104     def _place_circle(self, r):
105         # The guard number: if we don't place a circle within this number
106         # of trials, we give up.
107         guard = 500
108         while guard:
109             # Pick a random position, uniformly on the larger circle's interior
110             cr, cphi = ( self.R * np.sqrt(np.random.random()),
111                          2*np.pi * np.random.random() )
112             cx, cy = cr * np.cos(cphi), cr * np.sin(cphi)
113             if cr+r < self.R:
114             # The circle fits inside the larger circle.
115                 if not any(circle.overlap_with(self.CX+cx, self.CY+cy, r)
116                                     for circle in self.circles):
117                     # The circle doesn't overlap any other circle: place it.
118                     circle = Circle(cx+self.CX, cy+self.CY, r,
119                                  icolour=np.random.randint(len(self.colours)))
120                     self.circles.append(circle)
121                     return
122             guard -= 1
123         # Warn that we reached the guard number of attempts and gave up for
124         # for this circle.
125         #print('guard reached.')
126
127     def make_circles(self):
128         """Place the little circles inside the big one."""
129
130         # First choose a set of n random radii and sort them. We use
131         # random.random() * random.random() to favour small circles.
132         self.circles = []
133         r = self.rmin + (self.rmax - self.rmin) * np.random.random(
134                             self.n) * np.random.random(self.n)
135         r[::-1].sort()
136         # Do our best to place the circles, larger ones first.
137         for i in range(self.n):
138             self._place_circle(r[i])
```

```
139
140 circles = Circles(n=2000)
141 circles.make_circles()
142 circles.make_svg()
143
144 # Warning: very long wait before display
145 display(SVG(data=svgdata))
146 with open("circles.svg","w") as fo:
147     fo.write(svgdata)
```

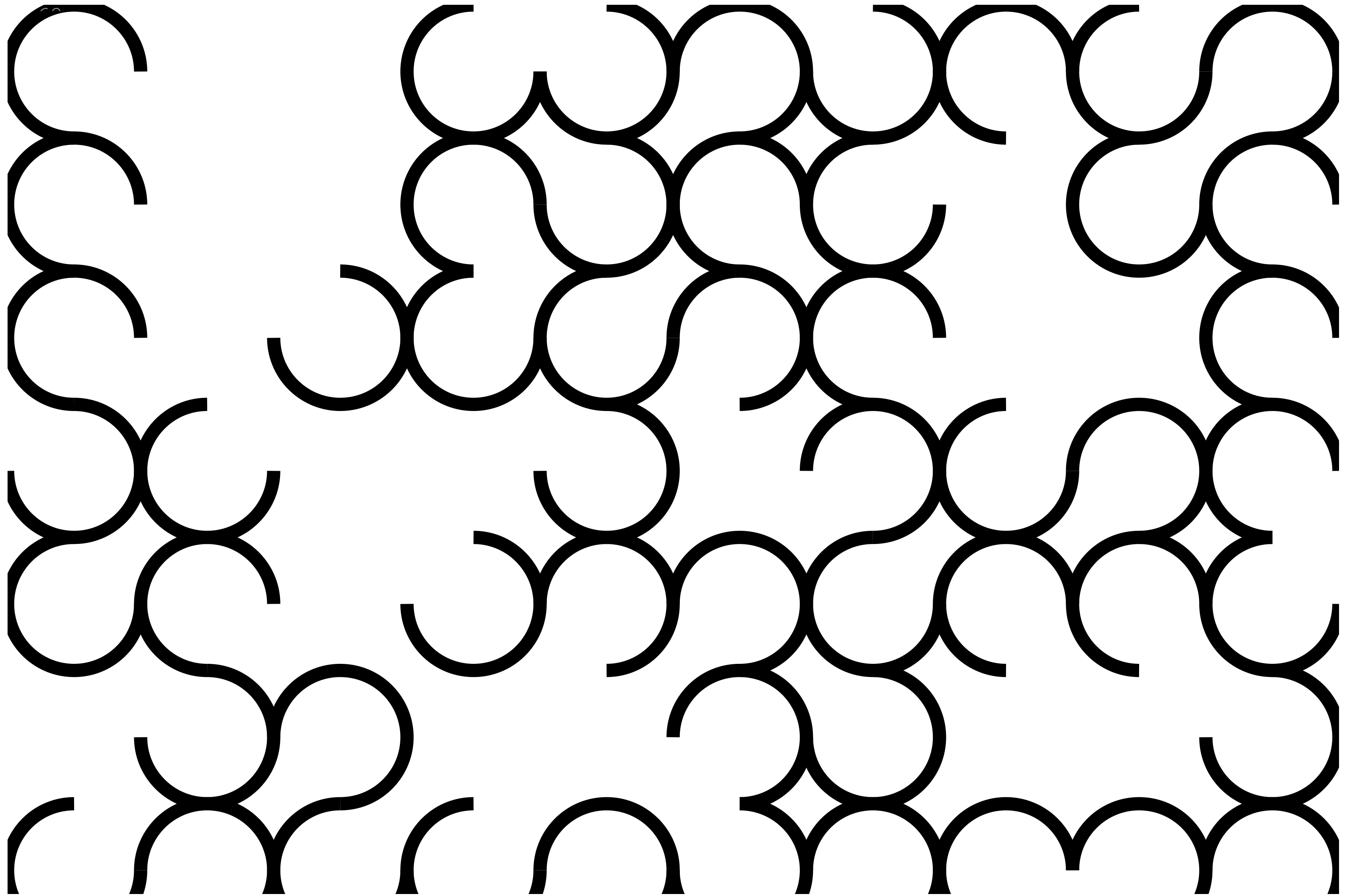


Arcs

Entrée [7]:

```
#draw a quarter of a square (symmetry coming)
import random

svgdata = ""

def addsvgdata(data):
    global svgdata
    svgdata += data

class Curve:
    def __init__(self,num,x,y,r,stroke,large_a_f=0,small_a_f=0,end_x=0,end_y=0):
        self.stroke = stroke
        self.start_x=x
        self.start_y=y
        self.r=r
        self.large_a_f = large_a_f
        self.small_a_f = small_a_f
        self.end_x = end_x
        self.end_y = end_y

    def get_path(self):
        return f'<path d="M {self.start_x} {self.start_y}\
            A {self.r} {self.r} 0 {self.large_a_f} {self.small_a_f} \
            {self.end_x} {self.end_y}" fill="none" stroke="black" \
            stroke-width="{self.stroke}"/>'

def main(num,imageSize):
    header = f'<svg viewBox="0 0 {imageSize} {imageSize}" xmlns="http://www.w3.org/2000/svg">\n'
    footer = f'</svg>'
    grid = {}
    c_size = imageSize/num
    addsvgdata(header)
    for r in range(num):
        for c in range(num):
            n = random.randint(0,4)
            s = 4
            rad = c_size/2
            twos = n//2
            ones = n%2

            x = c * c_size + (rad * int(twos != ones))
            y = r * c_size + rad + (rad * twos) - (rad * ones)
            end_x=c * c_size + rad + (rad * int(twos != ones))
            end_y=r * c_size + (rad * int(ones)) + (rad * int(twos))
            large_a_f = 1
            small_a_f = 1 * twos
```

```
47
48
49                curve = Curve(n,x,y,rad,s,large_a_f,small_a_f,end_x,end_y)
50                grid[(r,c)] =curve
51                addsvgdata(curve.get_path())
52
53
54      addsvgdata(footer)
55
56
57  main(10,400)
58  display(SVG(data=svgdata))
59  with open("arcs.svg","w") as fo:
60      fo.write(svgdata)
61
```



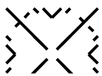
Diagonal lines

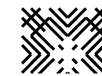
Entrée [8]:

```
#rewrite to make it all built out of diagonal lines.
import random

svgdata = ""

def addsvgdata(data):
    global svgdata
    svgdata += data

def draw_line(x, y, length, pos, size, sections, top_left_x, top_left_y, sw, sym=False, color='
    '''returns the svg xml for the line
    Params:
    x - starting x value
    y - starting y value
    length - number of squares across and down
    pos - whether the gradient is positive or negative
    size - the size of each square in the grid
    topleftX - the starting point for the tile x value
    topleftY - the starting point for the tile y value
    sw - strokewidth
    sym - whether this is the symmtrical duplicate of another line
    color of the stroke
    '''
    if sym:
        length = - length
    x2 = x + length
    if pos:
        y2 = y - length
    else:

        y2 = y + length

    if x2 < 0:
        y2 = y2 + x2
        x2 = 0
    if y2 < 0:
        x2 = x2 + y2
        y2 = 0
    #don't actually know why I need this -- dumb computers
    if sym and pos:
        temp = x2
        x2 = y2
```

```
43            y2 = temp
44
45        #depending on all the above conditions output the format for the line
46        return f'<line x1="{top_left_x + x * size}" y1="{top_left_y + y * size}" x2="{top_left_x + 
47        #return f'<line x1="{x * size + top_left_x}" y1="{y * size + top_left_y}" x2="{x2 * size + 
48
49    def create_lines(sections):
50        #Make lines (not actually unique) in the bottom right diagonal half
51        #of the bottom quadrant and then repeats in all the places
52        num_lines = random.randint(sections // 2, int(sections * 1.5))
53        point_set = set()
54        lines = []
55        neg = True
56        #repeats ignoring duplicate points
57        while len(lines) < num_lines:
58            #get a random even number for x adn y
59            x = random.randint(0, sections // 2 - 1) * 2
60            y = random.randint(0, x // 2) * 2
61
62            #calculate the maximum length
63            top = sections - x - y if x == y and x == 2 else sections-x
64            #pick a random length between 1 and the maximum
65            length = random.randint(1, top)
66            #alternate the positivity of the gradient depending on starting point
67            pos = (x in (2, 4)and y in (2, 4))
68
69            if (x, y) not in point_set:
70                lines.append((x, y, length, pos))
71                point_set.add((x, y))
72        return lines
73
74    def make_tiles(size, tiles ,image_size, sw=1):
75        #takes number of sections in each quadrant (indicates intricacy)
76        #tiles is the number of repetitions across and down
77        #imagesize is number of pixels across and down but it also affects
78        #the relative stroke width --> sw is stroke width
79        image_pad = 50
80        header = f'<svg viewBox="-{image_pad} -{image_pad} {image_size + 2 * image_pad} {image_size
81        addsvgdata(header)
82        tile_size = image_size / tiles
83        padding = image_size / tiles / 2
84        padding = 40
85        sections = size
86        for row in range(tiles):
87            for column in range(tiles):
88                #cutting off at the border is not working
89
90                #make a tile quadrant group (without <g> wrapping just yet)
```

```
91              group = ''
92              top_left_x = column * tile_size + padding / 2
93              top_left_y = row * tile_size + padding / 2
94              #print(top_left_x, top_left_y)
95              square_size = ((tile_size - padding) / size) / 2
96              lines= create_lines(sections)
97              for line in lines:
98                  #do the same thing (ish) in the four different quadrants
99                  x, y, length, pos = line
100                 x1 = x
101                 y1 = y
102
103                 #print(x1,y1,length,pos,a,b)
104                 l = draw_line(x1, y1, length, pos, square_size, sections, top_left_x, top_left_
105                 group += l +'\n'
106                 #x, y, length, pos, size, sections, top_left_x, top_left_y, sw, sym=False, colo
107                 if x != y or pos:
108                     x1 = y
109                     y1 = x
110                     l2 = draw_line(x1, y1, -length, pos, square_size, sections, top_left_x, top_
111                     #x, y, length, pos, size, sections, top_left_x, top_left_y, sw, sym=False, 
112                     group += l2 +'\n'
113
114             #repeat this quadrant 3 more times
115             for quad in range(4):
116                 rotation = quad * 90
117                 rotationX = top_left_x + (tile_size - padding) / 2
118                 rotationY = top_left_y + (tile_size - padding) / 2
119                 addsvgdata(f'<g transform="rotate({rotation} {rotationX} {rotationY})">{group}<
120
121
122
123     footer = f'</svg>'
124     addsvgdata(footer)
125
126
127 make_tiles(10, 10 ,1000, sw=2)
128
129 display(SVG(data=svgdata))
130 with open("diagonals.svg","w") as fo:
131     fo.write(svgdata)
132
133
```





Double pendulum

Entrée [9]:

```
import math
'''
Appreciation to the coding train who inspired this experiment
https://www.youtube.com/watch?v=uWzPe_S-RVE
'''
svgdata = ""

def addsvgdata(data):
    global svgdata
    svgdata += data

#setup basics
#radii
r1 = 100
r2 = 100
#masses
m1 = 5
m2 = 10
#angles
a1 = math.pi/2
a2 = math.pi/4
#velocities
a1_v = 0
a2_v = 0
#accelerations
a1_a = 0
a2_a = 0
#gravity
g = 1

previous_x2 = None
previous_y2 = None


#draw a quarter of a square (symmetry coming)
HEIGHT = 600
WIDTH = 600
header = f'<svg viewBox="-300 -300 {WIDTH} {HEIGHT}" xmlns="http://www.w3.org/2000/svg" style="l

style = f'''
        <filter id="glow">
            <feGaussianBlur class="blur" result="coloredBlur" stdDeviation="4"></feGaussianBlur
```

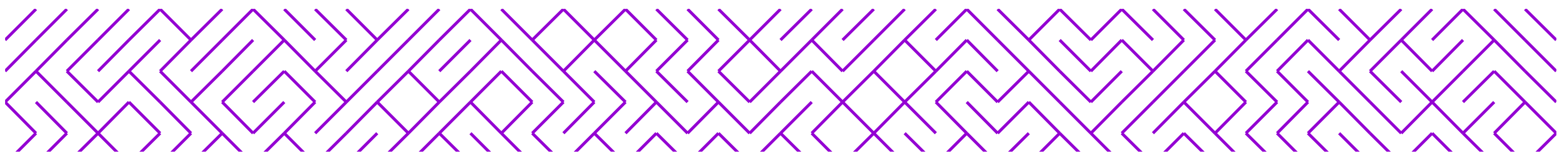
```
43              <feMerge>
44                  <feMergeNode in="coloredBlur"></feMergeNode>
45                  <feMergeNode in="coloredBlur"></feMergeNode>
46                  <feMergeNode in="coloredBlur"></feMergeNode>
47                  <feMergeNode in="SourceGraphic"></feMergeNode>
48              </feMerge>
49          </filter>
50          <g style="fill-opacity: 0; stroke-width: 2; stroke: green;filter:url(#glow)">
51
52          '''
53
54
55
56  def draw_circle(cx, cy, rad, color='black'):
57      return f'<circle cx="{cx}" cy="{cy}" r="{rad}" fill="{color}" stroke-width="0"/>'
58
59  def draw_line(x_start, y_start, x_end, y_end, color='black'):
60      return f'<line x1="{x_start}" y1="{y_start}" x2="{x_end}" y2="{y_end}" stroke="{color}"/>'
61
62  def first_point(x,y):
63      return f'<path d="M {x}, {y} '
64
65  def add_point(x,y):
66      return f'L{x}, {y}'
67
68  addsvgdata(header)
69  addsvgdata(style)
70
71  #make some lines
72  path = ""
73
74  for i in range(2000):
75      #increment accelerations (funky formula time)
76      num1 = -g * (2 * m1 + m2) * math.sin(a1)
77      num2 = -m2 * g * math.sin(a1-2*a2)
78      num3 = -2 * math.sin(a1 - a2) * m2
79      num4 = a2_v * a2_v * r2 + a1_v * a1_v * r1 * math.cos(a1 - a2)
80      den = r1 * (2 * m1 + m2 - m2 * math.cos(2 * a1 - 2 * a2))
81      a1_a = (num1 + num2 + num3 * num4) / den /10
82      #acceleration 2
83      num1 = 2 * math.sin(a1 - a2)
84      num2 = (a1_v * a1_v * r1 * (m1 + m2))
85      num3 = g * (m1 + m2) * math.cos(a1)
86      num4 = a2_v * a2_v * r2 * m2 * math.cos(a1 - a2)
87      den = r2 * (2 * m1 + m2 - m2 * math.cos(2 * a1 - 2 * a2))
88      a2_a = num1 * (num2 + num3 + num4) / den
89
90      #first pendulum
```

```
91      x1 = r1 * math.sin(a1)
92      y1 = r1 * math.cos(a1)
93      # print(draw_line(0, 0, x1, y1))
94      # print(draw_circle(x1, y1, m1, color=f'hsl({i*20},100%, 50% )'))
95      #second pendulum
96      x2 = x1 + r2 * math.sin(a2)
97      y2 = y1 + r2 * math.cos(a2)
98      #draw the second pendulum
99      # print(draw_line(x1, y1, x2, y2))
100     # print(draw_circle(x2, y2, m2, color=f'hsl({i*20},50%, 50% )'))
101     if previous_x2 == None:
102         path += first_point(x2, y2)
103     else:
104         path += add_point(x2, y2)
105
106
107     #increase velocity by accel
108     a1_v += a1_a
109     a2_v += a2_a
110
111     #increase angle
112     a1 = (a1 + a1_v)
113     a2 = (a2 + a2_v)
114
115
116     previous_x2 = x2
117     previous_y2 = y2
118     #print()
119
120 path+= '" />'
121
122
123 addsvgdata(path)
124 footer = '</g></svg>'
125 addsvgdata(footer)
126
127 display(SVG(data=svgdata))
128 with open("pendulum.svg","w") as fo:
129     fo.write(svgdata)
```

Maze

Entrée [10]:

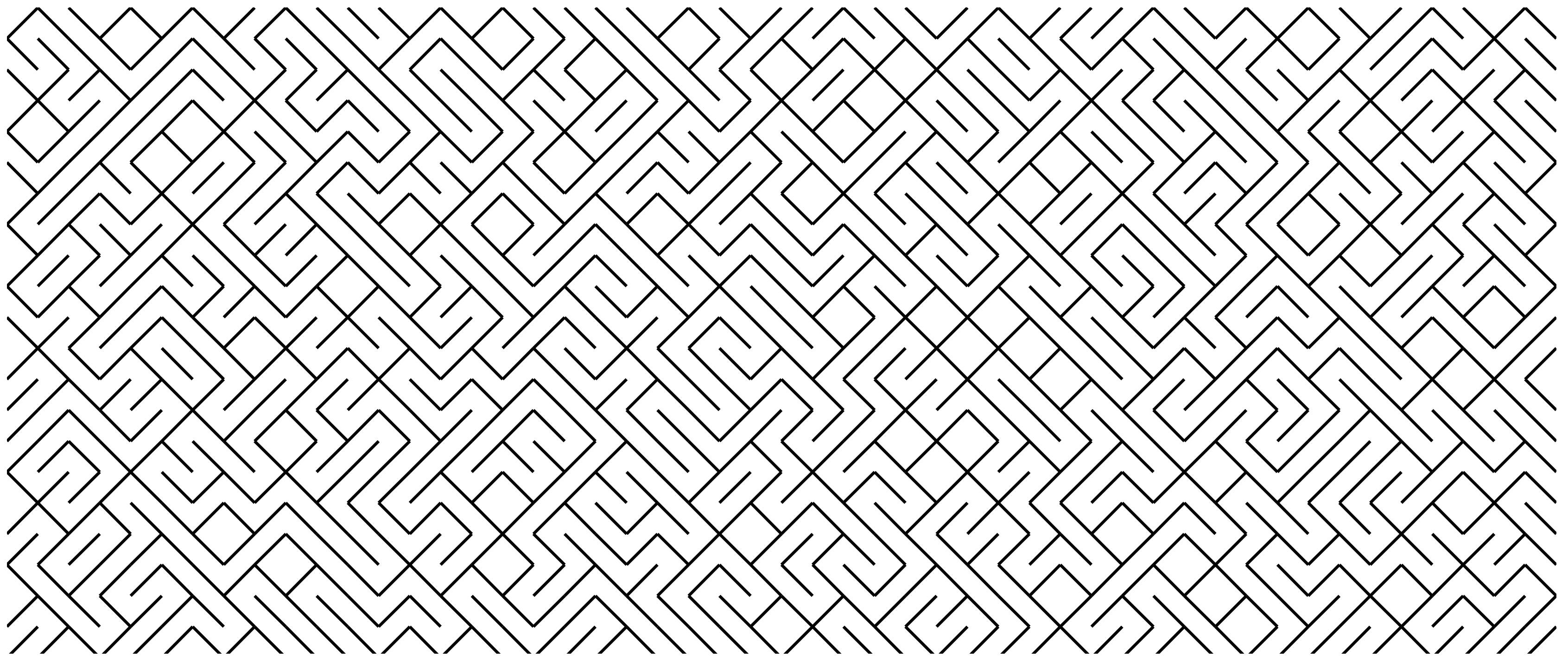
```python
import random

svgdata = ""

def addsvgdata(data):
    global svgdata
    svgdata += data

def draw_diagonal(x, y, x2, y2, color='black'):
    #format a line for svg
    return f'<line x1="{x}" y1="{y}" x2="{x2}" \
            y2="{y2}" stroke="{color}" />'

def make_maze(num, color = "black" , height = 500):
    #num is the number of diagonals across and down
    #filename is the output file name
    width = height
    header = f'<svg viewBox="0 0 {width} {height}" xmlns="http://www.w3.org/2000/svg">\n'

    step=height // num
    #write the header for the file
    addsvgdata(header)
    #loop through
    for row in range(0, height, step):
      for col in range(0, width, step):
        #choose a random direction for the diagonal line
        if random.choice([0, 1]) == 0:
            #write the line
            addsvgdata(draw_diagonal(col, row, col + step, row + step, color))
        else:
            #write the line
            addsvgdata(draw_diagonal(col + step, row, col, row + step, color))
    #write the footer
    footer = f'</svg>'
    addsvgdata(footer)


make_maze(50, color="darkviolet" )

display(SVG(data=svgdata))
with open("maze.svg","w") as fo:
    fo.write(svgdata)
```



maze for web: can be used in a web page

Entrée [11]:

```
import random

def drawDiag(x,y,x2,y2,color='black'):
    return f'<line x1="{x}" y1="{y}" x2="{x2}" y2="{y2}" stroke="{color}" />'

def makeMaze(num, color = "black" , height = 500):
    svgOut = ''
    width = height
    header = f'<svg viewBox="0 0 {width} {height}" xmlns="http://www.w3.org/2000/svg">\n'
    step = height // num
    svgOut += header

    for x in range(0, width, step):
      for y in range(0, height, step):

        if random.choice([0,1]) == 0:
            svgOut += drawDiag(x, y, x + step, y + step, color) + '\n'
        else:
            svgOut += drawDiag(x + step, y, x, y + step, color) + '\n'


    footer = f'</svg>'
    svgOut += footer
    return svgOut

svgdata = makeMaze(50)
```



Quadrilaterals

Entrée [12]:

```
#draw a quarter of a square (symmetry coming)
import random

svgdata = ""

def addsvgdata(data):
    global svgdata
    svgdata += data

class Quadrilateral:
    def __init__(self, x1, x2, y1, y2, col):
        '''
        Attributes:
        self.xTL x value of the top left
        self.xBL x value of the bottom left
        self.xTR x value of the top right
        self.xBR x value of the bottom right
        self.yTL y value of the top left
        self.yTR y value of the bottom left
        self.yBL y value of the top right
        self.yBR y value of the bottom right
        self.col color
        '''
        self.xTL = x1
        self.xBL = x1
        self.xTR = x2
        self.xBR = x2
        self.yTL = y1
        self.yTR = y1
        self.yBL = y2
        self.yBR = y2
        self.col = col




def draw_quad(quad):
##    M = moveto
##    L = lineto
    return f'<path d="M{quad.xTL} {quad.yTL} \
             L{quad.xTR} {quad.yTR} \
             L{quad.xBR} {quad.yBR} \
```

```
43              L{quad.xBL} {quad.yBL}Z" stroke="black" fill="{quad.col}"/>'
44
45  def make_quadrilaterals(num, imageSize, colors = ('#044BD9', '#0583F2', '#05AFF2', '#05DBF2', '
46      header = f'<svg viewBox="0 0 {imageSize} {imageSize}" xmlns="http://www.w3.org/2000/svg">\n
47      grid = {}
48      quad_size = imageSize/num
49      addsvgdata(header)
50      for r in range(num):
51          for c in range(num):
52              q = None
53              q = Quadrilateral(
54                              c * quad_size, c * quad_size + quad_size, r * quad_size,
55                              r * quad_size + quad_size, random.choice(colors))
56              grid[(r, c)] = q
57
58      #shifting stuff
59      for r in range(1, num):
60          for c in range(1, num):
61              diffx = random.randint(0, quad_size // 2) - quad_size // 4
62              diffy = random.randint(0, quad_size // 2) - quad_size // 4
63              q = grid[(r, c)]
64              qUpLeft = grid[(r - 1, c - 1)]
65              qUp = grid[(r - 1, c)]
66              qLeft = grid[(r, c - 1)]
67
68              #update all 4 quads for the square I'm working with's top left
69              q.xTL += diffx
70              q.yTL += diffy
71              qUpLeft.xBR += diffx
72              qUpLeft.yBR += diffy
73              qUp.xBL += diffx
74              qUp.yBL += diffy
75              qLeft.xTR += diffx
76              qLeft.yTR += diffy
77              addsvgdata(draw_quad(qUpLeft) )
78
79          addsvgdata(draw_quad(qUp) )
80
81      for c in range(num):
82          q = grid[(r, c)]
83          addsvgdata(draw_quad(q) )
84      footer = f'</svg>'
85      addsvgdata(footer)
86
87
88  make_quadrilaterals(30,450)
89
90  display(SVG(data=svgdata))
```

```
91 with open("quadrilaterals.svg","w") as fo:
92     fo.write(svgdata)
93
94
```

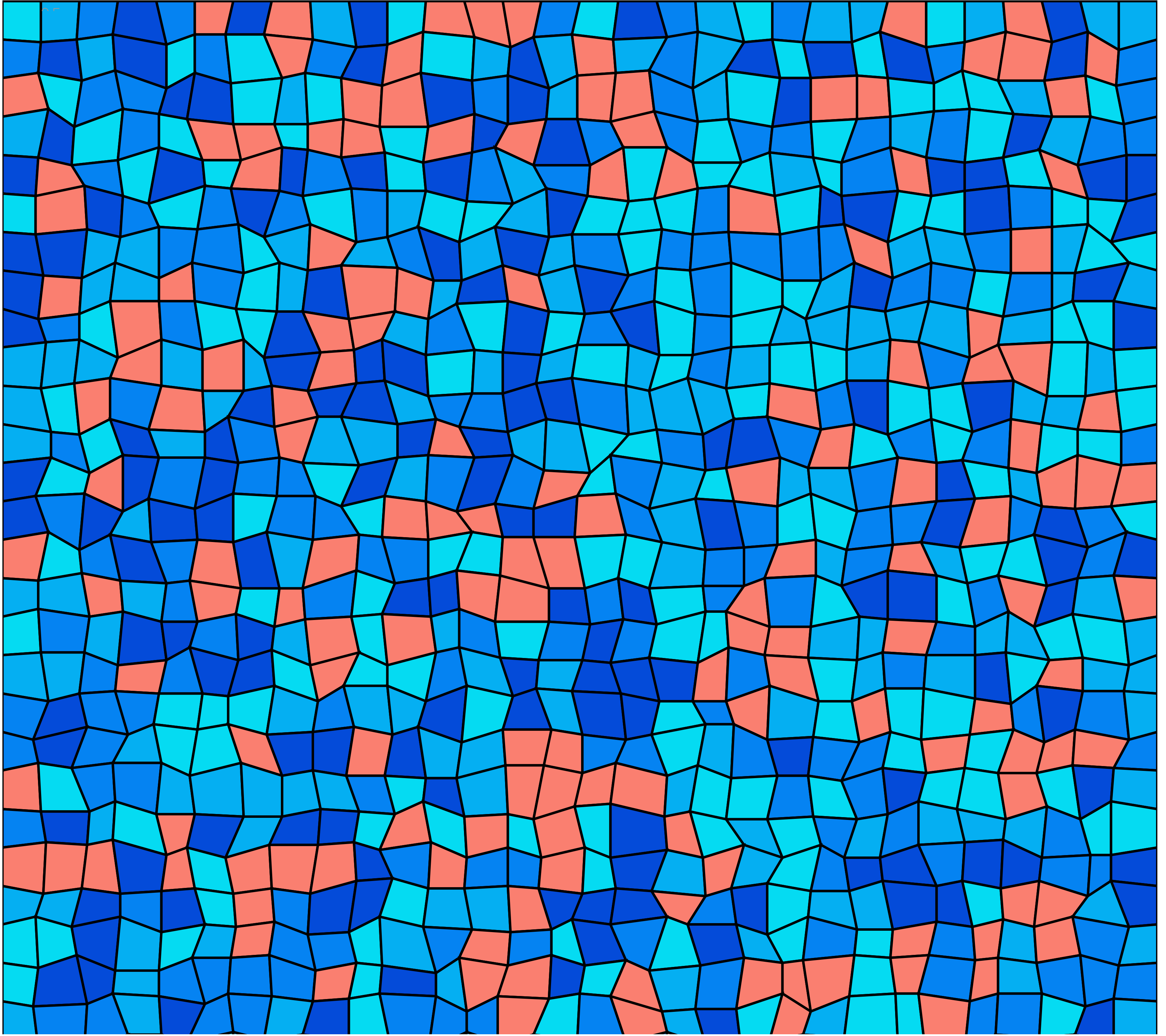


Image analysis

Entrée [4]:

```
from PIL import Image

def addsvgdata(data):
    global svgdata
    svgdata += data


class Circle:
    def __init__(self,cx,cy,pixels, topL):
        self.topL = topL
        self.cx = cx
        self.cy = cy
        self.rad = pixels.width/2
        self.pixels = pixels
        self.avg_col = self.get_avg_col()
        self.colour = pixels.getpixel((pixels.width//2, pixels.height//2))
        self.pixelDiff = self.get_diff()

        #print(self.colour)

    def get_diff(self):
        r,g,b = self.avg_col
        r1,g1,b1 = self.colour

        return abs(r-r1)+abs(g-g1)+abs(b-b1)

    def get_avg_col(self):
        red = []
        green = []
        blue = []
        for y in range(self.pixels.height):
            for x in range(self.pixels.width):
                pixel = self.pixels.getpixel((x, y))
                r, g, b = pixel
                red.append(r)
                green.append(g)
                blue.append(b)
        if len(red) ==0:
            return 0
        return sum(red)/len(red),sum(green)/len(green),sum(blue)/len(blue)


```

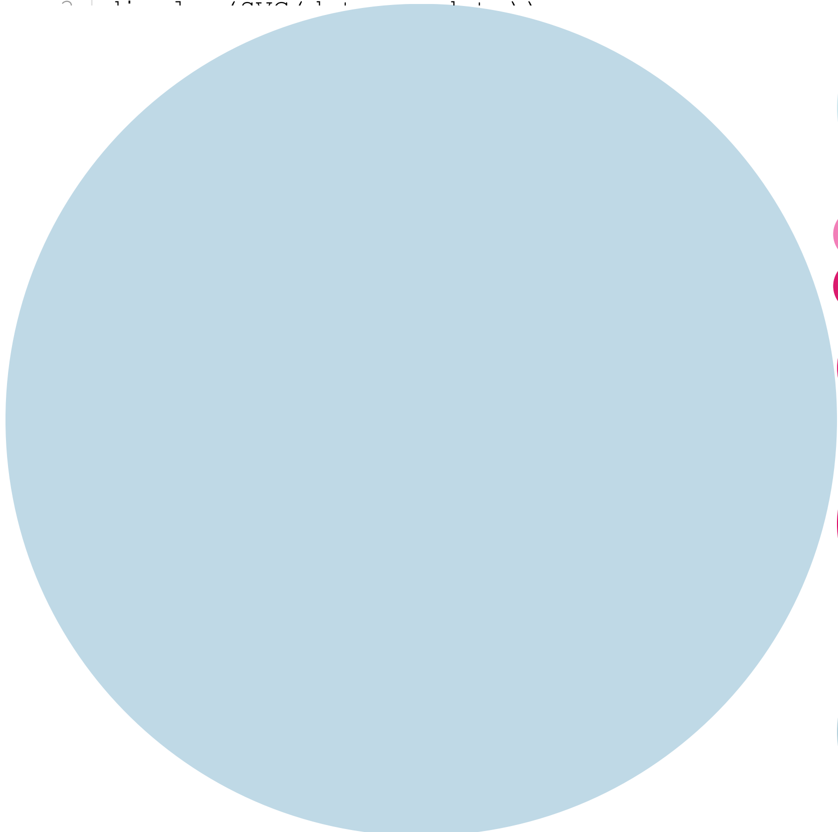
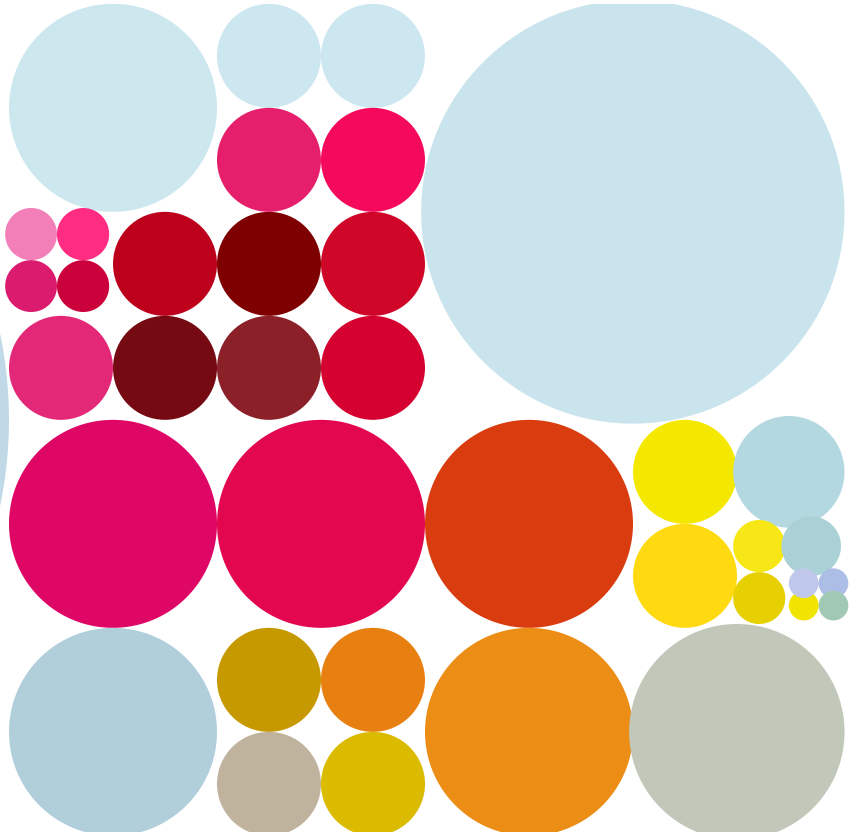
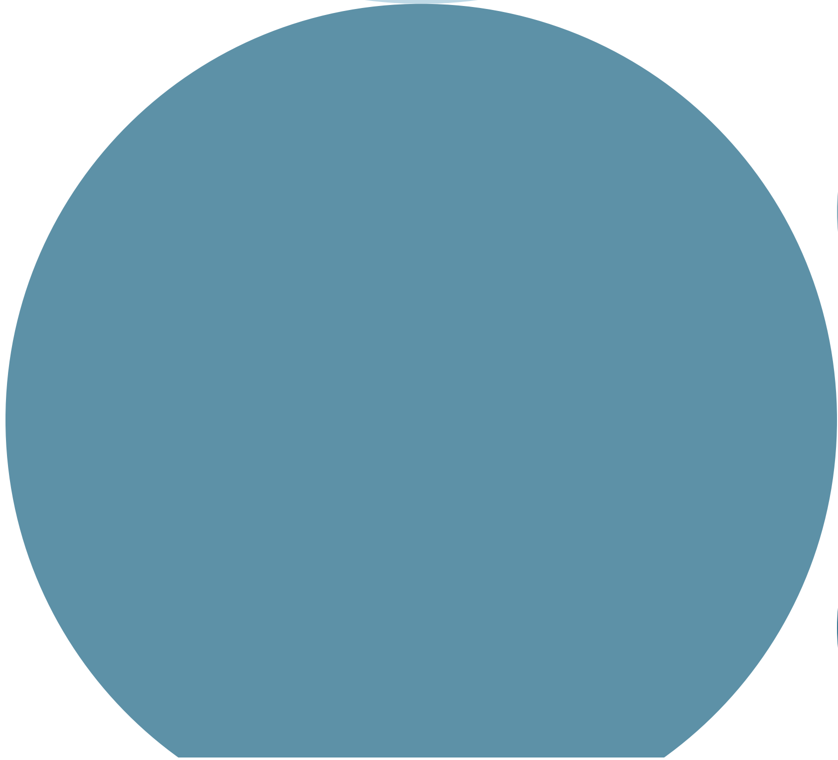
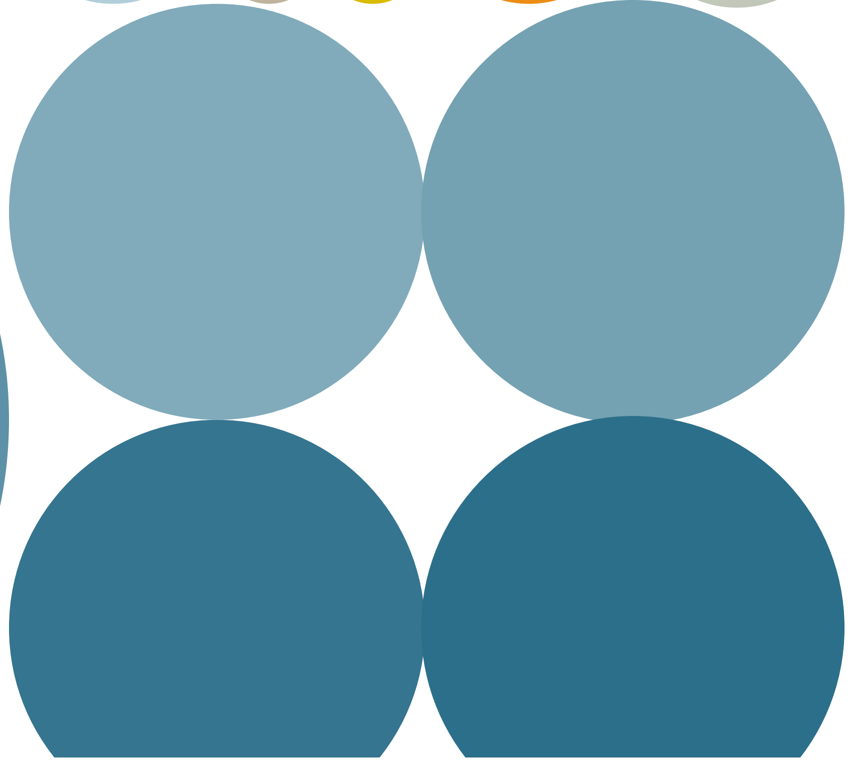
```python
43  def draw_circle(cx,cy,radius, colour):
44      return f'<circle cx="{cx}" cy="{cy}" r="{radius}" fill="rgb{colour}" stroke-width="0"/>'
45
46  def get_max(circles):
47      maxDiff = None
48      index = 0
49      temp=[]
50      for i in range(len(circles)):
51          pD = circles[i].pixelDiff
52          temp.append(str(pD))
53          if maxDiff is None or pD>maxDiff:
54              maxDiff = pD
55              index = i
56      return maxDiff,index
57
58  def analysePic(im):
59      #this should definitely be recursive
60      #biggest circle
61      topL = (0,0)
62      cx = im.width // 2
63      cy = im.height // 2
64      pixels = im.crop((0, 0, im.width, im.height))
65      circles = [Circle(cx, cy, pixels, topL)]
66      #set up the first 4 quadrants
67      radius = im.width // 2
68      index = 0
69      #split the new circle
70      newCircle = circles.pop(index)
71      newCircle.get_diff()
72      #repeat until the radius gets to 5
73      while radius > 2:
74          old_topL = newCircle.topL
75          topLx, topLy = old_topL
76          im = newCircle.pixels
77          radius = im.width // 4
78          #c1
79          topL1 = old_topL
80          cx = im.width // 4
81          cy = im.height // 4
82          im1 = im.crop((0, 0, im.width // 2, im.height // 2))
83          circles.append(Circle(cx, cy, im1, topL1))
84          #c2
85          #topL2 = old_topL[0]+radius*2,old_topL[1]
86          topL2 = topLx + im.width // 2,topLy
87          cx = im.width // 4
88          cy = im.height // 4
89          im2 = im.crop((im.width // 2, 0, im.width, im.height // 2))
90          circles.append(Circle(cx, cy, im2, topL2))
```

```
        #c3
        #topL3 = old_topL[0],old_topL[1]+radius*2
        topL3 = topLx, topLy + im.height // 2
        cx = im.width // 4
        cy = im.height // 4
        im3 = im.crop((0, im.height // 2, im.width // 2, im.height ))
        circles.append(Circle(cx, cy, im3, topL3))
        #c4
        topL4 = topLx + im.width // 2, topLy + im.height // 2
        cx = im.width // 4
        cy = im.height // 4
        im4 = im.crop((im.width // 2, im.height // 2, im.width, im.height))
        circles.append(Circle(cx, cy, im4, topL4))
        maxDiff, index = get_max(circles)
        #split the new circle
        newCircle = circles.pop(index)
    #put the final circle back
    circles.append(newCircle)
    return circles



def picCircles(image, filename):
    im = Image.open(image)
    width = im.width
    height = im.height
    header = f'<svg viewBox="0 0 {width} {height}" xmlns="http://www.w3.org/2000/svg">\n'
    footer = f'</svg>'

    sv = open(filename + ".svg","w")
    sv.write(header)
    addsvgdata(header)

    circles = analysePic(im)

    for circle in circles:
        topLx,topLy = circle.topL
        sv.write(draw_circle(circle.cx + topLx, circle.cy + topLy, circle.rad, circle.colour))
        addsvgdata(draw_circle(circle.cx + topLx, circle.cy + topLy, circle.rad, circle.colour)
    sv.write(footer)
    addsvgdata(footer)
    sv.close()
```

Entrée [5]:

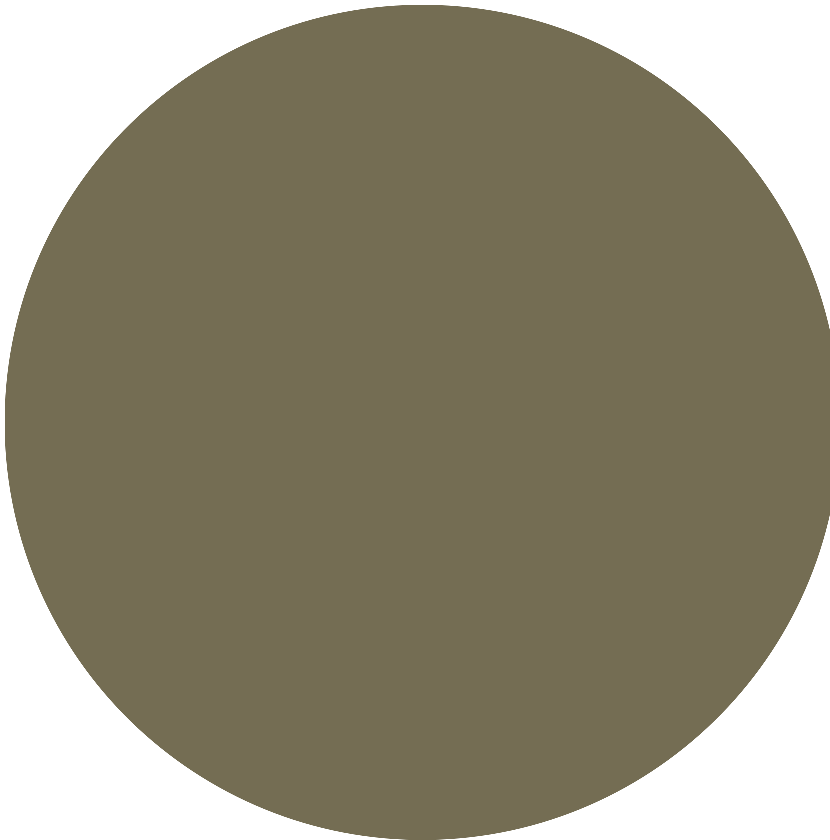
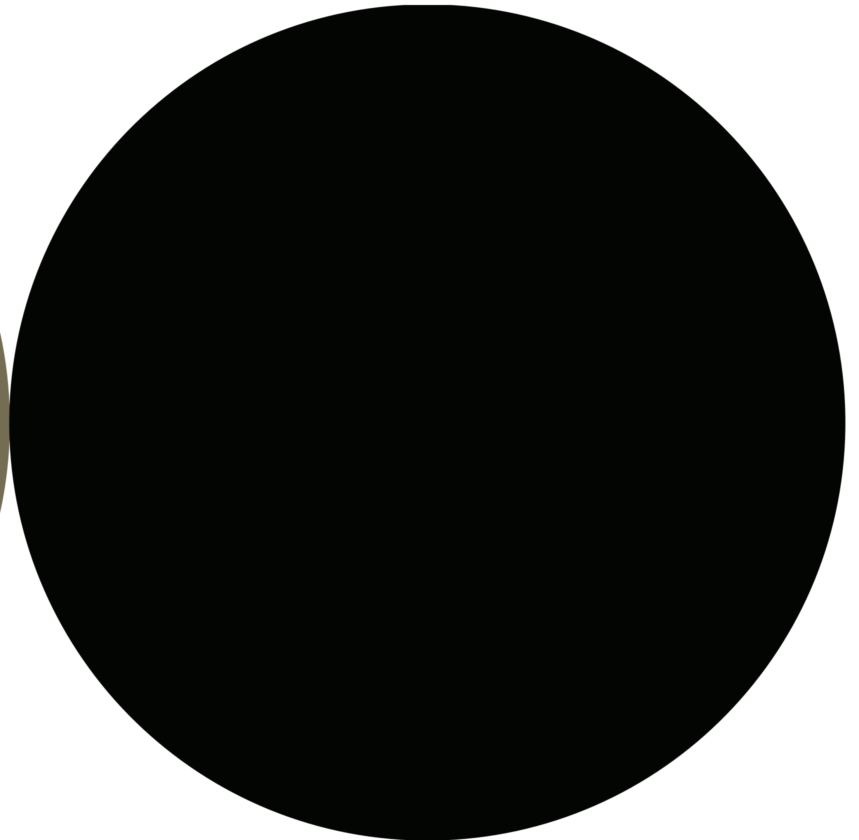
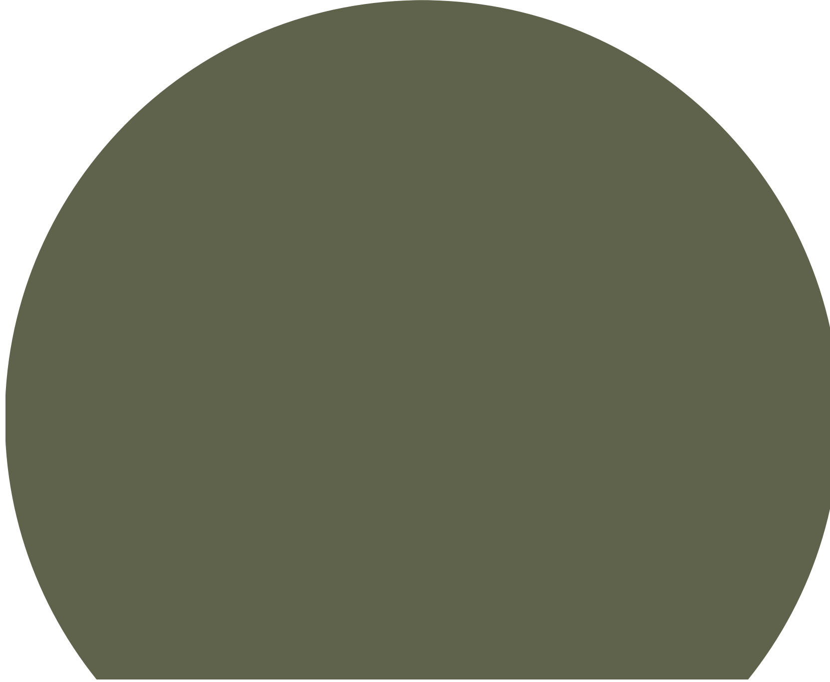
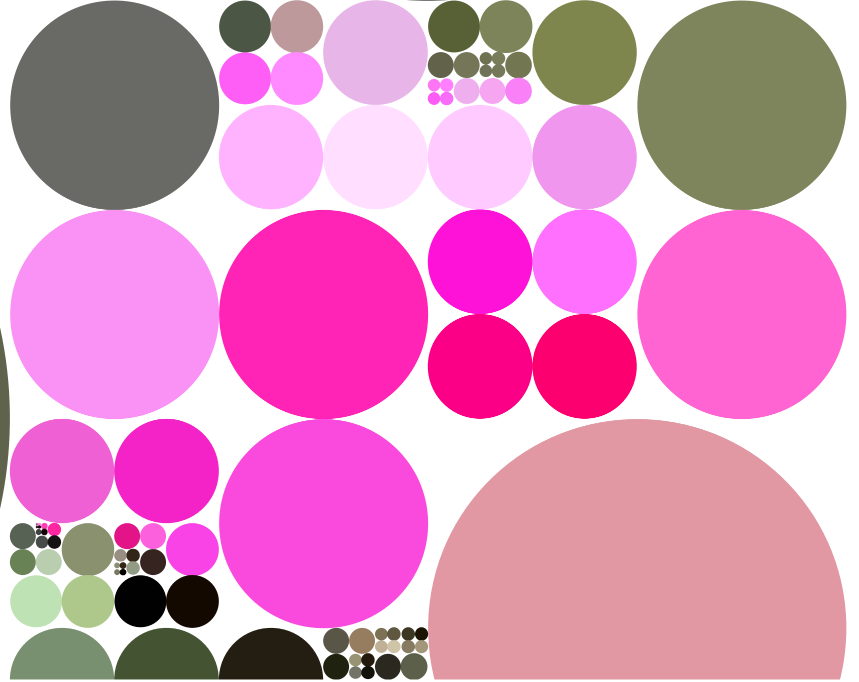
```
svgdata = ""
```

```
2 picCircles("gerberas.jpeg",'gerberas')
```

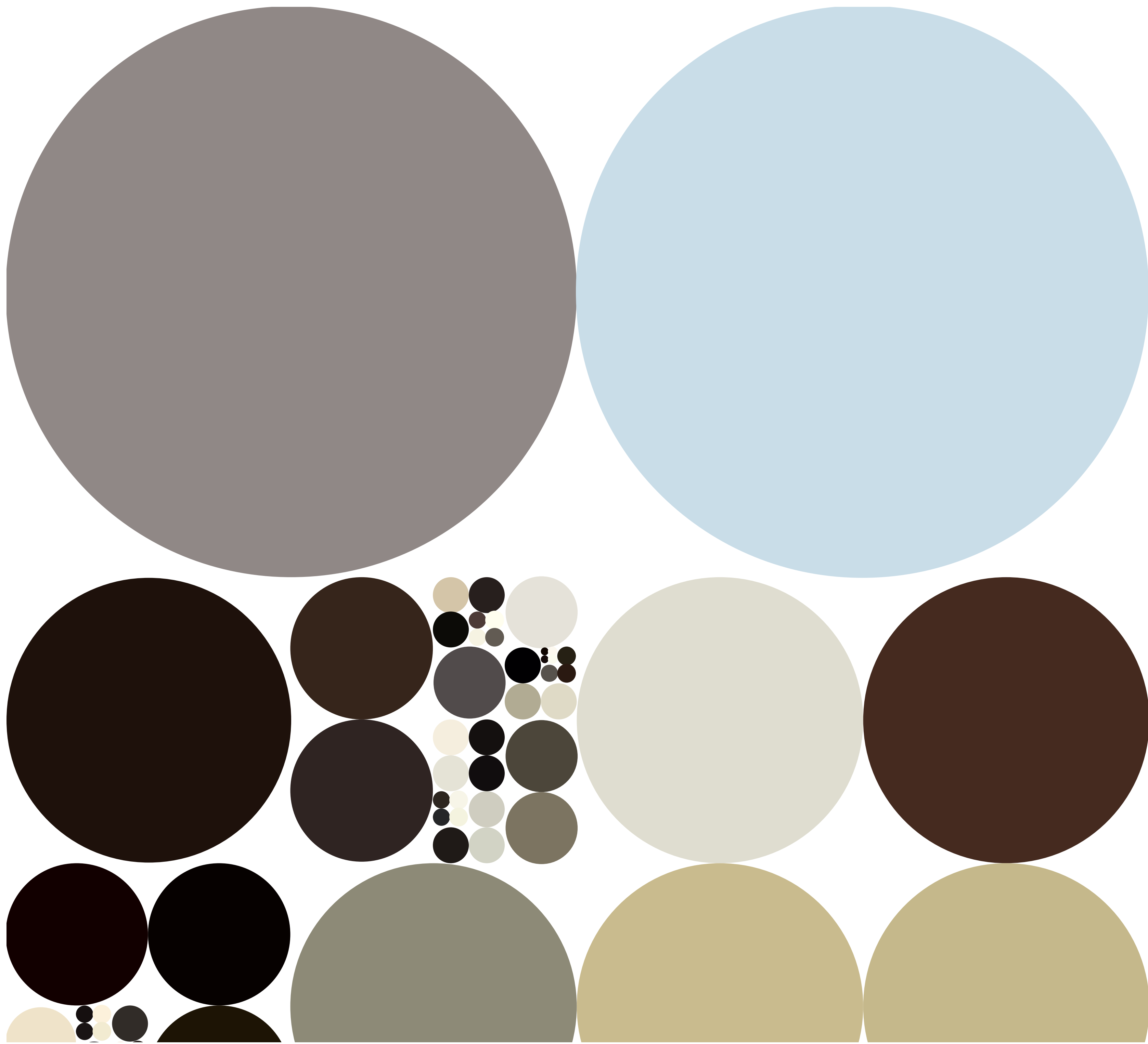
Entrée [6]:

```
1 svgdata = ""
2 picCircles("rose2.jpg",'rose2')
```

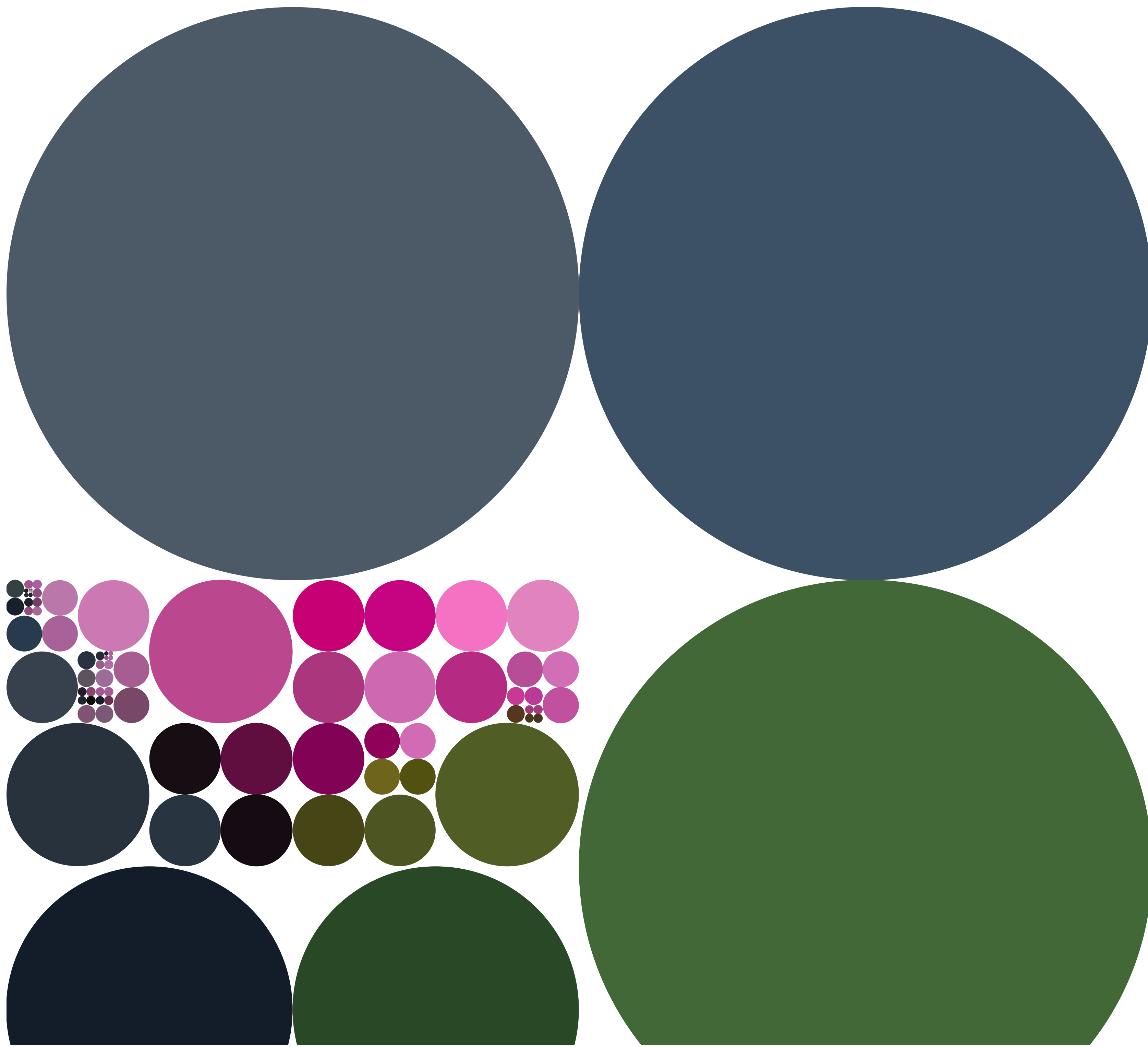
Entrée [16]:

```
1 svgdata = ""
2 picCircles("zebra.jpg",'zebra')
```



Entrée [17]:

```
svgdata = ""
picCircles("lotus.jpg",'lotus')
```



Logistic

Warning: Output take a very long time to appear

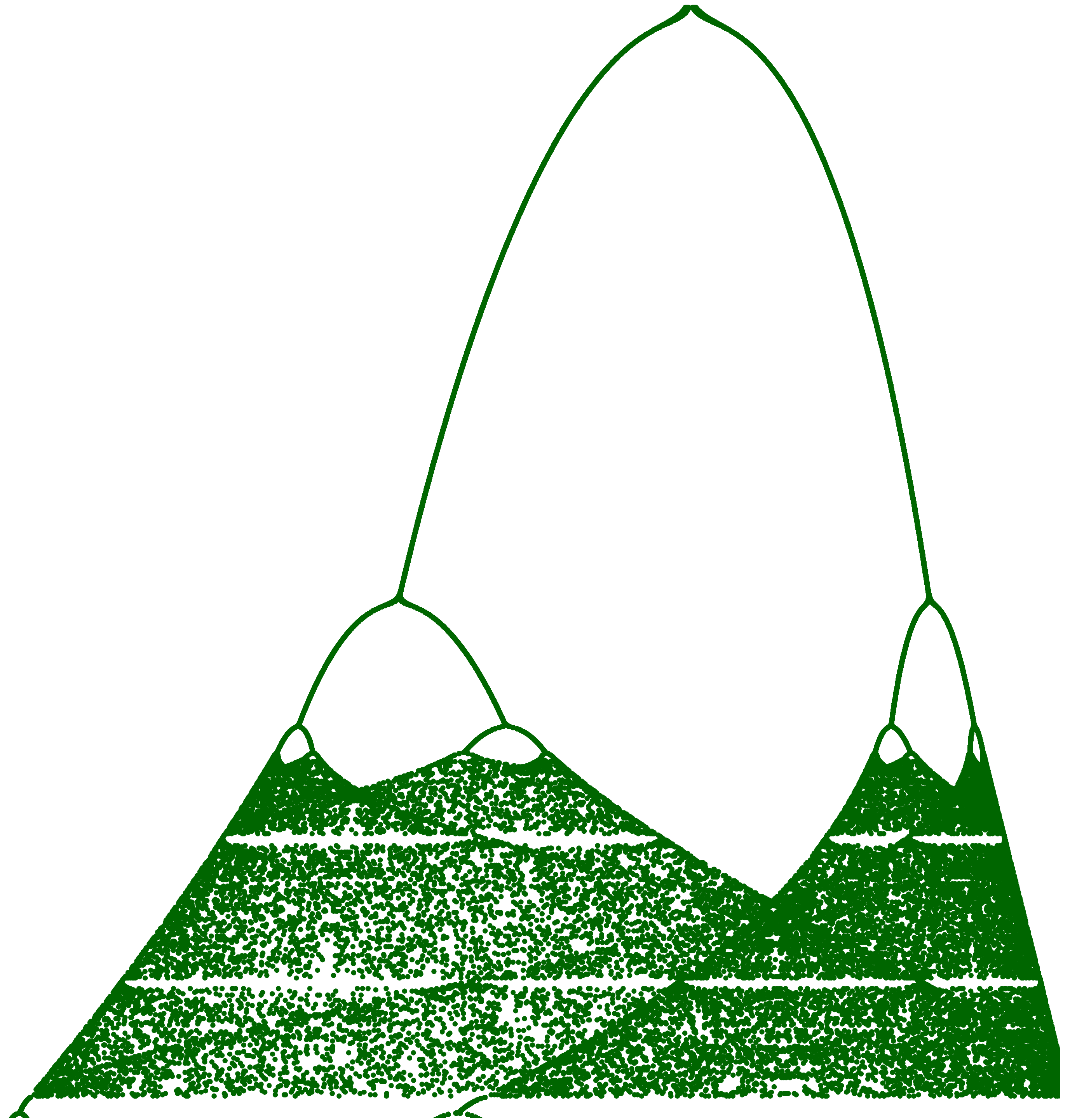
Entrée [18]:

```
import pandas as pd, numpy as np, matplotlib.pyplot as plt

svgdata = ""

def addsvgdata(data):
    global svgdata
    svgdata += data


'''
Full credit to Geoff Boeing whose code I borrowed to form the data frame
https://geoffboeing.com/2015/03/chaos-theory-logistic-map/
'''

class Circle:
    def __init__(self, cx, cy, r, color='black'):
        """Initialize the circle with its centre, (cx,cy) and radius, r.

        icolour is the index of the circle's colour.

        """
        self.cx, self.cy, self.radius = cx, cy, r
        self.color = color


def logistic_model(generations=20,
                   growth_rate_min=0.5,
                   growth_rate_max=4.0,
                   growth_rate_steps=7,
                   pop_initial=0.5):
    """
    returns a pandas dataframe with columns for each growth rate, row labels for each time step
    and population values computed by the logistic model: pop[t + 1] = pop[t] * rate * (1 - pop

    generations = number of iterations to run the model
    growth_rate_min = the first growth rate for the model, between 0 and 4
    growth_rate_max = the last growth rate for the model, between 0 and 4
    growth_rate_steps = how many growth rates between min (inclusive) and max (exclusive) to ru
    pop_initial = starting population when you run the model, between 0 and 1
    """
```

```
41
42      # convert the growth rate min and max values to floats so we can divide them
43      growth_rate_min = float(growth_rate_min)
44      growth_rate_max = float(growth_rate_max)
45
46      # calculate the size of each step
47      growth_rate_step = (growth_rate_max - growth_rate_min) / growth_rate_steps
48
49      # we want to go up to but not including the growth_rate_max
50      growth_rate_max -= 0.0000000001
51
52      # get a range of values to represent each growth rate we're modeling - these will be our c
53      growth_rates = np.arange(growth_rate_min, growth_rate_max, growth_rate_step)
54
55      # create a new dataframe with one column for each growth rate and one row for each timeste
56      pops = pd.DataFrame(columns=growth_rates, index=range(generations))
57      pops.iloc[0] = pop_initial
58
59      # for each column (aka growth rate) in the dataframe
60      for rate in pops.columns:
61
62          # pop is a copy of the pandas series of this column in the dataframe
63          pop = pops[rate]
64
65          # for each timestep in the number of iterations to run
66          for t in range(generations - 1):
67
68              # update the dataframe values by running this timestep of the logistic model
69              pop[t + 1] = pop[t] * rate * (1 - pop[t])
70
71      return pops
72
73  def get_bifurcation_points(pops, discard_gens):
74      """
75      convert a dataframe of values from the logistic model into a set of xy points that
76      you can plot as a bifurcation diagram
77
78      pops = population data output from the model
79      discard_gens = number of rows to discard before keeping points to plot
80      """
81
82      # create a new dataframe to contain our xy points
83      points = pd.DataFrame(columns=['x', 'y'])
84
85      # drop the initial rows of the populations data, if specified by the argument
86      if discard_gens > 0:
87          discard_gens = np.arange(0, discard_gens)
88          pops = pops.drop(labels=pops.index[discard_gens])
```

```
89
90      # for each column in the logistic populations dataframe
91      for rate in pops.columns:
92          # append the growth rate as the x column and all the population values as the y column
93          points = points.append(pd.DataFrame({'x':rate, 'y':pops[rate]}))
94
95      # reset the index and drop the old index before returning the xy point data
96      points = points.reset_index().drop(labels='index', axis=1)
97      return points
98
99  def bifurcation_plot(pops, discard_gens=1, xmin=0, xmax=4, ymin=0, ymax=1, height=6, width=10)
100     """
101     plot the results of the logistic model as a bifurcation diagram
102
103     pops = population data output from the model
104     discard_gens = number of rows to discard before keeping points to plot
105     xmin = minimum value on the x axis
106     xmax = maximum value on the x axis
107     ymin = minimum value on the y axis
108     ymax = maximum value on the y axis
109     height = the height of the figure to plot, in inches
110     width = the width of the figure to plot, in inches
111     """
112
113     # first get the xy points to plot
114     points = get_bifurcation_points(pops, discard_gens)
115     circles = []
116     for row in points.itertuples():
117         circles.append(Circle(row.x, row.y,2))
118
119     return circles
120
121 def drawCircle(cx, cy, radius, color):
122     return f'<circle cx="{cx}" cy="{cy}" r="{radius}" fill="{color}" stroke-width="0"/>'
123
124 def drawStuff(circles, width, mult, color="black"):
125     header = f'<svg viewBox="-50 -50 {width} {width}" xmlns="http://www.w3.org/2000/svg">\n'
126     addsvgdata(header)
127     for c in circles:
128         addsvgdata(drawCircle(c.cy * mult, c.cx * mult - 3 * mult, c.radius, color))
129     footer = f'</svg>'
130     addsvgdata(footer)
131
132
133
134 pops = logistic_model(generations=300, growth_rate_min=2.99, growth_rate_max=4, growth_rate_ste
135 circles = bifurcation_plot(pops, discard_gens=200, xmin=2.8, xmax=4)
136 drawStuff(circles, 1000, 1000, color='rgb(0,102,0)')
```

```
137
138 display(SVG(data=svgdata))
139 # Warning: very very very long wait before display
140 with open("logistic.svg","w") as fo:
```



Entrée [ ]:

```
1
```