riddler-22-09-09

September 12, 2022

1 Riddler Sept 9, 2022

[ ]:
```
import numpy as np
import math
import matplotlib.pyplot as plt
import matplotlib as mpl
```

1.1 Riddler Express

From Mike Strong comes a brief return to last week's battle:

In last week's Battle for Riddler Nation, you had to assign 100 phalanxes of soldiers to 10 castles, each worth a distinct number of points. For example, you could assign all 100 phalanxes to a single castle (and none to the others), split them evenly so that there were 10 phalanxes at every castle or arrange them in some other way.

What was the total possible number of strategies you could have submitted?

1.1.1 Solution

Here is a python script giving a total of 559,488 strategies.

[ ]:
```
%time



class stingray:
    def __init__(self):
        self._ray = [[1] * 101] + [[None] * 101] * 9

    def ray(self, i, j):
        ''' Number of ways of putting j identical things
            in i places.
        '''
        if i < 1 or i > 10 or j < 0 or j > 100:
            raise ValueError("{i,j}, need 1 <= i <= 10 and 0 <= j <= 100.")
        i -= 1
        if self._ray[i][j] is None:
            self._ray[i][j] = 0
            for jj in range(j + 1):
                self._ray[i][j] += self.ray(i, j - jj)
        return self._ray[i][j]


fish = stingray()
print(f"The answer is {fish.ray(10, 100):,d}")
```

```
CPU times: user 1 µs, sys: 0 ns, total: 1 µs
Wall time: 1.91 µs
The answer is 559,488
```

1.2 Riddler classic

On social media, I recently saw an image of cookies, which had presumably been circular when they were placed in the oven, emerging as expanded hexagons.

This made me wonder more about how circles could overlap to fill up a rectangular tray.

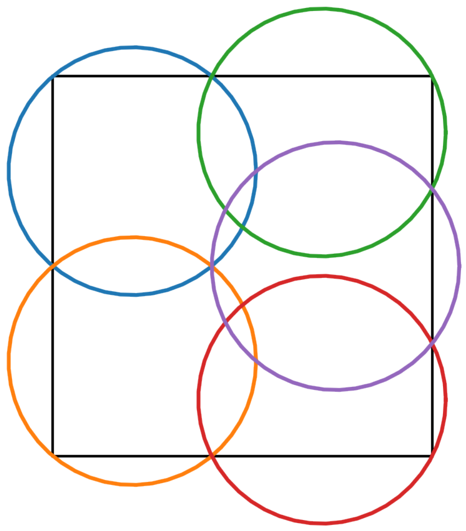
Suppose you have a unit square (i.e., with side length 1). If you also have four identical circles that can overlap, they would need to have a radius of $0.25 \cdot \surd 2$ to completely cover the square, as shown below:

A square that is completely covered by four overlapping circles of the same size. The four circles are centered over the four respective quadrants of the square, and all overlap in the squares center. Now suppose that, instead of four identical circles, you have five identical circles that can overlap. What is the minimum radius they would need to completely cover a unit square?

Extra credit: Suppose you have six identical circles that can overlap. What is the minimum radius they would need to completely cover a unit square?

1.2.1 Solution

Here is a pattern with 5 circles which covers the square with circles with radius of (about) $\sqrt{2}/4 * .924 = 0.326683$:



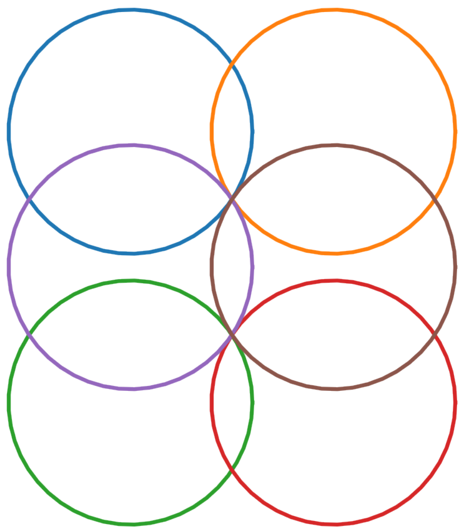
I have't tried to find a closed form for the solution; this estimated radius is one which makes the image look right.

However the pattern is the best possible. First there must be 4 circles, each covering one of the 4 corners of the square. The 5th circle can only intersect one edge of the square, so the other 3 edges must be covered by the first 4 circ ofles. The 5th circle must then pass through the intersections of the top right and bottom right circles with the right edge, and through the intersection of the left two circles.

1.2.2 Solution to extra credit.

Here is a pattern with 6 circles which covers the squares with circles with radius of

$$\sqrt{\frac{1}{6^2}+\frac{1}{4^2}} = \frac{\sqrt{52}}{24} = 0.3004626062886658\ldots\ .$$



In this case the exact value of the radius was easy to find, using the observation that half the height of the square is equal to three times the distance from the top of the square to the center of the top circles.

The proof that this pattern is optimal is (assuming symmetry, which I don't have a proof is necessary) is similar to that for 5 circles, except that this time there is alternative in which the two sides covered by 3 circles are adjacent rather than opposite. However this pattern, shown below doesn't quite cover the square with a radius of 0.304 ..., which is already larger than needed here.

1.3 Code for the solution.

Below is the code making the images. First are classes for drawing the circles and the square.

[ ]:
```
class Circle:
    def __init__(self, radius, x, y, ax=None, **kwds):
        self.radius = radius
        self.x = x
        self.y = y
        self._line = None
        self._ax = ax
        self.kwds = kwds
        if self._ax is not None:
            self.draw(self)

    @classmethod
    def two_point(cls, p1, p2, radius=1,  ax=None, **kwds):
        center = cls._center_from_two_points(p1, p2, radius)
        return cls(radius, *center, ax, **kwds)

    @classmethod
    def _center_from_two_points(cls, p1, p2, radius):
        p1 = np.array(p1)
        p2 = np.array(p2)
        mid = (p1 + p2) / 2
        dir = (p2 - p1) @ np.array([[0, -1], [1, 0]])
        dirlen = np.linalg.norm(dir)
        wanted_len = np.sqrt(radius ** 2 - dirlen ** 2 / 4)
        dir *= wanted_len / dirlen
        center = mid + dir
        return center

    def draw(self, ax=None, **kwds):
        if ax is None:
            ax = self._ax
        self.kwds.update(kwds)
        theta = np.linspace(0, 2 * np.pi)
        draw_x = self.x + self.radius * np.cos(theta)
        draw_y = self.y + self.radius * np.sin(theta)
        self._line = self._ax.plot(draw_x, draw_y, **self.kwds)

    def delete(self):
        self._line[0].remove()
        self._line = None
        self._ax = None

    def move(self, x=None, y=None, ax=None, radius=None):
        if ax is not None:
            self._ax = ax
        if radius is not None:
            self.radius = radius
        if x is not None:
            self.x = x
        if y is not None:
            self.y = y

    def remove(self):
        self._line[0].remove()
        self._line = None
```

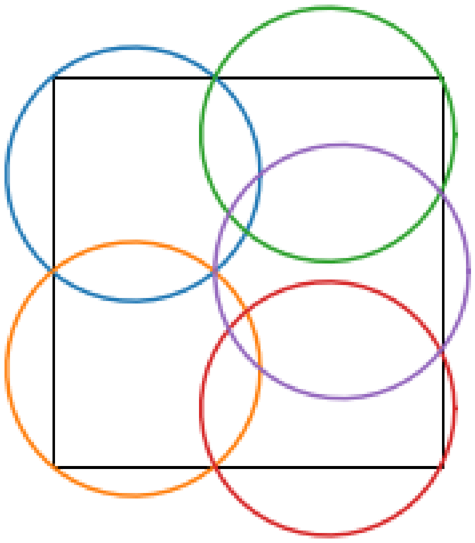
[ ]:
```
from matplotlib.path import Path as m_path
import matplotlib.patches as patches
```

```
class Square:
    def __init__(self, side=1, x=0, y=0, ax=None, **kwds):
        self.side = side
        self.x = x
        self.y = y
        self._line = None
        self.ax = ax
        self.kwds = kwds
        if self.ax is not None:
            self.draw()

    def draw(self, ax=None, x=None, y=None, side=None, **kwds):
        if ax is not None:
            self.ax = ax
        if x is not None:
            self.x = x
        if y is not None:
            self.y = y
        if side is not None:
            self.side = side
        self.kwds.update(kwds)

        verts = np.array(((-1, -1), (-1, 1), (1, 1), (1, -1), (0, 0)))
        verts = self.side / 2 * verts + (self.x, self.y)
        codes = [m_path.MOVETO] + [m_path.LINETO] * 3 + [m_path.CLOSEPOLY]
        path = m_path(verts, codes,)
        patch = patches.PathPatch(path, fill=False)
        self.ax.add_patch(patch)

    def delete(self):
        self._line[0].remove()
        self._line = None
        self._ax = None

    def move(self, x=None, y=None, ax=None, radius=None):
        if ax is not None:
            self.ax = ax
        if radius is not None:
            self.radius = radius
        if x is not None:
            self.x = x
        if y is not None:
            self.y = y

    def remove(self):
        self._line[0].remove()
        self._line = None
```

I then used these to draw the patterns. In each case, I fixed a radius r for the circles, and adjusted it until the pattern worked.

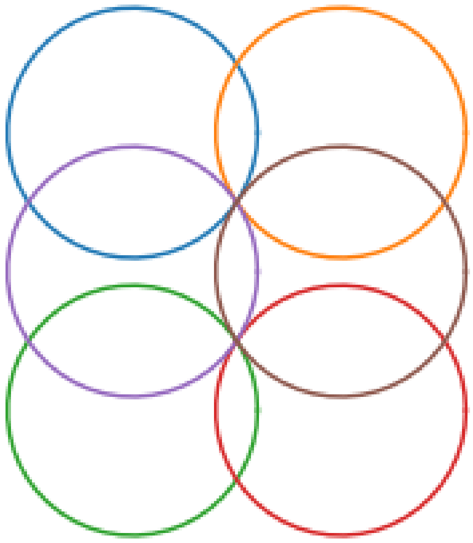
```
[ ]: fig, ax = plt.subplots()
     ax.set_aspect('equal')
     ax.axis(False)
     sqrt2 = np.sqrt(2)

     sq = Square(side=1, x=0, y=0, ax=ax)
     # sq2 = Square(side=1.2, ax=ax)


     r = sqrt2/4 * 0.92252144
     print(f"{r = }")
     # circle1 = Circle(radius=r, x=1/4, y=1/4, ax=ax)
     # circle2 = Circle(radius=r, x=-1/4 - .1, y=1/4, ax=ax)
     # circle3 = Circle(radius=r, x=1/4, y=-1/4, ax=ax)
     # circle4 = Circle(radius=r, x=-1/4 - .1, y=-1/4, ax=ax)
     c2 = Circle.two_point((-.5, 0), (-.5, .5), radius=r, ax=ax, label="c2")
     c4 = Circle.two_point((-.5, -.5), (-.5, 0), radius=r, ax=ax, label="c4")

     x1 = c2.x + (c2.x - (-.5))

     c1 = Circle.two_point((x1, .5), (.5, .5), radius=r, ax=ax, label="c1")
     c3 = Circle.two_point((.5, -.5), (x1, -.5), radius=r, ax=ax, label="c3")


     c5 = Circle.two_point((x1, 0), (2*r + x1, 0), radius=r, ax=ax, label="c5")
     # ax.legend()

     plt.savefig("images/5circles.png", dpi=300,)
```

```
r = 0.3261605830069894
```



[ ]:
```
fig, ax = plt.subplots()
ax.set_aspect('equal')
ax.axis(False)

sqrt2 = np.sqrt(2)

r = np.sqrt((16 + 36) / (16 * 36))
print(f"{r = }")

c11 = Circle.two_point((-.5, .5), (0, .5), radius=r, ax=ax, label="c11")
c12 = Circle.two_point((0, .5), (.5, .5), radius=r, ax=ax, label="c12")

c31 = Circle.two_point((0., -.5), (-.5, -.5), radius=r, ax=ax, label="c31")
c32 = Circle.two_point((.5, -.5), (0, -.5), radius=r, ax=ax, label="c32")

y0 = 2 * c12.y - 0.5

c21 = Circle.two_point((-.5, -y0), (-.5, y0), radius=r, ax=ax, label="c21")
c22 = Circle.two_point((.5, y0), (.5, -y0), radius=r, ax=ax, label="c22")

center22 = np.array((c22.x, c22.y))
center = Circle._center_from_two_points((0., -y0), (0., y0), radius=r)
print(f"{center22 = }, {center = },\ndifference = {center22 - center}.")

# ax.legend()
plt.savefig("images/6circles.png", dpi=300)
```

```
r = 0.3004626062886658
center22 = array([0.25, 0.  ]), center = array([0.25, 0.  ]),
difference = [2.77555756e-17 0.00000000e+00].
```



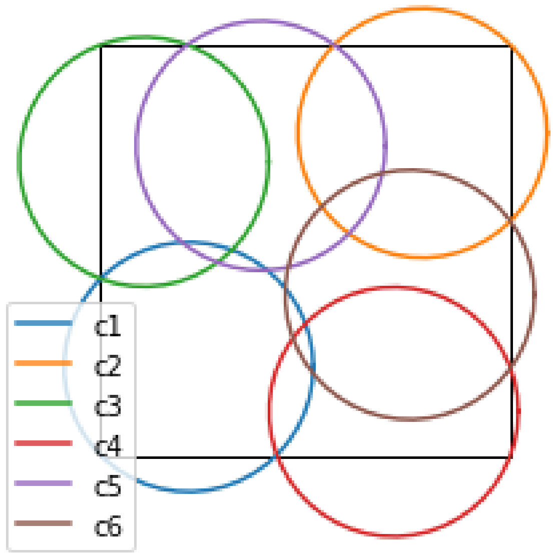
1.3.1 A second possibility for 6 circles

Perhaps the edges of the square covered by 3 circles could be adjacent? It turns out the circles are a bit larger than in the solution above.

```
[ ]: r = np.sqrt(2)/4 * .86
d = .3
fig, ax = plt.subplots()
ax.set_aspect('equal')
ax.axis(False)
sq = Square(side=1, x=0, y=0, ax=ax)

pts = [[-.5, -.5], (-5, np.sqrt(2) * r - 0.5), (-5, 5), ]

c1 = Circle.two_point((-0.5, -0.5), (-0.5, np.sqrt(2) * r - 0.5), radius=r,␣
 ↪ax=ax, label="c1")
c2 = Circle.two_point((0.5, 0.5), (0.5, 0.5 - np.sqrt(2) * r), radius=r, ax=ax,␣
 ↪label="c2")

c3 = Circle.two_point((-0.5, np.sqrt(2)* r - 0.5), (-0.5, 0.5), radius=r,␣
 ↪ax=ax, label="c3")
c4 = Circle.two_point((0.5, -0.5), (np.sqrt(2)* r - 0.5, -0.5),  radius=r,␣
 ↪ax=ax, label="c4")
```

```
c2.x - (.5 - c2.x)

c5 = Circle.two_point((2 * c3.x + 0.5, 0.5), (2 * c2.x - 0.5, 0.5), radius = r,␣
 ↪ax=ax, label="c5")
c6 = Circle.two_point((0.5, 2 * c2.y - 0.5), (0.5, 2 * c4.y + 0.5), radius = r,␣
 ↪ax=ax, label="c6")



ax.legend()
print(f"{r = }")
```

```
r = 0.30405591591021547
```



1.3.2 Calculation of radius for 5 circles

The next blocks calculate the radius for 5 circles to 14 decimal places.

[ ]:
```
from sympy import Point, Circle as sCircle, pi, sqrt, Symbol


def center_from_two_points(p1, p2, radius):
    mid = p1.midpoint(p2)
    dir = (p2 - p1).rotate(-pi / 2).unit
    center = mid + dir * sqrt(radius ** 2 - mid.distance(p2) ** 2)
    return center
```

[ ]:
```
r = Symbol('r')
p1 = center_from_two_points(Point(-.5, 0), Point(-.5, .5), r)
p2 = Point(2 * p1.x + 0.5, .5)
p3 = center_from_two_points(p2, Point(.5, .5), r)
p4 = Point(0.5, 2 * p3.y - 0.5)
c1 = sCircle(p1, r)
c2 = sCircle(Point(p1.x, -p1.y), r)
p5 = c1.intersection(c2)[1]
p6 = center_from_two_points(p5, p4, r)
# f = lambdify(r, p6.y, 'numpy')
print(p6.y.subs(r, np.sqrt(2)/4 * 0.92253).evalf())
p6.y.subs(r, np.sqrt(2)/4 * 0.93).evalf()
```

```
-5.22050044309613e-5
```

[ ]: -0.0404695641602829

[ ]:
```
def eval(rval):
    return p6.y.subs(r, sqrt(2)/4 * rval).evalf()
# eval(.92)
```

[ ]:
```
def solve_binary(rlow, rhi, func, epsilon=1E-14):
    ''' Find r so that func(r) = 0, with error < epsilon '''
    vlow = func(rlow)
    vhi = func(rhi)
    if vlow < 0 and vhi > 0:
        sign = 1
    elif vlow > 0 and vhi < 0:
        sign = -1
    else:
        raise ValueError("{func(rlow) = } and {func(rhi) = } must not have the␣
 ↪same sign.")
    n = 0
    while rhi - rlow > epsilon:
        print(f"{n:4n} rlow = {rlow}, vlow = {vlow},  rhi = {rhi}, vhi =␣
 ↪{vhi}{'':10}\r", end="")
        assert (vhi * sign > 0) and (vlow * sign < 0), print(f"{vhi =}, {vlow=␣
 ↪}")
        n += 1
        # rnew = rlow + (rhi - rlow) / (vlow - vhi) * vlow
        rnew = (rlow + rhi) / 2
        vnew = eval(rnew)

        if vnew * sign < 0:
            rlow, vlow = rnew, vnew
        else:
```

```python
            rhi, vhi = rnew, vnew
    return  (rlow + rhi) / 2
solution = solve_binary(0.92, 0.93, eval, )
print(f"\n\nr = {solution:.13f} gives an error of {eval(solution)}")
print(f"radius = sqrt(2)/4 * 4 = {np.sqrt(2) / 4 * r}")
```

```
  39 rlow = 0.9225214428199253, vlow = 6.48844256492004E-14,  rhi =
0.9225214428199435, vhi = -4.64762582527653E-14

r = 0.9225214428199 gives an error of -1.86409366364384E-14
radius = sqrt(2)/4 * 4 = 0.353553390593274*r
```