FiveThirtyEight's May 8, 2020 Riddler

Emma Knight

May 13, 2020

This is my (highly incomplete) solution to the riddler from May 8, 2020, which proposes a good way to keep your toddler distracted for half of a day:

Question 1. *A certain 2-year-old is eating his favorite snack: an apple. But he eats it in a very particular way. When he first receives the apple, and every minute thereafter, he rotates the apple to a random position and then looks down. If there's any skin of the apple left in the spot where he plans to take a bite, then he will indeed take that bite. But if there's no skin there (i.e., he's already taken bites at that spot), he won't take a bite and will rotate the apple for another minute. Once he has bitten off all the skin of the apple, he's done eating.*

Suppose the apple is a sphere with a radius of 4 centimeters, and that each bite of the apple is a circle of the sphere whose radius, as measured along the apple's curved surface, is 1 centimeter. On average, how many minutes will it take this 2-year-old to eat the apple?

This write-up will be divided into two parts: in the theory section, I will explain how to get upper and lower bounds on the answer. In the computation section, I will explain how I did some computations to get a numerical answer, and what I expect the true answer to be (roughly).

1 Theory

The goal of this part is to prove an upper and lower bound. However, I will actually talk about the version where the radius is is varying (and thinking in particular about the case when the radius goes to 0). Let r denote this radius, and x be the area of the big sphere divided by the area of the corresponding circle (so x is approximately r^{-2}). Then, one has the following:

Theorem 2. *The average amount of time the toddler takes to eat the apple is* $\Theta(x \ln(x))$.

I will need the following two claims:

Claim 3. *There exists a set of points* S_x *on the sphere such that*

- $|S_x| > Ax$ *for some constant* $A > 0$ *and for all* x *sufficiently large, and*

- *For any circle C of radius r, $|C \cap S_x| \leq 1$*

Claim 4. *There exists a partition P_x of the sphere such that*

- *$|P_x| < Bx$ for some constant $B > 0$ and for all x sufficiently large, and*
- *Every circle strictly contains one element of P_x.*

Proof of Theorem 2. The idea here is to use the coupon collector problem. This problem asks, if there are N types coupons and you start taking coupons one at a time, chosen uniformly at random among the types of coupons, how long on average does it take to collect all of the different types of coupons? This is a classical problem, and the answer is well known to be $NH_N \approx N\ln(N)$ where H_N is the N^{th} harmonic number.

Now, the claims come into focus. Since a collection of circles cannot cover the whole of the sphere without covering S_x, and each circle can only cover at most one point in S_x, then it takes at least $|S_x|H_{|S_x|}$ time on average. Since $|S_x| > Ax$, one gets that the average is at least $Ax\ln(Ax) > A'x\ln(x)$ for some slightly smaller constant A'.

Similarly, if each element of P_x is covered, then all of the sphere is. Each circle guarantees that one of the elements of P_x will be covered, and there are at most Bx elements of P_x, one gets an upper bound of $B'x\ln(x)$ for the same reason as before.

□

The proofs of the claims come down to the following point. Take the following standard parametrization of the sphere: $F(\theta, \phi) = (4\cos(\theta)\sin(\phi), 4\sin(\theta)\sin(\phi), 4\cos(\phi))$. Then there exists constants C_1, C_2 such that

- For all $(\theta_1, \phi_1), (\theta_2, \phi_2) \in [0, 2\pi] \times [0, \pi]$, the distance between $F(\theta_1, \phi_1)$ and $F(\theta_2, \phi_2)$ on the sphere is at most C_1 times the distance between (θ_1, ϕ_1) and (θ_2, ϕ_2) in the θ, ϕ-plane.
- For all $(\theta_1, \phi_1), (\theta_2, \phi_2) \in [0, 2\pi] \times [.001, \pi - .001]$, the distance between $F(\theta_1, \phi_1)$ and $F(\theta_2, \phi_2)$ on the sphere is at least C_2 times the distance between (θ_1, ϕ_1) and (θ_2, ϕ_2) in the θ, ϕ-plane.

Now, one proves the claims by choosing the correct mesh/partition in the θ, ϕ-plane scaled by the correct constant.

2 Computation

The idea that I used to do computations was to create a mesh on the sphere, and remove points from the mesh as the toddler ate those points. Here is the code that I ran to do a bunch of simulations (written in python):

```
import random
import math
import numpy as np
import matplotlib.pyplot as plt

# This generates a random point on the sphere of radius 4.
# More specifically, this generates a random point of distance
# between 1 and 4 from the origin, and then projects it onto the
# sphere of radius 4.
def generate_point():
    while True:
        x = random.uniform(-4, 4)
        y = random.uniform(-4, 4)
        z = random.uniform(-4, 4)
        r = math.sqrt(x**2 + y**2 + z**2)
        if ((1 < r) and (r < 4)):
            return [(4*x)/r, (4*y)/r, (4*z)/r]

# This generates a mesh of points on the sphere of radius 4.
# More speciffically, this generates every point on the unit sphere
# that has two of the coordinates rational, and denominator dividing
# gap, and then scales them up to the sphere of raidus 4.
def generate_mesh(gap):
    mesh = []
    for x in range((-1)*gap, gap+1):
        b = math.floor(math.sqrt((gap**2 - x**2)))
        for y in range(-b, b+1):
            z = math.sqrt((gap**2 - x**2 - y**2))
            mesh.append([4*x/gap, 4*y/gap, 4*z/gap])
            mesh.append([4*x/gap, 4*y/gap, (-4)*z/gap])
            mesh.append([4*x/gap, 4*z/gap, 4*y/gap])
            mesh.append([4*x/gap, (-4)*z/gap, 4*y/gap])
            mesh.append([4*z/gap, 4*x/gap, 4*y/gap])
            mesh.append([(-4)*z/gap, 4*x/gap, 4*y/gap])
    return mesh

# This is the square of the distance in R^3 betwee two points on the
# sphere of radius 4 that are 1 unit apart on the sphere.
dist = 32*(1-math.cos(1/4))

# This process simulates the toddler eating an apple.  It takes a mesh
# and starts to generate random points on the sphere.  For each point it
# generates, it then removes every point in the mesh that is at most
# sqrt(dist) away from it.  It then tells you how many points needed to
# be generated to destroy the whole mesh.  k is a dummy variable that is
# only being passed in to make sure that the code is still running.  The
# print step is unnecessary but good to verify that the code is still
# running.
```

```
def simulate(mesh, k):
    count = 0
    while (len(mesh)>0):
        count += 1
        p = generate_point()
        for i in range(len(mesh)-1, -1, -1):
            d = (p[0]-mesh[i][0])**2 + (p[1]-mesh[i][1])**2 + (p[2]-mesh[i][2])**2
            if (d <= dist):
                del(mesh[i])
        if (count % 20 == 0):
            print([k, len(mesh), count])
    return count

# These are the main variables that need to be passed in to the main loop.
# m is the mesh that I will use.  samples is the total number of samples I
# will take.  total is the cumulative sum of all of the results I have seen.
m = generate_mesh(100)
print(len(m))
samples = 10000
total = 0

# These are variables that are used in the graphics part.  results is the list
# of results from the simulations.  maxi/mini is the maximum/minimum of the
# results.  clump is the size of the clumps that I group results into (e.g.
# if clump is 20, then any result in the interval [500, 520) is counted as the
# same for displaying things).
results = []
maxi = 0
mini = 10000
clump = 20

# This is the main loop.  I run simulate samples times, and record the results.
for i in range(samples):
    result = simulate(m.copy(), i)
    print(result)
    total += result
    results.append(result//clump)
    maxi = max(result//clump, maxi)
    mini = min(result//clump, mini)

#This is the code for displaying everything.
displayset = []
for k in range(mini, maxi+1):
    results.append(k)
    displayset.append((results.count(k)-1)/samples)

plt.plot(range(clump*mini, clump*maxi+clump, clump), displayset, 'b-')
plt.show()
```

```
print(total/samples)
```

And here is code that I used to produce images:

```
import random
import math
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# This generates a random point on the sphere of radius 4.
# More specifically, this generates a random point of distance
# between 1 and 4 from the origin, and then projects it onto the
# sphere of radius 4.
def generate_point():
    while True:
        x = random.uniform(-4, 4)
        y = random.uniform(-4, 4)
        z = random.uniform(-4, 4)
        r = math.sqrt(x**2 + y**2 + z**2)
        if ((1 < r) and (r < 4)):
            return [(4*x)/r, (4*y)/r, (4*z)/r]

# This generates a mesh of points on the sphere of radius 4.
# More speciffically, this generates every point on the unit sphere
# that has two of the coordinates rational, and denominator dividing
# gap, and then scales them up to the sphere of raidus 4.
def generate_mesh(gap):
    mesh = []
    for x in range((-1)*gap, gap+1):
        b = math.floor(math.sqrt((gap**2 - x**2)))
        for y in range(-b, b+1):
            z = math.sqrt((gap**2 - x**2 - y**2))
            mesh.append([4*x/gap, 4*y/gap, 4*z/gap])
            mesh.append([4*x/gap, 4*y/gap, (-4)*z/gap])
            mesh.append([4*x/gap, 4*z/gap, 4*y/gap])
            mesh.append([4*x/gap, (-4)*z/gap, 4*y/gap])
            mesh.append([4*z/gap, 4*x/gap, 4*y/gap])
            mesh.append([(-4)*z/gap, 4*x/gap, 4*y/gap])
    return mesh

# This is the square of the distance in R^3 betwee two points on the
# sphere of radius 4 that are 1 unit apart on the sphere.
dist = 32*(1-math.cos(1/4))

# This process simulates the toddler eating an apple.  It takes a mesh
```

```
# and starts to generate random points on the sphere.  For each point it
# generates, it then removes every point in the mesh that is at most
# sqrt(dist) away from it.  It then outputs a list of the points that
# were needed to eat the apple.
def simulate(mesh):
    count = 0
    points = []
    while (len(mesh)>0):
        count += 1
        p = generate_point()
        points.append(p)
        for i in range(len(mesh)-1, -1, -1):
            d = (p[0]-mesh[i][0])**2 + (p[1]-mesh[i][1])**2 + (p[2]-mesh[i][2])**2
            if (d <= dist):
                del(mesh[i])
        if (count % 20 == 0):
            print([len(mesh), count])
    print(count)
    return points

# The next few processes are for drawing the circles needed to display the
# apple being eaten.  Ultimately, the final process takes in a point on the
# sphere of radius 4 and outputs three arrays needed for pyplot to show the
# the corresponding circle.
def newvect1(point):
    x = point[0]
    y = point[1]
    z = point[2]
    r = math.sqrt(x**2 + y**2)
    if (r > 0):
        z1 = z*math.cos(1/4) - r*math.sin(1/4)
        r1 = r*math.cos(1/4) + z*math.sin(1/4)
        x1 = (r1*x)/r
        y1 = (r1*y)/r
        return [x1-x, y1-y, z1-z]
    else:
        return [4*math.sin(1/4), 0.0, 0.0]

def newvect2(points):
    x0 = points[0][0]
    y0 = points[0][1]
    z0 = points[0][2]
    x1 = points[1][0]
    y1 = points[1][1]
    z1 = points[1][2]
    x2 = (y0*z1)-(y1*z0)
    y2 = (z0*x1)-(z1*x0)
    z2 = (x0*y1)-(x1*y0)
```

```
    r1 = math.sqrt(x1**2 + y1**2 + z1**2)
    r2 = math.sqrt(x2**2 + y2**2 + z2**2)
    return [(r1*x2)/r2, (r1*y2)/r2, (r1*z2)/r2]

def circle(point):
    p2 = newvect1(point)
    p3 = newvect2([point, p2])
    x1 = point[0]*math.cos(1/4)
    y1 = point[1]*math.cos(1/4)
    z1 = point[2]*math.cos(1/4)
    x2 = p2[0]
    y2 = p2[1]
    z2 = p2[2]
    x3 = p3[0]
    y3 = p3[1]
    z3 = p3[2]
    xvals = []
    yvals = []
    zvals = []
    for i in range(201):
        xvals.append(x1 + x2*math.cos((np.pi*i)/100) + x3*math.sin((np.pi*i)/100))
        yvals.append(y1 + y2*math.cos((np.pi*i)/100) + y3*math.sin((np.pi*i)/100))
        zvals.append(z1 + z2*math.cos((np.pi*i)/100) + z3*math.sin((np.pi*i)/100))
    return [xvals, yvals, zvals]

# This is the main drawing part.  The extra points in the figure are to keep
# the dimensions of the figure constant across all of the images.
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
pointset = simulate(generate_mesh(50))

xs = [5, -5]
ys = [5, -5]
zs = [5, -5]

# This loop puts all the points into the scatter plot.  Additionally, this
# creates a figure for each point in the pointset.  This figure has all the
# points up to the one that just got added in, as well as the circle around
# that point.  This figure isn't displayed but rather saved to the file
# slideshow(n).png.
for i in range(len(pointset)):
    ax = fig.add_subplot(111, projection='3d')
    xs.append(pointset[i][0])
    ys.append(pointset[i][1])
    zs.append(pointset[i][2])
    circ = circle(pointset[i])
    ax.plot(circ[0], circ[1], circ[2])
    ax.scatter(xs, ys, zs)
```

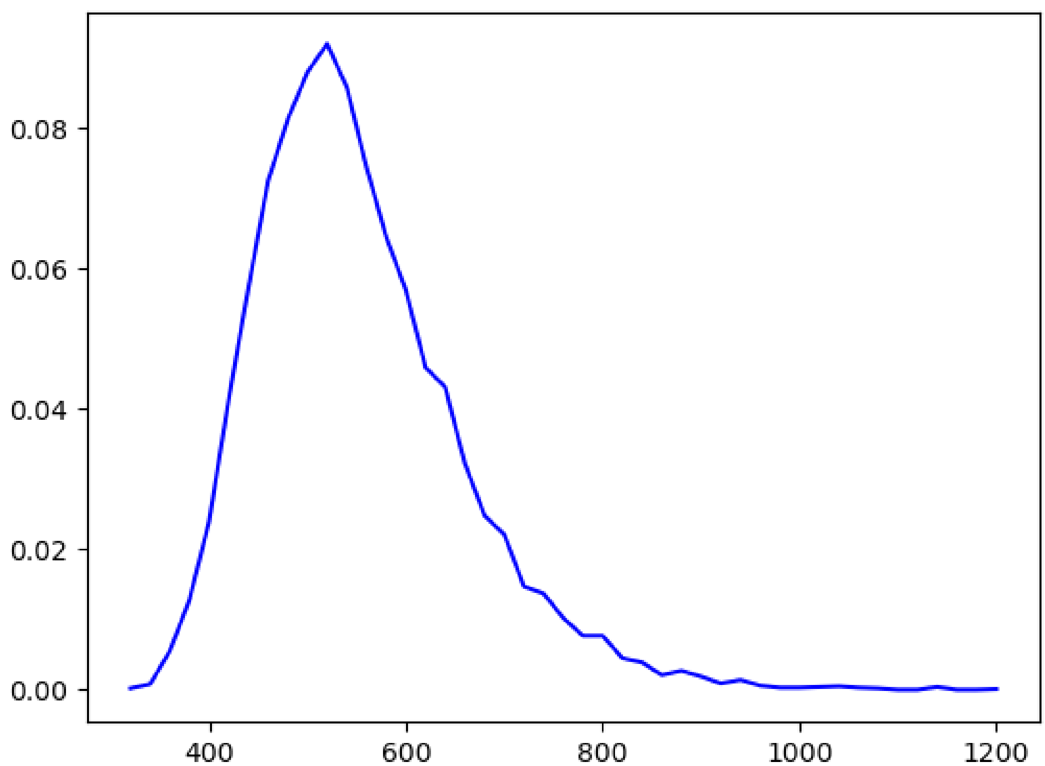
```
    s = 'slideshow' + str(i) + '.png'
    plt.savefig(s)
    plt.clf()

# Finally, we show the final scatterplot.
ax = fig.add_subplot(111, projection='3d')

ax.scatter(xs, ys, zs)

plt.show()
```

Simulating this 10000 times produced an average of 561.0749 minutes, and the following histogram:



In this histogram, I gathered together times in 20 minute intervals, determined what fraction of simulations took that much time. For example, roughly 9% of all simulations took between 580 and 59 minutes.

I suspect that this is an underestimate, however. Firstly, the mesh approach may lead to the simulation thinking everything is finished while it actually isn't. Secondly, there is a large tail to this distribution on the side of more time, but basically no tail on the other side. Thus, simulations may not catch the tail events enough to show the true average and instead underestimate it because of the bias of the tail.