CS 109 Challenge:

Author: Logan Mondal Bhamidipaty
Date: November 30, 2021

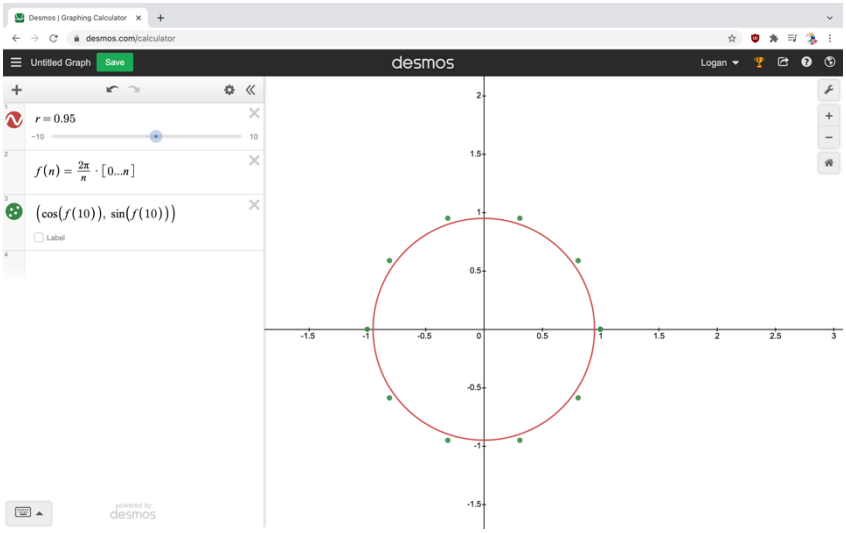
Introduction:

For my personal challenge, I decided to implement a toy problem that my TA Ian Tullis introduced me to in section. The problem, slightly-modified for rhetorical pizzazz, is as follows:

> You stare down at an infinitely large table and are given an infinite number of coins of uniform size, but before you celebrate becoming the wealthiest person on the planet you are presented a task. In order to keep all the coins, you must solve this puzzle. Given a random configuration of 10 points chosen at random over the entire table, find a way to cover all ten dots in such a way that no coin overlaps with any other. If you can, then you get to keep all the money.

I have posed this question to many of my friends (being acquainted with me means ~~enduring~~ enjoying spontaneous pop math questions). A common response was momentary puzzlement followed by a tentative answer: “It’s always possible isn’t it.”

Indeed, such was my initial reaction too, but I was dubious. While this is certainly true in most cases, I wondered about the edge cases: What happens if there is an extremely adversarial positioning of dots—say each of the 10 dots is placed slightly outside the area of one coin. How would you find a tiling that covered all the dots without any coin overlapping? Suddenly, the question becomes more intimidating.



A quick mock-up I made on Desmos to visualize a more challenging scattering

Perhaps surprisingly, our initial intuition was correct! Participants in the toy problem game are guaranteed infinite wealth, because it is always possible to cover 10 dots; though shockingly, choosing 10 dots is the boundary of this guarantee. Any scattering of more than 10 points may have no valid tilings even with infinite coins.

To understand this, please read on!

Math:

It turns out that even in this extremely adversarial dot positioning, there is a valid tiling. The trick to proving this lies in an elegant and clever merging of geometric thinking and probability, but before we present the nitty-gritty mathematics, it's important to first introduce a key result from a field of discrete math known as circle packing.

Circle packing is the study of arranging circles. In this case, our question is this: what tiling of uniformly-sized circles maximizes surface coverage without intersecting. If you'd like to learn more: there's an informative Wikipedia page on the subject with some pretty pictures.

The important result from circle packing is this: the tiling that covers the most surface area with uniformly-sized circles is a hexagonal packing.

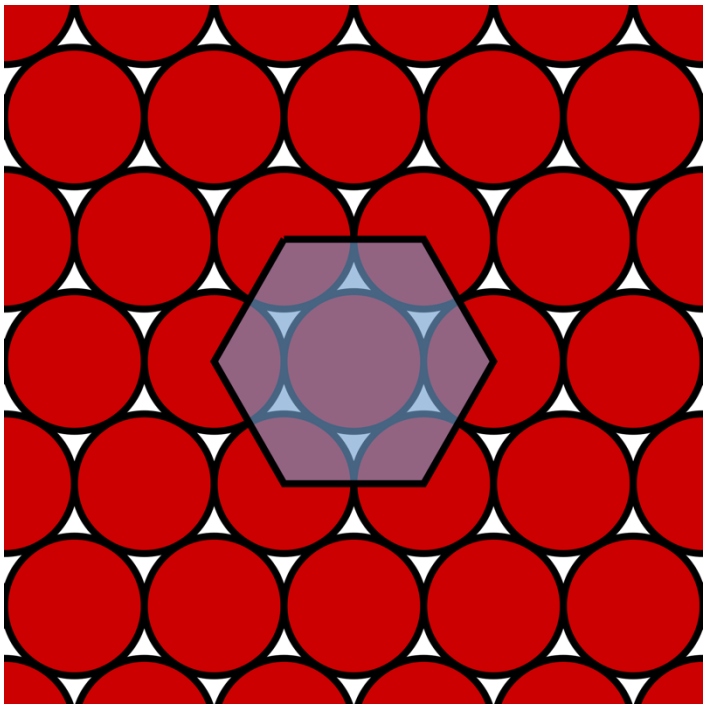


Image from Wikipedia page on Circle Packing

Using the hexagonal circle packing method, the maximum surface area coverage we can obtain is around 90.6 percent. With this result and our handy-dandy understanding of expectation and indicator variables, we have acquired all the tool we need to solve our toy problem completely.

Without further-ado, the proof:

Let $I_1, \ldots, I_{10}$ be indicator random variables for whether each dot is covered by a coin. The expectation that all dots are covered is $E[I_1 + \ldots + I_{10}]$.

We know that using a hexagonal packing of coins, we can achieve around 90.6 percent surface area coverage of the table. Thus, the expectation that each dot is covered is .906; mathematically, $E[I_i] = .906$ where $1 \leq i \leq 10$.

Using the linearity of expectation, we can rewrite our original sum:

$$E[I_1 + \ldots + I_{10}] = E[I_1] + \ldots + E[I_{10}] = .906 + \ldots + .906 = 9.06$$

We unpack our result using the formal definition of expectation:

$$E[C_{10}] = \sum_{i=1}^{10} i \cdot P(C_i) = 9.06.$$

where C_i is the event that i dots are covered by some hexagonal packing. This means that there must be some nonzero probability of covering all ten dots. After all, even if we considered the absurd and impossible case where $P(C_9) = 1$ and $P(C_1)$ through $P(C_9)$ all 0, our expectation of C_{10} would still have an upper bound at 9. Thus, there must be some probability of covering all 10 dots that bumps our expectation beyond 9. Therefore, no matter how adversarial our positioning, there must be a tiling that covers all 10 dots.

A surprising result from this proof is that the number 10 is critically important. While finding a tiling that covers all up to 10 dots is always possible, choosing 11 dots means that some positionings are impossible cover entirely.

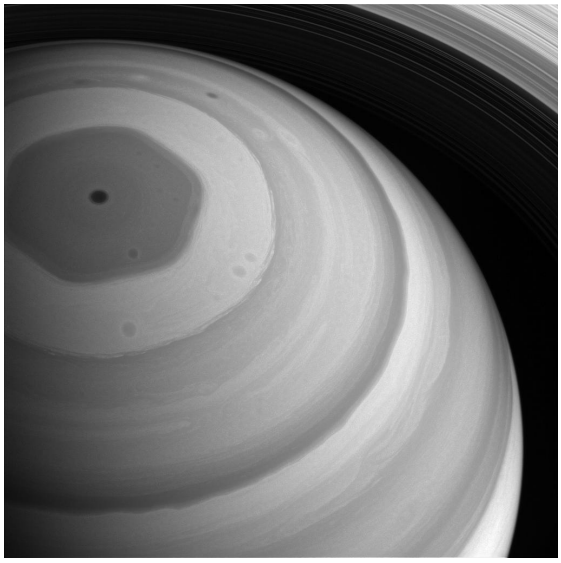
Acknowledgements and Applications:

While the problem itself is toy problem that has limited if any practical application, I did learn something about the field of circle packing itself.

First, hexagons and hexagonal lattices are extremely important. I will be the first to admit that I did not discover their importance through my own research entirely. Rather, these tidbits surfaced through my tendency to share what makes me happy with my friends and the people around me. The pure excitement that I get from tussling with math is invigorating. This joy existed in me before CS 109, but it was fostered by an amazing teaching team and pedagogy. For that, I would like to offer a special acknowledgement to Ian Tullis for his stellar mentorship and guidance. My wonderful experience in this course propelled me to dive deeply into this simple problem and talk about it with others.

With acknowledgements aside, I will proceed to describe the importance of hexagons. One hint at their importance comes from their frequency in nature: from honeycombs to tightly-packed

soap bubbles to a storm on a Saturn, they're just about everywhere you look![1] In chemistry and chemical engineering too, results from circle packing are used to understand molecular geometry and synthesize new drugs! While it may be a stretch to say, it would not be untrue to declare that the fight against diseases is literally built upon a foundation of hexagons.



Saturn's hexagon

The ubiquity of hexagons is truly remarkable, so it hopefully suffices as motivation for this simulation.

Code Explanation:

To implement my code, I went through a surprisingly long voyage that I will highlight now. I set several key milestones for myself throughout the development process:

1) Simulate the dots
2) Simulate the coins
3) Generate hexagonally packed coin tilings
4) Visualize the dots and coins together
5) Run largescale simulations

[1] For more interesting instance of hexagons in nature, see this article.

1: Simulate the dots

To simulate the points, I used NumPy's default random number generator (i.e. numpy.random.default_rng) to uniformly generate independent x and y coordinates within a specified boundary.

I encapsulated the process in a class and added some helper methods that aided with subsequent calculations later in the simulation.

2: Simulate the coins

I represented each coin as an instance of my Coin class. Each instance had a Point instance for its center and floating-point radius attribute. A point was covered if the distance from the center of the coin to the point was less than or equal to the coin's radius.

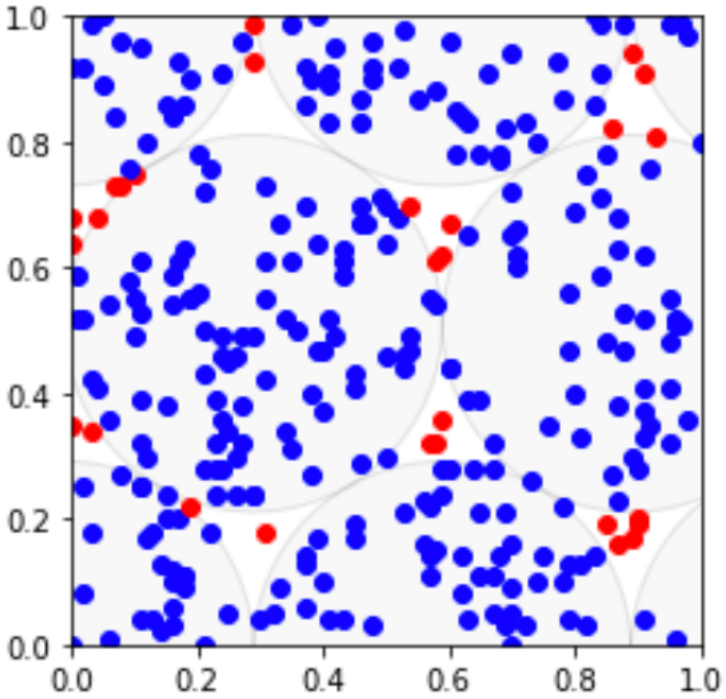
3: Generate hexagonally packed coin tilings

Generating the hexagons was the most difficult step, and it required the most math. In the end, the optimally efficient process I found for generating hexagonal tilings with the relevant information for hexagonal circle packing was this: first, use NumPy's mgrid function to generate coordinate pairs, then selectively shift and squash the coordinates so that each coordinate represents the center of a hexagonally-packed circle.

I arrived at this method of generating hexagonally-packed circle centers after considering other methods (e.g. transformation matrices, using polar coordinates, the roots of unity, and even Lie theory) I eventually discarded those ideas, because they were computationally expensive and difficult to implement and inefficient.

4: Visualize the dots and coins together

To visualize the dots and coins together I used Matplotlib. A sample result is displayed below and showcased in my video submission.



coin_radius = 0.03
num_points = 300
step_size = 0.01[2]

5: Run largescale simulations

To calculate the probabilities discussed below, I used for loops, time, and logging.

Results

I simulated over the square boundary 0 <= x <= 1 and 0 <= y <= 1. I used various coin radiuses and tested whether a tiling could be generated on various numbers of points.

The most salient results are these:

1) The more points you scatter, the more unlikely you will be to find a valid tiling.
2) Larger coin work better (in a limited area).
3) It is surprisingly easy to cover more than 10 points.

Result 1 follows from the math and is explained above, so I won't detail it here.

[2] The step_size parameter controls the discretization of the x_offset and y_offset. In simpler terms, this parameter answers the following question: if our current tiling of coins does not cover all points, how much should we shift each coin center in our next test?

Result 2 is more interesting. While a larger coin clearly works better within a limited boundary (if the coin covers the entire table, then all points on it will be covered with but a single coin), it would be irrelevant if we were to consider larger boundaries (as in the original problem statement).

Result 3 surprised me. It turns out that it is actually very easy to cover many small points. Take for instance, this snippet from some log output from a simulation I ran:

```
coin_radius=0.1
   num_points=10
      stats.fmean(loops_list)=6.331
      stats.fmean(valid_list)=1.0
   num_points=20
      stats.fmean(loops_list)=18.229
      stats.fmean(valid_list)=0.999
   num_points=30
      stats.fmean(loops_list)=42.336
      stats.fmean(valid_list)=0.888
   num_points=40
      stats.fmean(loops_list)=66.071
      stats.fmean(valid_list)=0.604
   num_points=50
      stats.fmean(loops_list)=85.567
      stats.fmean(valid_list)=0.274
   num_points=100
      stats.fmean(loops_list)=99.858
      stats.fmean(valid_list)=0.002
```

coin_radius and num_points are self-explanatory.
stats.fmean(loops_list) gives us the average number of shifts it took for us to find a valid covering of the random points.
stats.fmean(valid_list) tells us the probability that a valid tiling was found.

Conclusion

While I did not have as much time as I would've liked to analyze and extend my simulation, I am appreciative of the opportunity to share my project with you. Thank you for an amazing experience and all the best. –Logan