5.8 PROJECTS

Project 5.1 The magic of polling

> According to the latest poll, the president's job approval rating is at 45%, with a margin of error of ±3%, based on interviews with approximately 1,500 adults over the weekend.

We see news headlines like this all the time. But how can a poll of 1,500 randomly chosen people claim to represent the opinions of millions in the general population? How can the pollsters be so certain of the margin of error? In this project, we will investigate how well random sampling can really estimate the characteristics of a larger population. We will assume that we know the true percentage of the overall population with some characteristic or belief, and then investigate how accurate a much smaller poll is likely to get.

Suppose we know that 30% of the national population agrees with the statement, "Animals should be afforded the same rights as human beings." Intuitively, this means that, if we randomly sample ten individuals from this population, we should, on average, find that three of them agree with the statement and seven do not. But does it follow that every poll of ten randomly chosen people will mirror the percentage of the larger population? Unlike a Monte Carlo simulation, a poll is taken just once (or maybe twice) at any particular point in time. To have confidence in the poll results, we need some assurance that the results would not be drastically different if the poll had queried a different group of randomly chosen individuals. For example, suppose you polled ten people and found that two agreed with the statement, then polled ten more people and found that seven agreed, and then polled ten more people and found that all ten agreed. What would you conclude? There is too much variation for this poll to be credible. But what if we polled more than ten people? Does the variation, and hence the trustworthiness, improve?

In this project, you will write a program to investigate questions such as these and determine empirically how large a sample needs to be to reliably represent the sentiments of a large population.

1. Simulate a poll

In conducting this poll, the pollster asks each randomly selected individual whether he or she agrees with the statement. We know that 30% of the population does, so there is a 30% chance that each individual answers "yes." To simulate this polling process, we can iterate over the number of individuals being polled and count them as a "yes" with probability 0.3. The final count at the end of the loop, divided by the number of polled individuals, gives us the poll result. Implement this simulation by writing a function

```
poll(percentage, pollSize)
```

that simulates the polling of `pollSize` individuals from a large population in which the given `percentage` (between 0 and 100) will respond "yes." The function should return the percentage (between 0 and 100) of the poll that actually responded "yes."

Remember that the result will be different every time the function is called. Test your function with a variety of poll sizes.

2. Find the polling extremes

To investigate how much variation there can be in a poll of a particular size, write a function

```
pollExtremes(percentage, pollSize, trials)
```

that builds a list of `trials` poll results by calling `poll(percentage, pollSize)` `trials` times. The function should return the minimum and maximum percentages in this list. For example, if five trials give the percentages `[28, 35, 31, 24, 31]`, the function should return the minimum `24` and maximum `35`. If the list of poll results is named `pollResults`, you can return these two values with

```
return min(pollResults), max(pollResults)
```

Test your function with a variety of poll sizes and numbers of trials.

3. What is a sufficient poll size?

Next, we want to use your previous functions to investigate how increasing poll sizes affect the variation of the poll results. Intuitively, the more people you poll, the more accurate the results should be. Write a function

```
plotResults(percentage, minPollSize, maxPollSize, step, trials)
```

that plots the minimum and maximum percentages returned by calling the function `pollExtremes(percentage, pollSize, trials)` for values of `pollSize` ranging from `minPollSize` to `maxPollSize`, in increments of `step`. For each poll size, call your `pollExtremes` function with

```
low, high = pollExtremes(percentage, pollSize, trials)
```

and then append the values of `low` and `high` each to its own list for the plot. Your function should return the margin of error for the largest poll, defined to be `(high - low) / 2`. The poll size should be on the x-axis of your plot and the percentage should be on the y-axis. Plot both the low and high values. Be sure to label both axes.

Question 5.1.1 *Assuming that you want to balance a low margin of error with the labor involved in polling more people, what is a reasonable poll size? What margin of error does this poll size give?*

Write a `main` function (if you have not already) that calls your `plotResults` function to investigate an answer to this question. You might start by calling it with `plotResults(30, 10, 1000, 10, 100)`.

4. Does the error depend on the actual percentage?

To investigate this question, write another function

```
plotErrors(pollSize, minPercentage, maxPercentage, step, trials)
```

that plots the margin of error in a poll of `pollSize` individuals, for actual percentages ranging from `minPercentage` to `maxPercentage`, in increments of `step`. To find the margin of error for each poll, call the `pollExtremes` function as above, and compute `(high - low) / 2`. In your plot, the percentage should be on the x-axis and the margin of error should be on the y-axis. Be sure to label both axes.

Question 5.1.2 *Does your answer to the previous part change if the actual percentage of the population is very low or very high?*

You might start to investigate this question by calling the function with `plotErrors(1500, 10, 80, 1, 100)`.

Project 5.2 Escape!

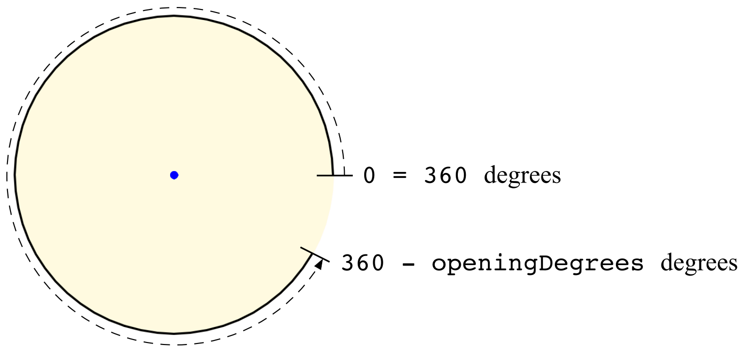
In some scenarios, movement of the "particle" in a random walk is restricted to a bounded region. But what if there is a small opening through which the particle might escape or disappear? How many steps on average does it take the particle to randomly come across this opening and escape? This model, which has become known as the *narrow escape problem*, could represent a forager running across a predator on the edge its territory, an animal finding an unsecured gate in a quarantined area, a molecule finding its way through a pore in the cell membrane, or air molecules in a hot room escaping through an open door.

1. Simulate the narrow escape

Write a function

```
escape(openingDegrees, tortoise, draw)
```

that simulates the narrow escape problem in a circle with radius 1 and an opening of `openingDegrees` degrees. In the circle, the opening will be between 360 – `openingDegrees` and 360 degrees, as illustrated below.



The particle should follow a *normally distributed* random walk, as described in Exercise 5.3.1. The standard deviation of the normal distribution needs to be quite small for the particle to be able to find small openings. A value of $\pi/128$ is suggested in [9]. Since we are interested in the number of steps taken by the particle (instead of the distance traveled, as before), the number of steps will need to be incremented in each iteration of the loop. When the particle hits a wall, it should "bounce" back to its previous position. Since the particle is moving within a circle, we can tell if it hits a wall by comparing its distance from the origin to the radius of the circle. If the particle moves out to the distance of the wall, but is within the angle of the opening, the loop should end, signaling the particle's escape.

Finding the current angle of the particle with respect to the origin requires some trigonometry. Since we know the x and y coordinates of the particle, the angle can be found by computing the arctangent of y/x: $\tan^{-1}(y/x)$. However, this will cause a problem with $x = 0$, so we need to check for that possibility and fudge the value of x a bit. Also, the Python arctangent ($\tan^{-1}$) function, `math.atan`, always returns an angle between $-\pi/2$ and $\pi/2$ radians (between −90 and 90 degrees), so the result needs to be adjusted to be between 0 and 360 degrees. The following function handles this for you.

```
def angle(x, y):
    if x == 0:       # avoid dividing by zero
        x = 0.001
    angle = math.degrees(math.atan(y / x))
    if angle < 0:
        if y < 0:
            angle = angle + 360   # quadrant IV
        else:
            angle = angle + 180   # quadrant II
    elif y < 0:
        angle = angle + 180       # quadrant III
    return angle
```

Below you will find a "skeleton" of the `escape` function with the loop and drawing code already written. Drawing the partial circle is handled by the function `setupWalls` below that. Notice that the function uses a `while` loop with a Boolean flag variable named `escaped` controlling the iteration. The value of `escaped` is initially `False`, and your algorithm should set it to `True` when the particle escapes. Most, *but not all*, of the remaining code is needed in the `while` loop.

```
def escape(openingDegrees, tortoise, draw):
    x = y = 0                         # initialize (x, y) = (0, 0)
    radius = 1                        # moving in unit radius circle
    stepLength = math.pi / 128        # std dev of each step

    if draw:
        scale = 300                   # scale up drawing
        setupWalls(tortoise, openingDegrees, scale, radius)

    steps = 0                         # count of steps taken
    escaped = False                   # has particle escaped yet?
    while not escaped:

        # one step of a random walk here
        # if the particle reaches the wall:
        #     if it is in the opening, then exit;
        #     otherwise, "bounce" back to previous saved position

        if draw:
            tortoise.goto(x * scale, y * scale)  # move particle

    if draw:
        screen = tortoise.getscreen()   # update screen to compensate
        screen.update()                 #  for high tracer value
    return steps
```

```
def setupWalls(tortoise, openingDegrees, scale, radius):
    screen = tortoise.getscreen()
    screen.mode('logo')                  # east is 0 degrees
    screen.tracer(5)                     # speed up drawing

    tortoise.up()                        # draw boundary with
    tortoise.width(0.015 * scale)        #   shaded background
    tortoise.goto(radius * scale, 0)
    tortoise.down()
    tortoise.pencolor('lightyellow')
    tortoise.fillcolor('lightyellow')
    tortoise.begin_fill()
    tortoise.circle(radius * scale)
    tortoise.end_fill()
    tortoise.pencolor('black')
    tortoise.circle(radius * scale, 360 - openingDegrees)
    tortoise.up()
    tortoise.home()

    tortoise.pencolor('blue')     # particle is a blue circle
    tortoise.fillcolor('blue')
    tortoise.shape('circle')
    tortoise.shapesize(0.75, 0.75)

    tortoise.width(1)            # set up for walk
    tortoise.pencolor('green')
    tortoise.speed(0)
    tortoise.down()              # comment this out to hide trail
```

2. Write a Monte Carlo simulation

Write a function

`escapeMonteCarlo(openingDegrees, trials)`

that returns the average number of steps required, over the given number of trials, to escape with an opening of `openingDegrees` degrees. This is very similar to the `rwMonteCarlo` function from Section 5.1.

3. Empirically derive the function

Write a function

`plotEscapeSteps(minOpening, maxOpening, openingStep, trials)`

that plots the average number of steps required, over the given number of trials, to escape openings with widths ranging from `minOpening` to `maxOpening` degrees, in increments of `openingStep`. (The x-axis values in your plot are the opening widths and y-axis values are the average number of steps required to escape.) This is very similar to the `plotDistances` function from Section 5.1.

Plot the average numbers of steps for openings ranging from 10 to 180 degrees, in

10-degree steps, using at least 1,000 trials to get a smooth curve. As this number of trials will take a few minutes to complete, start with fewer trials to make sure your simulation is working properly.

In his undergraduate thesis at the University of Pittsburgh, Carey Caginalp [9] mathematically derived a function that describes these results. In particular, he proved that the expected time required by a particle to escape an opening width of α degrees is

$$T(\alpha) = \frac{1}{2} - 2\ln\left(\sin\frac{\alpha}{4}\right).$$

Plot this function in the same graph as your empirical results. You will notice that the $T(\alpha)$ curve is considerably below the results from your simulation, which has to do with the step size that we used (i.e., the value of `stepLength` in the `escape` function). To adjust for this step size, multiply each value returned by the `escapeMonteCarlo` function by the square of the step size ($(\pi/128)^2$) before you plot it. Once you do this, the results from your Monte Carlo simulation will be in the same *time* units used by Caginalp, and should line up closely with the mathematically derived result.