lab9sol

November 2, 2014

1 Virtual Lab 9 Solution: Intro to Randomness (cont.)

1.0.1 EECS 70: Discrete Mathematics and Probability Theory, Fall 2014

Due Date: Monday, November 3rd, 2014 at 12pm **Name**:
Login: cs70-
Instructions:

- **Please fill out your name and login above.**
- **Please leave your answers in the Markdown cells, marked with `"YOUR COMMENTS HERE"`.**
- **Complete this lab by filling in all of the required functions, marked with `"YOUR CODE HERE"`.**
- If you plan to use Python, make sure to go over **Tutorial 1: Introduction to Python and IPython** and **Tutorial 2: Plotting in Python with Matplotlib** before attempting the lab.
- Make sure you run every code cell one after another, i.e. don't skip any cell. A shortcut for doing this in the notebook is Shift+Enter. When you finish, choose 'Cell > Run All' to test your code one last time all at once.
- Most of the solution requires no more than a few lines each.
- Please do not hardcode the result or change any function without the `"YOUR CODE HERE"` mark.
- Questions? Bring them to our Office Hour and/or ask on Piazza.
- Good luck, and have fun!

1.1 Table of Contents

The number inside parentheses is the number of functions or code blocks you are required to fill out for each question. Always make sure to double check before you submit.

- Introduction
- Part (a): Random Walk (3)
- Part (b): Normalized Random Walk (1)
- Part (c): q-curve (2)
- Part (d): Multiple q-curves (1)
- Part (e): Quartile Values (1)
- Part (f): Quartile Gap (1)
- Part (g): Scale the Gap (2)
- Part (h): Monty Hall (2)


```
In [1]: %pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
In [2]: from __future__ import division  # so that you don't have to worry about float division
        import random
        import math
```

Introduction

In this week's lab, we will continue our coin tossing example, but see it from a different perspective. Make sure you review the lab solution from Homework 8 before moving on.

Below, you will find sample implementations for some of the functions from last week's lab.

In [3]:
```
def count_heads(k):
    """
    Counts the number of heads in num_flips
    """

    return sum([round(random.random()) for _ in range(k)])
```

In [4]:
```
def count_heads_in_runs(k, n):
    """
    Returns a list of length n, where each element is the number
    of heads in k flips.
    """

    return [count_heads(k) for _ in range(n)]
```

In [5]:
```
def scaled_run(k, n):
    """
    Shifts the center of the horizontal axis to the origin by subtracting
    half the number of tosses from the number of heads
    """

    return [count_heads(k) - k//2 for _ in range(n)]
```

Part (a): Random Walk

Let's change gears a little bit from last time. Consider the following visualization of a sequence of coin flips. We start at zero. For every head we get, we add one. For every tail we get, we subtract one.

Hence, a sequence of 1000 coin tosses would be a path that starts at (0,0), and then goes to either $(1, 1)$ or $(1, -1)$, and continues wandering till $(1000, y)$ somewhere. Plot 20 such paths on the same plot based on randomly flipped coins. Each sample path should have 1000 coin tosses.

What do you observe about the paths?

Hint: First, implement the function `rand_one`, which generates 1 and -1 randomly with roughly 50% probability each. Then, implement the `path(n)` function, which returns a list of n elements that starts at 0 and every element thereafter is either one more or one less than the previous one. In Python, you can access the last element in a list with the syntax `lst[-1]`.

In [6]:
```
def rand_one():
    """
    Returns -1 roughly one half of a time, and 1 the other half

    YOUR CODE HERE
    """

    return -1 if random.random() < 0.5 else 1
```

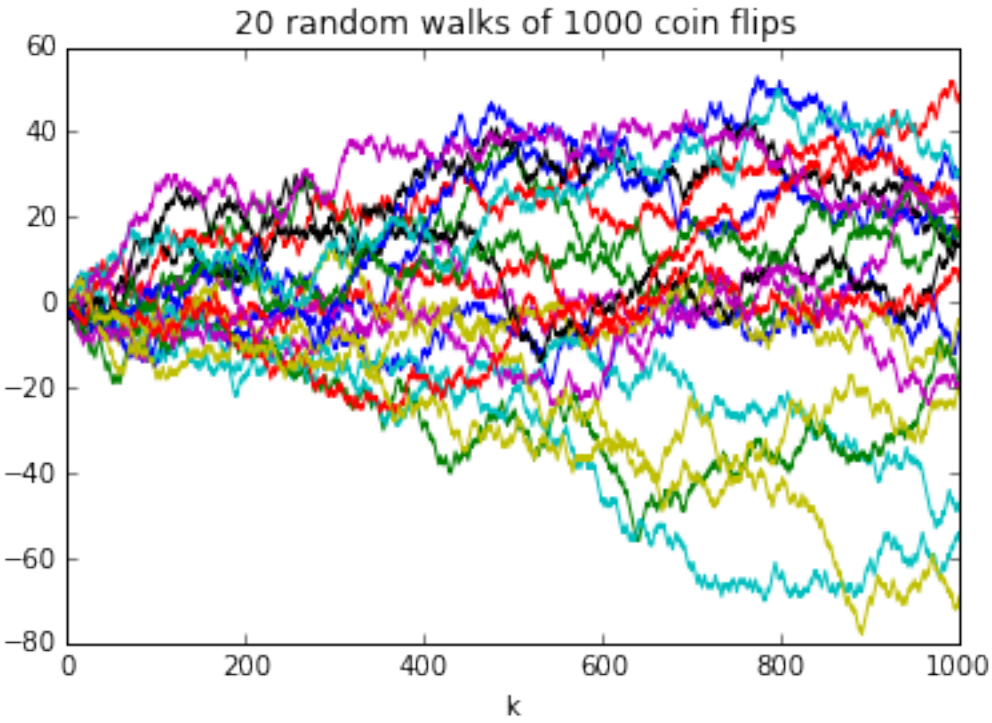
In [7]:
```
# Test your implementation by repeatedly running this code cell.
# You should see that the value fluctuates between -1 and 1.

rand_one()
```

Out[7]: 1

```
In [8]: def path(n):
            """
            An n-step random walk that starts at 0

            YOUR CODE HERE
            """

            walk = [0]
            for _ in range(1, n):
                walk.append(walk[-1] + rand_one())
            return walk
```

```
In [9]: def partA():
            """
            YOUR CODE HERE
            """

            for _ in range(20):
                plt.plot(path(1000))
            plt.title("20 random walks of 1000 coin flips")
            plt.xlabel("k")
            plt.show()
```

```
In [10]: partA()
```



YOUR COMMENTS HERE:

Part (b): Normalized Random Walk

Notice that the histograms you were plotting earlier were effectively looking at vertical slices in this picture and asking how many sample paths were crossing through a particular y coordinate. (If we are looking at

k tosses, then having exactly h heads is the same as this sample path crossing through $(k, h-(k-h)) = (k, 2h-k))$

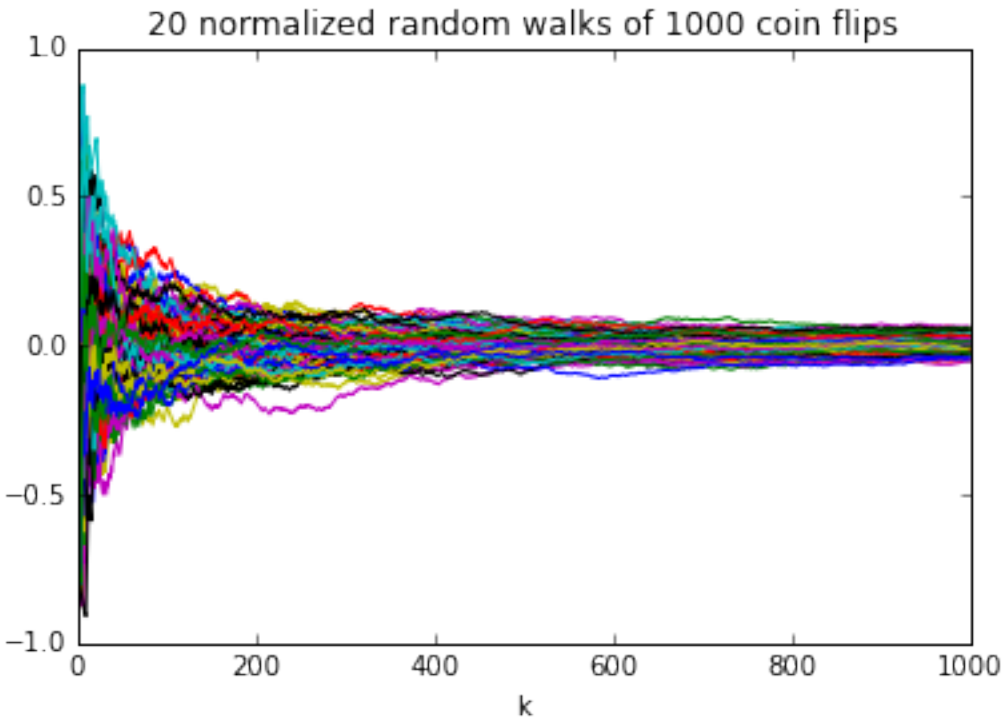
Now, let's see what the rescalings we were doing correspond to. The common-set-of-units scaling is what the previous part corresponded to. How would you change the scaling to correspond to the normalized set of units in part (e)? (in this plot, a sample path that consists of all heads should basically be a straight line that stays at the upper-limit — say 1. And a sample path that consists of all tails should be a straight line that stays at the lower limit – say -1).

Give this new scaling (it will depend on k — so it will change the visual shape of a path) and plot 100 sample paths of 1000 coin tosses each.

Comment on what this suggests relative to the earlier plots.

```
In [11]: def partB():
             """
             YOUR CODE HERE
             """

             for _ in range(100):
                 plt.plot([k/(i+1) for i, k in enumerate(path(1000))])
             plt.title("20 normalized random walks of 1000 coin flips")
             plt.xlabel("k")
             plt.show()
```

```
In [12]: partB()
```



YOUR COMMENTS HERE:

Part (c): q-curve

Shifting gears one more time, we are now going to look at the same basic experiment — tossing a fair coin k times — in a third way. Let R for a given run be the ratio of heads.

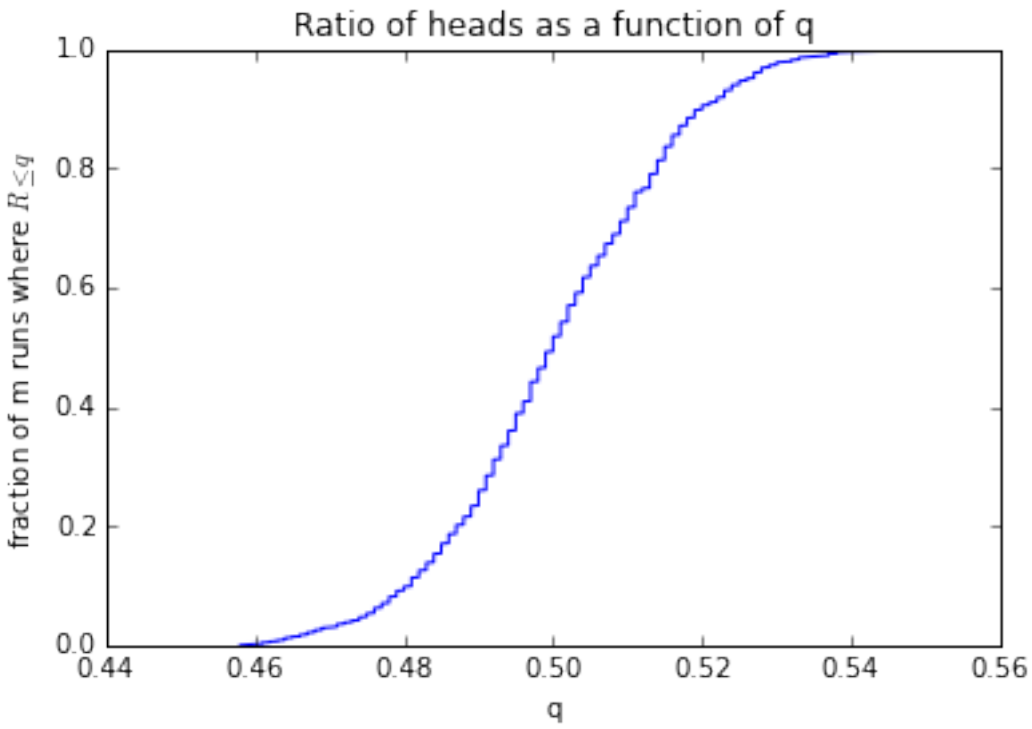
Fix $k = 1000$ to be the number of coin tosses in a run. Let $m = 1000$ be the number of runs. Plot how often $R \leq q$ as a function of q. The vertical axis should be (in linear scale) the fraction of the m runs in which $R \leq q$, while the horizontal scale should have q ranging from 0 to 1.

Hint: Implement the function `q_curve`, which returns the sorted fraction of heads for each of the m runs. You may find Python's built-in `sorted` function helpful here.

In [13]:
```
def q_curve(k, m):
    """
    Returns the sorted fraction R <= q for each of the m runs.

    YOUR CODE HERE
    """

    result = sorted([count_heads(k) / k for _ in xrange(m)])
    first_quartile = result[int(m * 0.25)]
    second_quartile = result[int(m * 0.5)]
    third_quartile = result[int(m * 0.75)]
    return result, first_quartile, second_quartile, third_quartile
```

In [14]:
```
# YOUR CODE HERE
# Hint: You can't use range with floats in Python. In that case,
# our good old friend np.linspace will help.

def partC():
    """
    YOUR CODE HERE
    """

    curve, _, _, _ = q_curve(1000, 1000)  # don't care about the quartile values here yet
    y_lim = np.linspace(0, 1, 1000)
    plt.plot(curve, y_lim)
    plt.title("Ratio of heads as a function of q")
    plt.xlabel("q")
    plt.ylabel("fraction of m runs where $R \leq q$")
    plt.show()
```

In [15]:
```
partC()
```



YOUR COMMENTS HERE:

Part (d): Multiple q-curves

Repeat the previous part for different values of k and put them all on the same plot. Try $k = 2, 10, 50, 100, 500, 1000, 10000$.

What do you see? Is this consonant with what you had observed in earlier plots?

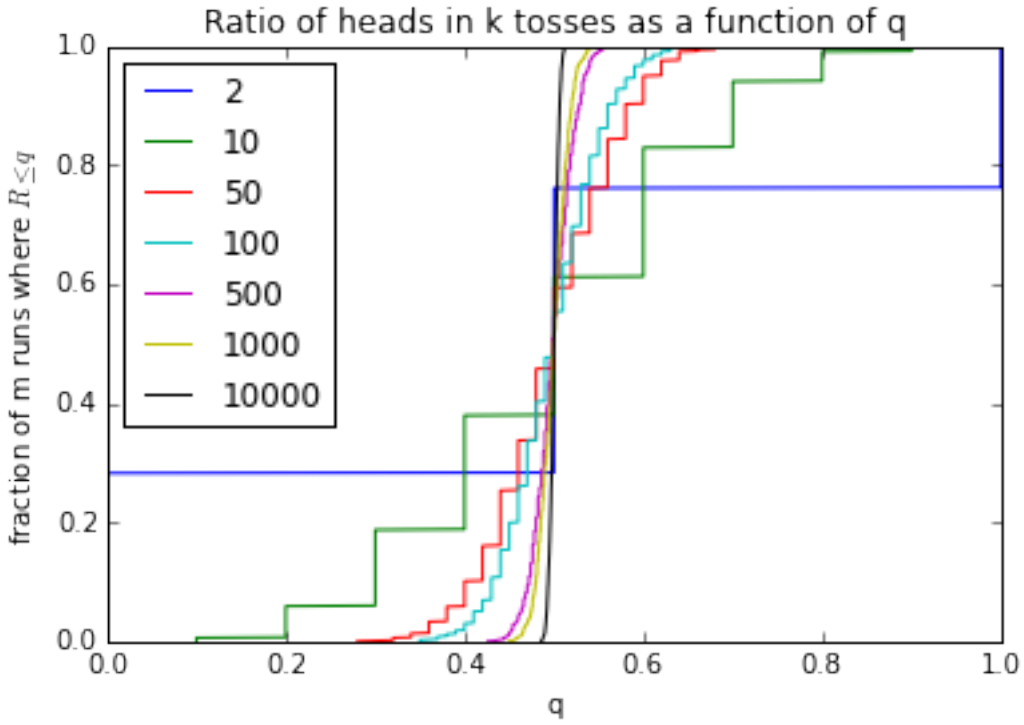
```
In [16]: ks = [2, 10, 50, 100, 500, 1000, 10000]
```

```
In [17]: # YOUR CODE HERE
```

```
def partD():
    """
    YOUR CODE HERE
    """

    y_lim = np.linspace(0, 1, 1000)
    first_Q, second_Q, third_Q = [], [], []

    for k in ks:
        curve, first, second, third = q_curve(k, 1000)
        plt.plot(curve, y_lim, label=str(k))

        # Append the quartile values for later parts
        first_Q.append(first)
        second_Q.append(second)
        third_Q.append(third)

    plt.title("Ratio of heads in k tosses as a function of q")
```

```
    plt.xlabel("q")
    plt.ylabel("fraction of m runs where $R \leq q$")
    plt.legend(loc=2)
    plt.show()
    return first_Q, second_Q, third_Q
```

In [18]:
```
first_Q, second_Q, third_Q = partD()
```



YOUR COMMENTS HERE:

Part (e): Quartile Values

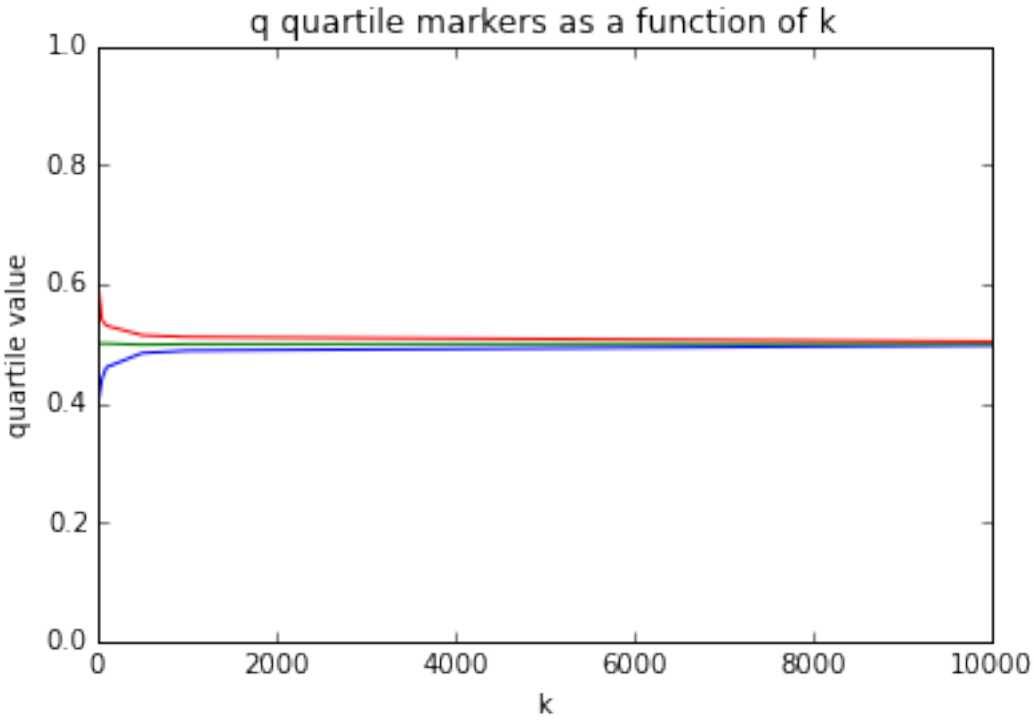
Now, think about rescaling the plots in the previous parts to see if there is something common about this shape. For each k, read off the q values where the curves seem to cross hypothetical horizontal lines at $0.25, 0.5, 0.75$. Call these the quartile markers. Compute these qs for your experiment. Plot them as a function of k. What do you observe?

In [19]:
```
# YOUR CODE HERE
```

```
def partE():
    """
    YOUR CODE HERE
    """

    plt.plot(ks, first_Q)
    plt.plot(ks, second_Q)
    plt.plot(ks, third_Q)
    plt.title("q quartile markers as a function of k")
    plt.ylim(0, 1)
    plt.xlabel("k")
```

```
            plt.ylabel("quartile value")
            plt.show()
```

```
In [20]: partE()
```



YOUR COMMENTS HERE:

Part (f): Quartile Gap

Notice that the gap between the 0.75 marker and the 0.25 marker is getting smaller as k gets larger. Notice also that the 0.5 marker seems to be sticking around $q = \frac{1}{2}$. As a scientific problem, suppose you wanted to discover how indeed this was scaling with k.

Plot the gap between the 0.75 and 0.25 marker as a function of k as a scatter plot. Use all of the traditional axes combinations: linear-linear, linear-log, log-linear, and log-log. Which one seems to offer some insight?

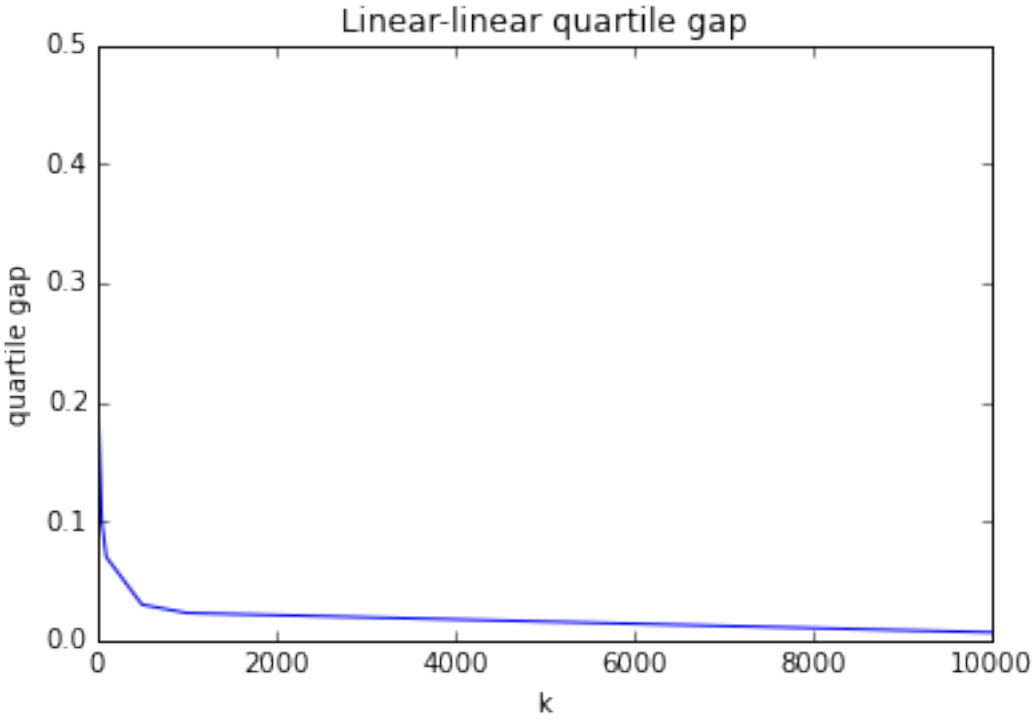
```
In [32]: gap = [third_Q[i] - first_Q[i] for i in range(len(first_Q))]
```

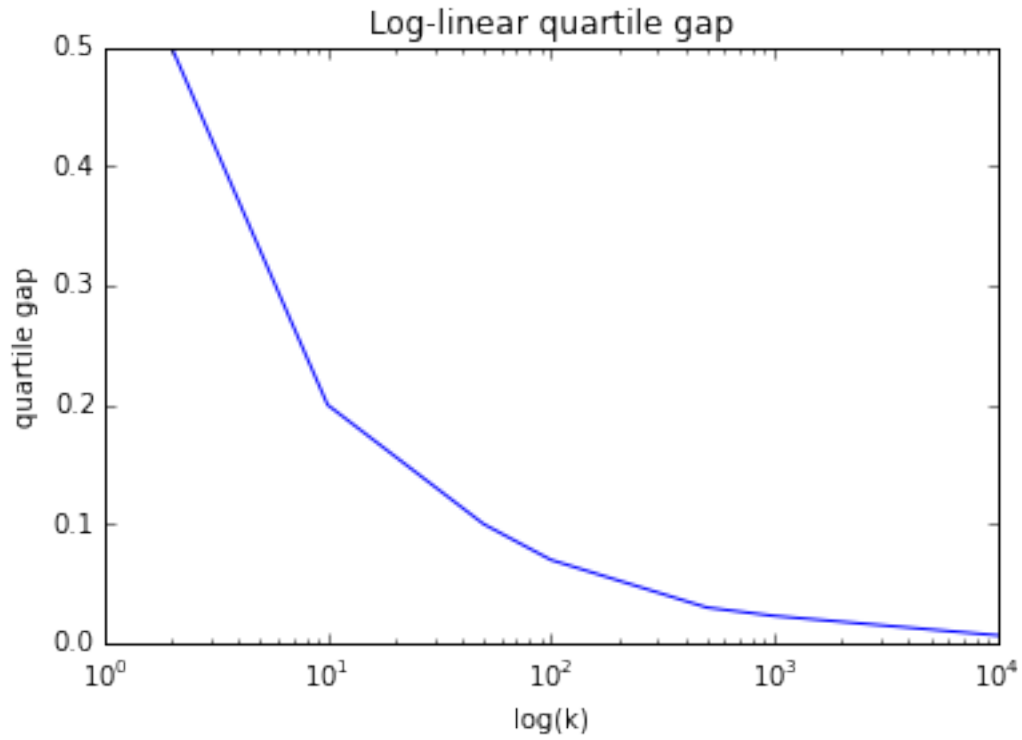
```
         def plot_quartile_gap(plot_fn, ks, gap):
             """
             Plot the quartile gap using plot_fn
             """

             plot_fn(ks, gap)
             plt.show()
```

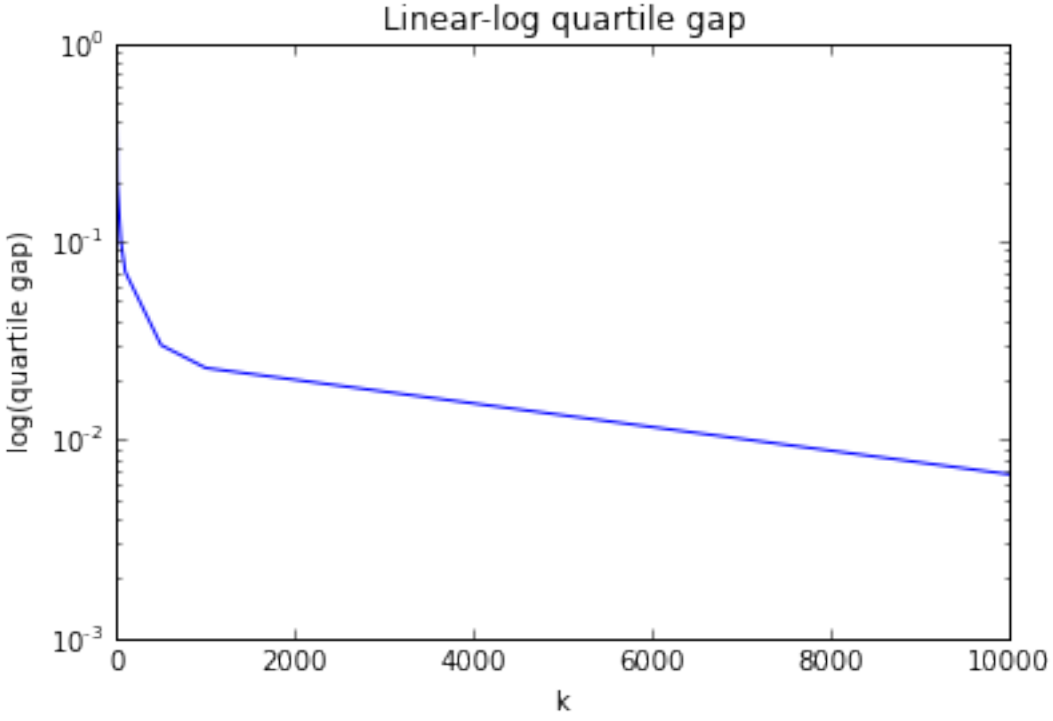
```
In [33]: def partF():
             """
             YOUR CODE HERE
             """
```

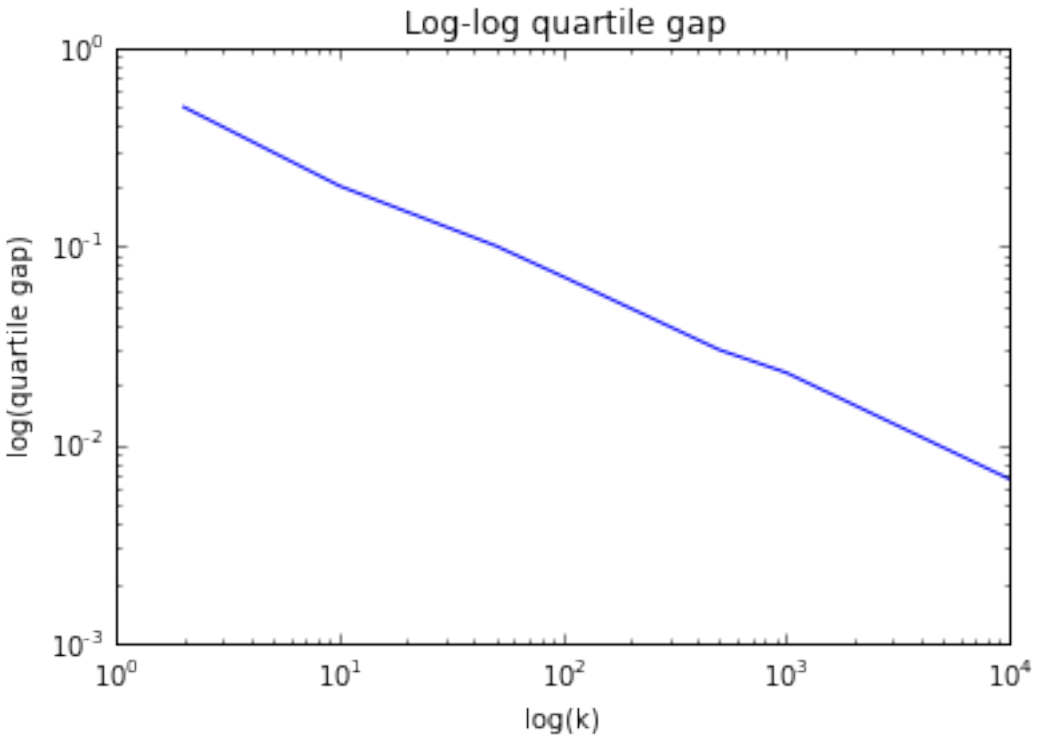
```
    plt.figure()
    plt.title("Linear-linear quartile gap")
    plt.xlabel("k")
    plt.ylabel("quartile gap")
    plot_quartile_gap(plt.plot, ks, gap)

    plt.figure()
    plt.title("Log-linear quartile gap")
    plt.xlabel("log(k)")
    plt.ylabel("quartile gap")
    plot_quartile_gap(plt.semilogx, ks, gap)

    plt.figure()
    plt.title("Linear-log quartile gap")
    plt.xlabel("k")
    plt.ylabel("log(quartile gap)")
    plot_quartile_gap(plt.semilogy, ks, gap)

    plt.figure()
    plt.title("Log-log quartile gap")
    plt.xlabel("log(k)")
    plt.ylabel("log(quartile gap)")
    plot_quartile_gap(plt.loglog, ks, gap)
```

In [34]: `partF()`




Log-linear quartile gap
quartile gap
log(k)
0.5
0.4
0.3
0.2
0.1
0.0
10^0
10^1
10^2
10^3
10^4


Linear-log quartile gap
log(quartile gap)
k
10^0
10^{-1}
10^{-2}
10^{-3}
0
2000
4000
6000
8000
10000



YOUR COMMENTS HERE:

Part (g): Scale the Gap

Based on what you observed in the previous set of plots, conjecture a scaling rule that lets you calculate the gap between the 0.75 marker and the 0.25 marker as a function of k for the fair coin tosses case. Explain your derivation in your writeup.

Use this rule to rescale the horizontal axis of the plots from three parts ago. What do you now observe about the curves for different values of k? By construction, they should be very close to each other in terms of where they are crossing $0.25, 0.5, 0.75$, but what about elsewhere?

Hint: Implement the function `q_curve_norm`, which does the same as `q_curve`, except now every point is normalized using your scaling rule. Think about the previous part and how it can help you come up with a scaling rule.

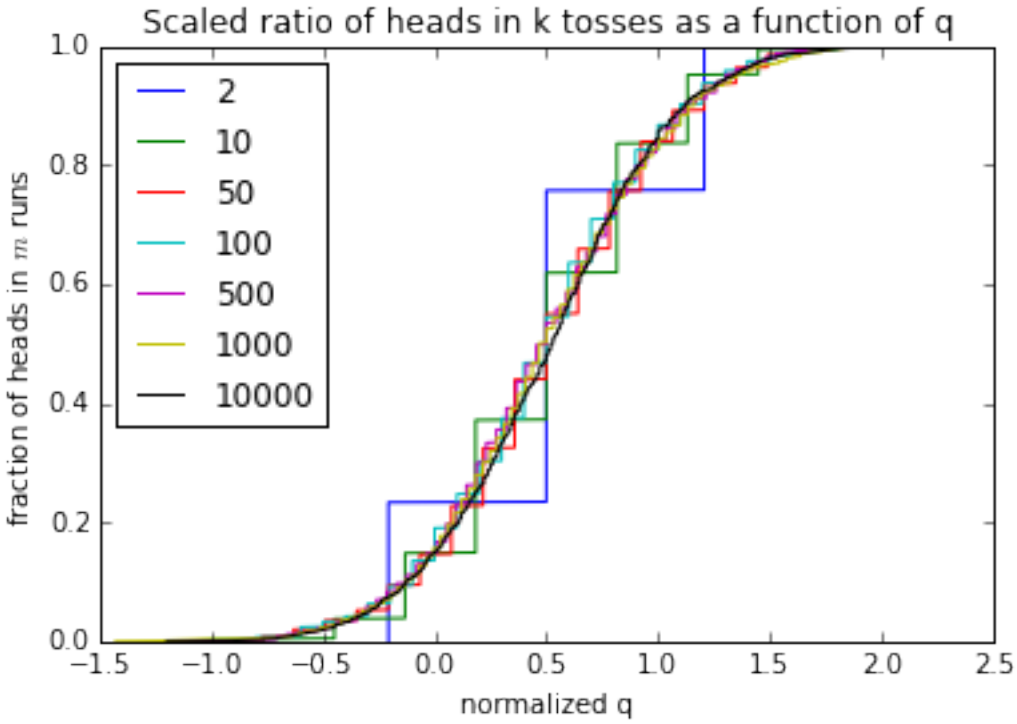
This is a challenging question. Don't worry if you get stuck here.

```
In [24]: def q_curve_norm(k, m):
             """
             Does the same as q_curve, except now every point is
             normalized using your scaling rule.

             YOUR CODE HERE
             """

             return sorted([(count_heads(k) / k - 0.5) * math.sqrt(k) + 0.5 for _ in xrange(m)])
```

```
In [35]: def partG():
             """
             YOUR CODE HERE
             """
```

```
    y_lim = np.linspace(0, 1, 1000)
    for k in ks:
        plt.plot(q_curve_norm(k, 1000), y_lim, label=str(k))
    plt.xlabel("normalized q")
    plt.ylabel("fraction of heads in $m$ runs")
    plt.title("Scaled ratio of heads in k tosses as a function of q")
    plt.legend(loc=2)
    plt.show()
```

In [36]: `partG()`



YOUR COMMENTS HERE:

Part (h): Monty Hall

Below, you will find a simple implementation that simulates the Monty Hall problem. There are n doors, and only one contains the prize. The contestant first picks a door, and then the host Monty will open all $n-2$ doors that don't contain the prize. The contestant is then given a choice to switch or stay with his current choice.

Your task will be to simulate 10000 trials of the Monty Hall problem. What is the probability of winning when the contestant switches? How about when he/she stays? Does it match your expectation and what was described in lecture and the lecture note? Please report the result in your answer.

Finally, if you want to understand how the algorithm work in general, make sure you set the `verbose` parameter to True.

In [27]:
```
def monty_hall(num_doors=3, switch=True, verbose=False):
    """
    Carries out the game for one contestant.  If 'switch' is True,
    the contestant will switch their chosen door when offered the chance.
    Returns True if the simulated contestant won, False otherwise.
```

```
    Doors are numbered from 0 up to num_doors-1 (inclusive).
    """

    # Randomly choose the door hiding the prize.
    winning_door = random.randint(0, num_doors-1)
    if verbose:
        print '\nPrize is behind door {}'.format(winning_door+1)

    # The contestant picks a random door, too.
    choice = random.randint(0, num_doors-1)
    if verbose:
        print 'Contestant chooses door {}'.format(choice+1)

    # The host opens all but two doors.
    closed_doors = list(range(num_doors))
    while len(closed_doors) > 2:
        # Randomly choose a door to open.
        door_to_remove = random.choice(closed_doors)

        # The host will never open the winning door, or the door
        # chosen by the contestant.
        if door_to_remove == winning_door or door_to_remove == choice:
            continue

        # Remove the door from the list of closed doors.
        closed_doors.remove(door_to_remove)
        if verbose:
            print 'Host opens door {}'.format(door_to_remove+1)

    # There are always two doors remaining.
    assert len(closed_doors) == 2

    # Does the contestant want to switch their choice?
    if switch:
        if verbose:
            print 'Contestant switches from door {} '.format(choice+1)

        # There are two closed doors left.  The contestant will never
        # choose the same door, so we'll remove that door as a choice.
        available_doors = list(closed_doors) # Make a copy of the list.
        available_doors.remove(choice)

        # Change choice to the only door available.
        choice = available_doors.pop()
        if verbose:
            print 'to {}'.format(choice+1)

    # Did the contestant win?
    won = (choice == winning_door)
    if verbose:
        if won:
            print 'Contestant WON'
        else:
```

```
                print 'Contestant LOST'
        return won
```

In [28]:
```
# Run this cell to understand the algorithm.
# Change the parameters (especially the second one).

monty_hall(3, True, verbose=True)
```

```
Prize is behind door 1
Contestant chooses door 2
Host opens door 3
Contestant switches from door 2
to 1
Contestant WON
```

Out[28]: True

In [29]:
```
num_trials = 10000
print "Simulating", num_trials, "trials."
won_switch_count = 0

# YOUR CODE HERE
# Please make sure you don't set `verbose` to True,
# otherwise the output is going to flood your notebook.

for i in range(10000):
    if monty_hall(3, True):
        won_switch_count += 1

# END YOUR CODE HERE

print 'Switching won {0:5} times out of {1} ({2}% of the time)'.format(\
    won_switch_count, num_trials, (won_switch_count / num_trials * 100))
print 'Staying won {0:5} times out of {1} ({2}% of the time)'.format(\
    num_trials-won_switch_count, num_trials, ((num_trials-won_switch_count) / num_trials * 100)
```

```
Simulating 10000 trials.
Switching won  6652 times out of 10000 (66.52% of the time)
Staying won  3348 times out of 10000 (33.48% of the time)
```

PASTE YOUR SIMULATION RESULT FROM ABOVE IN THIS CELL

Congratulations! You are done with Virtual Lab 9.

Don't forget to convert this notebook to a pdf document, merge it with your written homework, and submit both the pdf and the code (as a zip file) on glookup.

Reminder: late submissions are NOT accepted. If you have any technical difficulty, resolve it early on or use the provided VM.