CM1103 Coursework

David Buchanan

December 12, 2016

1. (a)

```python
def game(ra, rb):
  p = ra / (ra+rb)
  sa, sb = 0, 0

  while max(sa, sb) < 11 or abs(sa-sb) < 2:
    r = random.random()
    if r < p:
      sa += 1
    else:
      sb += 1

  return sa, sb
```

(b)

```python
def winProbability(ra, rb, n):
  total = 0
  for _ in range(n):
    sa, sb = game(ra, rb)
    if sa > sb:
      total += 1
  return total / n
```

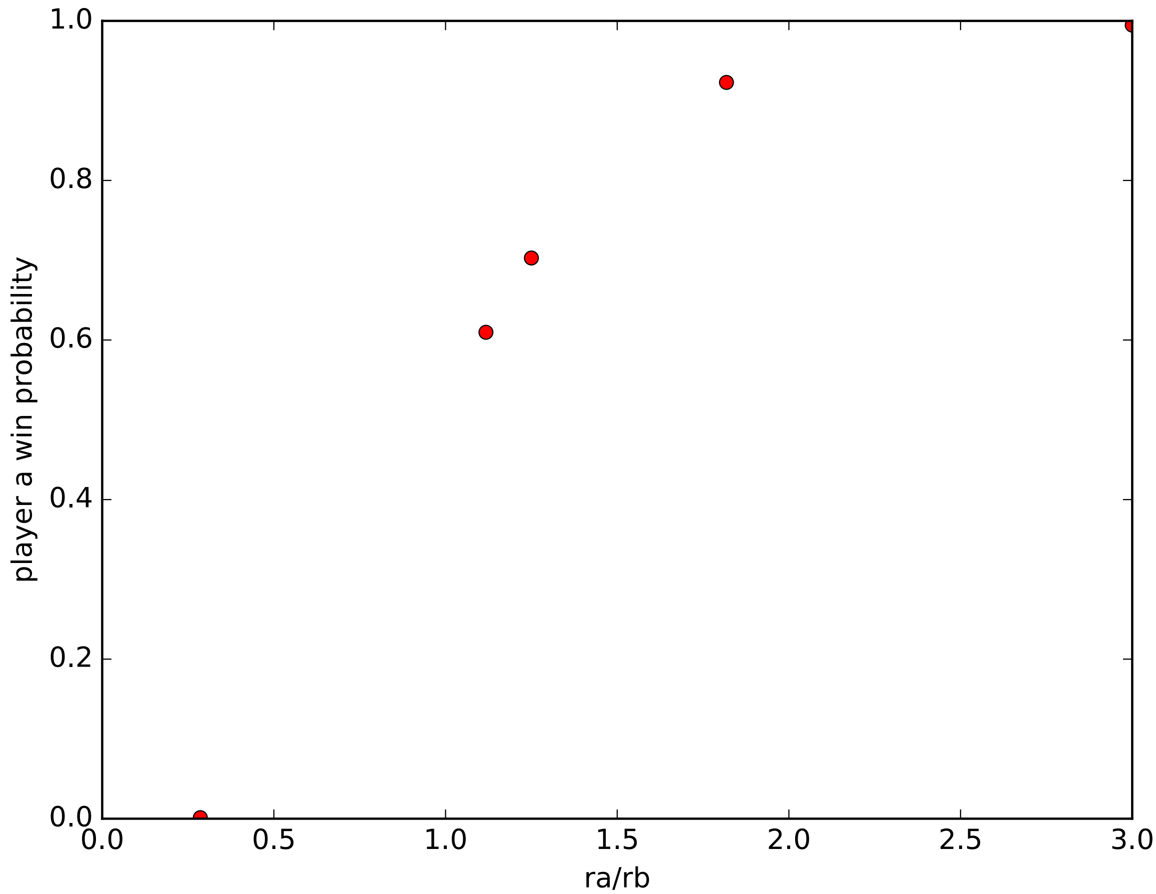
(c)

```python
def readCSV(filename):
  csvfile = open(filename, "r")
  csvfile.readline() # discard header
  return [tuple(map(int, re.findall(r"[0-9]+",line))) for line in csvfile.readlines()]
```

(d)

```python
def plotProbabilities(ratios):
  p1 = list(map(lambda r: r[0]/r[1], ratios))
  p2 = list(map(lambda r: winProbability(r[0], r[1], 10000), ratios))

  plt.plot(p1, p2, "ro")
  plt.xlabel("ra/rb")
  plt.ylabel("player a win probability")
  plt.savefig("csvfig.svg")
```

When this function is executed with the provided ratios, it generates the following graph:



(e) The probability that a wins a single game can be calculated by running the following python snippet:

```
>>> winProbability(60, 40, 100000)
0.83656
```

The number 100000 was chosen arbitrarily. From this output, we know that the probability of a winning a single game (p) is approximately 0.83656. If $n = 1$, then the probability that a wins the whole match (P) is also 0.83656. If $n = 2$, then we can calculate P as follows:

$$P = p^2 + 2(p^2)(1 - p) \approx 0.929$$

This is because there are three different ways a could win overall - By winning 2 in a row, "win, lose win", and "lose, win, win". Since $0.929 > 0.9$, the solution is $n = 2$.

2. The following python code generates the first half of my answer, for PARS:

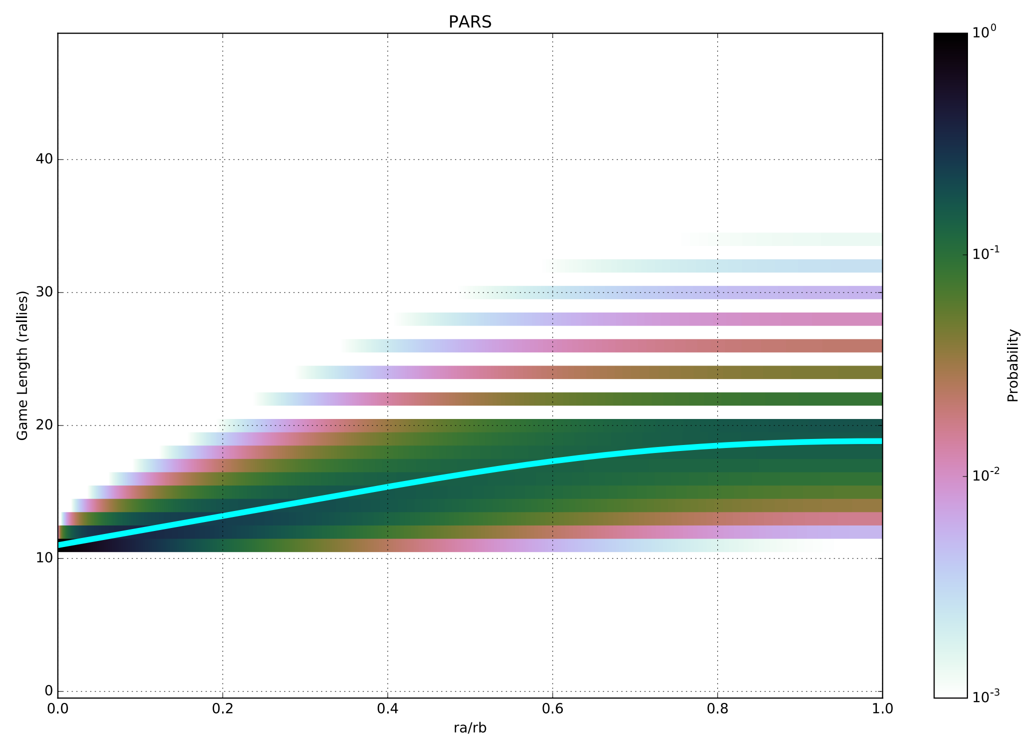
```
from functools import reduce
import numpy as np
import matplotlib as mpl
import matplotlib.colors as colors
import matplotlib.pyplot as plt

WIN_THRESHOLD = 11
REQUIRED_LEAD = 2
MAX_DURATION = WIN_THRESHOLD*2 - REQUIRED_LEAD
```

```python
TOTAL_DURATION = 50
HORIZONAL_RESOLUTION = 2048
ALMOST_ZERO=10e-9

xedges = np.linspace(0.0, 1.0, num=HORIZONAL_RESOLUTION+1)
yedges = np.linspace(-0.5, TOTAL_DURATION-0.5, num=TOTAL_DURATION+1)
H = np.full((TOTAL_DURATION, HORIZONAL_RESOLUTION), ALMOST_ZERO)

expectations = [None]*len(xedges)

def ncr(n, r):
  r = min(r, n-r)
  if r == 0: return 1
  numer = reduce(lambda a, b: a*b, range(n, n-r, -1))
  denom = reduce(lambda a, b: a*b, range(1, r+1))
  return numer//denom

for rarb in xedges[:-1]:
  rb = 1
  ra = rarb*rb
  r = ra/(ra+rb)
  n = int(rarb*HORIZONAL_RESOLUTION)

  remaining = 1

  for game_length in range(WIN_THRESHOLD, MAX_DURATION+1):
    score_of_loser = game_length - WIN_THRESHOLD
    probability = (r**WIN_THRESHOLD * (1-r)**score_of_loser +
      r**score_of_loser * (1-r)**WIN_THRESHOLD) * ncr(game_length-1, WIN_THRESHOLD-1)
    H[game_length, n] = probability
    remaining -= probability

  """
  the remaining probability is the chance of reaching 10/10

  The only way the game can end is if one player scores twice in a row
  """

  for length in range(MAX_DURATION+2, TOTAL_DURATION, 2):
    if remaining < ALMOST_ZERO:
      break
    prob = remaining * (r**2 + (1-r)**2)
    H[length, n] = prob
    remaining -= prob

  probs = H[:,n]
  expectations[n] = np.average(list(range(len(probs))), weights=probs)

X, Y = np.meshgrid(xedges, yedges)
plt.figure(figsize=(16,10))
plt.pcolormesh(X, Y, H, cmap="cubehelix_r",
  norm=colors.LogNorm(0.001, 1.0)).set_rasterized(True)
plt.axis([X.min(),X.max(),Y.min(),Y.max()])
plt.plot(xedges, expectations, lw=5, c="cyan")
plt.colorbar(label="Probability")
plt.grid()
```

```
66 plt.title("PARS")
67 plt.xlabel("ra/rb")
68 plt.ylabel("Game Length (rallies)")
69 plt.savefig("PARS.svg", dpi=300)
```

For every value of Ra/Rb between 0 and 1, the probability of each game length between 0 and 50 occurring is calculated. The main probability calculation occurs on lines 37-38 using the binomial distribution. However, this is only for games that do not reach a score of 10/10. After this point, different logic is used, shown on lines 48-53. The resulting probability is represented as a color on the output graph, using the "cubehelix" colourmap. This colourmap was chosen because it's intensity gradient is linear, so it still makes sense when viewed on a black-and white printout or by someone who is colourblind. A logarithmic colourmap was chosen, because I believe it shows the relevant details of the graph better.

The expected game length for each ability ratio is also graphed, represented by the cyan line:



The following python code generates the second half of my answer, for English scoring:

```
import numpy as np
import matplotlib as mpl
import matplotlib.colors as colors
import matplotlib.pyplot as plt

END_SCORE = 9
TOTAL_DURATION = 50
HORIZONAL_RESOLUTION = 1024
ALMOST_ZERO = 10e-9

xedges = np.linspace(0.0, 1.0, num=HORIZONAL_RESOLUTION+1)
yedges = np.linspace(-0.5, TOTAL_DURATION-0.5, num=TOTAL_DURATION+1)
```

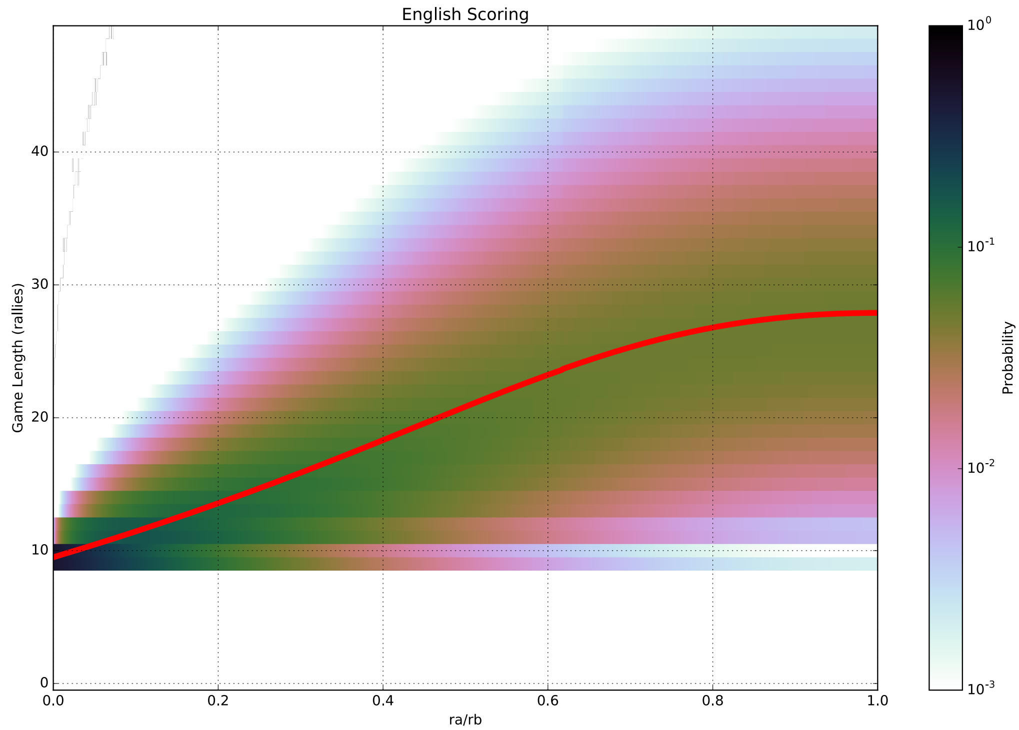
```
H = np.full((TOTAL_DURATION, HORIZONAL_RESOLUTION), ALMOST_ZERO)

expectations = [None]*len(xedges)

for rarb in xedges[:-1]:
  rb = 1
  ra = rarb*rb
  r = ra/(ra+rb)
  n = int(rarb*HORIZONAL_RESOLUTION)

  transition_table = {"start": { # initial "pseudostate" to decide who serves first
    (True, 0, 0, END_SCORE): 0.5,
    (False, 0, 0, END_SCORE): 0.5
  }}

  # transistion_table[i][j] will equal the probability of moving from state
  # i to state j in 1 step

  transient_states = ["start"]
  absorbing_states = []

  def calculate_probability(r, a_serves, a_score, b_score, playing_to):
    state = (a_serves, a_score, b_score, playing_to)
    transition_table[state] = {}

    if max(a_score, b_score) == playing_to:# or a_score == b_score == 4:
      absorbing_states.append(state)
      transition_table[state][state] = 1.0
      return

    if a_score == b_score == playing_to-1 == END_SCORE-1: # tie breaker
      if a_serves: # b chooses
        if (1-r)**2 - 3*(1-r) + 1 < 0:
          playing_to = END_SCORE+1
      else: # a chooses
        if r**2 - 3*r + 1 < 0:
          playing_to = END_SCORE+1

    transient_states.append(state)

    if a_serves:
      transition_table[state][(True, a_score+1, b_score, playing_to)] = r
      transition_table[state][(False, a_score, b_score, playing_to)] = 1-r
    else: # b serves
      transition_table[state][(True, a_score, b_score, playing_to)] = r
      transition_table[state][(False, a_score, b_score+1, playing_to)] = 1-r

    for new_state in transition_table[state].keys():
      # only if we haven't already checked:
      if new_state not in transient_states+absorbing_states:
        calculate_probability(r, *new_state)

  # recusrively construct transition table
  calculate_probability(r, True, 0, 0, END_SCORE);

  # convert transition table into numpy matrix, so we can do calculations with it
```

```
69   # more easily
70
71   P_array = []
72   transients = len(transient_states)
73
74   for i in transient_states+absorbing_states:
75     probabilities = []
76     for j in transient_states+absorbing_states:
77       probabilities.append(transition_table[i].get(j, 0))
78     P_array.append(probabilities)
79
80   P = np.matrix(P_array)
81   Q = P[0:transients, 0:transients]
82
83   prev = 0
84   cumulativeP = P*1 # make a copy of P
85
86   for game_length in range(TOTAL_DURATION):
87     cumulative = np.sum(cumulativeP[0,transients:])
88     cumulativeP *= P
89     H[game_length, n] = cumulative-prev
90     prev = cumulative
91
92   I = np.identity(transients)
93   N = np.linalg.inv(I-Q)
94
95   expectations[n] =  np.sum(N[0]) - 1 # subtract 1 to ignore starting state
96   print("Progress: {}%".format(rarb*100)) # this program runs quite slowly...
97
98 X, Y = np.meshgrid(xedges, yedges)
99 plt.figure(figsize=(16,10))
100 plt.pcolormesh(X, Y, H, cmap="cubehelix_r",
101   norm=colors.LogNorm(0.001, 1.0)).set_rasterized(True)
102 plt.axis([X.min(),X.max(),Y.min(),Y.max()])
103 plt.plot(xedges, expectations, lw=5, c="red")
104 plt.colorbar(label="Probability")
105 plt.grid()
106 plt.title("English Scoring")
107 plt.xlabel("ra/rb")
108 plt.ylabel("Game Length (rallies)")
109 plt.savefig("English.svg", dpi=300)
```

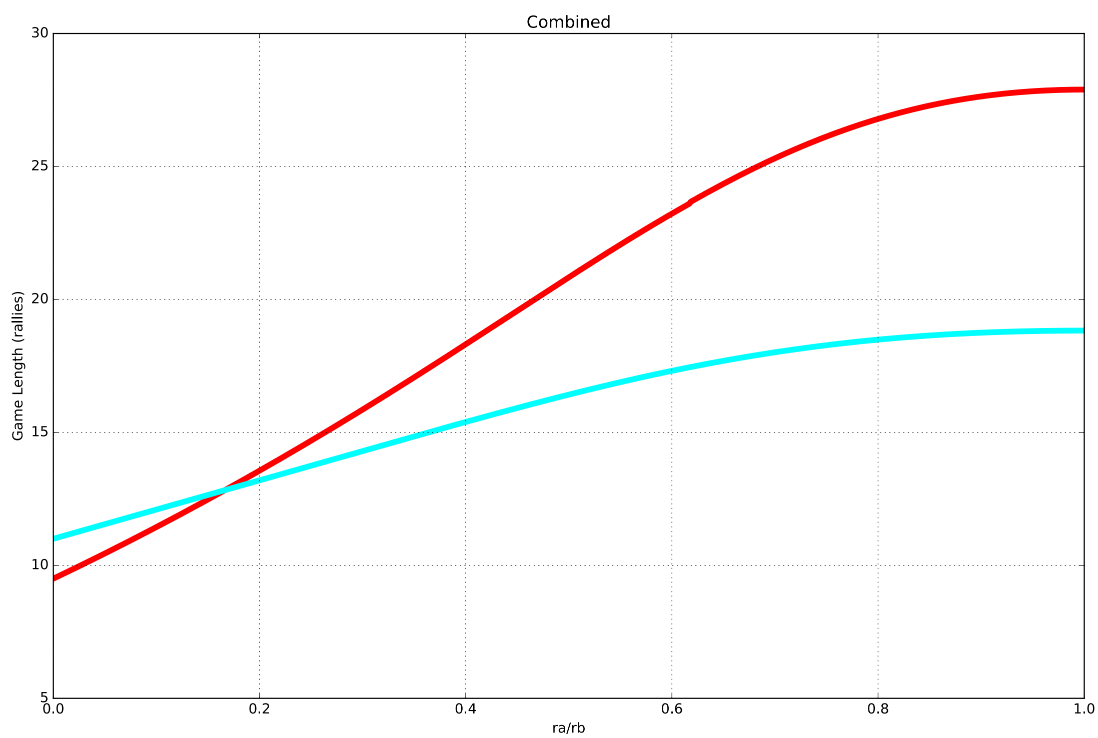
Because the game state can return to a previous state (any time the server loses a rally), there are an infinite number of paths to get from one state from a previous state. Therefore, it is not easy to use simple maths to calculate the expected game length. By representing the game as a Markov chain, we can generate a transition matrix, and use various matrix operations to calculate the required probabilities.

Lines 34-66 construct a transition table recursively, using nested dicts, for convenience. Lines 71-80 convert this data structure into a numpy matrix to make mathematical calculations easy.

The expected game length is represented by the magenta line on the graph:



If we combine the results for the expected game lengths for both scoring systems, we get the following graph (cyan=PARS, magenta=English):



For the English scoring system, I assumed that when a player gets to choose to play to 9 or 10, they make the statistically optimal choice, based on calculations from this article: `https://nrich.maths.org/1390`

In both of these programs, I assumed that the probability of a winning a point was constant, and that the game never lasted longer than 50 points.

I used this article to learn the maths required for the last program: `https://en.wikipedia.org/wiki/Absorbing_Markov_chain`