609 Final Project, Fall 2016

Project Team: James Topor, Youqing Xiang

December 16, 2016

Chapter 5.3, Project 1

> **1.** *Blackjack*—Construct and perform a Monte Carlo simulation of blackjack (also called twenty-one). The rules of blackjack are as follows:

NOTE: The **R** code used for Blackjack simulation can be found in the attached **Appendix**.

Project 1 in Chapter 5.3 asks us to construct a Monte Carlo simulation of the card game called Blackjack. We are told that in addition to the basic rules of Blackjack, the simulation must adhere to the following requirements:

1. Bets of $2 will be placed on each hand by the "player", with no limit on the number of $2 bets that can be placed.
2. A win of a hand will earn the player $2 above and beyond their initial $2 bet. A loss of a hand will result in the player losing the entirety of their $2 bet. If a hand results in a tie between the player and the dealer, no money is either won or lost.
3. The game will start using two decks of cards. Hands will be played until all cards from the two decks have been used.
4. When the two decks of cards are exhausted, the player's running total winnings/losses for all hands played with the two decks of cards are saved.
5. When two decks of cards are exhausted, a new set of two decks is initiated and play resumes, with the player's winnings/losses tally being accummulated anew.
6. A total of 12 sets of two decks of cards should be "played".
7. When all 12 set of two decks have been played, we must average the winning/losses from all 12 iterations to determine the overall performance of the player.
8. The player cannot see any of the dealer's cards

The simulation must therefore be capable of generating multiple two-deck sets of cards for purposes of enabling the required 12 iterations two-deck play. Furthermore, the simulation must also carefully manage the dealing of cards to both the player and the dealer, with a variety of constraints being checked as each card is dealt.

For example, since the required simulation requires the play of nearly 12 * 2 * 52 = 1248 cards, it is simply not practical to require a player to interactively play the blackjack game. In fact, the requirements for the simulation state quite clearly that we must choose a strategy to play for the player and then play that strategy throughout the entire simulation. As such, we assume that the player will always accept another card if their current hand totals 15 or less, and will always stand whenever their hand totals 16 or more.

Similarly, per the instructions the dealer will always accept another card if their hand totals 16 or less and will always stand whenever their hand totals 17 or more.

The 'main' portion of the simulation generates and shuffles a 2-deck set of cards, with each face card represented by its numeric Blackjack value. For example, jacks, queens, and kings are all treated as '10' while aces are initially treated as '11'.

The shuffled 2-deck set of cards is then "played" until all cards have been used. A total of 12 iterations of 'play' are simulated, with the player's winnings from each simulation stored in a numeric vector. Finally, the vector of winnings is averaged to determine the player's average winnings per hand, with the results displayed onscreen.

iteration	res.winnings
1	-14
2	16
3	-8
4	2
5	6
6	-2
7	14
8	16
9	8
10	0
11	-8
12	2

```
## Average Winnings per Hand =  2.66666666666667
```

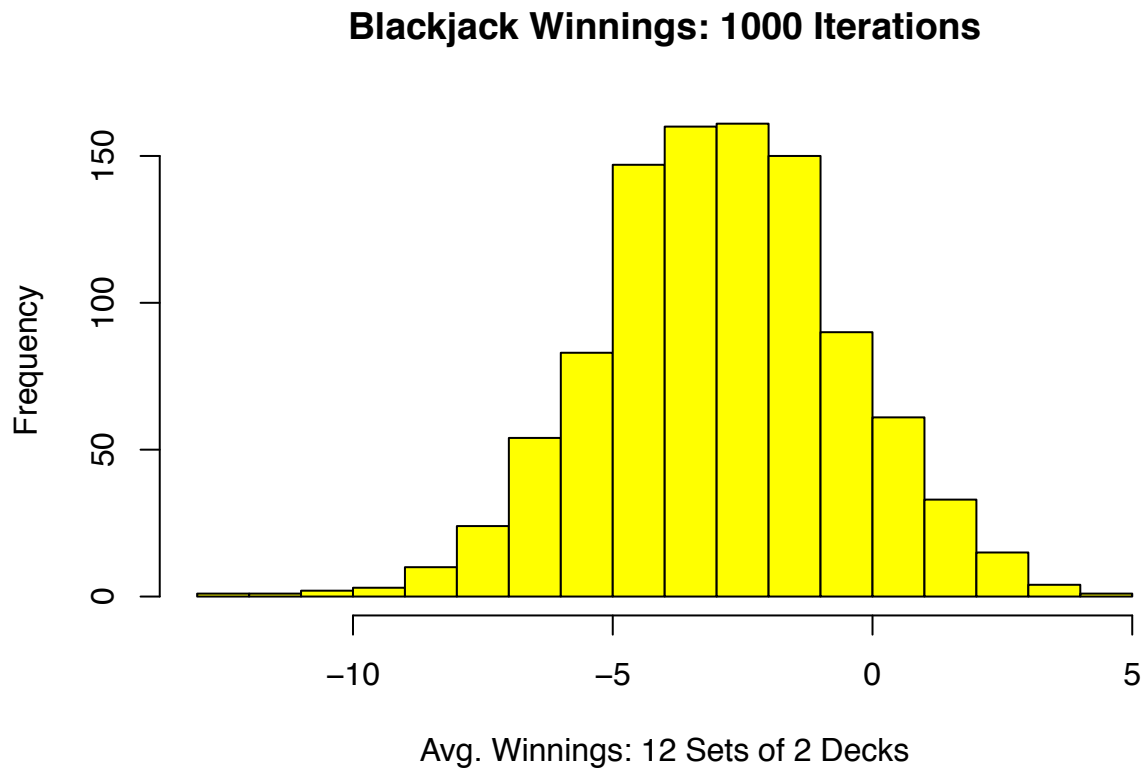
The results shown above represent a fairly limited sample of possible player winnings relative to the betting strategy described in the requirements for this project. To get a better sense of what a player's likely winnings would be we can run a large number of iterations (e.g., 1000) of the above simulation and then calculate the average winnings for all iterations.

Summary statistics for the player's average winnings from 1000 iterations of playing 12 sets of 2 decks of Blackjack with the player standing if the value of their hand is 16 or greater are shown below:

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -12.670  -4.333  -2.833  -2.816  -1.167   4.667
```

Based on the Blackjack rules we were required to adhere to, the player's average losses when playing 12 sets of 2 decks of cards is approximately ($2.81). As such, a player should expect to lose money if they adhere to the "stand at 16 or greater" strategy.

A histogram of the average winnings of 1000 iterations of playing 12 sets of 2 decks of Blackjack shows that the distribution of the average winnings is approximately normal, and also reinforces the conclusion that the player should on average expect to lose money when adhering to the "stand at 16 or greater" strategy.



We can attempt to pinpoint the player strategy that minimizes the expected average losses by evaluating "stand" values ranging from 11 to 20 via further simulation iterations:

stand	avg_winnings
11	-5.148667
12	-3.723333
13	-3.248833
14	-2.930833
15	-2.589167
16	-2.683167
17	-3.011333
18	-4.290167
19	-7.613667
20	-12.870833

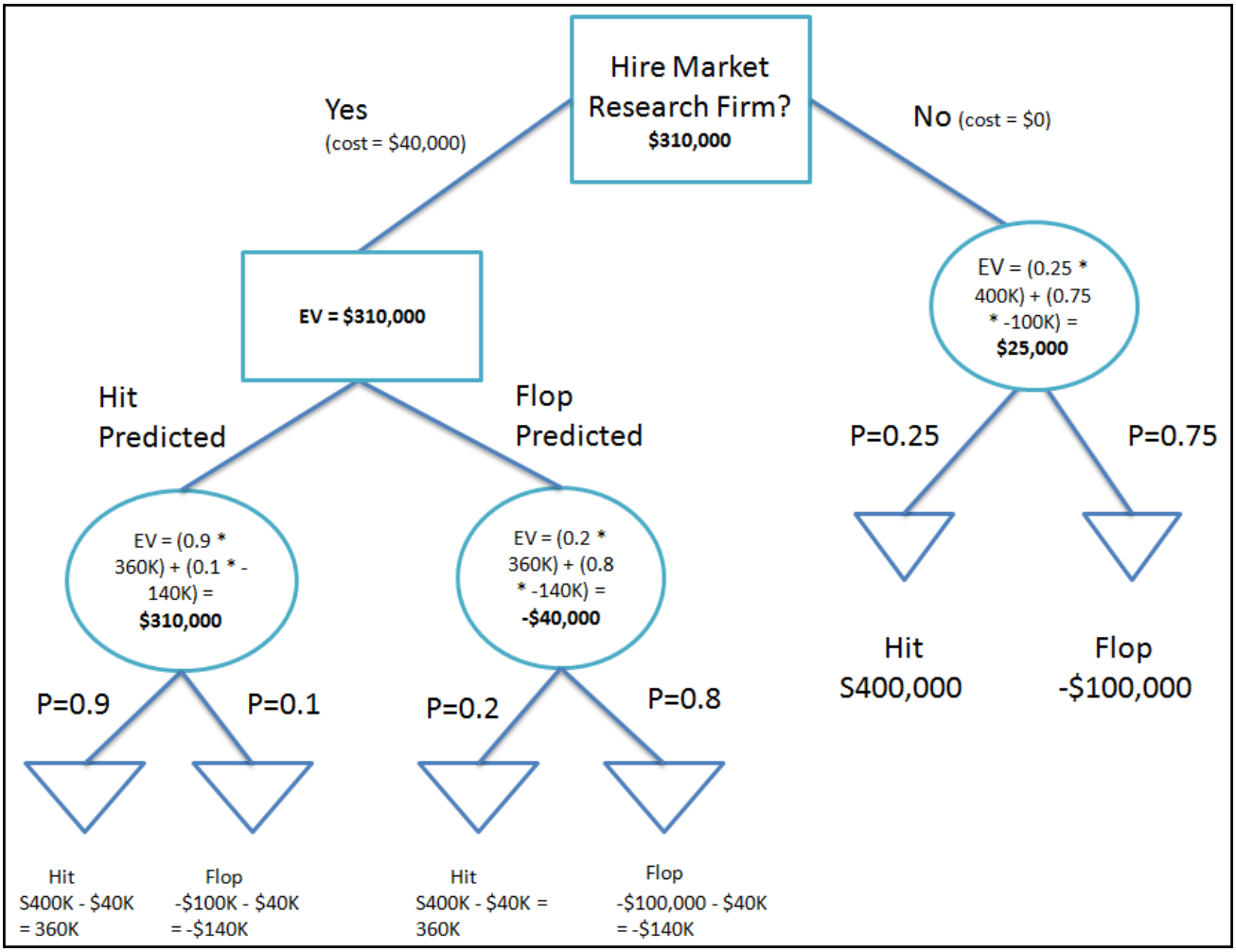
The table above shows that the average player loss gradually declines as they increase their Blackjack "stand at" value from 11 to 15. However, increasing the "stand at" hurdle to any value above 15 will result in average losses greater than would be expected if the player adhered to a "stand at" value of 15. As such, if the player wishes to minimize their average losses, they should adhere to a Blackjack strategy of "stand at 15".

Chapter 9.4, Project 4

4. The NBC TV network earns an average of $400,000 from a hit show and loses an average of $100,000 on a flop (a show that cannot hold its rating and must be canceled). If the network airs a show without a market review, 25% turn out to be hits, and 75% are flops. For $40,000, a market research firm can be hired to help determine whether the show will be a hit or a flop. If the show is actually going to be a hit, there is a 90% chance that the market research firm will predict a hit. If the show is going to be a flop, there is an 80% chance that the market research will predict the show to be a flop. Determine how the network can maximize its profits over the long haul.

To best determine how NBC may maximize its profits we need to determine whether the expected value of hiring the market research firm exceeds that of simply airing a new show without it having been evaluated by the market research firm.

A decision tree for this problem is shown below:



The righthand branch of the decision tree allows us to evaluate the expected value of NBC choosing to not hire the market research firm prior to airing a new show. We can calculate the expected value of not hiring the market research company as follows:

```
Hit_earn = 400000
Flop_cost = -100000

# probabilities of hits + flops
pH <- 0.25 # prob of hit
pF <- 0.75 # prob of flop

# now calculate expected value of market research
ev_nomr <- (Hit_earn * pH ) + (Flop_cost  * pF)

ev_nomr
```

```
## [1] 25000
```

As we can see, if NBC chooses not to hire a market research firm prior to airing a show, the expected value of airing the show will be **$25,000**.

The lefthand branch of the decision tree allows us to evaluate the expected value of NBC hiring the market research firm prior to airing a new show. We can calculate the expected value of hiring the market research company as follows:

```
Hit_earn = 400000
Flop_cost = -100000
MR_cost = -40000

# probabilities of hits + flops
pHH <- 0.9 # prob of hit given that hit predicted
pFH <- 0.1 # prob of flop given flop predicted

pHF <- 0.2 # prob of hit given flop predicted
pFF <- 0.8 # prob of flop given flop predicted

# now calculate expected value of market research
ev_mrHIT <- ( ( (Hit_earn + MR_cost) * pHH ) +
           ((Flop_cost + MR_cost) * pFH) )
ev_mrFLOP <- ( ((Hit_earn + MR_cost) * pHF) +
           ((Flop_cost + MR_cost) * pFF) )
ev_mrHIT
```

```
## [1] 310000
```

```
ev_mrFLOP
```

```
## [1] -40000
```

If the research firm predicts that the show will be a hit, NBC should air the show since the expected value of airing the show will be **$310,000**, inclusive of the cost of the research report. Conversely, if the research firm predicts a flop, the expected value of airing the show is **(-$40,000)**, an amount that coincidentally matches the cost of the market research report. As such, NBC should *never* air a show if the market research firm predicts a flop.

Therefore, relying on the results of the market research firm's analysis yields an expected value of **$310,000** per show, while not using the research firm yields an expected value of only **$25,000** per show. Based on this result, the network should hire a market research firm if it wishes to maximize its profits over the long haul

---

Chapter 12.5, Project 2

> **2.** Using the improved Euler's method, approximate the solution to the harvesting predator–prey problem in Problem 7. Compare the new solution to the one obtained in Problem 7 using the same step size $\Delta t = 1$ over the interval $0 \leq t \leq 4$. Graph the solution trajectories for both solutions.

Problem 7 in Section 12.5 reads as follows:

> **7.** The following system is a predator–prey model in which harvesting occurs for both species. Use Euler's method with step size $\Delta t = 1$ over $0 \leq t \leq 4$ to numerically solve
>
> $$\frac{dx}{dt} = x - xy - \frac{3}{4}y$$
> $$\frac{dy}{dt} = xy - y - \frac{3}{4}x$$
>
> subject to $x(0) = \frac{1}{2}$ and $y(0) = 1$.

As shown above, for Problem 7 we are asked to apply Euler's method with step size $\Delta t = 1$ where $0 <= t <= 4$. Therefore, we need to calculate a total of four approximations to each curve $x(t)$ and $y(t)$ over the interval $I : t_0 <= t <= 4$. The following R code snippet is used to calculate the required approximations:

```
# Euler's Method

# initialize data frame to store results
res <- data.frame(x = numeric(5), y = numeric(5))
rownames(res) <- c("t0", "t1", "t2", "t3", "t4")

# initialize x0 and y0
res$x[1] <- 1/2
res$y[1] <- 1

# initialize delta
delta <- 1

# now loop starting at 2 since res[1] contains values of X0 and y0 at t0
for (i in 2:5) {
  # dx/dt = x - xy - 3/4*y
  res$x[i] <- res$x[(i-1)] + ( res$x[(i-1)] - (res$x[(i-1)] * res$y[(i-1)]) -
                               (3/4 * res$y[(i-1)] ) ) * delta

  # dy/dt = xy - y - 3/4*x
  res$y[i] <- res$y[(i-1)] + ( (res$x[(i-1)] * res$y[(i-1)]) -
              res$y[(i-1)] - (3/4 * res$x[(i-1)] ) ) * delta
}
```

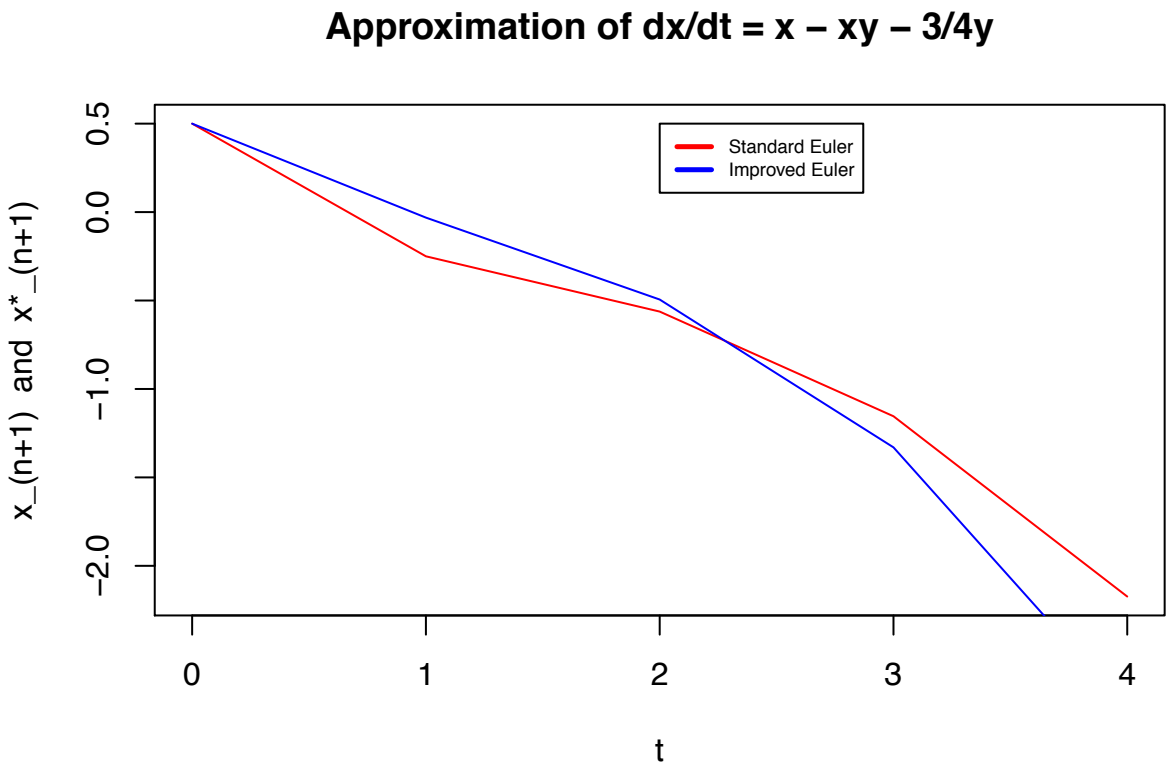
Project 12.5.2 asks us to apply the Improved Euler's Method as described on page 567 in the textbook to approximate the solution to the harvesting predator-prey scenario described in Problem 7 (solved above). The Improved Euler's Method is calculated as follows:

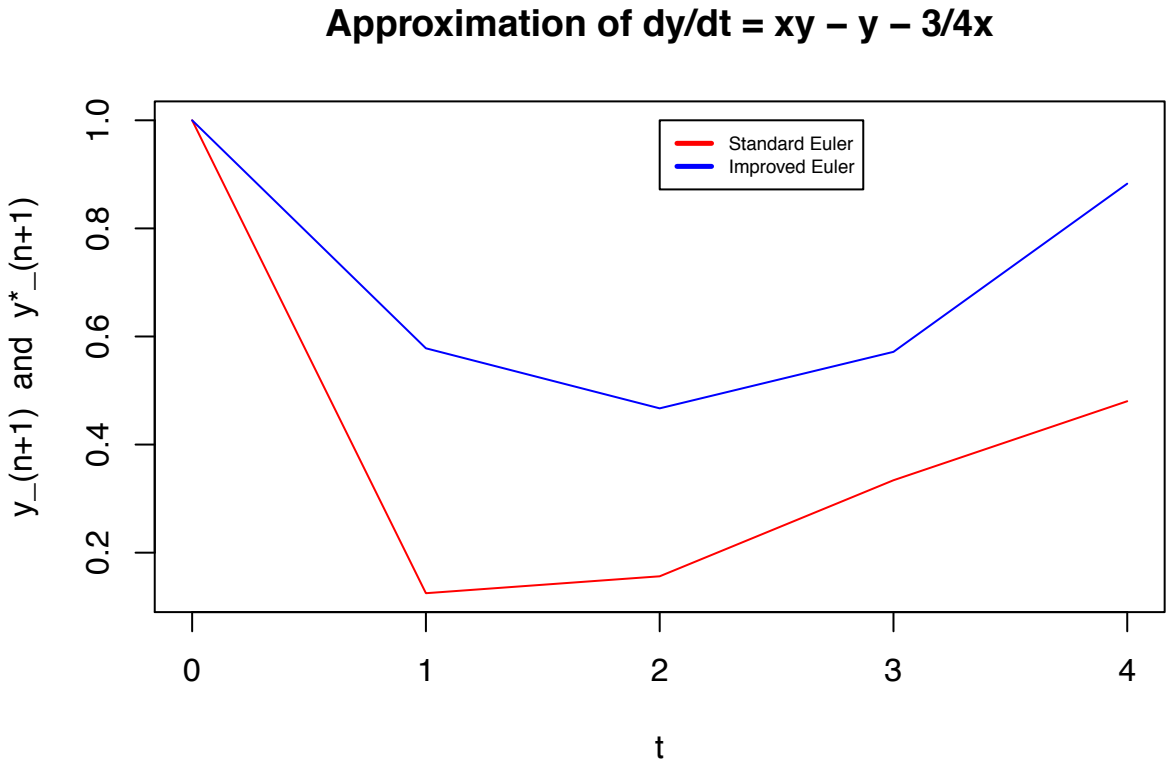
```
# Improved Euler's Method

# initialize data frame to store results
res2 <- data.frame(x = numeric(5), y = numeric(5))
rownames(res2) <- c("t0", "t1", "t2", "t3", "t4")

# initialize x0 and y0
res2$x[1] <- 1/2
res2$y[1] <- 1

# initialize delta
delta <- 1

# now loop starting at 2 since res[1] contains values of X0 and y0 at t0
for (i in 2:5) {
  # find x*_(n+1). Use variable 'xsnp1' to indicate "x*_n+1"
      # equation is: x_n+1 = x_n + (x_n + (x_n * y_n) - 3/4(x_n) ) * delta
  xsnp1 <- res2$x[(i-1)] + ( res2$x[(i-1)] - (res2$x[(i-1)] * res2$y[(i-1)]) -
                           (3/4 * res2$y[(i-1)] ) ) * delta

  # find y*_(n+1). Use variable 'ysnp1' to indicate "y*_n+1"
      # equation is: y_n+1 = y_n + ( (x_n * y_n) - y_n - 3/4(x_n) ) * delta
  ysnp1 <- res2$y[(i-1)] + ( (res2$x[(i-1)] * res2$y[(i-1)]) -
              res2$y[(i-1)] - (3/4 * res2$x[(i-1)] ) ) * delta

  # calculate x - xy - 3/4*y WITHOUT multiplying by delta OR adding x_n to result
  fxn <- res2$x[(i-1)] - (res2$x[(i-1)] * res2$y[(i-1)]) - (3/4 * res2$y[(i-1)] )

  # calculate xy - y - 3/4*x WITHOUT multiplying by delta OR adding y_n to result
  gyn <- (res2$x[(i-1)] * res2$y[(i-1)]) - res2$y[(i-1)] - (3/4 * res2$x[(i-1)] )

   # calculate f(x*_n+1)
  fxsnp1 <- xsnp1 - xsnp1*ysnp1 - 0.75*ysnp1

  # calculate g(y*_n+1)
  gysnp1 <- xsnp1 * ysnp1 - ysnp1 - 0.75*xsnp1

  # now calculate approximations for x and y
  res2$x[i] <- res2$x[(i-1)] + ( (fxn + fxsnp1) * (delta/2) )
  res2$y[i] <- res2$y[(i-1)] + ( (gyn + gysnp1) * (delta/2) )
}
```

The approximations obtained via both methods are shown below in tabular format.

Euler.X	Euler.Y	Imp_Euler.X	Imp_Euler.Y
0.500000	1.0000000	0.5000000	1.0000000
-0.250000	0.1250000	-0.0312500	0.5781250
-0.562500	0.1562500	-0.4943827	0.4670390
-1.154297	0.3339844	-1.3302892	0.5715652
-2.173565	0.4802055	-2.8066576	0.8827161

A side-by-side graphical comparison of the standard Euler's method and the Improved Euler's method is shown below for the approximated x_{n+1}, x^*_{n+1}, y_{n+1} and y^*_{n+1} values.



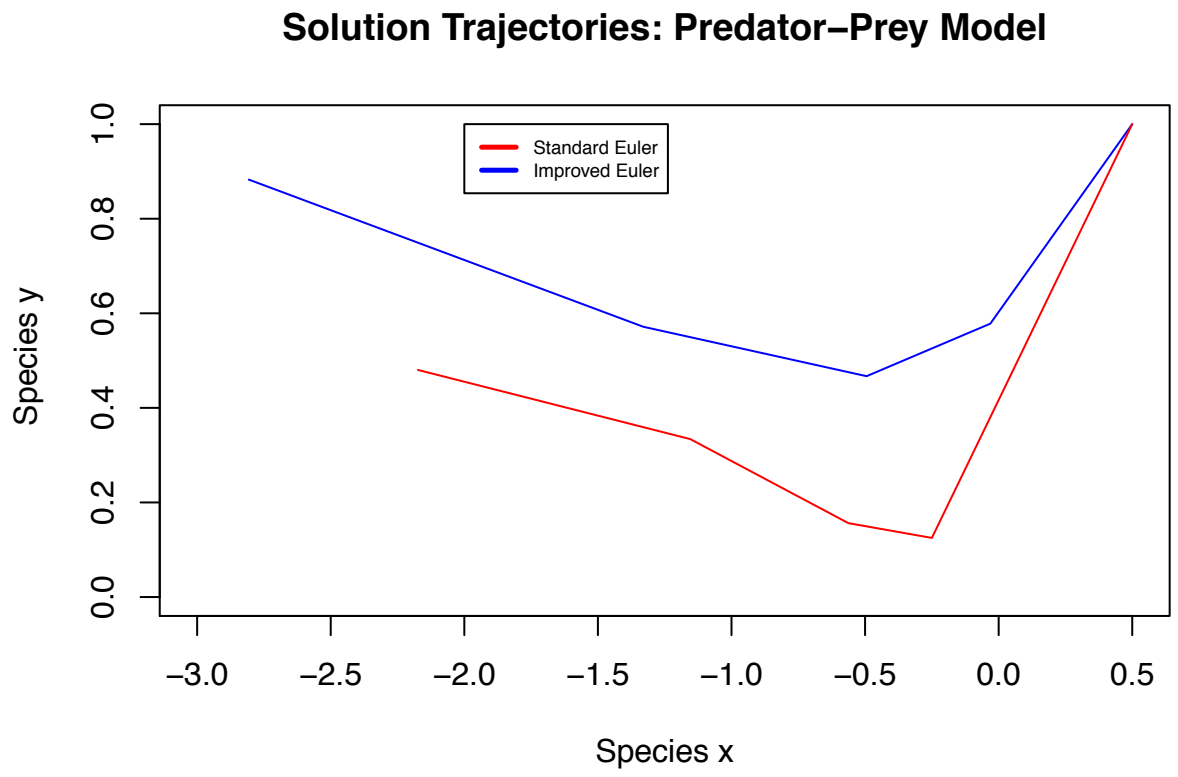


As we can see in the plots shown above, x^*_{n+1} values derived using the Improved Euler's Method do not appear to

vary very much from the x_{n+1} values derived using the standard Euler's method. In both instances, the plot shows a continuous decline in *Species x* over time.

However, the y^*_{n+1} values derived via the Improved Euler's Method do appear to vary less than the y_{n+1} values derived via the standard Euler's method as evidenced by the relatively narrow range of values exhibited in the plot. However, both methods show clear evidence of an initial decline in *Species y* followed by a recovery of the species.

Combining the respective derived values into a single plot yields the solution trajectories for both methods:



As stated in Chapter 12.5, Problem 7, the origin of each trajectory occurs at $x0 = 0.5, y0 = 1$. Both trajectories then follow a path down and to the left before making an upward turn toward the upper lefthand corner of the plot. As we can see in the plot, both species decline initially. However, as *Species x* continues to decline *Species y* begins to recover. Finally, no evidence of periodicity is seen in the plot.

Appendix

R Code for Blackjack Simulation (Chapter 5.3, Project 1)

We start by defining a collection of functions that will be used to facilitate the "play" of both the dealer and the player.

```
#############################################################################
# Function sum_cards: sums the values of the cards in a blackjack hand. Aces
# are counted as either 11 or 1 depending on the sum total of non-ace cards in the hand
#
```

```r
# Parameters:
# - hand: a blackjack hand that has been dealt to either a player or dealer.
#
# NOTE: Blackjack hands in this simulation consist solely of integer values since the
#       cards do not need to be displayed graphically. Suits of cards also are irrelevant
#       for similar reasons
# ============================================================================

sum_cards <- function(hand) {

  aces <- FALSE
  ace_locs <- numeric()
  sum <- 0

  # if the hand is only 2 cards, just sum and exit
  if (length(hand) == 2) {
    sum <- hand[1] + hand[2]
    return(sum)
  }

  # find aces
  for (k in 1:length(hand)) {
    if(hand[k] == 11) {
      aces <- TRUE
      ace_locs <- c(ace_locs, k)
    }
  }

  # if no aces were found, sum all items and return
  if (aces == FALSE) {
    for (i in hand) {
       sum <- sum + i
    }
    return(sum)
  }

  # otherwise aces were found so first sum items that are NOT aces
  not_aces <- subset(hand, !(hand %in% c(11)))
  for (i in not_aces) {
    sum <- sum + i
  }

  # if sum of non-aces > 10 all aces must be treated as 1's
  if (sum > 10) {
    for (i in ace_locs) {
      sum <- sum + 1
    }
    return(sum)
  }

  # if sum of non-aces <= 9, need to check count of aces.
  # If > 1 one can be treated as 11 while other must be treated as 1
  if (sum <= 9 & length(ace_locs) > 1) {
    sum <- sum + 11
    for (i in 1:(length(ace_locs) - 1)) {
```

```
      sum <- sum + 1
    }
    return(sum)
  }

  # if sum of non-aces == 10, need to check count of aces.
  # if == 1, then add 11 to sum and exit. if > 1, treat all aces as 1's
  if (sum == 10 & length(ace_locs) == 1) {
    sum <- sum + 11
    return(sum)
  } else {
    for (i in 1:length(ace_locs)) {
      sum <- sum + 1
    }
    return(sum)
  }

}

##############################################################################
# Function player_h: implements standard blackjack rules logic for the player's hand
# Parameters:
# - p_hand: the initial 2 cards dealt to the player
# - cards: a vector containing a pre-shuffled set of cards
# - i: an index variable that indicates the position of the card most recently dealt
#
# NOTE: Player strategy will be to take another card any time hand is 15 or less. This
#       implies that the player will stand with any hand of 16 or greater.
# ============================================================================

player_h <- function(p_hand, cards, i, p_stand){

  ncards <- length(cards)
  stand <- FALSE

  while (stand != TRUE) {
    # sum current hand
    current <- sum_cards(p_hand)

    # if card index == number of cards in decks, time to stop
    if (i == ncards) {
      stand <- TRUE

    # else if hand is < p_stand value, take another card
    } else if (current < p_stand) {
      i <- i + 1
      p_hand <- c(p_hand, cards[i])
    } else {
      stand <- TRUE
    }

  } # end while stand != TRUE

  return(c(current, i))
}
```

```r
##############################################################################
# Function dealer_h: implements standard blackjack rules logic for the dealer's hand
# Parameters:
# - d_hand: the initial 2 cards dealt to the dealer
# - cards: a vector containing a pre-shuffled set of cards
# - i: an index variable that indicates the position of the card most recently dealt
#
# NOTE: Dealer MUST stand on hand total of 17 or greater, so any time dealer hand is <= 16
#       another card must be drawn
# ============================================================================

dealer_h <- function(d_hand, cards, i){

  ncards <- length(cards)
  stand <- FALSE

  while (stand != TRUE) {
    # sum current hand
    current <- sum_cards(d_hand)

    # if card index == number of cards in decks, time to stop
    if (i == ncards) {
      stand <- TRUE

    # else if hand is <= 16, take another card
    } else if (current <= 16) {
      i <- i + 1
      d_hand <- c(d_hand, cards[i])
    } else {
      stand <- TRUE
    }

  } # end while stand != TRUE

  return(c(current, i))
}

##############################################################################
# Function play_bj: plays blackjack until 'cards' are exhausted and tallies player's
# winnings and losses
#
# Parameters:
# - cards: a pre-shuffled set of cards
# - bet: the dollar amount of bets placeable by the player for each blackjack hand
#
# NOTE: need to sample WITHOUT replacement from cards. Since they are
# pre-shuffled before calling this function, just deal from top of deck and
# use and index to keep track of which cards haven't been dealt
# ============================================================================

play_bj <- function(cards, bet, p_stand){

  # initialize the number of cards to be played
  ncards <- length(cards)
```

```
# initialize index for cards
i <- 0

# initialize winnings accummulator
winnings <- 0

# now loop until all cards used
while (i < ncards) {

  # New hand needed: check to see whether at least 4 cards remain unused
  if (i <= (ncards - 4)) {

    #### Deal 2 cards to player
    # increment card index
    i <- i + 2
    p_hand <- c(cards[i-1], cards[i])

    ### Deal 2 cards to dealer
    # increment card index
    i <- i + 2
    d_hand <- c(cards[i-1], cards[i])

  } else {
    # else cards are exhausted so exit
    return(winnings)
  }

  #### logic for player's hand
  p_res <- player_h(p_hand, cards, i, p_stand)

  # update card index
  i <- p_res[2]

  # if player didn't go over 21 then apply dealer hand logic
  if (p_res[1] <= 21) {
    d_res <- dealer_h(d_hand, cards, i)

    # update card index
    i <- d_res[2]
  }

  # ------------------------------------------------------------
  # now compare player's hand to dealer's hand to find out who won

  # if player hand exceeded 21 subtract 2 from winnings
  if (p_res[1] > 21) {
    winnings <- winnings - 2

  # else if dealer went over 21 dealer has lost so add 2 to winnings
  } else if (d_res[1] > 21) {
    winnings <- winnings + 2

  # else if player hand > dealer hand add 2 to winnings
  } else if (p_res[1] > d_res[1]) {
```

```
      winnings <- winnings + 2

    # else if player hand < dealer hand subtract 2 from winnings
    } else if (p_res[1] < d_res[1]) {
      winnings <- winnings - 2
    }


  } # end while (i < ncards)
  return(winnings)
}
```

The 'main' portion of the simulation generates and shuffles a 2-deck set of cards, with each face card represented by its numeric Blackjack value. For example, jacks, queens, and kings are all treated as '10' while aces are initially treated as '11'. (NOTE: Aces can also be treated as having a value of '1' depending upon the values of other cards present in any given Blackjack hand. The programming logic for determining the value of an ace in any given hand is contained within the **sum_Cards** function defined above).

```
#############################################################################
# This is the 'main' portion of the simulation

# NOTE: Suits of cards don't really matter at all. All we really need to track is the number remaining fo

# set number of 2-deck sets to play
N <- 12

# set size of bets
bet <- 2

# set hand total for player to stand at
p_stand <- 16

# generate a deck of cards
deck <- rep(c(2:10, 10, 10, 10, 11), 4)

# create a 2 deck set of cards
two_d <- rep(deck,2)

# create a vector to store results of play
res <- data.frame(N = 1:12, winnings = numeric(N))

for (i in 1:N){
  # shuffle 2 new decks of cards
  s_decks <-sample(two_d,length(two_d), replace = FALSE)
  s_decks

  # play until 2 decks are exhausted
  res$winnings[i] <- play_bj(s_decks, bet, p_stand)
}

# create a table of winnings for each iteration
iteration <- 1:N
kable(data.frame(iteration, res$winnings))

# average the winnings and print to screen
```

```r
avg_winnings <- mean(res$winnings)
message(paste("Average Winnings per Hand = ", avg_winnings))
```

R code for 1000 iterations of simulation

```r
# set number of simulation iterations to be run
iters <- 1000

# create a vector to store results of play
res2 <- data.frame(N = 1:iters, avg_winnings = numeric(iters))

for (k in 1:iters) {
  for (i in 1:N){
    # shuffle 2 new decks of cards
    s_decks <-sample(two_d,length(two_d), replace = FALSE)
    s_decks

    # play until 2 decks are exhausted
    res$winnings[i] <- play_bj(s_decks, bet, p_stand)
  } # for i

  # save the average the winnings to the results vector
  res2$avg_winnings[k] <- mean(res$winnings)

} # for k
```

Evaluating "stand" values ranging from 11 to 20 via further simulation iterations:

```r
set.seed(123)

# create a vector to store the average winnings for various player "stand" values
playw <- data.frame(stand = 11:20, avg_winnings = numeric(10))

p_stand <- 11

for (z in 1:10) {
  res4 <- data.frame(N = 1:iters, avg_winnings = numeric(iters))

  for (k in 1:iters) {
    for (i in 1:N){
      # shuffle 2 new decks of cards
      s_decks <-sample(two_d,length(two_d), replace = FALSE)
      s_decks

      # play until 2 decks are exhausted
      res$winnings[i] <- play_bj(s_decks, bet, p_stand)
    } # for i

  # average the winnings and print to screen
  res4$avg_winnings[k] <- mean(res$winnings)

  } # for k

  # save the average winnings to the results vector
```

```
  playw$avg_winnings[z] <- mean(res4$avg_winnings)

  # increase the player's stand value
  p_stand <- p_stand + 1

} # for z

kable(playw)
```