MKTG 551-3, Homework 5

Rayyan Sayeed

August 3, 2018

1 Introduction

Here, we implement a period utility function for a consumer with dynamic inventory facing two product pack sizes. We use a dynamic programming model to compute the value function and policy function, and plot results.

2 Code

The code provided below implements the period utility function, and obtains value and policy functions using linear interpolation and 100 steps.

```
setwd("C:/Users/Rayyan Sayeed/Documents/Northwestern Spring 2018/MKTG 551-3/Homework 5")
library(dplyr, tidyr)
library(MASS)
library(ggplot2)
library(gridExtra)

##### UTILITY AND VALUE FNS
#####
#####
```

```r
# Define parameters

J = 2 # 2 pack sizes, {2,5}
I = 20 # Max inventory holding
beta = 0.99; alpha = 4; gam = 4

# Transportation cost

tau = cbind(0,0.5)

# Holding cost

cost=function(i_t){
  return(0.05*i_t)
}



# Period Utility function, consumer in state i_t

# Define p_jt matrix, price of choice of pack size j, time t
nj = c(2,5)
p_1=rbind(1.2,3)
p_2=rbind(2,5)
p_matrix = cbind(p_1,p_2)

util = function(i_t,p_vec,j){
    if(j != 0){
      return(-alpha*p_vec[j]-tau[j]+gam - cost(i_t+nj[j]-1))
    }
    else if(j == 0 & i_t >= 1 ){
      return(gam-cost(i_t-1))
    }
```

```
    else{
      return(0)
    }
  }

# Value function, weighted by prob.

vfunc.rhs = function(i_t,p_vec,j,s,vf){
  if(j != 0){
  return( util(i_t,p_vec,j) + beta*( .16*vfunc.approx(i_t-1+nj[j],1,vf)) +
      (1-.16)*vfunc.approx(i_t-1+nj[j],2,vf) )
  }
 else{return( util(i_t,p_vec,j) + beta*( .16*vfunc.approx(i_t-1,1,vf)) +
     (1-.16)*vfunc.approx(i_t-1,2,vf) )}
}

### Value function approximation by either linear or cubic interpolation

vfunc.approx = function(s,p,vf){

    # linear interpolation
    if(s >= ss[length(ss)]) {
      return(vf[length(ss),p])
    }
    else{
      s.idx.lower = floor((s/ss.grid.step)) + 1
      s.idx.upper = s.idx.lower + 1
      return(vf[s.idx.lower] +
               (s-ss[s.idx.lower])*(vf[s.idx.upper] - vf[s.idx.lower]))/ss.grid.step
    }
  }
```

```
# Dont need policy func approx, will get it from iteration later

##### SOLVE VALUE AND POLICY FNS
#####
#####

# Solve for the value and policy functions on the state space grid
vfunc.solve = function(ss){

  ## ss is state space
  ss.n = length(ss)

  # empty vfunc and pfunc current fns
  vfunc.curr = matrix(0,nrow=ss.n,ncol=2)
  pfunc.curr = matrix(0,nrow=ss.n,ncol=2)

  # With no cake, nothing to eat and no utility
  vfunc.curr[1] = 0
  pfunc.curr[1] = 0

  iter = 1
  curr.tol = 9e9

  while(curr.tol > tol.conv & iter < 100){

    # Create empty value and policy functions
    vfunc.new = matrix(0,nrow=ss.n,ncol=2)
    pfunc.new = matrix(0,nrow=ss.n,ncol=2)

    ########################
```

```
# Solve for rest of state space
for(s in 2:ss.n){
  # iterates through pack sizes
  # vfunc.rhs = function(i_t,p,j,s,vf)
  for(p in 1:2){
    pack_vec=p_matrix[,p]
    approx_j0= vfunc.rhs(ss[s],pack_vec,0,s,vfunc.curr)
    approx_j1= vfunc.rhs(ss[s],pack_vec,1,s,vfunc.curr)
    approx_j2= vfunc.rhs(ss[s],pack_vec,2,s,vfunc.curr)

    vfunc.new[s,p]=log(exp(approx_j0)+exp(approx_j1)+exp(approx_j2))
    #print(approx_j0)
    #print(approx_j1)
    #print(approx_j2)
    pfmax = which.max(c(approx_j0,approx_j1,approx_j2))
    pfunc.new[s,p]= pfmax-1
  }
}
#print(vfunc.new)
#print(vfunc.curr)
# Check convergence criterion
curr.tol = max(abs(vfunc.new - vfunc.curr))
if(!is.null(print)) {
  cat("k =", iter,
      " | norm =", round(curr.tol,6),
      " |", round(vfunc.new[c(2:3,floor(ss.n/2),(ss.n-1):ss.n)], 6), "\n")
}
if(curr.tol < tol.conv){ cat("Converged in", iter, " iterations with tol",
    curr.tol) }

# Update for the next iteration
vfunc.curr = vfunc.new
```

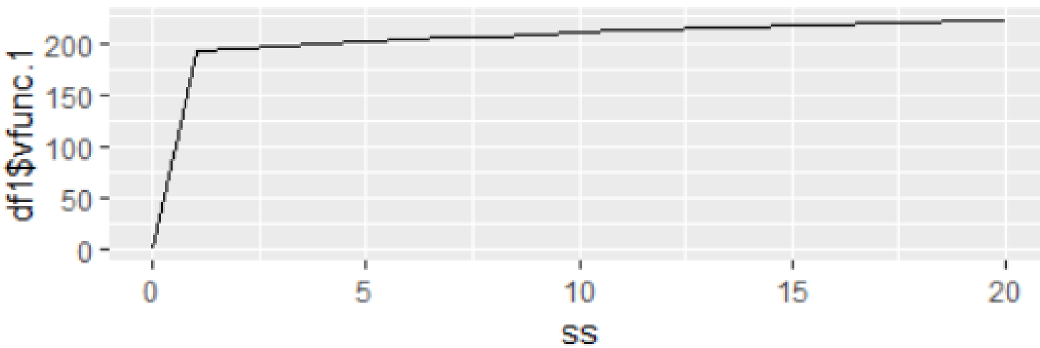
```
  pfunc.curr = pfunc.new
  iter = iter + 1
}
return(data.frame(ss=ss,vfunc=vfunc.curr, pfunc=pfunc.curr))

##### DYNAMIC DECISION SIMULATION
#####
#####

# Simulate the problem to verify the solution
vfunc.sim = function(df, s.init, T.sim=100, vfunc.curr.approx=NULL,
    pfunc.curr.approx=NULL){

  ss = df$ss
  vfunc.curr = df$vfunc
  pfunc.curr = df$pfunc

  # Initial state
  s1 = ss[s.init]



    # Law of motion
    curr.s = curr.s - c.star1
  }

  ##### POLICY FUNCTION SIMULATION
  #####
  #####





  curr.s = s1
```

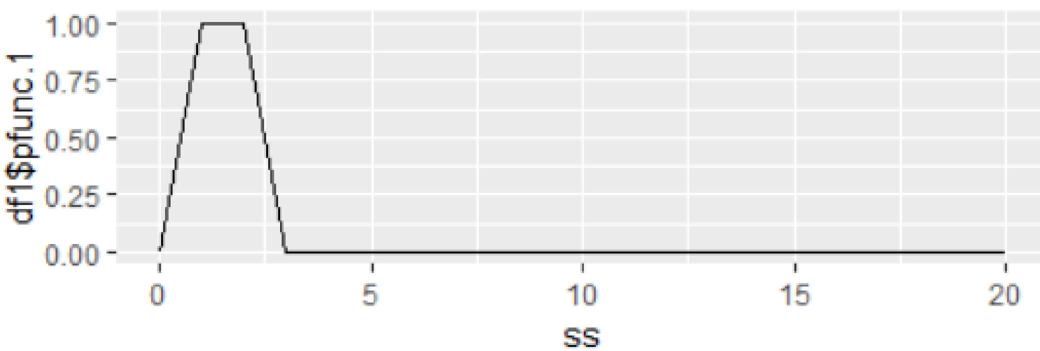
```
  for(t in 1:T.sim){
    # Using the policy function
    c.star2 = pfunc.approx(curr.s, pfunc.curr, spline.func=pfunc.curr.approx)
    state.seq2[t] = curr.s
    cons.seq2[t] = c.star2
    util.seq2[t] = util(c.star2)
    util.discounted.sum2 = util.discounted.sum2 + (beta^(t-1))*util.seq2[t]

    # Law of motion
    curr.s = curr.s - c.star2
  }

  print(paste("Initial State: s_1 =", s1))
  print(paste("Discounted sum of utils (vfunc):",
              util.discounted.sum1, "|",
              (abs(vfunc.curr[s.init]-util.discounted.sum1))))
  print(paste("Discounted sum of utils (pfunc):",
              util.discounted.sum2, "|",
              (abs(vfunc.curr[s.init]-util.discounted.sum2))))
  print(paste("Value function at s_1:       ",
              vfunc.curr[s.init], "| diff"))
}

##### LINEAR INTERPOLATION, CASE 1: STEP SIZE = 1
#####
#####



beta = 0.99
ss.grid.step = 1.0
ss = seq(from=0,to=20,by=ss.grid.step)
```

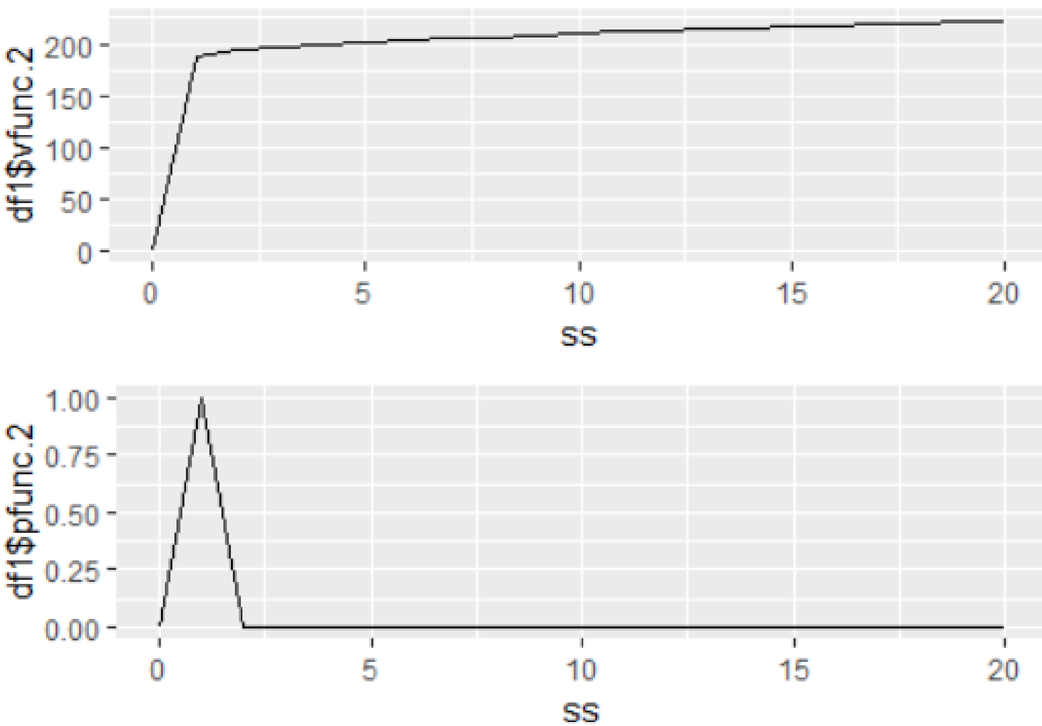
```
tol.conv = 1e-6

df1 = vfunc.solve(ss)

# Plots for pack size 2


p1_1 <- ggplot(df1, aes(x = ss, y=df1$vfunc.1)) + geom_line()
p2_1 <- ggplot(df1, aes(x = ss, y=df1$pfunc.1)) + geom_line()
grid.arrange(p1_1, p2_1)

# Plots for pack size 5

p1_2 <- ggplot(df1, aes(x = ss, y=df1$vfunc.2)) + geom_line()
p2_2 <- ggplot(df1, aes(x = ss, y=df1$pfunc.2)) + geom_line()
grid.arrange(p1_2, p2_2)
```

3 Results

The implementation above returns the following plots for the policy and value functions for each pack size.






df1$vfunc.2
200
150
100
50
0
0
5
10
15
20
ss
df1$pfunc.2
1.00
0.75
0.50
0.25
0.00
0
5
10
15
20
ss