Monetary policy and asset pricing - Assignment no.1

Riccardo Dal Cero

06/11/2022

Initial Operations

```
# Clear the variables
rm(list = ls())

# Install packages
packages <- c("matlib", "rmarkdown", "tinytex")
new.packages <- packages[!(packages %in% installed.packages()[, "Package"])]
if (length(new.packages)) install.packages(new.packages)
invisible(lapply(packages, library, character.only = TRUE))
# Load packages
library(matlib)
options(digits = 15)
```

POINT 2

We define a bisection alghorithm that max the value function which represents the following preferences: risk premium rate > 4% and risk free rate -> Epsilon

Risk computation

the risk computation function take m as input and compute the risk free / premium rate and recall the value function with such values.

```
riskComputation <- function(m,param){
    #parameters
    phi     <-  param[4]
    beta    <- 0.96
    gamma   <- param[3]
    h       <- param[1]
    l       <- param[2]
    epsilon <- 0.012

    #Generate pi Matrix
    valuesPi        <-   c(phi, epsilon, 1 - phi - epsilon, 1 / 2,
     0, 1 / 2, 1 - phi - epsilon, epsilon, phi)
    Pi              <-   matrix(valuesPi, byrow = TRUE, nrow = 3, ncol = 3)

    #a vector that contains the three states of the system
    values_states   <-   c(h,m,l)

    #ouput vector
    out <-  c()
```

```
    #stationary prob.
    st_prob <- c(0.5, epsilon, 0.5) * (1 + epsilon)

    #Computing the zero-cupon bond
    q   <-  beta * (Pi %*% (values_states ^- gamma))

    #compute risk-free rate vector
    risk_free   <-   (1 / (q)) - 1

    #compute mean risk free rate using stationary prob. as a weight
    rf <- risk_free[, 1] %*% st_prob
    as.numeric(rf)

    #risky_rate
    values_matrix <- diag(3) * (values_states^(1 - gamma))

    #Computing Pi star
    Pi_star <-  Pi %*% values_matrix

    I       <- diag(3) #diagonal matrix
    v1      <- c(1, 1, 1) #[1,1,1]

    #Computing risk premium prices
    p       <- (inv(I - beta * Pi_star)) %*% (beta * (Pi_star %*% v1))

    #Converting prices in rates

    # values in diagonal matrix
    values     <- c((1 + p) * values_states)

    #generate the diagonal matrix
    v2      <- matrix(diag(values), 3, 3)

    #generate the matrix
    v3      <- matrix((p)^(-1), 3, 3)

    #computing the matrix to convert the prices into rates
    v4      <- v3 %*% v2

    #computing the avarage risk rate using an equal weight for the three status
    rr      <- t(c(rep(1 / 3, 3))) %*% v4 %*% st_prob
    as.numeric(rr)

    #compute index
    i       <- v((rr - 1) * 100, rf * 100)

    out     <- c(rf * 100, (rr - 1) * 100, log(i))

    return(out)

}
```

Value function

Value function: given the risk premium rate = 'dummy' and 'x' = risk free rate as input return a value which is higher when x -> 1 and dummy > 4

```
v <- function(dummy, x) {

  for (t in seq_along(dummy)) {
    if (dummy[t] > 4) {
      dummy[t]  <-  2
    }
    else {
      dummy[t]  <-  1
    }
  }
  value <-  abs(1 / (1 - x)) * dummy
  return(value)
}
```

Bisection alghorithm

```
bisection <- function(param) {
    #bisection variable
    #setting the max min value for m
    max     <- 20.0000
    min     <- 0.0000
    half    <- (max - min) / 2 + min
    i       <- 0 #values from value function
    count   <- 1 #numeber of iteration
    out <-c()
    while (i < 20) {
      print(cbind("Iteration: ", count))

      #A vector which takes the first quartile and the last
      #from the set of possibile values (max-min)
      m1  <- c((max - min) / 4, (max - min) * 3 / 4) + min

      bis <- matrix(NA, nrow = 2, ncol = 4)
      #   index    m     free    p   val
      #   1       m1[1]
      #   2       m1[2]

      bis[, 1] <- as.numeric(m1)

      #recall a function to compute risk rate for m[1] and the index
      bis[1, 2:4] <- riskComputation(m1[1], param)

      #recall a function to compute risk rate for m[1] and the index
      bis[2, 2:4] <- riskComputation(m1[2], param)

      #select the one with max index(value)
      if (bis[1, 4] > bis[2, 4]) {
        #the m in the first quartile is the one with greater index(value)
        #so the m value that max the value function is in the first half
```

```
        #of the distribution
        #so now I will restrict the possibile values of m by half, selecting
        #the first half of the distribution

        max     <- half
        half    <- (max - min) / 2 + min
        print(bis[1,])
        i       <- bis[1, 4]  #output of the value function given m
        out <- bis[1,]

      } else if (bis[1, 4] < bis[2, 4]) {
        #the same as above, but we take the other half of the distrib.

        min     <- half
        half    <- (max - min) / 2 + min
        print(bis[2,])
        i       <- bis[2, 4]
        out <-  bis[2,]

      } else {
        #In case the value function return the same value for the two m
        #we have convergence so I stop the cycle

        print(bis[1, ])

        i <- 100
      }
      count <- count + 1
    }
    return(out)
}
```

Main function

```
main <- function() {
  phi     <- 0.43
  gamma   <- 2
  h       <- 1.054
  l       <- 0.982
  param   <- c(h, l, gamma, phi)
  m       <- bisection(param)
  return(m)
}
main()
```

```
##                      count
## [1,] "Iteration: " "1"
## [1]  5.00000000000000  9.00417241999373 16.37178889289230 -1.38681577765787
##                      count
## [1,] "Iteration: " "2"
## [1]  2.50000000000000  8.83878206073685 12.91190965546454 -1.36593629233859
##                      count
## [1,] "Iteration: " "3"
## [1]  1.25000000000000  8.18215972525146 10.91522811534871 -1.27845295465851
```

```
##                     count
## [1,] "Iteration: " "4"
## [1]  0.625000000000000  5.632275672601348  9.400674010092835 -0.839901072750121
##                     count
## [1,] "Iteration: " "5"
## [1]  0.312500000000000 -3.478912864698422  7.675430760854862 -0.806233172289428
##                     count
## [1,] "Iteration: " "6"
## [1]  0.4687500000000000  3.1106675009936162  8.7466048346652503
## [4] -0.0538570680813541
##                     count
## [1,] "Iteration: " "7"
## [1] 0.390625000000000 0.692032424216795 8.292627157322086 1.870907955541296
##                     count
## [1,] "Iteration: " "8"
## [1] 0.429687500000000 2.051844654152190 8.534748544013461 0.642601744304008
##                     count
## [1,] "Iteration: " "9"
## [1] 0.41015625000000 1.41597624067666 8.41798154798743 1.57027431466990
##                     count
## [1,] "Iteration: " "10"
## [1] 0.40039062500000 1.06596353460526 8.35645187549399 3.41180037618420
##                     count
## [1,] "Iteration: " "11"
## [1] 0.395507812500000 0.882117460382905 8.324836406826840 2.831213757816644
##                     count
## [1,] "Iteration: " "12"
## [1] 0.397949218750000 0.974803630028115 8.340717072183113 4.374202524135429
##                     count
## [1,] "Iteration: " "13"
## [1] 0.39916992187500 1.02057232065208 8.34860253638789 4.57695594492983
##                     count
## [1,] "Iteration: " "14"
## [1] 0.398559570312500 0.997735413448177 8.344664340085227 6.783510255095465
##                     count
## [1,] "Iteration: " "15"
## [1] 0.39886474609375 1.00916569475797 8.34663457852132 5.38543477556215
##                     count
## [1,] "Iteration: " "16"
## [1] 0.398712158203125 1.003453514999350 8.345649747575834 6.361509906772996
##                     count
## [1,] "Iteration: " "17"
## [1] 0.398635864257812 1.000595204944698 8.345157122868496 8.119751947415141
##                     count
## [1,] "Iteration: " "18"
## [1] 0.398597717285156 0.999165494438833 8.344910765100000 7.781818331373649
##                     count
## [1,] "Iteration: " "19"
## [1] 0.398616790771484 0.999880395994595 8.345033966422699 9.724471407560991
##                     count
## [1,] "Iteration: " "20"
## [1] 0.398626327514648 1.000237812044386 8.345095537824632 9.037177106322064
##                     count
## [1,] "Iteration: " "21"
```

```
## [1]  0.398621559143066  1.000059106913290  8.345064742907882 10.429309844818214
##                    count
## [1,] "Iteration: " "22"
## [1]  0.398619174957275  0.999969752177433  8.345049331690380 11.099233537883405
##                    count
## [1,] "Iteration: " "23"
## [1]  0.398620367050171  1.000014429726207  8.345057030410397 11.839367339775446
##                    count
## [1,] "Iteration: " "24"
## [1]  0.398619771003723  0.999992090997026  8.345053203041530 12.440656010933829
##                    count
## [1,] "Iteration: " "25"
## [1]  0.398620069026947  1.000003260372931  8.345055112116517 13.326816153821831
##                    count
## [1,] "Iteration: " "26"
## [1]  0.398619920015335  0.999997675687815  8.345054132950702 13.665233577685846
##                    count
## [1,] "Iteration: " "27"
## [1]  0.398619994521141  1.000000468031081  8.345054616359260 15.267878311870151
##                    count
## [1,] "Iteration: " "28"
## [1]  0.398619957268238  0.999999071859622  8.345054387866657 14.583230026646037
##                    count
## [1,] "Iteration: " "29"
## [1]  0.398619975894690  0.999999769945385  8.345054526087248 15.978096279659901
##                    count
## [1,] "Iteration: " "30"
## [1]  0.398619985207915  1.000000118988250  8.345054564271992 16.637388271306332
##                    count
## [1,] "Iteration: " "31"
## [1]  0.398619980551302  0.999999944466817  8.345054544095332 17.399432288588205
##                    count
## [1,] "Iteration: " "32"
## [1]  0.398619982879609  1.000000031727533  8.345054575346156 17.959228149274193
##                    count
## [1,] "Iteration: " "33"
## [1]  0.398619981715456  0.999999988097175  8.345054543108965 18.939637292291639
##                    count
## [1,] "Iteration: " "34"
## [1]  0.398619982297532  1.000000009912354  8.345054547747743 19.122631109818901
##                    count
## [1,] "Iteration: " "35"
## [1]  0.398619982006494  0.999999999004771  8.345054540094932 21.421194978474951
```

```
## [1]  0.398619982006494  0.999999999004771  8.345054540094932 21.421194978474951
```