The Monte-Carlo pricer for Asian options with variance reduction features

Przemyslaw Rys - Advanced R Programming @ 2018 WNE UW

May 13, 2018

Contents

Introduction

The main purpose of the implemented functions is to estimate Asian arithmetic average option by the Monte Carlo method, within the Black-Scholes model assumptions. The main idea of the Monte Carlo is to simulate many possible scenarios - underlying price processes and then calculate option payoff in each of them. The fair price is the discounted payoff expected value, under the martingale measure. The price is estimated by the average of the calculated discounted payoffs. Any price other than the fair one results in arbitrage possibility. The method precision depends on the number of simulations. Moreover, the estimated value converges to the theoretical one, when the simulated path sample size increase to infinity.

In the project implementation of the two functions has been presented. The first one follows the simple intuition, by simulating the price series and calculating the payoffs inside the *for* loop. The second function uses the more efficient *apply* function instead and contains additional features, improving the method precision. The variance reduction methods, such as the antithetic and control variates have been included, as well as the Quasi-Monte Carlo engine for the particular case of the European options.

The report is composed of the three parts. The first one contains the functions presentation and implementation in R. The main purpose of the second one is to compare precision and execution time, by the example of specified call and put options. In the last part, the advanced function code is split into the segments for the purpose of deeper code analyze, especially different implementation time comparison and find the bottlenecks.

1 Description and implementation

1.1 Main functions

1.1.1 Standard function

The standard function performs the Monte-Carlo pricing in a simple way, using *for* loop. The algorithm simulates the underlying stochastic process in the specified points and uses prices to calculate option payoff. The option price is estimated by the average discounted payoff across all simulations, generated in accordance with risk-free measure (Black-Scholes) and specified parameters, such as the interest rate and underlying price volatility. The considered function checks if the parameters are proper, by calling the external function and returns confidence interval for the estimated price at specified confidence level (*95%* by default). The default value of simulations number is *10 000*.

```
getAsianOptionPriceMC <- function(S0, r, sigma, TTM, K, n, M = NULL, type, confLevel = 0.95){
  # Verifying type.
  if(type == "call"){
    payoffFunction <- function(x) ifelse(mean(x) > K, mean(x) - K, 0)
  } else if( type == "put"){
    payoffFunction <- function(x) ifelse(K > mean(x), K - mean(x), 0)
  } else {
    stop("Wrong option type. Please set type as a 'call' or 'put'.")
  }

  # Verifying M and use default if needed.
  if(is.null(M)){
    warning("The number of Monte Carlo trajectories simulation (M) has not
been specified.\n The default value of 10 000 is used.\n")
    M <- 10000
  }

  # Verifying parameters.
  verify <- verifyMCParameters(S0, r, sigma, TTM, K, n, M, confLevel)
  if (verify$assert == 1) {
    stop("Execution has been haulted. \n")
  } else{
    n         <- verify$n
    M         <- verify$M
    confLevel <- verify$confLevel
  }

  # Predefined values.
  deltaT            <- TTM / n
  discountedPayoffs <- numeric(M)

  # Running the Monte-Carlo engine.
  for (i in 1:M)
    discountedPayoffs[i] <-
    payoffFunction(S0 * exp(cumsum((r - 0.5 * sigma ^ 2) * deltaT +
                                  sqrt(deltaT) * sigma * rnorm(n)))) *
    exp(-r * TTM)

  # Calculating price and confidence interval.
  price             <- mean(discountedPayoffs)
  confInterval      <- price + c(-1, 1) * qnorm(confLevel + (1-confLevel) / 2) *
    sd(discountedPayoffs) / sqrt(M)
  result             <- list(price, confInterval)
  names(result)      <-
    c("price", paste0("confInterval", format(100*confLevel, digits = 4), "%"))

  result
}
```

1.1.2 Advanced function

The second function has the same goal, as previous- to approximate fair option price, using Monte-Carlo simulations within Black-Scholes model framework. The main technical difference is that the method generates the random normal matrix and uses the *apply* function to transform the rows into the separated underlying process realizations. That approach should be more efficient, than the one based on loops because of the R specificity. The function has some conceptual differences as well. The antithetic variates method is used to double simulations immediately, by using the sequences opposite to the generated normal ones. Since the standard normal distribution is symmetric, hence the opposite sequences come from the same distribution. Additionally, the simulations are no longer independent of each other. There is a high negative correlation between the corresponding sequences, what lower the estimator variance, as well as doubling the number of simulations. The other variance-reduction method used in the algorithm is based on *control variables*. The highly correlated payoffs from auxiliary option (on discrete geometric average) are used to reduce the payoffs deviation from the expected value impact. In consequence, the method precision is significantly improved. The last feature is using the Halton low discrepancy sequences to generate uniformly distributed numbers in the case of the plain vanilla option. The method, called Quasi-Monte Carlo, has a big precision advantage over the standard Monte-Carlo in that case, but loss it while applying for higher dimension problems due.

```
getAsianOptionPriceMCAdv <-
  function(S0, r, sigma, TTM, K, n, M = NULL, type, confLevel = 0.95){

  # Verifying type.
  if (type == "call") {
    payoffFunction         <-
      function(x) ifelse(mean(x) > K, mean(x) - K, 0)

    auxilaryPayoffFunction <-
      function(x) ifelse(exp(mean(log(x))) > K, exp(mean(log(x))) - K, 0)

  } else if (type == "put") {

    payoffFunction         <-
      function(x) ifelse(K > mean(x), K - mean(x), 0)

    auxilaryPayoffFunction <-
      function(x) ifelse(K > exp(mean(log(x))), K - exp(mean(log(x))), 0)

  } else {
    stop("Wrong option type. Please set type as a 'call' or 'put'.")
  }

  # Verifying M and use default if needed.
  if(is.null(M)){
    warning("The number of Monte Carlo trajectories simulation (M) has not been
specified.\n The default value of 10 000 is used.\n")
```

```r
  M <- 10000
}

# Verifying parameters.
verify <- verifyMCParameters(S0, r, sigma, TTM, K, n, M, confLevel)
if (verify$assert == 1) {
  stop("Execution has been haulted. \n")
} else{
  n         <- verify$n
  M         <- verify$M
  confLevel <- verify$confLevel
}

if (n == 1) { # Plain vanilla option.
  warning("The considered option is vanilla, so the Halton series has been
          used to improve precision.\n")

  # Redefining payoff functions.
  if (type == "call") {
    payoffFunction         <- function(x) ifelse(x > K, x - K, 0)
  } else if (type == "put") {
    payoffFunction         <- function(x) ifelse(K > x, K - x, 0)
  }

  brownianMotions   <-
    generateGeometricBrownianMotionsEnd(S0, r, sigma, TTM, M)

  discountedPayoffs <-
    payoffFunction(brownianMotions) * exp(-r * TTM)

  price             <-
    mean(discountedPayoffs)

  confInterval      <-
    price + c(-1, 1) * qnorm(confLevel + (1-confLevel) / 2) *
    sd(discountedPayoffs) / sqrt(M)

} else{ # Stricte Asian option.
  brownianMotions           <-
    generateAntitheticGeometricBrownianMotions(S0, r, sigma, TTM, K, n, M)

  discountedPayoffs         <-
    apply(brownianMotions,
          1,
          FUN = function(x) payoffFunction(x)) * exp(-r * TTM)

  auxilaryDiscountedPayoffs <-
    apply(brownianMotions,
          1,
          FUN = function(x) auxilaryPayoffFunction(x))  * exp(-r * TTM)

  if(type == "call"){
    auxilaryPrice <-
```

```
        calculatePriceAsianGeometricDiscreteCall(S0, r, sigma, TTM, K, n)
    } else {
      auxilaryPrice <-
        calculatePriceAsianGeometricDiscretePut(S0, r, sigma, TTM, K, n)
    }

    theta           <-
      -cov(discountedPayoffs, auxilaryDiscountedPayoffs) /
      var(auxilaryDiscountedPayoffs)

    modifiedPayoffs <-
      discountedPayoffs + theta * (auxilaryDiscountedPayoffs - auxilaryPrice)

    price           <-
      mean(modifiedPayoffs)

    confInterval <-
      price + c(-1, 1) * qnorm(confLevel + (1-confLevel) / 2) *
      sd(modifiedPayoffs) / sqrt(M)
  }

  result            <- list(price, confInterval)
  names(result)     <-
    c("price", paste0("confInterval", format(100*confLevel, digits = 4), "%"))
  result
}
```

1.2 Random numbers and stochastic processes generation module

The module is composed of functions simulating the stochastic processes and implementing the Halton low discrepancy sequences for purposes of the Monte Carlo pricing. Functions use build-in normal random numbers generator *rnorm* and *runif.halton* from *fOptions* package. The first function- *generateGeometricBrownianMotions* simulates the Geometric Brownian Motions, used to model the underlying price process in the Black-Scholes model. The second one, called *generateAntitheticGeometricBrownianMotion* simulates the Geometric Brownian Motions in the same way, as the previous one, but additionally produces new trajectories from the same distribution, using the *antithetic variates*. Therefore, the function returns two times more simulations and provides the lower variance estimator in further Monte Carlo application.

```
generateGeometricBrownianMotions <- function(S0, r, sigma, TTM, K, n, M){
  deltaT <- TTM / n
  t(apply(matrix(rnorm(M*n), M, n),
          1,
          function(x) S0 * exp(cumsum((r - 0.5 * sigma ^ 2) *
                                        deltaT + sqrt(deltaT) * sigma * x))))
}

generateAntitheticGeometricBrownianMotions <- function(S0, r, sigma, TTM, K, n, M){
  deltaT <- TTM / n
  normM  <- matrix(rnorm(M*n), M, n)

  t(apply(rbind(normM, -normM),
          1,
          function(x) S0 * exp(cumsum((r - 0.5 * sigma ^ 2) *
                                        deltaT + sqrt(deltaT) * sigma * x))))

}
```

The European plain vanilla option is the special case of Asian option with only one period. Therefore, the presented main function could be used to price that option. For sake of precision and computation time, the Halton deterministic sequences have been used to simulate the sample from a uniform distribution and then transformed by the *Box-Muller* method to sample similar to the normally distributed one. The numbers are not random or even pseudo-random, but purely deterministic - however, due to the low discrepancy, the (Quasi) Monte Carlo estimator, based on it, converges faster. The function, called *transformBoxMuller* transforms two numbers from the uniform distribution on $[0, 1]$ into the ones from standard normal distribution, and the second function, called *generateGeometricBrownianMotionsEnd* generate the ending points of the Black-Scholes simulations, from the Halton sequence.

```
transformUniformToNormalByBoxMuller <- function(u1, u2){
  r     <- sqrt(-2 * log(u1))
  theta <- 2 * pi * u2

  c(r * cos(theta), r * sin(theta))
}

generateGeometricBrownianMotionsEnd <- function(S0, r, sigma, TTM, n, M){
  unifQMC   <- runif.halton(ceiling(M/2), 2, 1)
  normQMC   <- as.numeric(apply(unifQMC,
                                1,
                                FUN = function(x) transformBoxMuller(x[1], x[2])))[1:M]
}
```

1.3 The control variates module

The control variates feature, implemented in the advanced version of the function, are based on the geometric discrete average Asian option. In consequence we need to calculate the price of that auxiliary option by the closed-form formula.

```
calculatePriceAsianGeometricDiscreteCall <- function(S0, r, sigma, TTM, K, n){
  b1 <- (log(S0 / K) + (r - 0.5 * sigma ^ 2) * TTM * (n + 1) / (2 * n) +
           sigma ^ 2 * TTM * (n + 1) * (2 * n + 1) / (6 * n ^ 2)) /
    ((sigma / n) * sqrt(TTM * (n + 1) * (2 * n + 1) / 6))

  b2 <- (log(S0 / K) + (r - 0.5 * sigma ^ 2) * TTM * (n + 1) / (2 * n)) /
    ((sigma / n) * sqrt(TTM * (n + 1) * (2 * n + 1) / 6))

  exp(-r * TTM + (r - 0.5 * sigma ^ 2) * TTM * (n + 1 ) / (2 * n) +
               sigma ^ 2 * TTM * (n + 1) * (2 * n + 1)/ (12 * n  ^ 2)) *
    S0 * pnorm(b1) - exp(-r * TTM) * K * pnorm(b2)

}

calculatePriceAsianGeometricDiscretePut <- function(S0, r, sigma, TTM, K, n){
  b1 <- (log(S0 / K) + (r - 0.5 * sigma ^ 2) * TTM * (n + 1) / (2 * n) +
           sigma ^ 2 * TTM * (n + 1) * (2 * n + 1) / (6 * n ^ 2)) /
    ((sigma / n) * sqrt(TTM * (n + 1) * (2 * n + 1) / 6))

  b2 <- (log(S0 / K) + (r - 0.5 * sigma ^ 2) * TTM * (n + 1) / (2 * n)) /
    ((sigma / n) * sqrt(TTM * (n + 1) * (2 * n + 1) / 6))

  exp(-r * TTM) * K * pnorm(-b2) -
           exp(-r * TTM + (r - 0.5 * sigma ^ 2) * TTM * (n + 1 ) / (2 * n) +
      sigma ^ 2 * TTM * (n + 1) * (2 * n + 1)/ (12 * n  ^ 2)) * S0 * pnorm(-b1)

}
```

1.4 The parameters verification module

The considered module is composed of one function, used in both versions of Monte-Carlo pricer. The function checks if the assumptions of Black-Scholes model are satisfied and the Monte-Carlo parameters are set properly. Function output the *assertVar* variable, determining if the main program should be stopped and the values of *n*, *M*, *confLevel* parameters, which could be changed to meet requirements in the verification process.

```
verifyMCParameters <- function(S0, r, sigma, TTM, K, n, M, confLevel){
  assertVar <- 0
  if (S0 <= 0) {
    message("Initial underlying price need to be positive, due to the
         Black-Scholes model assumptions.\n
         Please call function again, with the proper value.\n")
    assertVar <- 1
  }

  if (r <= 0)
    warning("The interest rate (r) is negative.\n.")

  if (sigma < 0) {
    message("The underlying price volatility (sigma) need to be positive. \n
         Please call function again, with the proper value.\n")
    assertVar <- 1
  }

  if (sigma == 0)
    warning("The underlying price is deterministic, due to the zero volatility
            (sigma).")

  if (TTM <= 0) {
    message("Time to maturity (TTM) need to be positive. \n
         Please call function again, with the proper value.\n")
    assertVar <- 1
  }

  if (n != round(n)) {
    warning(paste0("The number of periods for average purposes has been
                   rounded to", round(n), ".\n"))
    n <- round(n)
  }
  if (n < 1) {
    message("The number of periods for average purposes (n) need to higher
or equal to 1. \n
         Please call function again, with the proper value.\n")
    assertVar <- 1
  }


  if (M != round(M)) {
    warning(paste0("The number of Monte Carlo iterations (M) has been rounded to ", round(M), ".\n"))
    M <- round(M)
  }
  if(M <= 1){
    message("The number of Monte Carlo iterations (M) need to be higher or equal to 1. \n
         Please call function again, with the proper value.\n")
    assertVar <- 1
  }

  if (confLevel < 0.5 | confLevel > 1) {
    warning("The confidence level should be in (0.5, 1). The default value of 95% has been used. \n")
    confLevel <- 0.95
  }

  results        <- list(assertVar, n, M, confLevel)
  names(results) <- c("assert", "n", "M", "confLevel")
  return(results)
}
```

2 The functions comparison

The functions will be compared in terms of the computation time and results stability for the exemplary call and put options with the following parameters:

- initial price *S0* - 50,
- risk-free interest rate r - 0.02,
- underlying volatility *sigma* - 0.3,
- time to maturity TTM - 1.5 (years),
- strike K - 40,
- average periods n - 100,
- Monte Carlo loops M - 10 000 (default value).

Figure 1: The price and time of MC functions - call option.

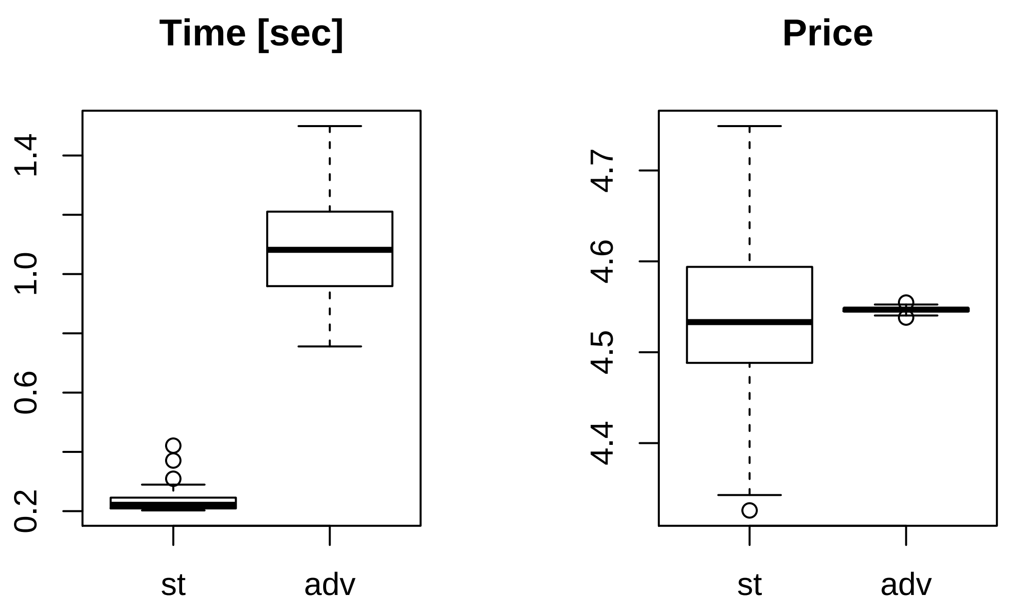
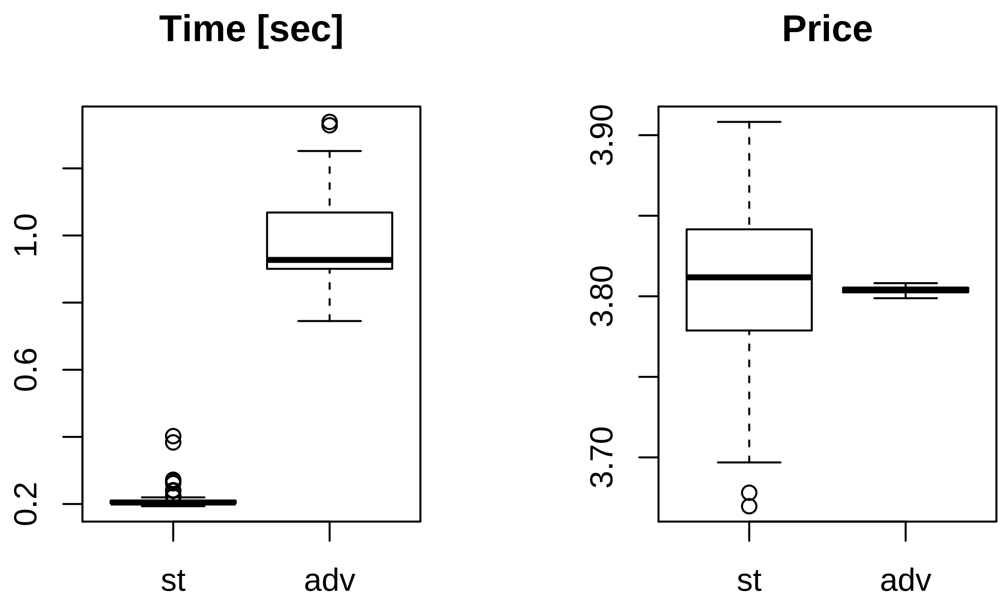


Figure 2: The price and time of MC functions - put option.



The advanced version has precision incomparably higher, than the standard one. However, it cannot be fairly compared, because of the big execution time difference. Consequently, the advanced method seems to be more precise, but far slower than the algorithm without additional features. To perform the better comparison, the number of Monte Carlo simulations in the standard method will be *4.74* times higher than in the advanced method. Thus, methods should have similar execution time and therefore the comparison of precision could be made.

Figure 3: The price and time of MC functions - call option (normalized by a number of iterations).

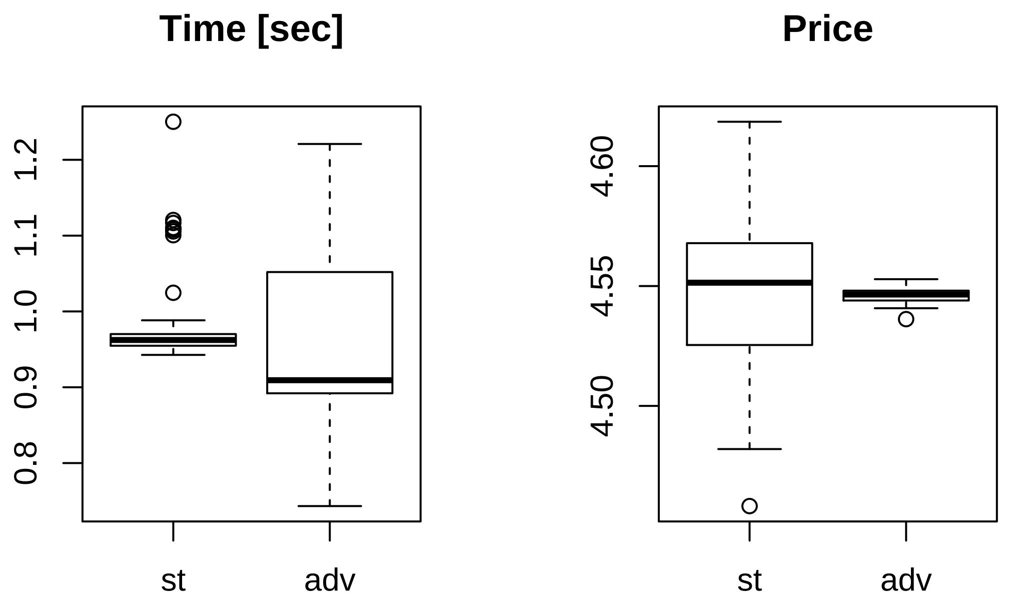
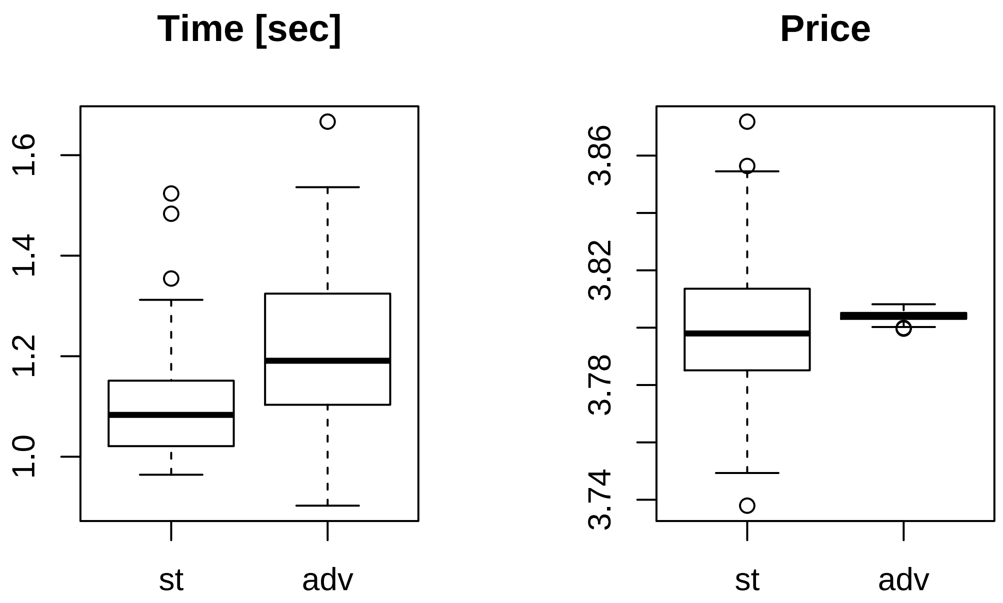


Figure 4: The price and time of MC functions - put option (normalized by a number of iterations).



In both cases, the advanced method provides higher results precision, than the standard one in similar computation time. For the call option the confidence interval of price obtained by the standard method in *47 400* iterations is $[4.480297, 4.615259]$, when the advanced method in *10 000* iterations gives results with confidence interval *10* times narrower - $[4.535946, 4.549997]$ (both for *95%* confidence level). Analogously, for the put option, the standard method confidence interval is $[3.747623, 3.838595]$, while the advanced method confidence interval is $[4.535946, 4.549997]$. The standard Monte Carlo pricer allows the results differ from the real value, by up to the *0.091* and the advanced one, up to the *0.014*. The precision advantage of the second method is irrefutable, despite the lower number of simulations.

The next comparison will be provided for the European options - with $n = 1$, because in that case the advanced function uses Halton sequence feature. The option characteristics will be the same as for the previous one, except the *n.* The number of iterations in both methods will be the same and equal to *10 000.*

Figure 5: The price and time of MC functions - call European option.

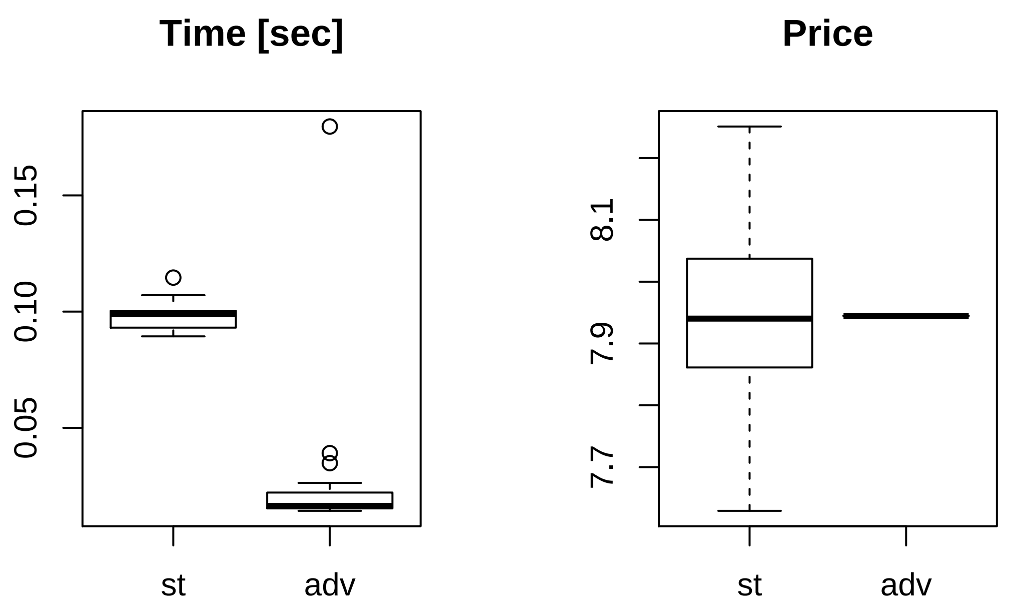
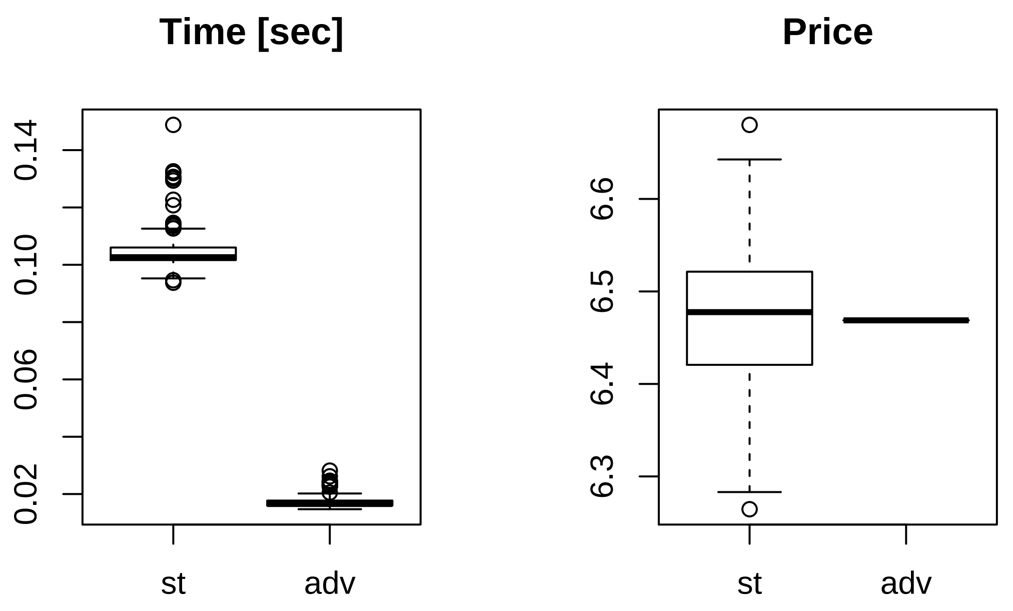


Figure 6: The price and time of MC functions - put European option.



In average, the Quasi-Monte Carlo is five times faster than the regular Monte Carlo, so in an analogous way as previously, the computations will be repeated for the different number of simulation, implying similar computation time for both methods. From now on, the M parameter for Quasi-Monte Carlo is equal to *53 750*.

Figure 7: The price and time of MC functions - call European option (normalized number of iterations).

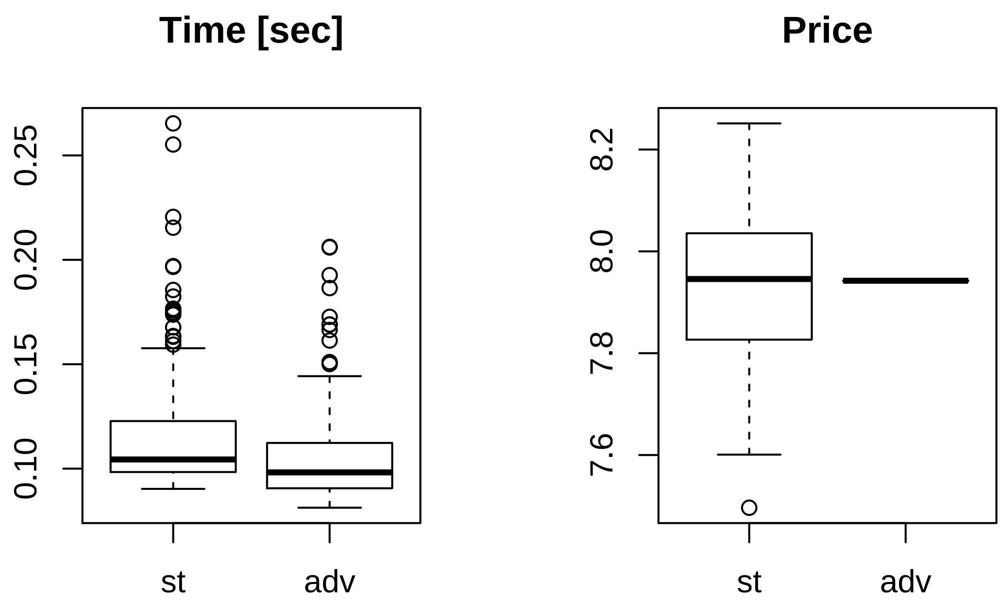
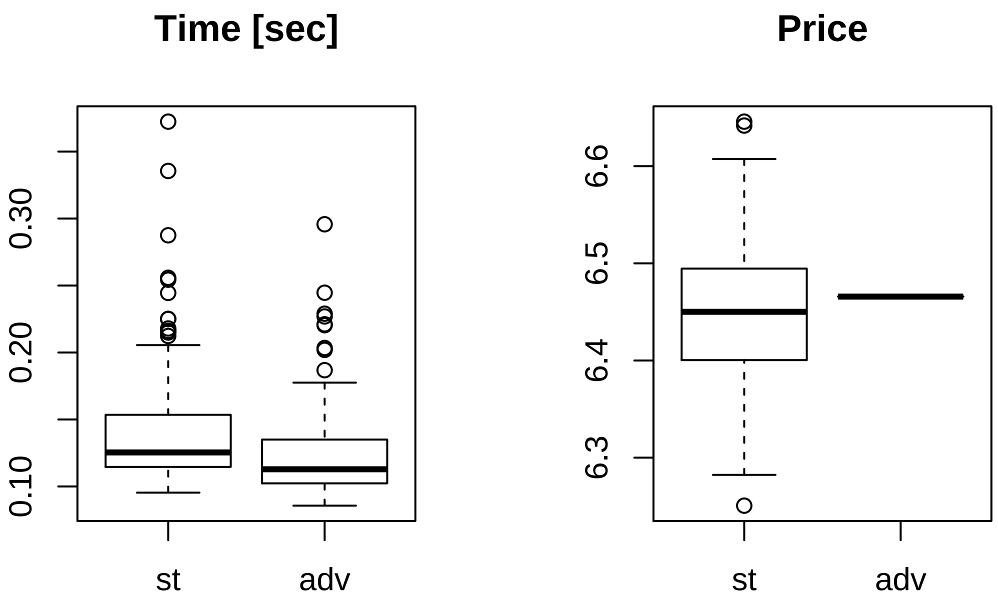


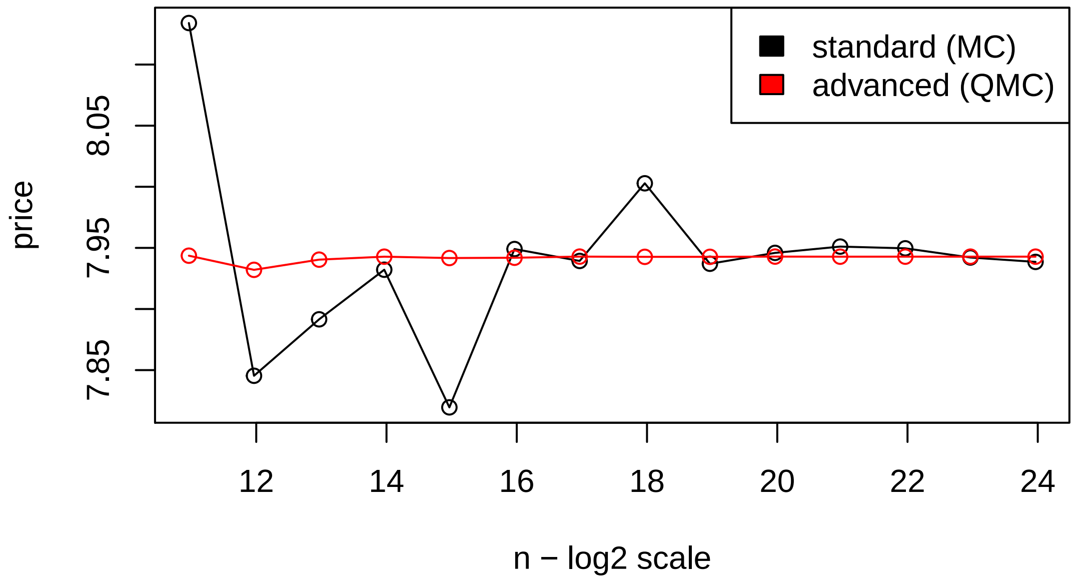
Figure 8: The price and time of MC functions - call European option (normalized number of iterations).



It is worth notice, that the Quasi-Monte Carlo is deterministic, so the variance of prices generated by the independent executions is equal to zero. However, the results are the only estimator of the theoretical value, thus the method precision level cannot be observed on boxplot. With the normalized numbers of iterations, both method execution time is comparable. For considered call option the Quasi-Monte Carlo results in price significance interval $[7.825249, 8.059376]$, when the random execution of standard Monte Carlo in $[7.468791, 8.001387]$. The advanced method has over 2 times narrower confidence interval (with significance level *95%*). For the corresponding put option advanced method interval is $[6.396814, 6.534700]$, when the one for a standard method is more than 2 times wider - $[6.330829, 6.651643]$.

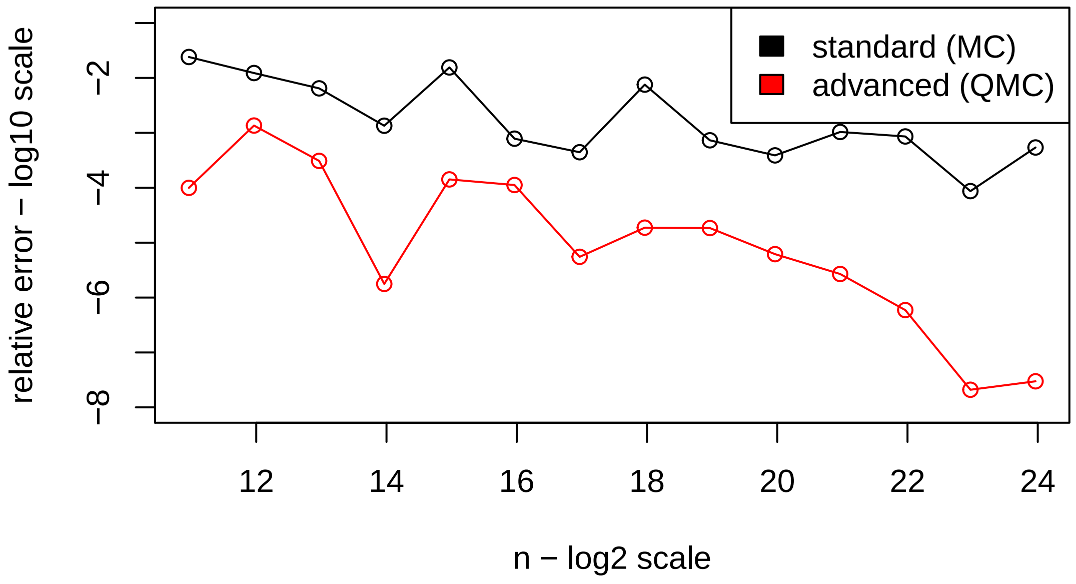
The next, crucial property of Quasi-Monte Carlo is higher convergence rate. While increasing number of simulations, the price estimator converges to fair price much faster, than the corresponding Monte Carlo one. That property could be verified by calculating prices for the sequence of different increasing M. The pricing procedure will be conducted for the call option considered before with $M \in \{500 * 2^n : n \in \mathbb{N} \cap [1, 15]\}$.

Figure 9: The estimated European call option price in accordance with the number of simulations.



One can see, that the Quasi-Monte Carlo (the one provided by the advanced method) has higher stability and its results converge to the limit faster, than the ones from the standard Monte Carlo. Since there is a closed-form formula for European options, hence one can analyze the relative estimator error. The formula implemented for discrete geometric average Asian option will be used to price European option, treated as a special case of Asian, with *n=1* periods.

Figure 10: The relative error of European call option price estimator in accordance with the number of simulations



The Quasi-Monte Carlo seems to have not only strong time but also a precision advantage over the classic method. It is worth to remind, that theoretically, Quasi-Monte Carlo errors are asymptotically equal to $\frac{1}{n}$, when the error of standard Monte Carlo to $\frac{1}{\sqrt{n}}$. Therefore, increasing the number of Monte Carlo iterations *100* times results in approximately *10* times lower error of the standard method estimator, but circa *100* times lower error of the advanced one. It is worth to remind, that the antithetic and control variates features are useless for the estimating European option price by Quasi-Monte Carlo. The Halton sequence already has antithetic variates for the most of numbers, so there is no need to generate it one more time. The control variates used in the specified form cannot be used due to the equivalence of the arithmetic and geometric options for n=*1*, where both are just the European one. However, the different control variate, such as the underlying price at maturity could be used.

3 Time analyses of the advanced function

The advanced function is composed of the three modules:

- parameters verification,
- geometric Brownian motion simulations,
- the payoff calculations and estimating price with a confidence interval.

The time of each module execution has been calculated by the *benchmark* function from package *rbenchmark*.

Table 1: The computation time comparison of the advanced function modules

test	replications	elapsed	relative	user.self	sys.self	user.child	sys.child
verify	100	0.0	1.00	0.00	0.00	0	0
simulate	100	37.8	12600.33	37.60	0.02	0	0
rest	100	53.2	17734.33	53.22	0.00	0	0

The computation time of the parameters verification module is negligible, compared to the other, more complex modules, which shares in time are equal to *41.54%* for the simulation part, and respectively *58.46%* for the further calculations. Therefore, the calculations, including payoffs for every simulated path and the control variable based variance reduction, are the slowest part of the program. However, it is worth to check the reasons - if the control variates module is expensive in time?

Table 2: The computation time comparison of the advanced function calculations with and without control variates

test	replications	elapsed	relative	user.self	sys.self	user.child	sys.child
without control var.	100	59.158	2.642	58.980	0.016	0	0
with control var.	100	22.393	1.000	22.296	0.008	0	0

According to the results, including the control variates makes the execution time of that part approximately *3.5* times longer. Nevertheless, the price estimation based on payoffs calculations takes a long time too.

The first function, called *standard* uses the simple loop to generates the price series, when the advanced one uses the *apply* function to perform the same operation, but with additional antithetic variates feature. The comparison will be conducted for the loops, single *apply* and *apply* with the antithetic module.

Table 3: The computation time comparison of the three ways of generating geometric Brownian motion

test	replications	elapsed	relative	user.self	sys.self	user.child	sys.child
apply	100	17.471	1.000	17.476	0.000	0	0
apply with anti	100	33.267	1.904	33.252	0.024	0	0
for loop	100	21.251	1.216	21.248	0.004	0	0

The *apply* functions allow to reduce computation time by more than *15%*, but using the antithetic variates results in time *90%* longer. As has been noticed before, the precision improvement caused by using antithetic variates is significant and fully compensate the higher execution time.

For the European option case the Halton sequence has been used (so-called Quasi-Monte Carlo method). The following table presents the comparison of the Halton sequence and standard generator time efficiency. Because the simple version uses the loops, that method will be considered too.

Table 4: The computation time comparison of the generators based on the Halton sequence and standard random numbers generators in vectorized and loop-based versions

test	replications	elapsed	relative	user.self	sys.self	user.child	sys.child
halton	100	1.370	1.000	1.372	0	0	0
for loop	100	9.939	7.255	9.940	0	0	0
vectorized	100	1.780	1.299	1.780	0	0	0

The advantage of vectorization is clearly visible here. Moreover, the Halton sequence based generator run faster, than the other ones. Probably it is caused by the low discrepancy sequences specificity. The generator is not trying to simulate random behavior, but rather to produce the sequences covering the unit interval in a regular way.