
Universität Zürich[UZH]

# Assignment 2, due April 24, 2023

## Topics in Advanced Time Series Econometrics

Prof. Dr. Marc Paolella

presented by

**Migle Kasetaite: 21-733-779**

**Kilian Sennrich: 18-051-060**

**Abstract**

This is our submission for Assignment 2 of the lecture *Topics in Advanced Time Series Econometrics*. In the first part, a simulation study is conducted, where three methods for parameter estimation in the MA(q) model are compared. The second part holds a simulation study for investigating the use of AIC and BIC for parameter selection in the ARMA model.

# 1 Comparison of different moving average estimation procedures

The estimation of MA models presents numerous challenges in practical applications. Maximum likelihood estimation methods have encountered convergence issues, often resulting in estimated zeros that fall precisely on the unit circle. This has led to the exploration of alternative robust algorithms for determining MA parameters, including constrained optimization of the likelihood function, and various other attempts to derive approximate maximum likelihood estimators for MA models. However, nonlinear maximum likelihood estimators have been known to fail to converge under certain circumstances. To date, no definitive consensus has been reached in the literature regarding the preferred method for MA estimation.

## 1.1 Durbins Method

The old approach of Durbin (1959) to estimating MA models involves two stages of linear estimation that replace the non-linear estimation problem. In the first stage, $k$ AR parameters are estimated from the time series data. In the second stage, the MA parameters are computed using the estimated AR parameters' lagged products. The theoretical equivalence of AR($\infty$) and MA($q$) processes forms the basis for Durbin's MA method. Therefore, optimal estimates for $b$ can be achieved with $k \to \infty$. Estimating an AR($\infty$) model is not feasible. This is why a finite AR($k$) model is estimated in practice, where $k$ can be chosen arbitrarily. With increasing k, the computational complexity of the Durbin method increases exponentially. Reasearchers suggest $k$ to be dependent on the sample size (Mentz, 1977). Large k does not necessarily lead to better estimates of $k$. Therefore, for ideal results, k must be dealt with as a hyper parameter (Broersen, 2009).
Durbin (1959) uses the fact that an MA($p$) process

$$x_t = \epsilon_t + \beta\epsilon_{t-1} \quad (t = \mathbf{1}, ..., n) \tag{1}$$

can be rewritten as $x_t + \alpha_1' x_{t-1} + \alpha_2' x_{t-2} + \ldots = \epsilon_t,$, an infinite AR process. He argues, that the infinite AR process can be approximated with $k$, for sufficiently large $k$:

$$x_t + \alpha_1 x_{t-1} + \ldots + \alpha_k x_{t-k} = \epsilon_t \tag{2}$$

In an MA(1) model, the parameter $b$ as an estimate of $\beta$ for an MA(1) model can the calculated as:

$$b = -\frac{\sum_{i=0}^{k-1} a_i a_{i+1}}{\sum_{i=0}^{k} a_i^2}. \tag{3}$$

The code for Durbins MA(1) model can be found in Code-Section 1.
Durbin generalizes the one parameter case for MA($q$) models as follows:

$$\begin{bmatrix} b_1 \\ \\ b_2 \\ \vdots \\ b_h \end{bmatrix} = - \begin{bmatrix} \sum_{i=0}^{k-1} a_i a_{i+1} \\ \sum_{i=0}^{k-2} a_i a_{i+2} \\ \vdots \\ \sum_{i=0}^{k-h} a_i a_{i+h} \end{bmatrix}^T \begin{bmatrix} \sum_{i=0}^{k} a_i^2 & \sum_{i=0}^{k-1} a_i a_{i+1} & \sum_{i=0}^{k-2} a_i a_{i+2} & \dots & \sum_{i=0}^{k-h+1} a_i a_{i+h-1} \\ \sum_{i=0}^{k-1} a_i a_{i+1} & \sum_{i=0}^{k} a_i^2 & & & \vdots \\ \vdots & & & & \vdots \\ \sum_{i=0}^{k-h+1} a_i a_{i+h-1} & \dots & \dots & \dots & \sum_{i=0}^{k} a_i^2 \end{bmatrix}^{-1} . \quad (4)$$

The code for this linear system can be found in Code-Section 2.
Various algorithms have been devised for AR estimation. The Yule-Walker (YW) AR method employs estimates of the autocorrelation function based on lagged-products. It is known to be biased. This can lead to significant estimation error. Therefore, Broersen (2009) proposes the use of the Burg estimator. The Burg estimator works by iteratively estimating the MA coefficients of the model while keeping the AR coefficients fixed. At each iteration, the MA coefficients are estimated by minimizing a weighted sum of the squared errors between the observed data and the model's predicted values. The weights used in the optimization are determined by the estimated AR coefficients from the previous iteration.

## 1.2 Paolellas Approximation Method

Paolella (2018) presents a computationally efficient approximate method to estimating the parameters $b$ of an MA model. Similar to Durbins method, Paolellas approximation method starts by estimating an AR($q^*$) model, where $q^*$ is a function of the sample size $T$. Other than Durbins method, Paolella (2018) then uses the residuals $\hat{U}_t, t = q^* + 1, \dots, T$ to compute the model

$$Y_t - \widehat{U}_t = \sum_{i=1}^{p} a_i Y_{t-i} + \sum_{j=1}^{q} b_j \widehat{U}_{t-j} + \xi_t, \quad t = q^* + 1 + q, \dots, T. \quad (5)$$

"It is noteworthy that method [...] is far faster than (certainly exact, but also conditional) maximum likelihood." (p.320, Paolella, 2018) The code for Paolellas approximation method can be found in Code-Section 4.

## 1.3 Hannan-Rissanen Method

The Hannan-Rissanen (HR) algorithm is a statistical method used for estimating the parameters of an AR model, but it can also be used to estimate the parameters of an MA model. In the case of an MA(1) model, the HR algorithm works as follows:

1. Choose an initial guess for the MA(1) coefficient $b$ and estimate $\sigma^2$
2. Compute the residuals of the model using $b$ and $\sigma^2$
3. Estimate $b$ and $\sigma^2$ using the residuals from step 2
4. Repeat steps 2 and 3 until $b$ and $\sigma^2$ converge.

In practice, this algorithm can be computationally intensive, especially for large $T$. The code for the Hannan-Rissanen method can be found in Code-Section 5.

## 1.4 Simulation Study

A simulation study was conducted in order to compare three different estimation procedures. The study was sectioned into 3 parts:

1. Estimation of $\hat{b}$ in the MA(1) model for $b \in \{-0.9, -0.8, \ldots, 0.8, 0.9\}$.
2. Estimation of $\hat{b}_1$ and $\hat{b}_2$ in the MA(2) model.
3. Parametrically bootstrapping Confidence Intervals (CIs) in the MA(1) model around $b = 0.5$.

### 1.4.1 Estimation of $\hat{b}$ in the MA(1) model for $b \in \{-0.9, -0.8, \ldots, 0.8, 0.9\}$

A simulation study with $N = 1000$ simulations with $b \in \{-0.9, -0.8, \ldots, 0.8, 0.9\}$ was conducted for each of the three methods: Durbins method, Paolellas approximation method and Hannan-Rissanen method. For that purpose, an MA(1) process was simulated with $T = 100$ observations. The code for the simulation study can be found in Code-Section 6.
The results are presented in graphical form in Figures 1 (Durbin method), 2 (Paolella approximation method) and 3 (Hannan-Rissanen method). The Durbin method under-performs the other two methods. Various authors had reported, that the Durbin method doesn't perform well in practice (e.g. Stoica, 2000). However, Broersen (2009) has pointed out that the bad performance in previous studies was due to the wrong chosen number $k$ of estimated AR parameters. Broersen goes on to describe an algorithm on how to find the optimal $k$:

1. Compute the AR models of orders $0, 1, \ldots, N/2$ with Burg's method. The highest AR model order may be limited to 1000 for $N > 2000$.
2. Select the AR order $k$ with the minimum of the order selection criterion CIC.
3. Take as the intermediate AR order $M_{sw} = 2k + q$.
4. Use (3) or (4) to compute the MA(q) parameters.

From Broersens algorithm it is evident that the quality of the parameter estimation is dependent on $k$, which is dependent of the number of observations $T$. This justifies our observation that the Durbin method does not perform as well as the Paolella approximation method nor the Hannan Rissanen method, as we fixed $k = 10$.
Paolellas approximation method surprisingly delivered very consistant and stable results with $\hat{\sigma}_b^2$ being consistent over all true $b$ and $\hat{\mu}_b$ lying consistently close to the true parameter for *any* $b$. For $b$ close to $|1|$, Paolellas method seemed to produce even better results as the Hannan-Rissanen method, which is evident from comparing $\hat{\sigma}_b^2$ in Figures 2 and 3.

| | $\bar{x}$-Runtime | $\hat{\sigma}$-Runtime |
|---|---|---|
| Durbin Estimation | 9 min 30 s | 7.91 s |
| Paolella Estimation | 7.85 s | 3.90 s |
| Hannan Rissanen Estimation | 11 min 52 s | 6.42 s |

Table 1: $\bar{x}$ and $\hat{\sigma}$-Runtimes of the simulation study per estimation method computed over $N = 14$ runs of the simulations.

| k | $\bar{x}_{b_1}$ | $\bar{x}_{b_2}$ | Time |
|---|---|---|---|
| 10 | -0.809308 | 0.051698 | 22 s |
| 20 | -0.717379 | -0.049528 | 1 min 43 s |
| 30 | -0.683727 | -0.069623 | 3 min 42 s |
| 35 | -0.676013 | -0.078139 | 15 min 3 s |

Table 2: Mean estimates for $[\hat{b}_1, \hat{b}_2]^T$ for increasing $k$, with T = 100. It seems that with increasing k, $[\hat{b}_1, \hat{b}_2]^T$ converge to true $[b_1, b_2]^T$. Computation times increase exponentially with k.

Computation Times for all three methods were assessed. For that, the above described simulation study was run 14 times for each of the three methods. The results are displayed in Table 1. Computation Times of the tree methods differ significantly as evident from Table 1. Paolellas Approximation Method grossly outperforms the other two methods in terms of computation time. Durbins method and Hannan-Rissanens method seem to perform at approximately computation time.

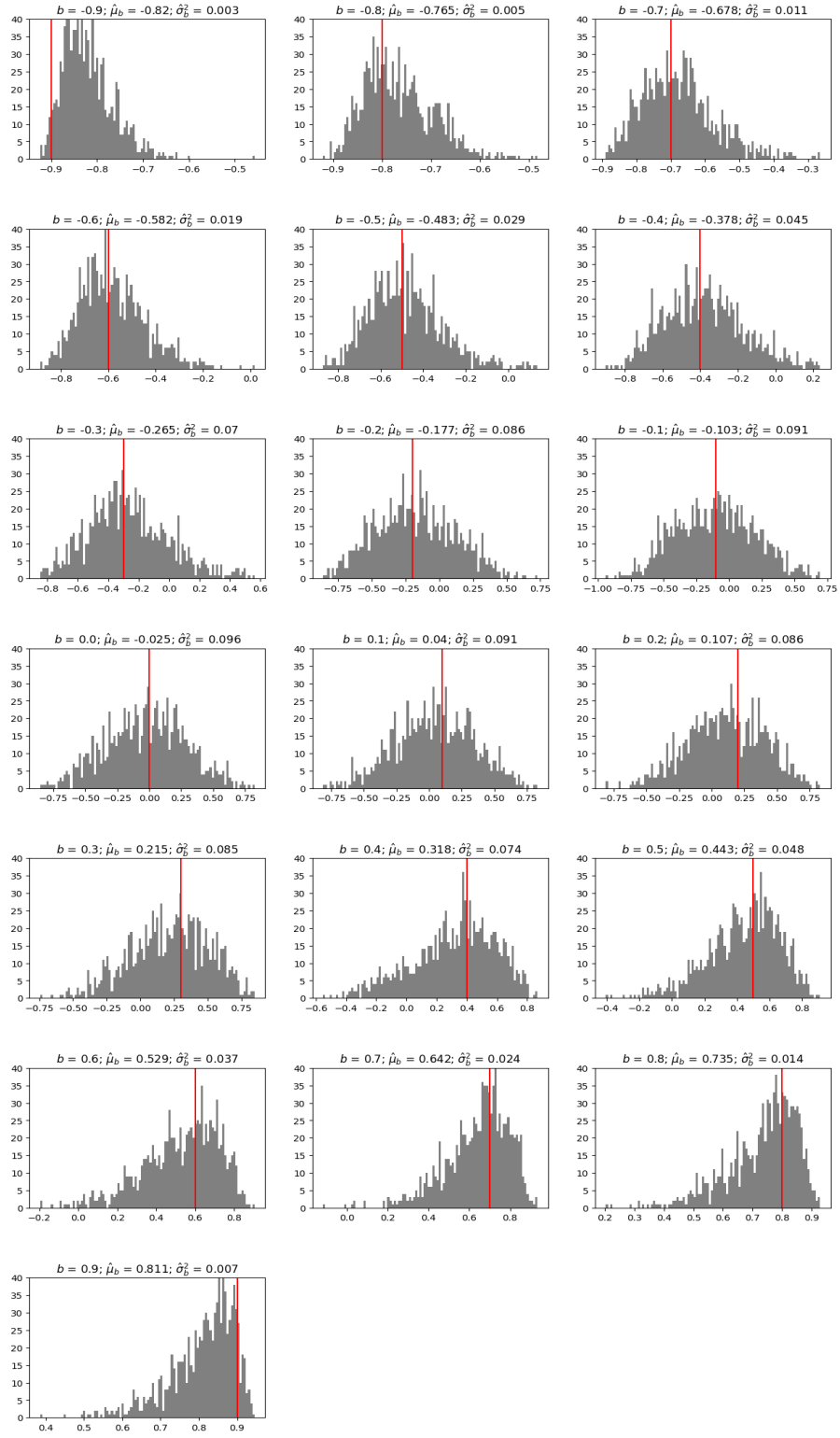


Figure 1: Results for the Durbin Method: The parameter estimation works well for $b$ close to 0, but underestimates the true parameter for b close to $|1|$. The red line indicates the true parameter $b$, the histogram displays 1000 estimates of $b$

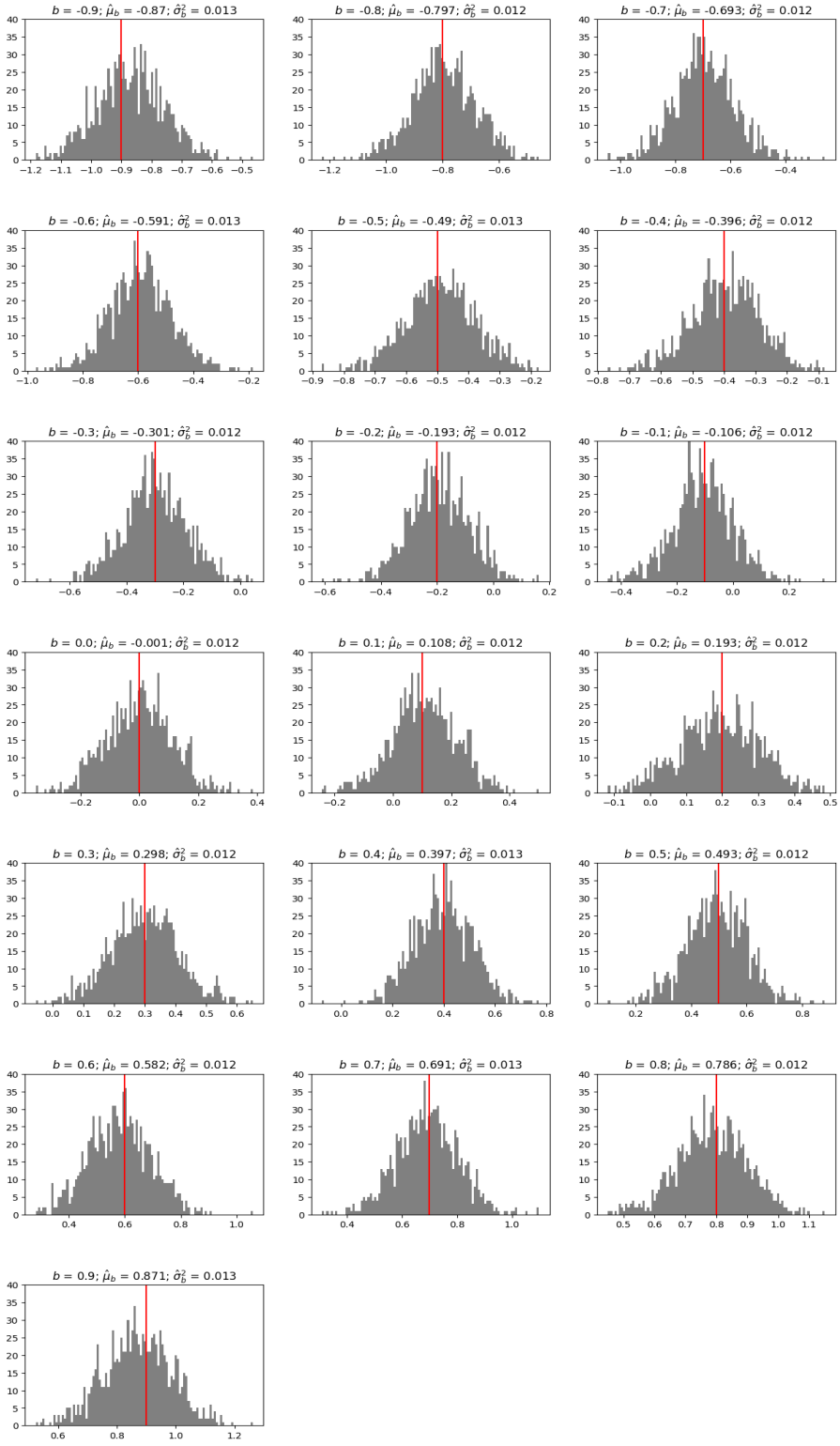


Figure 2: Results for the Paolella approximation method: The parameter estimation works well for any $b$. The red line indicates the true parameter $b$, the histogram displays 1000 estimates of $b$

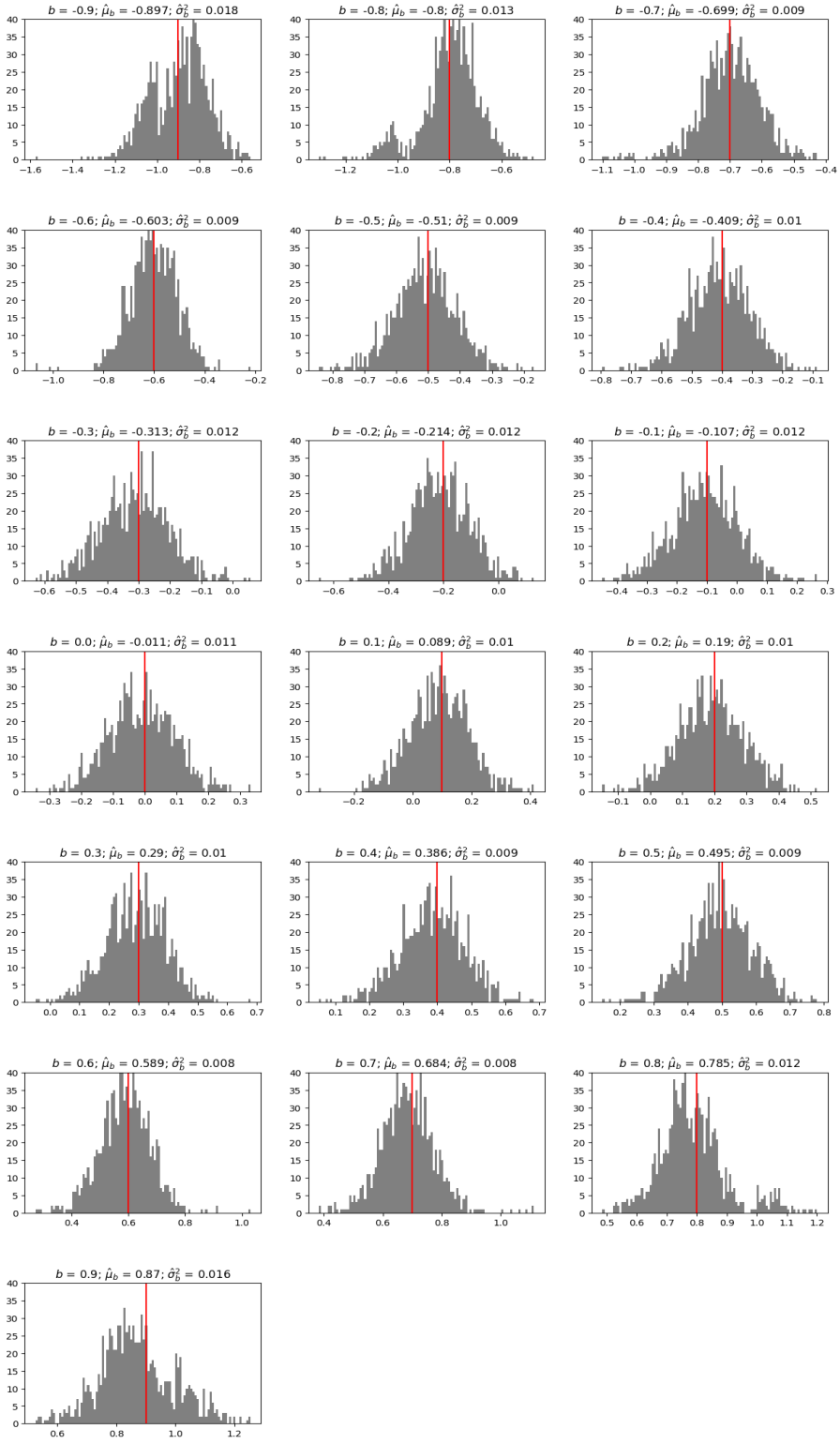


Figure 3: Results for the Hannan-Rissanen method: The parameter estimation works well for any $b$, though, indicated by $\hat{\sigma}_b^2$, the estimate slightly decreases in accuracy as $b$ approaches $|1|$. The red line indicates the true parameter $b$, the histogram displays 1000 estimates of $b$

| | T | $\bar{x}_{b_1}$ | $\hat{\sigma}_{b_1}$ | MSE $b_1$ | $\bar{x}_{b_2}$ | $\hat{\sigma}_{b_2}$ | MSE $b_2$ |
|---|---|---|---|---|---|---|---|
| Durbin | 100 | -0.717379 | 0.180025 | 0.079663 | -0.049528 | 0.162456 | 0.062671 |
| Durbin | 1000 | -0.910571 | 0.057345 | 0.171857 | 0.100530 | 0.057549 | 0.119272 |
| Paolella | 100 | -0.491414 | 0.111409 | 0.012044 | -0.234116 | 0.124672 | 0.015509 |
| Paolella | 1000 | -0.499774 | 0.032884 | 0.001154 | -0.238683 | 0.037368 | 0.001338 |
| Hannan Rissanen | 100 | -0.507167 | 0.120164 | 0.014491 | -0.238157 | 0.118717 | 0.014097 |
| Hannan Rissanen | 1000 | -0.499015 | 0.033543 | 0.001126 | -0.240090 | 0.032067 | 0.001028 |

Table 3: Results of the simulation study of 1.4.2 per estimation method and per T. $b_1 = -0.5$, $b_2 = -0.24$. The results of the Durbin method were estimated using $k = 20$. Notice that Durbin's method doesn't perform well at all since $k$ is fixed to 20, rather than an estimated constant as described in Broersen (2009). From Table 2 it is evident, that Durbins method can achieve much more accurate results, when $k$ is chosen correctly.

### 1.4.2 Estimation of $\hat{b}_1$ and $\hat{b}_2$ in the MA(2) model

To further assess the performance of the MA estimation methods, a simulation study was conducted for an MA(2) model. Again, 1000 repetitions were performed, but this time varying the number of observations $T \in \{100, 1000\}$. $[b_1, b_2]^T$ was fixed to $[-0.5, -0.24]^T$. Mean estimates for $b_1$ and $b_2$, it's standard errors and the mean squared error $MSE_{b_i} = E\left[(\hat{b}_i - b_i)^2\right]$ are displayed in Table 2.
Paolellas approximation method and the Hannan-Rissanen method performed equally well indifferent of the sample size T. The standard errors for both were considerably smaller for $T = 1000$ ($\approx 0.03$) than for $T = 100$ ($\approx 0.12$). The same pattern was observed for the MSE ($T = 100 \rightarrow \approx 0.013$ & $T = 1000 \rightarrow \approx 0.001$).

### 1.4.3 Parametrically bootstrapping Confidence Intervals (CIs) in the MA(1) model around $b = 0.5$

As a last step of the evaluation of the three methods for parameter estimation in MA processes, bootstrapped confidence intervals were evaluated in a simulation study. Parametric bootstrapping was used for this in oppose to the heavily used non-parametric bootstrapping. In parametric bootstrapping is a statistical resampling technique used to estimate the sampling distribution of a statistic, usually the mean or variance, when the underlying population distribution is unknown but can be estimated from the sample data. The parametric bootstrap involves creating a large number of resamples from the original sample data by generating random samples based on a specific assumed parametric distribution. The parameters of the assumed distribution are estimated from the original sample data, and the resampled data are generated using these estimated parameters. The parametric bootstrap assumes that the population distribution can be well approximated by a specific parametric distribution, which may not always be true.
For each of the methods, a total of 1000 simulations were conducted. During each simulation, 90% CIs were derived using parametric bootstrapping with $N_b = 100$. Evaluating Cis on such a small number of bootstraps may lead to a lot of bias, but such a small number had to be chosen in order to limit the already significant runtime. For all three methods, the variance for the bootstrapping was set to 1. This is due to equal treatment of the three methods, since Durbin, in his 1959 publication, doesn't describe a way to estimate the variance of $b$.

| | Durbin | Paolella | Hannan-Rissanen |
|---|---|---|---|
| mean 0.05 quantile | -0.031603 | 0.306967 | 0.368453 |
| mean 0.95 quantile | 0.702275 | 0.670185 | 0.670330 |
| mean length of 90% CI | 0.733878 | 0.363218 | 0.301876 |
| actual coverage | 0.896 | 0.892 | 0.885 |
| mean $\hat{b}$ | 0.434007 | 0.494321 | 0.511880 |
| $b$ | 0.5 | 0.5 | 0.5 |

Table 4: Results of the simulation study from 1.4.3. A total of $N_{sim} = 1000$ repetitions were computed. Bootstapped 90% CIs were derived using $N_B = 100$ parametric bootstraps. Note that the Durbin method used $k = 10$

The results are displayed in Table 4. The Hannan-Rissanen method and the Paolella approximation method work well with an average length of CI of 0.36 for the Paolella method and 0.30 for the Hannan-Rissanen method. The difference in the average length of the CI is mainly driven by the 0.05 quantile. The slightly better results of the Hannan-Rissanen method may be explained by the fact, that Paolellas method is an approximate method. The Durbin method misses the true parameter in mean by 0.07, which is in line with the results from 1.4.1. The confidence Interval is significantly longer with an average length of 0.70. This is also due to the fact that $k = 10$ was fixed in this experiment. In terms of actual coverage, that is the number of times, the true estimate lies within the confidence interval, all three methods are approximately equivalent (though this is evident, given the confidence intervals are not the same length).

# 2 p and q selection in ARMA(p,q) process using AIC and BIC measures

A popular time series model for forecasting is the Autoregressive Moving Average (ARMA) model. The moving average order (q) and the autoregressive order (p) are its two primary parameters. To produce forecasts that are accurate and trustworthy, it is essential to choose the right values for p and q.
Model selection criteria, such as used to find the optimal number of parameters in the ARMA(p,q) model, frequently use the Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC). These criteria penalize overly complicated models in order to prevent overfitting by taking into account both the model's complexity and goodness of fit. Better model fit is indicated by a lower AIC/BIC value.

## 2.1 Simulation Study

In this part of the study, we first picked parameters for a nontrivial stationary and invertible ARMA(3,2) process. In order to ensure the invertibility and stationarity of an ARMA(3,2) the following conditions have to be met:

1. The AR polynomial should have all its roots outside the unit circle: This can be checked by making sure that the absolute values of the roots of the AR polynomial are greater than 1. In other words, the AR parameters $\phi_1, \phi_2, \phi_3$ should be chosen such that the AR part of the model is stable and does not cause explosive behavior.

2. The MA polynomial should have all its roots outside the unit circle: Similarly, the absolute values of the roots of the MA polynomial should be greater than 1.

3. Additionally, the ARMA model should be invertible, meaning that it can be written as an infinite order AR model. This requires that the AR parameters do not have any common roots with the MA parameters.

In our model we picked parameters $p = [0.4, -0.5, -0.2]^T$ for AR(3) and $q = [0.65, 0.35]^T$ for MA(2). $p$ was described in Paolella (2018) as a feasible choice and $q$ was found online. Additionally, we checked the model's invertibility and stationarity by using Python's inbuilt functions.
We then simulated 10'000 times T=100 and T=1000 length realizations of our specification for the stationary, invertible ARMA(3,2) process. We estimated the AR(p) parameters using the *Brug* method as this method is computationally cheap and yields accurate results (Broersen, 2009). For the estimation of the MA(q) parameters, we used the *Innovations* method. We decided against using Durbins method as it seems to yield inaccurate results for $q > 1$ and is computationally rather expensive. Using the AIC measure, the optimal p for an AR(p) model of size T=100 and T=1000 was determined. For all 10'000 realizations of length T, we generated AIC values of $p = \{1, \ldots, 10\}$ and selected the order of the AR(p) process by selecting the lowest AIC. The formula to compute the AIC is $AIC = -2 * \log(L) + 2 * k$, where:

- L = Maximum likelihood estimate of the likelihood function of the model, given the data.
- k = Number of estimated parameters in the model, including both the intercept and the other parameters.

Results for $T = 100$ can be found in Table 5. The results for $T = 1000$ are listed in Table 6 Next, we determined the optimal q for an MA(q) model using the AIC measure. Similar to above, we generated AIC values of $q = \{1, \ldots, 10\}$ and selected the $q$ value, where the AIC was the lowest. The results are displayed in Table 5 (T=100) and Table 6 (T=1000). We further used the BIC measure to estimate the optimal number of parameters. The formula for calculating the BIC measure is $BIC = -2 * \log(L) + k * \log(n)$, where:

- L = Maximum likelihood estimate of the likelihood function of the model, given the data.
- k = Number of estimated parameters in the model, including both the intercept and the other parameters.
- n = Number of data points in the sample.

Results for $T = 100$ can be found in Table 5 and results for $T = 1000$ are shown in Table 6.

| | p count (AIC) | q count (AIC) | p count (BIC) | q count (BIC) |
|---|---|---|---|---|
| 1 | – | – | 1 | – |
| 2 | 762 | 514 | 3179 | 3724 |
| 3 | 4538 | 275 | 5908 | 720 |
| 4 | 1271 | 3624 | 577 | 4107 |
| 5 | 1345 | 1355 | 221 | 685 |
| 6 | 1040 | 1750 | 96 | 519 |
| 7 | 407 | 911 | 15 | 156 |
| 8 | 231 | 731 | 3 | 67 |
| 9 | 227 | 501 | – | 16 |
| 10 | 179 | 339 | – | 6 |

Table 5: Count of optimal p and q values for ARMA(p,q) model picked by AIC and BIC measures. A total of Rep = 10000 repetitions were computed for sample size T = 100.

| | p count (AIC) | q count (AIC) | p count (BIC) | q count (BIC) |
|---|---|---|---|---|
| 1 | – | – | – | – |
| 2 | – | – | – | – |
| 3 | 117 | – | 4076 | – |
| 4 | 28 | 11 | 166 | 812 |
| 5 | 2438 | 7 | 4096 | 205 |
| 6 | 4654 | 1940 | 1637 | 5872 |
| 7 | 907 | 756 | 20 | 949 |
| 8 | 877 | 3376 | 5 | 1757 |
| 9 | 618 | 1492 | – | 238 |
| 10 | 361 | 2418 | – | 167 |

Table 6: Count of optimal p and q values for ARMA(p,q) model picked by AIC and BIC measures. A total of $Rep = 10000$ repetitions were computed for sample size T = 1000.

## 2.2 Results

The results of the simulation study can be found in Table 5 (count of AIC and BIC selected p and q values for T=100) and Table 6 (count of AIC and BIC selected p and q values for T=1000) and visual representations can be seen in Figure 4 (distribution of p and q selected by AIC) and Figure 5 (distribution of p and q selected by BIC).
From the tables and plots, we observed the below results.

1. Sample size T = 100:

   - AIC:
     - Optimal p values: The highest count is for p = 3 (4538), which is the true AR order. The other orders selected are lower or higher than the true value.
     - Optimal q values: The highest count is for q = 4 (3624), which is slightly higher than the true MA order (2). The count for q = 2 is 514.

- BIC:
  - Optimal p values: The highest count is for p = 3 (5908), which is the true AR order. The other orders selected are lower or higher than the true value.
  - Optimal q values: The highest count is for q = 4 (4107), which is slightly higher than the true MA order (2). The count for q = 2 is 3724.

2. Sample size T = 1000:

   - AIC:
     - Optimal p values: The highest count is for p = 6 (4654), which is higher than the true AR order (3). The count for p = 3 is 117.
     - Optimal q values: The highest count is for q = 8 (3376), which is significantly higher than the true MA order (2). The true MA order (2) has not been selected by the AIC estimator at all.
   - BIC:
     - Optimal p values: The highest count is for p = 5 (4096), which is higher than the true AR order (3). The count for p = 3 is 4076.
     - Optimal q values: The highest count is for q = 6 (5872), which is higher than the true MA order (2). The true MA order (2) has not been selected by AIC estimator at all.

For the sample size T = 100, both AIC and BIC tend to select the true AR order, but a slightly higher MA order (q = 4) than the true value (q = 2).
For the larger sample size (T = 1000), the BIC measure tends to select almost always the true AR order (p = 5)(p = 3 and p = 5 were selected almost to the same extent), but both AIC and BIC tend to select a higher MA order than the true one (AIC: q = 8 and BIC: q = 6). The performance of AIC and BIC seems to be better for smaller sample sizes in detecting the true model order, even though they still select a slightly higher MA order than the true value.

This simulation study shows that in real-life applications, the selected model may not necessarily be the correct underlying model, especially for larger sample sizes, when using AIC or BIC to choose the optimal ARMA model for a given time series. Overfitting can result from a model that is excessively complex and captures noise rather than the underlying structure of the data. Poor forecasts on new, unseen data can be the result of overfitting.
Therefore, when using AIC or BIC for model selection, especially with large sample sizes, practitioners should use caution. Cross-validation, model validation on a holdout dataset, or other model selection criteria must all be taken into account in order to make sure that the chosen model is reliable and readily generalizes to new situations.

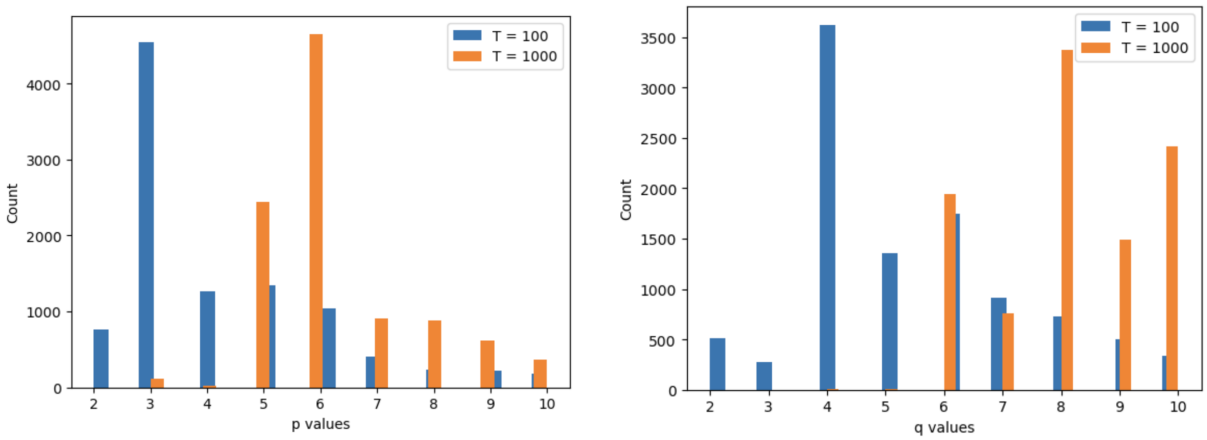


Figure 4: Plots showing the distribution of optimal p and q values selected by AIC measure.

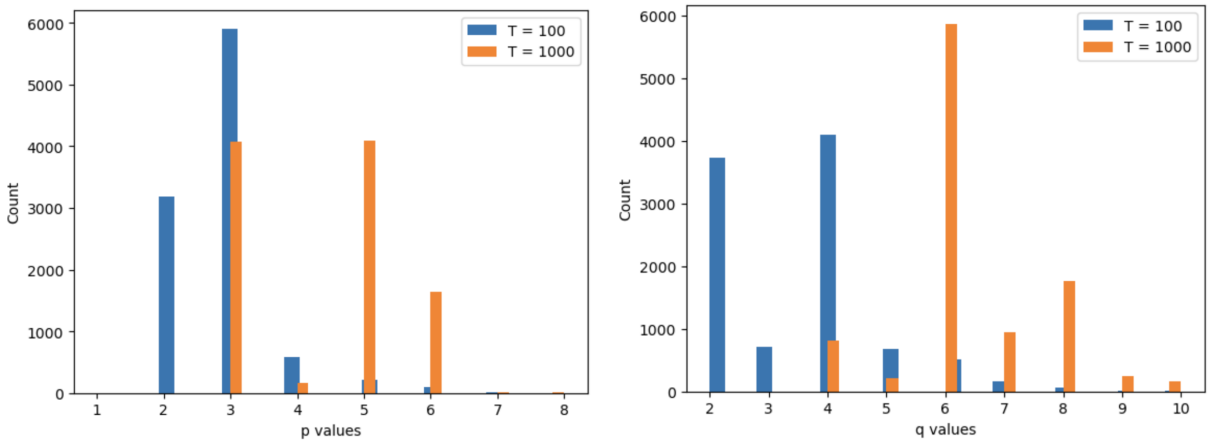


Figure 5: Plots showing the distribution of optimal p and q values selected by BIC measure.

# 3 Bibliography

Broersen, P. M. (2009). Modified Durbin method for accurate estimation of moving-average models. IEEE Transactions on Instrumentation and Measurement, 58(5), 1361-1369.

Mentz, R. P. (1977). Estimation in the first-order moving average model through the finite autoregressive approximation: Some asymptotic results. *Journal of Econometrics, 6*(2), 225-236.

Paolella, M. S. (2018). *Linear models and time-series analysis: regression, ANOVA, ARMA and GARCH*. John Wiley & Sons.

Stoica, P., McKelvey, T., & Mari, J. (2000). MA estimation in polynomial time. *IEEE Transactions on Signal Processing, 48*(7), 1999-2012.

# 4 Appendix

```
def durbinMA1(e, k):
    # k is the number of lags to use in the AR(p) model
    # e is the input data

    # Estimate all parameters up to k of the AR(p) model on the input data
    # The first parameter is the estimated intercept, the last parameter is
                                                the estimated variance
    # We don't need either of these, so we slice the array to exclude them
    A = sm.tsa.arima.ARIMA(e, order=(k, 0, 0)).fit(method = "burg").params[0
                                                :(-1)]

    # Calculate the MA(1) parameter according to Durbin's formula
    upper = sum(A[i] * A[i+1] for i in range(0, (k+1)-1))
    lower = sum(A**2)

    durbin = upper / lower

    return -durbin
```

Code-Section 1: Durbins method of estimating $\hat{b}$ in the MA(1) model.

```
def durbinMAq(e, k, q):
    # k is the number of lags to use in the AR(p) model
    # q is the order of the MA(q) model
    # e is the input data

    #The Durbin paper suggests in equation 15 the following formula A_mat*
                                            b_vec = -c_vec
    # it is straightforward to see that b_vec = -c_vec*A_mat^(-1)

    if q >= len(e):
        raise ValueError("q must be less than or equal to k")

    # Estimate all parameters up to k of the AR(p) model on the input data
    A = sm.tsa.arima.ARIMA(e, order=(k, 0, 0)).fit(method = "burg").params[0
                                            :(-1)]

    #compute the sum from 0 to k-1 of A[i] * A[i+1]
    c_vec = np.zeros(q)
    for j in range(1, q+1):
        c_vec[j-1] = sum(A[i] * A[i+j] for i in range(0, k+1-j))

    #init a 3x3 matrix of zeros using np.zeros
    A_mat = np.zeros((q, q))

    #fill the matrix with the appropriate values
    for row in range(0, q):
        for column in range(0, q):
            if row != column:
                A_mat[row, column] = sum(A[p] * A[p+abs(column-row)] for p
                                                    in range(0, k+1-abs(
                                                    column-row)))
            else:
                A_mat[row, column] = sum(A[p] * A[p] for p in range(0, k+1))

    #inverse the matrix
    A_mat_inv = np.linalg.inv(A_mat)

    #compute b_vec
    b_vec = -c_vec @ A_mat_inv

    return  b_vec
```

Code-Section 2: Durbins method of estimating $\hat{b} = [\hat{b}_1, \ldots, \hat{b}_q]^T$ of an MA(q) model

```
def durbinMA1959(e, k, q):
    if q == 1:
        return np.array([durbinMA1(e, k)])
    else:
        return durbinMAq(e, k, q)
```

Code-Section 3: Durbins method for estimating the parameters $\hat{b} = [\hat{b}_1, \ldots, \hat{b}_q]^T$ in the MA(q) model optimized for computation time.

```
def paolellaApprox(y, p, q):
    # Assumes zero mean stationary invertible ARMA(p,q)
    # param = [AR terms, MA terms, sigma]
    L = np.ceil(np.sqrt(len(y))).astype(int)
    z = y[L+1:]
    Z = toeplitz(y[L:-1], y[L::-1])
    uhat = (np.eye(len(z)) - Z @ np.linalg.inv(Z.T @ Z) @ Z.T) @ z
    sigmahat = np.std(uhat)
    yy = z - uhat
    X = np.empty((len(z) - max(p, q), p+q))
    for i in range(1, p+1):
        X[:, i-1] = z[max(p-i, 0):-i]
    for i in range(1, q+1):
        X[:, p+i-1] = uhat[max(q-i, 0):-i]
    yuse = yy[max(p, q):]
    ARMAparam = np.linalg.inv(X.T @ X) @ X.T @ yuse
    param = np.concatenate((ARMAparam, np.array([sigmahat])))
    return param
```

Code-Section 4: Paolellas approximation method for estimating $\hat{b} = [\hat{b}_1, \ldots, \hat{b}_q]^T$ in the MA(q) model.

```
def estimate_hannan_rissanen(e, q):
    # Set up ARIMA model with MA(1) component
    mod = sm.tsa.arima.ARIMA(e, order=(0, 0, q), enforce_invertibility=False
                                                 )

    # Fit model using Hannan-Rissanen algorithm
    res = mod.fit(method='hannan_rissanen')

    # Return estimated parameters

    return res.params[1:-1]

estimate_hannan_rissanen(e, q=2)
```

Code-Section 5: Hannan-Rissanens method of estimating the parameters $\hat{b} = [\hat{b}_1, \ldots, \hat{b}_q]^T$ in the MA(q) model.

```
N_sim = 1000
n = 100


def MA1_sim(N_sim, n, method = ["durbinMA1959", "paolellaApprox", "
                                        hannanRissanen"]):

    # output matrix
    results = np.zeros((N_sim, 20))

    for sim in range(N_sim):

        col = 0
        for b_int in [i/10 for i in range(-9, 9+1, 1)]:

            # simulate MA(1) process
            e = sm.tsa.arma_generate_sample([1], [1, b_int], n)

            # chosen estimation method
            if method == "durbinMA1959":
                results[sim, col] = durbinMA1959(e, k=10, q=1)[0]
            if method == "paolellaApprox":
                results[sim, col] = paolellaApprox(e, p=0, q=1)[0]
            if method == "hannanRissanen":
                results[sim, col] = estimate_hannan_rissanen(e, q=1)[0]

            col += 1

        # print every 100th simulation
        if sim % 100 == 0:
            if sim == 0:
                print("Simulation started...")
            else:
                print(f"Simulation {sim} complete")

    print("Simulation finished")

    return results

# Durbin
results_durbin = MA1_sim(N_sim=N_sim, n=n, method = "durbinMA1959")

#Paolella
results_paolella = MA1_sim(N_sim=N_sim, n=n, method="paolellaApprox")

#Hannan Rissanen
results_hannan_rissannen = MA1_sim(N_sim=N_sim, n=n, method="hannanRissanen"
                                        )
```

Code-Section 6: Simulation study for 1.4.1.

```
def MA2_sim(N_sim, n, b1_true, b2_true, method = ["durbinMA1959", "
                                        paolellaApprox", "hannanRissanen"]):

    # Matrix for the results
    results = np.zeros((N_sim, 2))

    for sim in range(N_sim):

        # generate a sample MA(2) process
        e = sm.tsa.arma_generate_sample([1], [1, b1_true, b2_true], n)

        #choose the method of parameter estimation
        if method == "durbinMA1959":
            bs = durbinMA1959(e, k=30, q=2)
            results[sim, 0] = bs[0]
            results[sim, 1] = bs[1]

        if method == "paolellaApprox":
            bs = paolellaApprox(e, p=0, q=2)
            results[sim, 0] = bs[0]
            results[sim, 1] = bs[1]

        if method == "hannanRissanen":
            bs = estimate_hannan_rissanen(e, q=2)
            results[sim, 0] = bs[0]
            results[sim, 1] = bs[1]

        # print every 100th simulatio
        if sim % 100 == 0:
            if sim == 0:
                print("Simulation started...")
            else:
                print(f"Simulation {sim} complete")

    print("Simulation finished")

    return results

# Durbin T = 100 & T = 1000
results_t100_durbin = MA2_sim(N_sim=N_sim, n=100, b1_true=b1_true, b2_true=
                                        b2_true, method = "durbinMA1959")
results_t1000_durbin = MA2_sim(N_sim=N_sim, n=1000, b1_true=b1_true, b2_true
                                        =b2_true, method = "durbinMA1959")
# Paolella T = 100 & T = 1000
results_t100_paolella = MA2_sim(N_sim=N_sim, n=100, b1_true=b1_true, b2_true
                                        =b2_true, method = "paolellaApprox")
results_t1000_paolella = MA2_sim(N_sim=N_sim, n=1000, b1_true=b1_true,
                                        b2_true=b2_true, method = "
                                        paolellaApprox")
# Hannan Rissanen T = 100 & T = 1000
results_t100_hannan_rissanen = MA2_sim(N_sim=N_sim, n=100, b1_true=b1_true,
                                        b2_true=b2_true, method = "
                                        hannanRissanen")
results_t1000_hannan_rissanen = MA2_sim(N_sim=N_sim, n=1000, b1_true=b1_true
                                        , b2_true=b2_true, method = "
                                        hannanRissanen")
```

Code-Section 7: Simulation study for 1.4.2.

```
#MA(1) parametric bootstrap
def MA1_parametric_bootstrap(N_sim, N_bootstraps, b_true, method = ["
                                    durbinMA1959", "paolellaApprox", "
                                    hannanRissanen"]):

    results = np.zeros((N_sim, 4))
    for sim in range(N_sim):

        # generate a time series with the true parameter
        e = sm.tsa.arma_generate_sample(ar=[1], ma=[1, b_true], nsample=100)

        # estimate the b parameter
        if method == "durbinMA1959":
            b_est = durbinMA1959(e, k=10, q=1)[0]
        if method == "paolellaApprox":
            b_est = paolellaApprox(e, p=0, q=1)[0]
        if method == "hannanRissanen":
            b_est = estimate_hannan_rissanen(e, q=1)[0]

        # Bootstrap
        result_bootstrap = list()
        for bootstrap in range(N_bootstraps):

            # generate a time series with the estimated parameter (resample
                                                    timeseries)
            e_boot = sm.tsa.arma_generate_sample(ar=[1], ma=[1, b_est],
                                                    nsample=100)

            #re-estimate the b parameter
            if method == "durbinMA1959":
                b_boot = durbinMA1959(e_boot, k=10, q=1)[0]
            if method == "paolellaApprox":
                b_boot = paolellaApprox(e_boot, p=0, q=1)[0]
            if method == "hannanRissanen":
                b_boot = estimate_hannan_rissanen(e_boot, q=1)[0]

            result_bootstrap.append(b_boot)

        # compute 90% CI of the original parameter: 0.5 and 0.95 quantiles
        quant05 = np.quantile(result_bootstrap, 0.05)
        quant95 = np.quantile(result_bootstrap, 0.95)
        results[sim, 0] = b_est
        results[sim, 1] = quant05
        results[sim, 2] = quant95

        #is b_true between the 0.05 and 0.95 quantiles --> then 1 else 0
        results[sim, 3] = 1 if (b_true > quant05) & (b_true < quant95) else
                                                0

        if sim % 10 == 0 and sim == 0:
            print("Simulation started...")
        if sim % 10 == 0 and sim != 0:
            print(f"{sim} simulations done")
    print("Simulation finished")

    return results
```

Code-Section 8: Simulation study for 1.4.3.

```
import numpy as np
import statsmodels.api as sm
from statsmodels.tsa.arima_process import ArmaProcess


# Setting ARMA parameters ,include zero-th lag
ar = np.array([1, 0.4,-0.5,-0.2])
ma = np.array([1, 0.65, 0.35])

# Sample size and number of repetitions
T1 = 100
T2 = 1000
Rep = 10000

# Simulating realizations of the process
realiz_t100 = np.zeros((Rep, T1))
realiz_t1000 = np.zeros((Rep, T2))

for i in range(Rep):
    # Generating a realization of T = 100
    realiz_t100[i,:] = sm.tsa.arma_generate_sample(ar, ma, T1)

    # Generating a realization of T = 1000
    realiz_t1000[i,:] = sm.tsa.arma_generate_sample(ar, ma, T2)

# Checking if model is invertibe and stationary
arma_t = ArmaProcess(ar, ma)
print ("Is my model invertibe? ", arma_t.isinvertible)
print("Is my model stationary?", arma_t.isstationary)

# Generating optimal p and q for ARMA(p,q) using AIC and BIC measures

p_q_max = 10 #setting up max for p and q

def AIC_BIC(data, measure):

    # Initializing empty array for aic values
    aic_bic_p= np.zeros((Rep,p_q_max))
    aic_bic_q = np.zeros((Rep,p_q_max))


    optimal_p = []
    optimal_q = []


    for i in range(Rep):

        # Initializing empty array for aic values

        aic_bic_p= np.zeros((Rep,p_q_max))
        aic_bic_q = np.zeros((Rep,p_q_max))

        for p in range(p_q_max):

            # Fit the AR(p) and MA(q) model to the data for T=100
            ar_model_p = sm.tsa.ARIMA(data[i,:], order=(p+1,0,0),
                                                    enforce_invertibility=
                                                    False).fit(method = "
                                                    burg")
```

```
            ar_model_q = sm.tsa.ARIMA(data[i,:], order=(0,0,p+1),
                                                  enforce_invertibility=
                                                  False).fit(method = "
                                                  innovations")

            if measure == "aic":
                # Compute AIC of p and q
                aic_bic_p[i,p] = ar_model_p.aic
                aic_bic_q[i,p] = ar_model_q.aic

            else:
                aic_bic_p[i,p] = ar_model_p.bic
                aic_bic_q[i,p] = ar_model_q.bic

        # Determine the optimal value of p based on the AIC
        optimal_p.append(np.argmin(aic_bic_p[i]) + 1)
        optimal_q.append(np.argmin(aic_bic_q[i]) + 1)

    return optimal_p, optimal_q

AIC_optimal_p_100, AIC_optimal_q_100 = AIC_BIC(realiz_t100, "aic")
AIC_optimal_p_1000, AIC_optimal_q_1000 = AIC_BIC(realiz_t1000, "aic")
BIC_optimal_p_100, BIC_optimal_q_100 = AIC_BIC(realiz_t100, "bic")
BIC_optimal_p_1000, BIC_optimal_q_1000 = AIC_BIC(realiz_t1000, "bic")


# loop for printing AIC and BIC selected values
res = {"AIC_optimal_p_100":AIC_optimal_p_100, "AIC_optimal_q_100":
                                        AIC_optimal_q_100, "
                                        AIC_optimal_p_1000":
                                        AIC_optimal_p_1000,
       "AIC_optimal_q_1000":AIC_optimal_q_1000, "BIC_optimal_p_100":
                                            BIC_optimal_p_100, "
                                            BIC_optimal_q_100":
                                            BIC_optimal_q_100,
       "BIC_optimal_p_1000": BIC_optimal_p_1000, "BIC_optimal_q_1000":
                                            BIC_optimal_q_1000}

for k, v in res.items():
    print(k, dict(zip(sorted(list(v)),[sorted(list(v)).count(i) for i in
                                        sorted(list(v))])))
```

Code-Section 9: Simulation study for getting optimal p and q values for ARMA(p,q) using AIC and BIC measures.

```
    import matplotlib.pyplot as plt

    def plot(x,y, x_label):
        plt.hist(x, bins = 30, label= "T = 100")
        plt.hist(y, bins = 30, label= "T = 1000")
        plt.xlabel(x_label)
        plt.ylabel('Count')
        plt.legend(loc='upper right')
        plt.show()

    plot(AIC_optimal_p_100, AIC_optimal_p_1000, "p values")
    plot(AIC_optimal_q_100, AIC_optimal_q_1000, "q values")
    plot(BIC_optimal_p_100, BIC_optimal_p_1000, "p values")
    plot(BIC_optimal_q_100, BIC_optimal_q_1000, "q values")
```

Code-Section 10: Plotting the count of optimal p and q values picked by AIC and BIC measures.