The George Washington University

Laboratory Nine:

Estimating ARMA Model Parameters with Levenberg-Marquardt Algorithm

Fernando Zambrano

DATS 6450: Multivariate Modeling

Dr. Reza Jafari

15 April 2020

Abstract:

Implementing ARMA forecasting for a dataset require estimating several parameters relative to that specific dataset. The three most important are: the autoregressive (AR) order parameter, the moving average (MA) order parameter, and finally the value of the coefficients of the AR and MA parameters. The coefficients can be estimated using Maximum Likelihood Estimation or MLE in conjunction with the Levenberg-Marquardt algorithm implemented in a Python program. This study will look at several examples which LM estimates the parameters of ARMA processes.

Introduction:

ARMA models are composed of both an AR (na), and an MA(nb) models. ARMA models are popular and powerful in the forecasting industry due to its reliability and flexibility in predicting future values of single variable base on past behavior. However, one of the shortcomings of using ARMA models for experimental and research purposes is that their performance is dependent on using the proper order or lag value of the AR and MA components for a dataset. This information is not always known nor is it trivial to detect. For this reason, the Generalized Partial Autocorrelation, or GPAC, method is used to estimate the best order of an ARMA process for a given dataset. Once order determination is established the next step is parameter estimation of *theta*, which represents the coefficients for AR and MA processes. There are a couple of algorithms that can estimate parameters, but this study will focus on the Levenberg-Marquardt algorithm or LM. Estimating one or two coefficients may possible by hand, but some processes may have several more parameters to estimate in which case the best solution is to use a programming language, such as Python, to assist with fast and recursive operations.

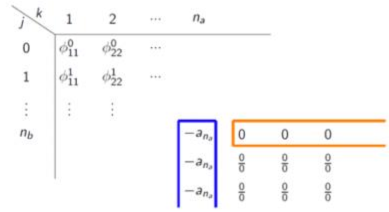
Methods & Theory:

The Autoregressive Moving Average (ARMA) is a combination of AR (*na*) and MA (*nb*) models. The ARMA model is one the most popular methods used for time stationary time series analysis since it offers the minimum number of parameters to forecast the unknown. The equation for the ARMA (*na,nb*) model is below.

$$y(t) + a_1 y(t-1) + a_2 y(t-2) + \ldots + a_{n_a} y(t-n_a) = \epsilon(t) + b_1\epsilon(t-1) + b_2\epsilon(t-2) + \ldots + b_{n_b}\epsilon(t-n_b)$$

GPAC uses the Autocorrelation Function or ACF of a dataset to estimate the order of the ARMA process by constructing a table with the possible combinations for the ARMA orders. The parameters of this this method are *j*, *k*, and *phi*. Ry(*j*) represents the estimated autocorrelation at lag *j*. *j* also denotes the estimated value *nb* for the MA process. *k* denotes the estimated value for *na* for the AR process. *Phi* is the result the division between the determinate of the matrices of the given formula below:

$$\phi^{j}_{kk} = \frac{\begin{vmatrix} \hat{R}_y(j) & \hat{R}_y(j-1) & \dots & \hat{R}_y(j+1) \\ \hat{R}_y(j+1) & \hat{R}_y(j) & \dots & \hat{R}_y(j+2) \\ \vdots & \vdots & \vdots & \vdots \\ \hat{R}_y(j+k-1) & \hat{R}_y(j+k-2) & \dots & \hat{R}_y(j+k) \end{vmatrix}}{\begin{vmatrix} \hat{R}_y(j) & \hat{R}_y(j-1) & \dots & \hat{R}_y(j-k+1) \\ \hat{R}_y(j+1) & \hat{R}_y(j) & \dots & \hat{R}_y(j-k+2) \\ \vdots & \vdots & \vdots & \vdots \\ \hat{R}_y(j+k-1) & \hat{R}_y(j+k-2) & \dots & \hat{R}_y(j) \end{vmatrix}}$$



Where $\hat{R}_y(j)$ is the estimated autocorrelation of y(t) at lag j.

The result of the GPAC table will highlight the number rows and columns equal to the order of the process. For example, if there are two highlighted columns and 1 highlighted row, the GPAC is estimating an ARMA (2,1) order. The rest of the table will most likely be filled with zeros.

MLE uses Bayes Theorem to maximize the likelihood of estimating the parameter theta. In other words, based upon a given dataset what parameter has the highest probably of explaining the given data. Below is the equation for Bayes Theorem.

Let θ be the unknown parameter and $\mathbf{y}$ be the set of observations. Using the *Bayes Theorem*:

$$P(\theta|\mathbf{y}) = \frac{P(\mathbf{y}|\theta)P(\theta)}{P(\mathbf{y})}$$

$$posterior = \frac{Likelihood \times prior}{evidence}$$

The LM algorithm is a combination of two popular optimization algorithms that take MLE into account are Gauss-Newton method, and gradient descent method to solve non-linear equations. The loss function that drives this algorithm is the Least Square Estimate or LSE. This loss function aims to minimize the sum of squared error (SSE) between the data and the estimated function that fits the data.

There are three main steps in which LM functions. The first step is initializing the vector *theta*, which contains all the coefficients of the parameters for *na* and *nb* components to 0. This sets a baseline for SSE in the case that all the parameters are 0 in which case the data is simply white noise and cannot be estimated. This step also demonstrates the importance of the GPAC table. Without having an initial guess of the number of parameters for LM to estimate for the AR and MA portion renders the LM algorithm useless since it will not know how many parameters to estimate.

Step 2 adds a small change, *delta* ($\delta = 10^{-6}$ or 10^{-7}) to each parameter within the theta vector to calculate a new SSE. Within this same step there is damping factor *mu* (10) which scales the amount of change *delta* will add *theta.*

Step 3 is where the bulk of the algorithm performs due to its recursive nature. LM will change its optimization method depending on the SSE after each iteration. If the new SSE calculated in step 2 is less than the SSE calculated in step 1, then LM will begin to optimize the loss function using the Gauss-Newton method (GN) because the current estimation of *theta* parameters is close to their optimal value. LM will estimate theta again using the old *theta* as its initialized value. To avoid overshooting the optimal value and increasing the precision of the estimate, *mu* becomes smaller to decrease the amount change *delta* will add to *theta*. The algorithm will remain estimating *theta* using GM until either the SSE becomes greater than the previous, or the second norm of estimated *theta* is less than *epsilon* (= 10^{-3} or 10^{-4}). In the case of the later, this is an indication that the loss function has converged to its minimum, or that the minimum SSE has been reached and the algorithm is completed. In the case of the former, the estimated parameters overshot their optimal values and are now far away. LM switches to steepest gradient descent (SGD) to estimate *theta*. Instead of *delta* adding a small change, mu is scaled up in order to make large changes to *theta* until the new SSE is lower than the previous in which case LM switches back to GN until convergence is reached or max iterations (typically 100) within the program have been exhausted. If max iterations have elapsed, then there might be an error in the program and code should be debugged. The sudo-code of step 3 is below to visually show the flow of switching between GN and SGD until convergence or iteration exhaustion is reached.

if $\#$ *of iterations* $< MAX$ **then**
- **if** $SSE(\theta_{new}) < SSE(\theta_{old})$ **then**
  - **if** $\|\Delta\theta\| < \epsilon(10^{-3})$ **then**
    - $\hat{\theta} = \theta_{new}$;
    - $\hat{\sigma_e}^2 = \frac{SSE(\theta_{new})}{N-n}$;
    - $\hat{cov}(\hat{\theta}) = \hat{\sigma_e}^2.A^{-1}$;
    - stop;
  - **else**
    - $\theta = \theta_{new}$;
    - $\mu = \mu/10$;
- **while** $SSE(\theta_{new}) >= SSE(\theta_{old})$ **do**
  - $\mu = \mu * 10$;
  - **if** $\mu > \mu_{max}$ **then**
    - Print out results and print error message;
    - stop;
  - Return to step 2
- $\#$ of iterations $+=1$;
- **if** $\#$ *of iterations* $> MAX$ **then**
  - Print out the results;
  - Error Message;
  - stop;
- $\theta_{old} = \theta_{new}$;
- Return to step 1;
- Return to step 2;

Evaluating the results of LM are important to make sure the results are reliable and to validate the order determination from GPAC. The best indicator demonstrating the reliability of the estimated parameters is a 95% confidence interval. The formula for calculating the threshold of the confidence interval for each individual parameter is below which is taking the square root of the value of each element in the diagonal of the covariance matrix will calculate the standard deviation. 2x +- the standard deviation will determine the 95% confidence interval.

Let $\Sigma\hat{\theta} = \hat{cov}(\hat{\theta})$ Then the confidence interval can be calculated as :

$$\hat{\theta}_i \pm 2\sqrt{[\Sigma\hat{\theta}]_{ii}}$$

for $1 \leq i \leq n$

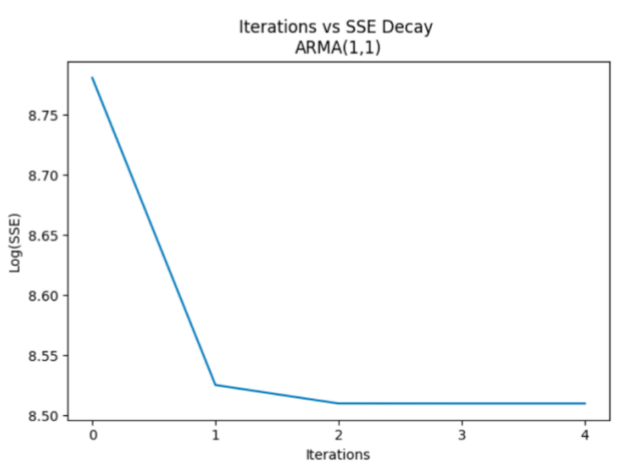
If a confidence interval for a parameter includes zero, then that parameter should be dropped and lower the order of the corresponding AR or MA portion by one. The last evaluation criteria is to check for Zero/Pole cancelation of the parameters. If parameters in the AR and MA portion share common roots, they will cancel each other out and the order for AR and MA should be decreased by one. The LM should be processed again with the updated number of parameters.

Implementation and Results:

Implementing LM in Python on eight examples with synthetic data in order to validate the accuracy of the estimated parameters.

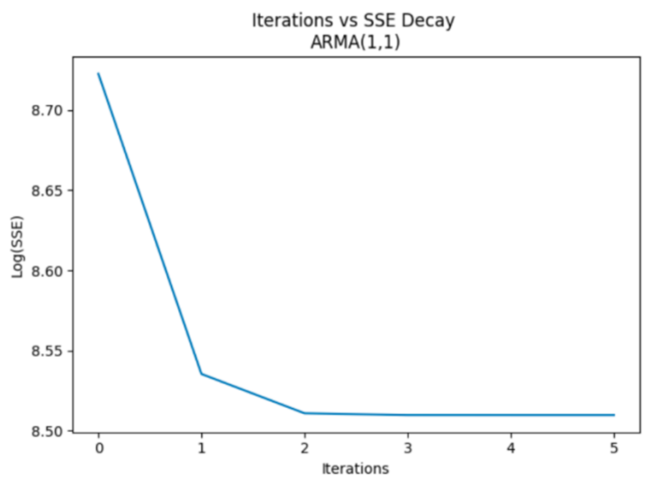
1. $y(t) - 0.5y(t-1) = e(t)$

```
Total # of Iterations: 4
True Theta:
[['a1' '-0.5']
 ['b1' '0.0']]
Estimated Theta:
[['a1' '-0.4891862673290693']
 ['b1' '-0.002285165728973532']]
95% Confidence Interval:
[['Theta' ' Theta - 2xSTD ' ' Theta  + 2xSTD  ']
 ['a1' '-0.5398351756551626' '-0.43853735900297597']
 ['b1' '-0.06035594668764842' '0.05578561522970135']]
Covariance Matrix of Theta:
[['a1' '0.0006413279786562515' '0.0006421374166692866']
 ['b1' '0.0006421374166692865' '0.0008430539002875993']]
Theta STD:
[['a1' '0.025324454163046666']
 ['b1' '0.029035390479337442']]
Root Checks:
Num(bn): [1, -0.002285165728973532]
Den(an): [1, -0.4891862673290693]
Roots Num: [0.00228517]
Roots Den: [0.48918627]
Variance of Error:
0.9929955653612291

END
Relevant Parameters:
[['a1' '-0.4891862673290693']]
MA/Num parameters: [1, 0.0]
AR/Den parameters: [1, -0.4891862673290693]
```



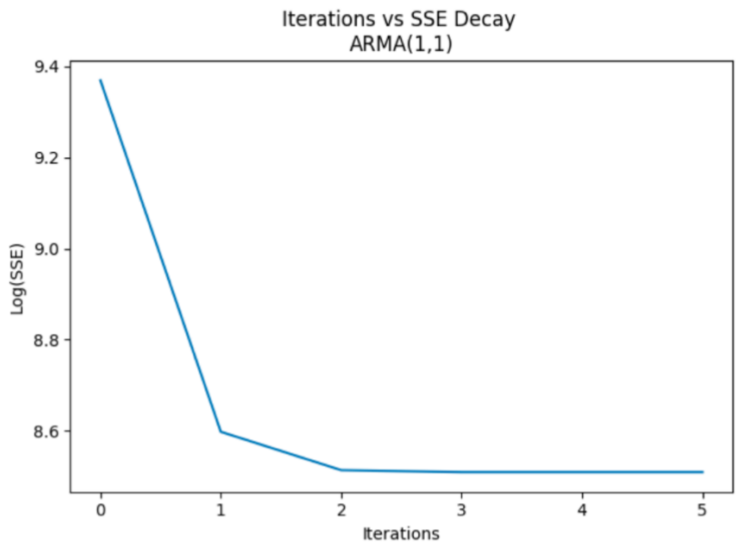
2. *ARMA (0,1) : y(t) = e(t) + 0.5e(t-1)*

```
Total # of Iterations: 5
True Theta:
[['a1' '0.0']
 ['b1' '0.5']]
Estimated Theta:
[['a1' '0.029386480035949097']
 ['b1' '0.5163143765233729']]
95% Confidence Interval:
[['Theta' ' Theta - 2xSTD ' ' Theta  + 2xSTD  ']
 ['a1' '-0.02779987054354196' '0.08657283061544016']
 ['b1' '0.46733753749398893' '0.5652912155527567']]
Covariance Matrix of Theta:
[['a1' '0.0008175696731501143' '0.0006085946093448969']
 ['b1' '0.0006085946093448968' '0.0005996826903275461']]
Theta STD:
[['a1' '0.028593175289745528']
 ['b1' '0.024488419514691964']]
Root Checks:
Num(bn): [1, 0.5163143765233729]
Den(an): [1, 0.029386480035949097]
Roots Num: [-0.51631438]
Roots Den: [-0.02938648]
Variance of Error:
0.9929531799237961

END
Relevant Parameters:
[['b1' '0.5163143765233729']]
MA/Num parameters: [1, 0.5163143765233729]
AR/Den parameters: [1, 0.0]
```



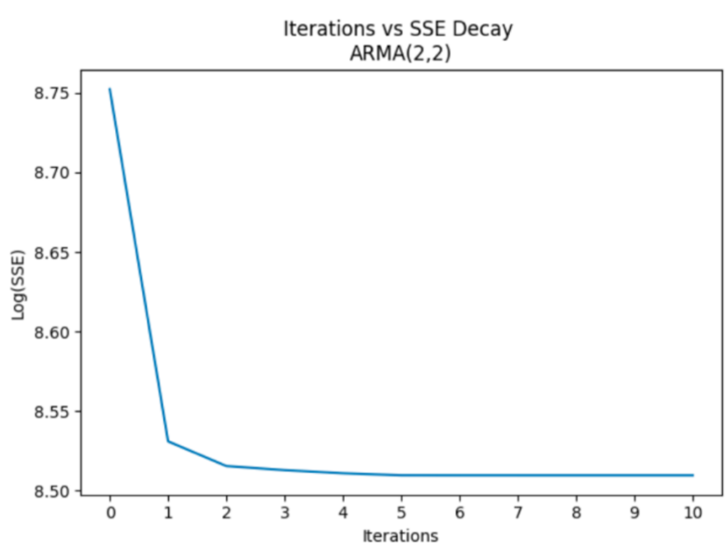
3. *ARMA (1,1) : y(t) + 0.5y(t-1) = e(t) - 0.5e(t-1)*

```
Total # of Iterations: 5
True Theta:
[['a1' '0.5']
 ['b1' '-0.5']]
Estimated Theta:
[['a1' '0.4979481124255066']
 ['b1' '-0.5135240240351686']]
95% Confidence Interval:
[['Theta' ' Theta - 2xSTD ' ' Theta  + 2xSTD  ']
 ['a1' '0.4674884265884612' '0.528407798262552']
 ['b1' '-0.5436640224587019' '-0.4833840256116353']]
Covariance Matrix of Theta:
[['a1' '0.00023194811532287624' '0.00013602536701651905']
 ['b1' '0.00013602536701651905' '0.00022710487624264759']]
Theta STD:
[['a1' '0.015229842918522706']
 ['b1' '0.015069999211766655']]
Root Checks:
Num(bn): [1, -0.5135240240351686]
Den(an): [1, 0.4979481124255066]
Roots Num: [0.51352402]
Roots Den: [-0.49794811]
Variance of Error:
0.9929950555929329

END
Relevant Parameters:
[['a1' '0.4979481124255066']
 ['b1' '-0.5135240240351686']]
MA/Num parameters: [1, -0.5135240240351686]
AR/Den parameters: [1, 0.4979481124255066]
```



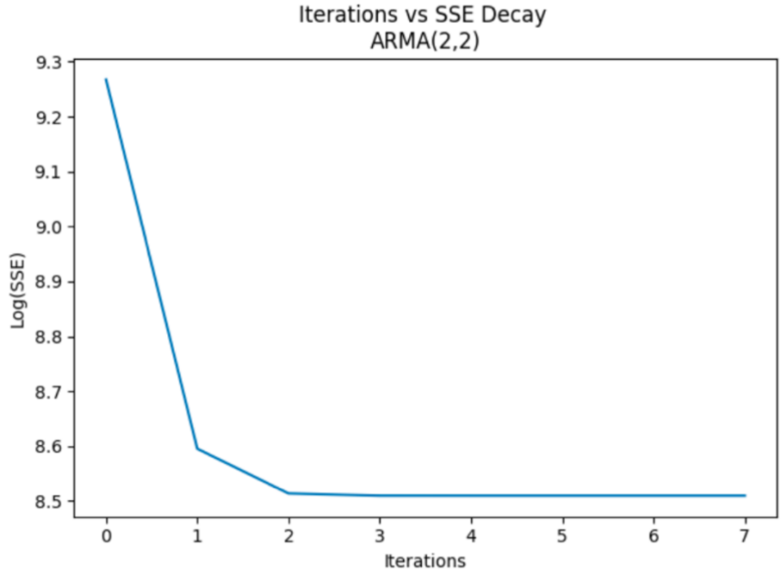
4. *ARMA(2,0) : y(t) + 0.5y(t-1) + 0.2y(t-2) = e(t)*

```
Estimated Results:

Total # of Iterations: 10
True Theta:
[['a1' '0.5']
 ['a2' '0.2']
 ['b1' '0.0']
 ['b2' '0.0']]
Estimated Theta:
[['a1' '0.6488993076481807']
 ['a2' '0.20793634751062753']
 ['b1' '0.1361431659282407']
 ['b2' '-0.07518549877274583']]
95% Confidence Interval:
[['Theta' ' Theta - 2xSTD ' ' Theta  + 2xSTD  ']
 ['a1' '0.3789684176490685' '0.9188301976472929']
 ['a2' '0.1499996558387961' '0.26587303918245897']
 ['b1' '-0.13515752447026017' '0.4074438563267415']
 ['b2' '-0.20775785054489335' '0.05738685299940169']]
Covariance Matrix of Theta:
[['a1' '0.018215671343928198' '0.0013922813058604004'
  '0.01820827534636795' '-0.007815134217885595']
 ['a2' '0.0013922813058604112' '0.0008391650604692154'
  '0.001363544963267842' '0.00018806836776008947']
 ['b1' '0.01820827534636795' '0.0013635449632678298'
  '0.018401016152675802' '-0.007813009143590689']
 ['b2' '-0.00781513421788558' '0.00018806836776009603'
  '-0.007813009143590675' '0.004393857113599506']]
Theta STD:
[['a1' '0.1349654449995561']
 ['a2' '0.028968345835915715']
 ['b1' '0.13565034519925043']
 ['b2' '0.06628617588607376']]
Root Checks:
Num(bn): [1, 0.1361431659282407, -0.07518549877274583]
Den(an): [1, 0.6488993076481807, 0.20793634751062753]
Roots Num: [-0.35059457  0.21445141]
Roots Den: [-0.32444965+0.32041968j -0.32444965-0.32041968j]
Variance of Error:
0.9932331527652096

END
Relevant Parameters:
[['a1' '0.6488993076481807']
 ['a2' '0.20793634751062753']]
MA/Num parameters: [1, 0.0, 0.0]
AR/Den parameters: [1, 0.6488993076481807, 0.20793634751062753]
```



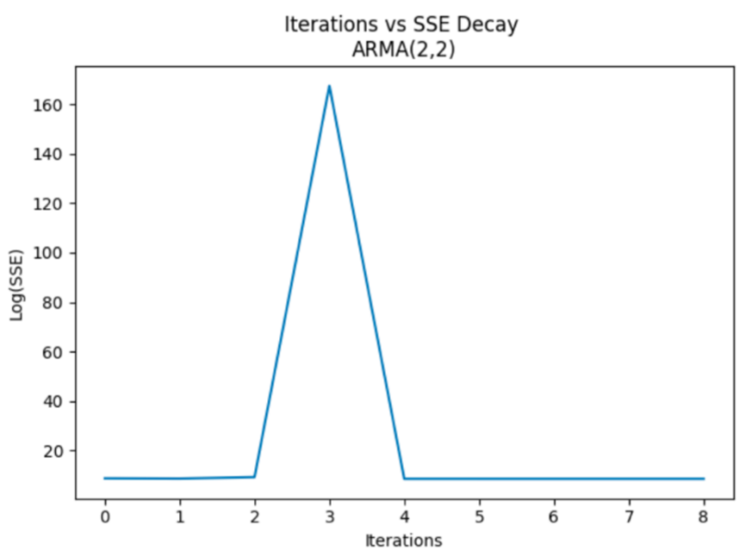
5. *ARMA(2,1) : y(t) + 0.5y(t-1) + 0.2y(t-2) = e(t) - 0.5e(t-1)*

```
Estimated Results:

Total # of Iterations: 7
True Theta:
[['a1' '0.5']
 ['a2' '0.2']
 ['b1' '-0.5']
 ['b2' '0.0']]
Estimated Theta:
[['a1' '0.5315028900250532']
 ['a2' '0.2116072415294408']
 ['b1' '-0.4814710698427051']
 ['b2' '-0.018933674188785873']]
95% Confidence Interval:
[['Theta' ' Theta - 2xSTD ' ' Theta  + 2xSTD  ']
 ['a1' '0.3313456595546894' '0.7316601240946375']
 ['a2' '0.1409482966125532' '0.2822661864463284']
 ['b1' '-0.6837675862178407' '-0.2791745534675695']
 ['b2' '-0.15791636103666162' '0.12004901265908989']]
Covariance Matrix of Theta:
[['a1' '0.010015729587596592' '0.003183258412280292'
  '0.010023257699226101' '-0.006747605645564177']
 ['a2' '0.0031832584122802906' '0.0012481716241919388'
  '0.0031817422150066704' '-0.002004580326201307']
 ['b1' '0.010023257699226101' '0.003181742215006671'
  '0.010230970134378875' '-0.006853286938091194']
 ['b2' '-0.006747605645564179' '-0.002004580326201308'
  '-0.006853286938091196' '0.004829046810863675']]
Theta STD:
[['a1' '0.10007861703479216']
 ['a2' '0.0353294724584438']
 ['b1' '0.1011482581875678']
 ['b2' '0.06949134342393788']]
Root Checks:
Num(bn): [1, -0.4814710698427051, -0.018933674188785873]
Den(an): [1, 0.5315028900250532, 0.2116072415294408]
Roots Num: [ 0.51802108 -0.03655001]
Roots Den: [-0.26575145+0.37547758j -0.26575145-0.37547758j]
Variance of Error:
0.9933766171270915

END
Relevant Parameters:
[['a1' '0.5315028900250532']
 ['a2' '0.2116072415294408']
 ['b1' '-0.4814710698427051']]
MA/Num parameters: [1, -0.4814710698427051, 0.0]
AR/Den parameters: [1, 0.5315028900250532, 0.2116072415294408]
```



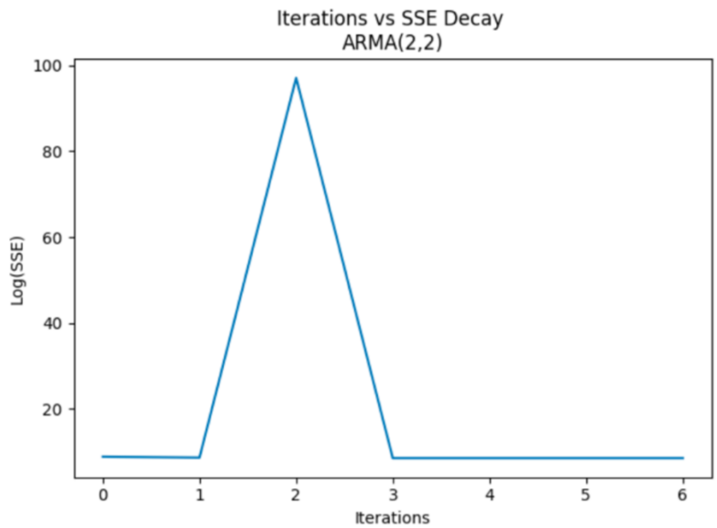
6. *ARMA(1,2) : y(t) + 0.5y(t-1) = e(t) + 0.5e(t-1) - 0.4e(t-2)*

```
Estimated Results:

Total # of Iterations: 8
True Theta:
[['a1' '0.5']
 ['a2' '0.0']
 ['b1' '0.5']
 ['b2' '-0.4']]
Estimated Theta:
[['a1' '0.5083693468392865']
 ['a2' '0.0024640669548186586']
 ['b1' '0.4954245378322593']
 ['b2' '-0.411451225990503 2']]
95% Confidence Interval:
[['Theta' ' Theta - 2xSTD ' ' Theta  + 2xSTD  ']
 ['a1' '0.42891082600820196' '0.587827867670371']
 ['a2' '-0.06599962849149732' '0.07092776240113463']
 ['b1' '0.42112697495228' '0.5697221007122386']
 ['b2' '-0.4842606563479169' '-0.33864179563308955']]
Covariance Matrix of Theta:
[['a1' '0.0015784141331659717' '0.0011883367328886587'
  '0.001379130321185114' '0.0013068756029052107']
 ['a2' '0.0011883367328886594' '0.0011718193985414767'
  '0.001087470208809741' '0.001129872486030602']
 ['b1' '0.0013791303211851131' '0.0010874702088097396'
  '0.0013800319624761195' '0.0013076923286260839']
 ['b2' '0.0013068756029052 1' '0.0011298724860306008'
  '0.0013076923286260843' '0.0013253032872427672']]
Theta STD:
[['a1' '0.03972926041554224']
 ['a2' '0.03423184772315799']
 ['b1' '0.03714878143998965']
 ['b2' '0.036404715178706824']]
Root Checks:
Num(bn): [1, 0.4954245378322593, -0.4114512259905032]
Den(an): [1, 0.5083693468392865, 0.0024640669548186586]
Roots Num: [-0.93532596  0.43990143]
Roots Den: [-0.50347523 -0.00489412]
Variance of Error:
0.9932249130768274

END
Relevant Parameters:
[['a1' '0.5083693468392865']
 ['b1' '0.4954245378322593']
 ['b2' '-0.4114512259905032']]
MA/Num parameters: [1, 0.4954245378322593, -0.4114512259905032]
AR/Den parameters: [1, 0.5083693468392865, 0.0]
```



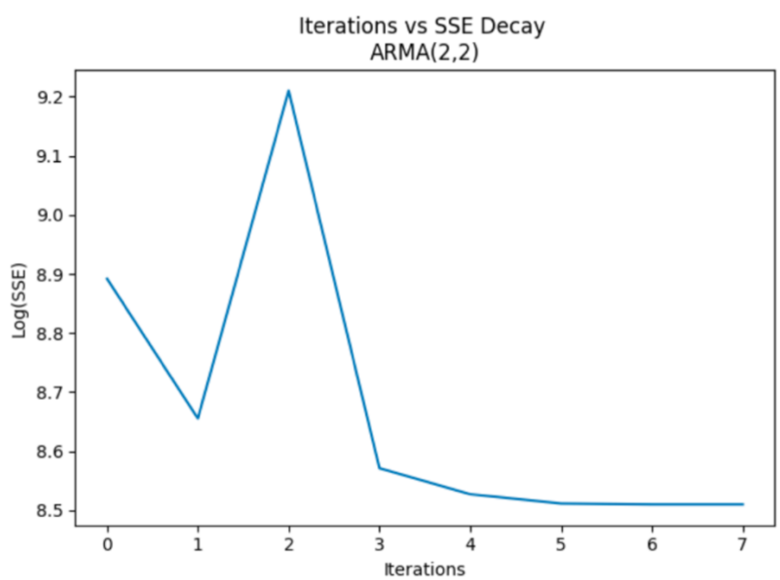
7. *ARMA(0,2) : y(t) = e(t)+ 0.5e(t-1) - 0.4e(t-2)*

```
Estimated Results:

Total # of Iterations: 6
True Theta:
[['a1' '0.0']
 ['a2' '0.0']
 ['b1' '0.5']
 ['b2' '-0.4']]
Estimated Theta:
[['a1' '0.0012505720144759943']
 ['a2' '-0.005404842275722472']
 ['b1' '0.4883101755627548']
 ['b2' '-0.41808642690682507']]
95% Confidence Interval:
[['Theta' ' Theta - 2xSTD ' ' Theta  + 2xSTD  ']
 ['a1' '-0.13311749376551685' '0.1356186377944688']
 ['a2' '-0.07465635886656907' '0.06384667431512414']
 ['b1' '0.35709728887782755' '0.619523062247682']
 ['b2' '-0.544089076971489' '-0.29208377684216114']]
Covariance Matrix of Theta:
[['a1' '0.00451369427536412' '0.0020060183075290496'
  '0.00430885746924766' '0.004103439787002129']
 ['a2' '0.0020060183075290496' '0.0011989431375330757'
  '0.002003795948192438' '0.001973891311081317']
 ['b1' '0.00430885746924766' '0.00200379594819244' '0.004304205408047888'
  '0.004099219648364628']
 ['b2' '0.004103439787002125' '0.0019738913110813197'
  '0.004099219648364626' '0.003969166955829541']]
Theta STD:
[['a1' '0.06718403288999641']
 ['a2' '0.0346257582954233']
 ['b1' '0.06560644334246361']
 ['b2' '0.06300132503233198']]
Root Checks:
Num(bn): [1, 0.4883101755627548, -0.41808642690682507]
Den(an): [1, 0.0012505720144759943, -0.005404842275722472]
Roots Num: [-0.93531219  0.44700201]
Roots Den: [-0.07414558  0.07289501]
Variance of Error:
0.9932212696225474

END
Relevant Parameters:
[['b1' '0.4883101755627548']
 ['b2' '-0.41808642690682507']]
MA/Num parameters: [1, 0.4883101755627548, -0.41808642690682507]
AR/Den parameters: [1, 0.0, 0.0]
```



8. *ARMA(2,2) :y(t)+0.5y(t-1)+0.2y(t-2)=e(t)+0.5e(t-1) - 0.4e(t-2)*

```
Estimated Results:

Total # of Iterations: 7
True Theta:
[['a1' '0.5']
 ['a2' '0.2']
 ['b1' '0.5']
 ['b2' '-0.4']]
Estimated Theta:
[['a1' '0.5026164460381817']
 ['a2' '0.1952637411413557']
 ['b1' '0.4896959392027901']
 ['b2' '-0.4172042178107724']]
95% Confidence Interval:
[['Theta' ' Theta - 2xSTD ' ' Theta  + 2xSTD  ']
 ['a1' '0.44813339666161967' '0.5570994954147437']
 ['a2' '0.14830324480955706' '0.24222423747315433']
 ['b1' '0.4378959552853531' '0.5414959231202271']
 ['b2' '-0.46837969655314693' '-0.36602873906839783']]
Covariance Matrix of Theta:
[['a1' '0.0007421006673422232' '0.0004850234485936707'
  '0.0006070298372619232' '0.0005712443074132682']
 ['a2' '0.0004850234485936706' '0.0005513220539322183'
  '0.0004426625189071244' '0.0004812337419882l736']
 ['b1' '0.0006070298372619233' '0.0004426625189071244'
  '0.0006708095834616836' '0.0006275343972979151']
 ['b2' '0.0005712443074132682' '0.0004812337419882l736'
  '0.0006275343972979151' '0.0006547324061278075']]
Theta STD:
[['a1' '0.027241524688281']
 ['a2' '0.023480248165899317']
 ['b1' '0.02589999195871851']
 ['b2' '0.025587739371187277']]
Root Checks:
Num(bn): [1, 0.4896959392027901, -0.4172042178107724]
Den(an): [1, 0.5026164460381817, 0.1952637411413557]
Roots Num: [-0.93561186  0.44591592]
Roots Den: [-0.25130822+0.36346653j -0.25130822-0.36346653j]
Variance of Error:
0.9932060632218003

END
Relevant Parameters:
[['a1' '0.5026164460381817']
 ['a2' '0.1952637411413557']
 ['b1' '0.4896959392027901']
 ['b2' '-0.4172042178107724']]
MA/Num parameters: [1, 0.4896959392027901, -0.4172042178107724]
AR/Den parameters: [1, 0.5026164460381817, 0.1952637411413557]
```



Conclusion:

The results of this study show that LM algorithm is accurate, efficient, and robust. The parameter estimations were accurate for all examples except example 4. The original ARMA (2,0) with AR parameters 0.5 and 0.2 were estimated at 0.65 and 0.19, but also with MA parameters 0.13 and -0.04. Even though these estimations were not accurate, this brings up the importance of validating estimations with confidence intervals. The confidence interval for these estimations show that the AR parameters are within the 95% confidence interval, meanwhile the MA parameters do not because they contain 0. Therefore, the ARMA process remains ARMA (2,0). Furthermore, convergence was reached within 5 – 10 iterations demonstrating opination within a handful of seconds. Finally, LM is robust as it seamlessly changes between GN and SGD in order to handle the different scenarios of parameters estimations being close and far from their optimal values. However, one issue that needed to be resolved were not due to the algorithm but with Python. Some calculation SSE were too big they approached infinity or “NaN”, which called for resetting SSE to 9999 to avoid errors and restarting the process.

APPENDIX:

```
import numpy as np
import pandas as pd
from scipy import signal
import matplotlib.pyplot as plt
import math

def step0(num_true,den_true):
    print("\nStep 0:")
    # create the true theta parameters vector based on na,nb parameters
    if len(den_true) == 1:
        theta_true = np.vstack((0,np.vstack(num_true[1:])))
    elif len(num_true) == 1:
        theta_true = np.vstack((np.vstack(den_true[1:]),0))
    else:
        theta_true = np.vstack((np.vstack(den_true[1:]),np.vstack(num_true[1:])))
    theta_true = [aa for bb in theta_true for aa in bb]
    # Initialize theta vector to zero based on the # of paremeters
    # Revmove the 1 at the begining of num and den since they are for dlsim package and NOT ture parameters
    nb = len(num_true) -1
    na = len(den_true) -1
    # number of parameters
    NP = na + nb
    # intialize parameters to 0 for the length of na + nb
    theta = np.zeros(NP)
    theta = theta.reshape(len(theta),1)
    print("Theta true:", theta_true)
    print("# na parm:",na)
    print("# nb parm:",nb)
    print("Theta shape:",theta.shape)

    return theta_true,theta,na,nb



# Function that takes in theta parameters and y
# Calculates white noise of function y
def calsim(y, theta, na, nb):
```

```python
    # make sure num and den have same size arrays for dlsim calculations

    # put AR in num
    den = [1] + list(theta[:na].flatten())
    # put MA in den
    num = [1] + list(theta[na:].flatten())

    system = (den, num, 1)

    _, e = signal.dlsim(system, y)
    # Change e from tuple to an array
    enew = np.zeros(len(e))
    for i in range(len(e)):
        enew[i] = e[i][0]

    enew = enew.reshape(len(e), 1)
    # print("Shape of e: ",enew.shape)
    # print("Length of e: ",len(enew))

    return enew



# Function that calcuates the SSE old
# Creates A and g paramters
def step1(y,theta,delta,na,nb):
    #print("Step 1:")

    e = calsim(y,theta,na,nb)
    SSEo = float(np.dot(e.T,e))
    X = np.zeros([na+nb,len(e)])
    for i in range(na+nb):
        #print("\n theta {} + delta:".format(i+1))
        theta2 = theta.copy()
        theta2[i] = theta[i] + delta
        e2 = calsim(y,theta2,na,nb)
        X[i] = ((e - e2)/delta).flatten()
        #print("Shape of X:",X.shape)

    # Need to transpose the X matrix for the proper shapes
    X = X.T
```

```
    #print("\nShape of X:",X.shape)
    A = X.T @ X
    #print("\nMatrix A:",A)
    #print("Shape of A:", A.shape)
    g = X.T @ e
    #print("\ng:",g)
    #print("Shape of g:", g.shape)

    return A,g,SSEo




# Step 2 update theta
def step2(y,theta,A,g,mu,na,nb):
    print("Step 2:")
    # Create Identity matrix with dimensions NP x NP
    I = np.identity(na+nb)
    #print("\nmu:",mu)
    muI = np.dot(mu,I)
    # Calculate change in theta
    dtheta = np.dot(np.linalg.inv(A + muI),g)
    #print("\nDelta Theta:")
    #print(dtheta)
    #print(dtheta.shape)
    #print("\nTheta old:")
    #print(theta)
    # Add the change in theta to old theta parameters
    thetaNew = np.add(theta,dtheta)
    #print("\nTheta New:")
    #print(thetaNew)
    # Calcualte new error with updated theta parameters
    e3 = calsim(y,thetaNew,na,nb)
    # Calculate new SSE term
    SSEn = float(np.dot(e3.T,e3))
    # make sure there are no Nan values, make it really big in case
    if math.isnan(SSEn) == True:
        print("SSE = NAN",SSEn)
        SSEn = 9999
    if np.isinf(SSEn) == True:
```

```python
        print("SSE = INF", SSEn)
        SSEn = 9999
    #print("\nSSE New:",SSEn)

    return SSEn,dtheta,thetaNew




# step 3 Convergence
def step3(y,thetaNew,dtheta,SSEn,SSEo,A,g,mu,muf,muMax,delta,na,nb,epsilon,theta_true,theta):
    MAX_ITER = 100
    iterations = 0
    SSEs = [SSEo]

    while iterations < MAX_ITER:
        #print("\nITERATION:",iterations)
        #print("\nSSE New",SSEn,"vs","SSE Old",SSEo)
        SSEs.append(SSEn)
        #print("Norm 2:",np.linalg.norm(dtheta,2)," vs Epsilon",epsilon)
        if SSEn < SSEo:
            print("Optimizing with Gauss-Netown")
            if np.linalg.norm(dtheta,2) < epsilon:
                print("Reached Convergance")

                theta = thetaNew
                variance_e = SSEn/ (len(y) - (na + nb))
                cov_theta = np.multiply(variance_e,np.linalg.inv(A))

                print("\nEstimated Results:")
                print("\nTotal # of Iterations:", iterations + 1)

                THETA, CI = results(theta,na,nb,cov_theta,variance_e,theta_true)

                iterations = np.arange(0, iterations + 2,1).tolist()

                plotSSE(SSEs,iterations,na,nb)

                pselect(THETA, CI)

                return theta, cov_theta,CI
```

```
            else:
                #print("Not Converged")
                theta = thetaNew
                #print("mu old:",mu)
                mu /= muf
                #print("mu new",mu)

        while SSEn >= SSEo:
            print("Optimizing with Gradient Descent")
            #print("mu old:",mu)
            mu *= muf
            #print("mu new:",mu)
            if mu > muMax:
                print("ERROR")
                return None
            SSEn, dtheta, thetaNew = step2(y,theta,A,g,mu,na,nb)

        iterations += 1

        if iterations > MAX_ITER:
            print("ERROR")

            return None

        #print("Theta:")
        #print(thetaNew)
        theta = thetaNew

        A,g,SSEo = step1(y,theta,delta,na,nb)

        SSEn, dtheta, thetaNew = step2(y,theta,A,g,mu,na,nb)







def results(theta,na,nb,cov_theta,variance_e,theta_true):
    # find the std for theta from covariance matrix
    std_theta = np.zeros(len(theta))
```

```
for i in range(len(std_theta)):
    std_theta[i] = np.sqrt(cov_theta[i][i])

# create confidence interval matrix
confi = np.zeros([len(theta),2])
for i in range(len(theta)):
    for j in range(len(theta)):
        if j == 0:
            confi[i][j] = theta[i] - (2 * std_theta[i])
        if j == 1:
            confi[i][j] = theta[i] + (2 * std_theta[i])

# Map CI to theta
NP = na + nb
VARS = list(np.zeros(NP))
for i in range(na):
    VARS[i] = "a" + str(i + 1)
for i in range(na,NP):
    VARS[i] = "b" + str(i - na + 1)

VARS = np.array(VARS)
VARS = VARS.reshape(NP,1)

headers = np.array(["Theta"   ," Theta - 2xSTD "," Theta  + 2xSTD  "])
headers = headers.reshape(1,3)

CI = np.concatenate((VARS,confi),axis = 1)
CI = np.concatenate((headers,CI,),axis = 0)

THETA = np.concatenate((VARS,theta),axis = 1)
COV = np.concatenate((VARS,cov_theta),axis = 1)

std_theta = std_theta.reshape(len(std_theta),1)
STD = np.concatenate((VARS,std_theta),axis = 1)

theta_true = np.array(theta_true)
theta_true = theta_true.reshape(len(theta_true),1)
TRUE = np.concatenate((VARS,theta_true),axis = 1)

print("True Theta:")
print(TRUE)
```

```python
    print("Estimated Theta:")
    print(THETA)
    print("95% Confidence Interval:")
    print(CI)
    print("Covariance Matrix of Theta:")
    print(COV)
    print("Theta STD:")
    print(STD)
    print("Root Checks:")
    num = [1] + list(theta[na:].flatten())
    den = [1] + list(theta[:na].flatten())
    print("Num(bn):",num)
    print("Den(an):",den)
    print("Roots Num:",np.roots(num))
    print("Roots Den:",np.roots(den))
    print("Variance of Error:")
    print(variance_e)
    print("\nEND")

    #df = pd.DataFrame(data=CI)

    return THETA,CI




def plotSSE(SSEs,iterations,na,nb):
    plt.plot(iterations,np.log(SSEs))
    plt.title("Iterations vs SSE Decay \nARMA({0},{1})".format(na,nb))
    plt.xlabel("Iterations")
    plt.ylabel("Log(SSE)")
    plt.xticks(list(range(0, len(iterations),1)))
    plt.show()



def pselect(theta, ci):
    # find whhich parameters do not include 0 in their CI
    x = []
    for i in range(len(theta)):
        if float(ci[i + 1][1]) < 0 > float(ci[i + 1][2]):
            x.append(ci[i + 1][0])
```

```
    if float(ci[i + 1][1]) > 0 < float(ci[i + 1][2]):
        x.append(ci[i + 1][0])
    # collect the relevant theta values
y = []
for i in range(len(theta)):
    for j in range(len(x)):
        if theta[i][0] == x[j]:
            y.append(float(theta[i][1]))

theta1 = np.array(x)
theta1 = theta1.reshape(len(theta1), 1)
values = np.array(y)
values = values.reshape(len(values), 1)
prediction_vars = np.concatenate((theta1, values), axis=1)

print("Relevant Parameters:")
print(prediction_vars)

# formate parameters for dlsim
num = [1]
den = [1]

for i in range(len(prediction_vars)):
    if (str(prediction_vars[i][0]).startswith("a")) == True:
        den.append(float(prediction_vars[i][1]))
    else:
        num.append(float(prediction_vars[i][1]))

na = len(den)
nb = len(num)

if na < nb:
    zeros = np.zeros(nb - na)
    den = den + list(zeros)
if nb < na:
    zeros = np.zeros(na - nb)
    num = num + list(zeros)

print("MA/Num parameters:", num)
print("AR/Den parameters:", den)
```

```
    return prediction_vars, num, den



def LM(N,num_true,num_den):
    delta = 1e-6
    mu = 0.1
    muf = 10
    muMax = 10e10
    epsilon = 0.001
    np.random.seed(42)
    mean = 0
    std = 1
    e = std * np.random.randn(N) + mean

    # Create the synthetic data to which we want to estimate parameters
    system = (num_true,den_true,1)
    # noinspection PyTupleAssignmentBalance
    x,y = signal.dlsim(system,e)

    # Step 0 intialize theta
    theta_true,theta, na, nb = step0(num_true,den_true)
    # step1
    A,g,SSEo = step1(y,theta,delta,na,nb)
    # step2
    SSEn, dtheta, thetaNew = step2(y,theta,A,g,mu,na,nb)
    # step3
    theta,cov_theta, CI = step3(y,thetaNew,dtheta,SSEn,SSEo,
                                A,g,mu,muf,muMax,delta,
                                na,nb,epsilon,theta_true,theta)




print("Print Example 1")
# y(t) - 0.5y(t - 1) = e(t)
N = 5000
num_true = [1,0]
den_true = [1,-0.5]
LM(N,num_true,den_true)
```

```
print("Example 2")
#====Example 2=======================
# ARMA(0,1) : y(t) = e(t) + 0.5e(t-1)

num_true = [1,0.5]
den_true = [1,0]
LM(N,num_true,den_true)




print("Example 3")
#====Example 3===================================
# ARMA(1,1) : y(t) + 0.5y(t-1) = e(t) - 0.5e(t-1)

num_true = [1,-0.5]
den_true = [1,0.5]
LM(N,num_true,den_true)




print("Example 4")
#====Example 4===================================
# ARMA(2,0) : y(t) + 0.5y(t-1) + 0.2y(t-2) = e(t)
num_true = [1,0,0]
den_true = [1,0.5,0.2]
LM(N,num_true,den_true)




print("Example 5")
# ====Example 5===============================================
# ARMA(2,1) : y(t) + 0.5y(t-1) + 0.2y(t-2) = e(t) - 0.5e(t-1)
num_true = [1,-.5,0]
den_true = [1,0.5,0.2]
```

```
LM(N,num_true,den_true)



print("Example 6")
#====Example 6===============================================
# ARMA(1,2) : y(t) + 0.5y(t-1) = e(t) + 0.5e(t-1) - 0.4e(t-2)

num_true = [1,.5,-0.4]
den_true = [1,0.5,0]
LM(N,num_true,den_true)




print("Example 7")
#====Example 7================================================
# ARMA(0,2) : y(t) = e(t)+ 0.5e(t-1) - 0.4e(t-2)

num_true = [1,0.5,-0.4]
den_true = [1,0,0]
LM(N,num_true,den_true)



print("Example 8")
#====Example 8================================================
# ARMA(2,2) :y(t)+0.5y(t-1)+0.2y(t-2)=e(t)+0.5e(t-1) - 0.4e(t-2)

num_true = [1,.5,-0.4]
den_true = [1,0.5,0.2]
LM(N,num_true,den_true)
```

SOURCES: