ANALYZING ORACLE PERFORMANCE USING TIME SERIES MODELS

Chen (Gwen) Shapira,HP Software-as-a-Service

1. Introduction

Part of the work of an Oracle DBA is to monitor key performance indicators such as CPU utilization, load average, number of concurrent users on the system, response times and data sizes. These performance indicators change constantly and tracking the changes is critical for multiple reasons:

- Knowing that the current behavior of the system is abnormal and respond. For example when a critical query that normally takes 3 seconds starts taking 8, the DBA will want to find the cause for the problem and solve the issue.
- Noticing changes in the system, even when they are not an immediate cause for alert. For example, if in the last week the average number of users on the system changed from 16 to 35, it may be a cause for investigation.
- Using long term trends to predict future needs of the system. For example, the responsible administrator will note that data size is slowly growing and order new disks long before she encounters space shortage.
- Knowing the normal performance variation of the system, it is possible to negotiate for realistic and responsible SLAs with the business – using hard data instead of wishful thinking and guesswork.

Merely monitoring the system and collecting information is not enough. In order to perform the tasks described above, the DBA needs to analyze the data and separate the expected variations in system behavior from the variations that indicate a problem or a substantial change.

Currently, to perform these tasks database administrators use either simple statistical analysis tools such as averages and standard deviations, regression analysis or queue analysis. [1]

While all these techniques are useful, they do not take into account the variations in the system behavior that occur over time. Recognizing abnormal system load, for example, requires knowing that 80% cpu utilization is normal on Mondays but can represent a problem on Fridays.

In this paper I show how to use several techniques from the field of time series analysis that can be used to understand the system behavior and make predictions regarding future behavior.

I will begin by briefly explaining how and why to prepare the data for analysis. This is a critical first step that can be easily overlooked with disastrous results.

Then we will look in detail at the three major components of every time series: Long term trends, seasonal variation and short term correlation effects. I will show techniques to analyze and model each component separately and once analyzed I will show how to remove its effects from the data to allow analyzing the next component without interference.

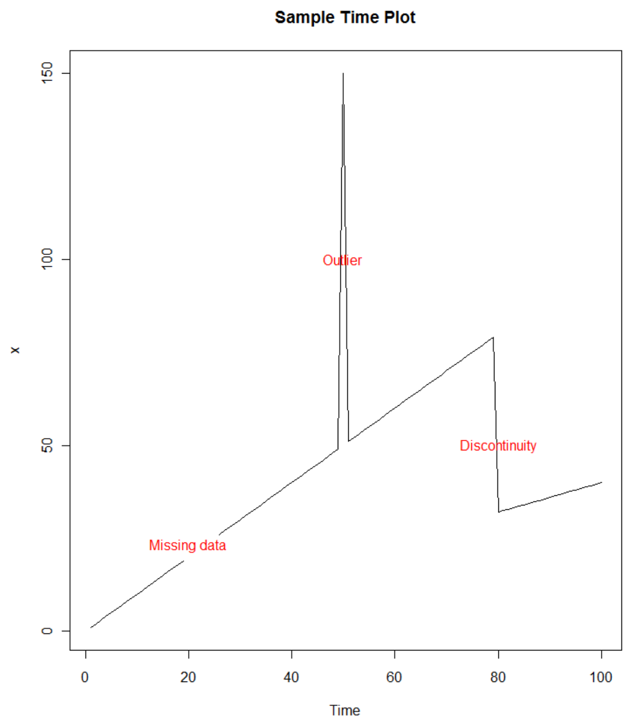
Once all the components are modeled, I will show how to use the results to recognize abnormal behavior in our system and how to use the models we have built to predict future system growth.

For each technique that I will explain and demonstrate, I will also provide scripts that perform the analysis I describe. All the code I used to demonstrate techniques, generate data and analyze real data sets is in a file named R_code.txt which should be found with this paper, as well as the data sets I used throughout this paper.

2. Preparing the Data

Even the best analysis in the world will yield useless results if the data does not lend itself to correct analysis.

The first step in any time series analysis is to generate a graph of the observations over time. The graph, called time plot, will show the important features of the data - trend, seasonality, outliers, missing data points and discontinuities (points where the nature of the data completely changes).



If the plot shows outliers and discontinuities, it will be necessary to correct the data before we start analyzing it - or they may affect our analysis and will make our results meaningless.

Correcting the data can be done using the following methods:

1. Replace outliers and missing data with average value. Note that if your data has strong trend or seasonality, you will need to use local average values because the overall average will be meaningless. The process of replacing missing data (or outliers) with meaningful average data is also known as interpolation and this is probably the most common and most intuitive way of fixing data.
2. If the data is missing in a systematic way then replacing missing data with averages is not a good idea. For example, a monitoring limitation causes data to be routinely missing when the load is very high. In this case, we know that the missing data was not average. What we can do is replace the missing values with maximum values of the data we did measure.
3. If the data has discontinuities in it, analyze each part of the data separately and fit a different model to each part. In forecasts you will probably want to use only the latest model, assuming you have reasons to believe that this behavior will continue into the future.

3. Time Series Analysis Techniques

Through the first section of the paper we will use fabricated data to demonstrate the analysis techniques. The data is attached to this presentation is a file named FakeCPU3.csv.

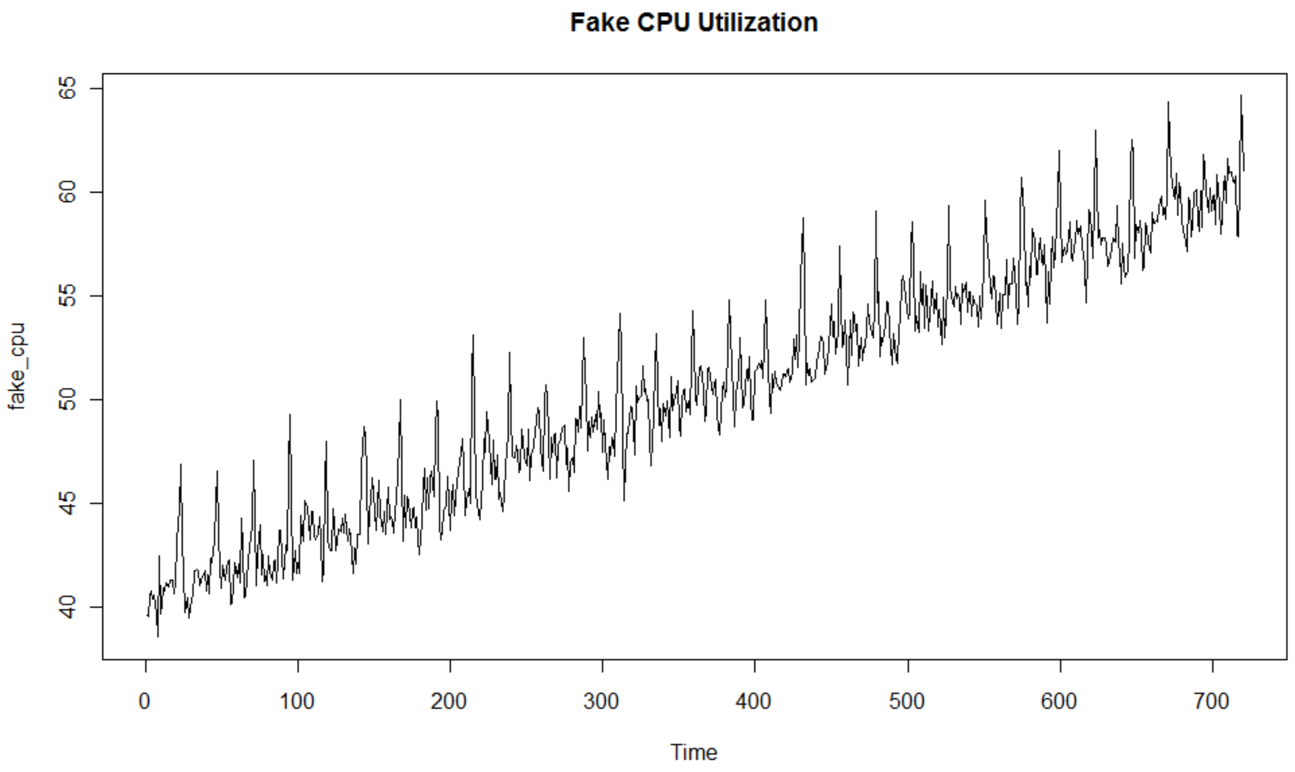
The following R code loads the fabricated data:

```
fake_cpu <- read.csv(file="C:/TS_Presentation/FakeCPU3.csv")
```

fake_cpu <- fake_cpu[,2]

Fabricated data has the distinct advantage of containing all the time-series components I am interested in demonstrating, while real-life data (presented and analyzed in section 6) often only has one or two of the components.

Here is the time plot of the data we are about to analyze:



The data in the graph represents measurements of CPU utilization on a single server, measured every hour over the course of 30 days.

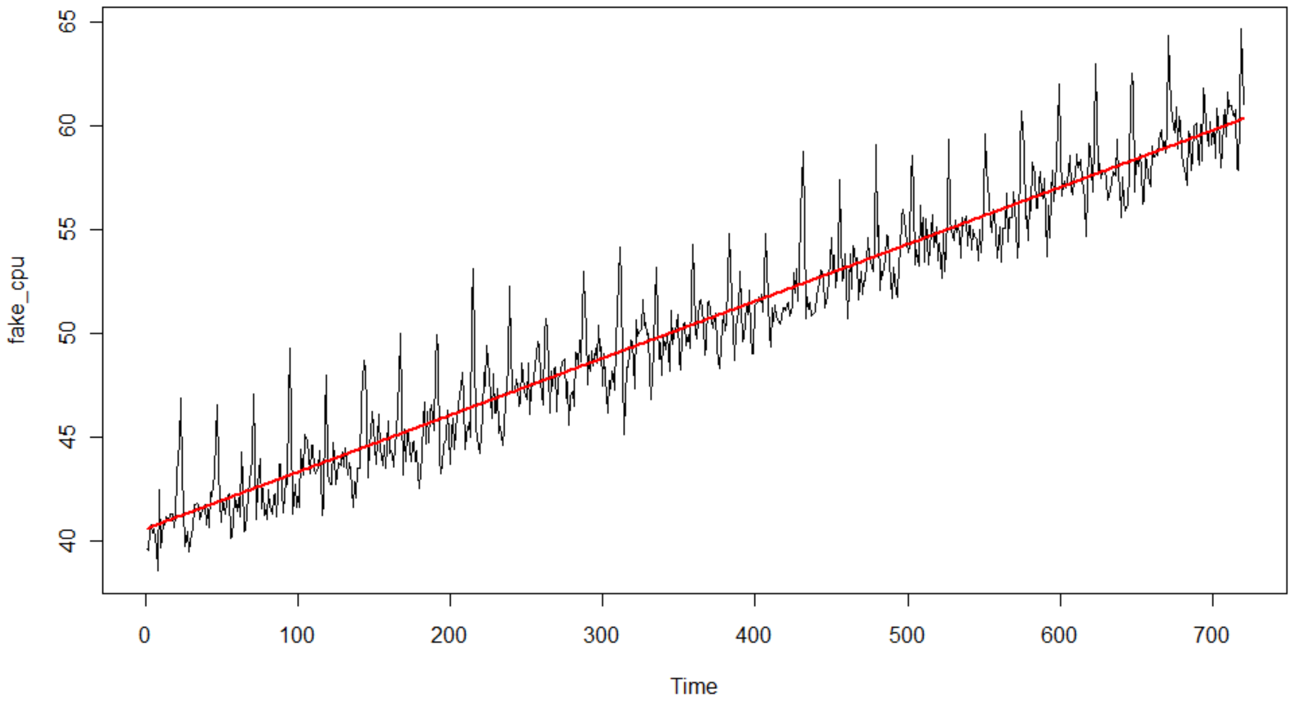
Trend

Trend is the long term variation in the average level. In our case, we can see from the graph that the average CPU on our system rises from 40% at the beginning of the month to 60% at the end. This means that we cannot describe the data using a simple average - saying "This system has 50% average CPU utilization" is extremely misleading. We need a different way to describe the changing average levels of our CPU utilization over time.

The easiest and perhaps most well known way to describe the trend in a data set is the **linear model**.

Our data shows the value of the CPU utilization at consecutive times 1,2,3...720. We can use a linear model to describe the average CPU utilization at time t using the following formula:

```
CPU = 40 + 0.02*t + error
```



At each point t the local average is 40+0.02*t. This means that when we start measuring (time 0) our CPU utilization is around 40%, each additional measurement shows the CPU around 0.02 higher, so 10 days (and 240 measurements later) CPU is at around 45% utilization.

To generate this model in R we ran:

```
   t <- c(1:720)
   plot.ts(fake_cpu)
   model <- lm(fake_cpu~t) #This part generates a linear model
   lines(model$fit,col=2,lwd=2)
```

This kind of analysis works very well on my demo data, but it will rarely work on real performance measurements. In practice, linear trends generally provide unrealistic data models.

One possibility of a more realistic model is piecewise linear trend - where we fit a local linear model to each part of the data. Another approach can be to fit a more complex function such as a polynomial curve.

Instead of attempting to describe the local averages in the data using increasingly complex functions, we can use a moving average, also called “linear filter” or “smoothing function” to calculate the local averages directly.

The technique for calculating a moving average is simple:

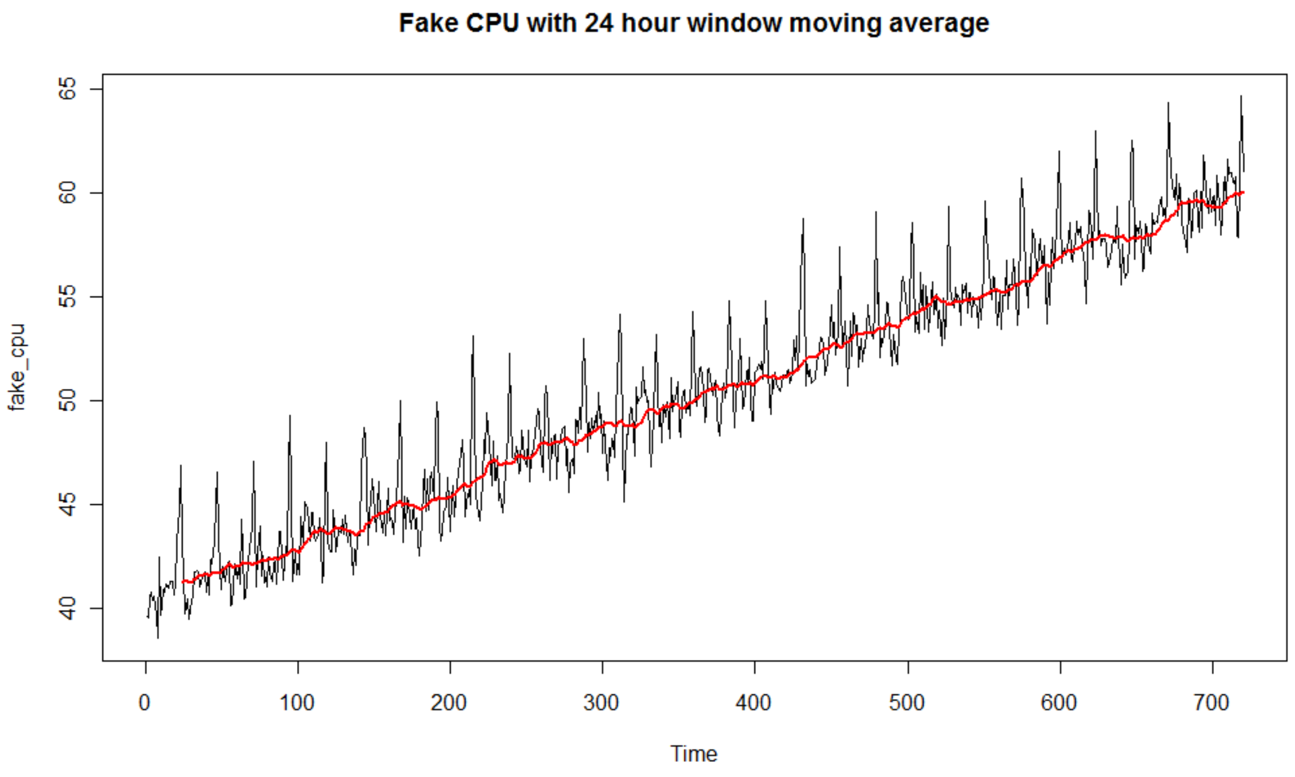
We decide on a window size - the number of measurements that will be including in each average. Suppose we pick 24 measurements as our window. We start by calculating the average of measurements 1-24, the average of measurements 2-25, and so on. The collection of resulting averages is the trend of our series.

While it does not give a simple formula as a model, it does describe the trend of our data, since it gives the average CPU utilization for each point in time. As will be shown in section 5, moving averages can be used for predicting future changes.

While the technique is simple – there are many possible variations. How do we pick the window size? Smaller windows will describe the data more accurately, but will be more influenced by accidental variations. Larger window sizes will have a larger delay until they reflect changes in the data, but they will include less variance.

In addition, not all measurements in the window need to be given an identical weight. The moving average can easily be a weighted average. It makes sense to give less weight to measurements that are farther into the past and more weight to more recent measurements.

Statistical tools such as R allow you to create a window of any size and assign any weights. In my data analysis I used a simple average with no weights on a window of 24 measurements (a day). There are well researched domains such as stock analysis or weather predictions where there are different windows and weights that are frequently used and considered useful by researchers of the domain. There is no such body of knowledge for time-series analysis for server performance measurements so some experimentation was required in order to find the window that best describes the data.

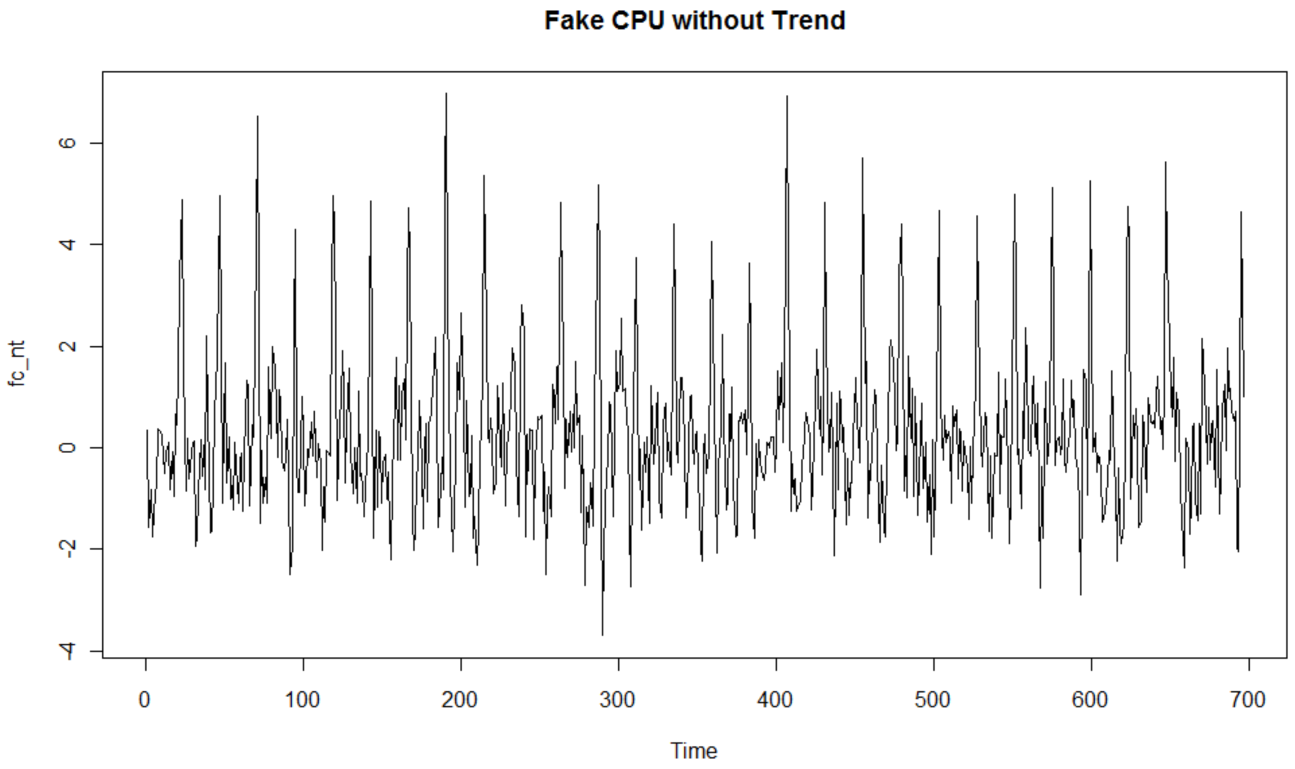


```
#Apply moving average to fake CPU
# window size is 24, each measurement has equal weight (1/24)
ffc <- filter(fake_cpu,rep(1/24,24),sides="1")

#Draw the time series with the moving average
plot.ts(fake_cpu)
lines(ffc,col="red",lwd=2)
```

Once the trend was discovered and modeled, it should be removed from the data to allow analysis of local fluctuations.

Removing the trend is done by taking the value given by the model for each point in time (either by calculating the result of the linear function or by taking the average calculated over a window for this point) and subtracting it from the value in the original data.

```
# Since we have took a moving average with window size of 24,
# the first 24 points will be missing from result
# This means that in order to remove the trend,
# we need to remove the first 24 measurements from our data

fake_cpu <- fake_cpu[-(1:24)]
ffc <- ffc[-(1:24)]

# After handling this difficulty
# We simply substract the averages from our data
fc_nt <- fake_cpu - ffc
```



You can see that this graph has no visible trend. In addition the average level at every point in time is 0 – a direct result of subtracting the previous average level from the data.

Seasonality

Once the trend has been discovered and removed, we can look for seasonal variations.

Many performance indicators exhibit cyclic variation - archive log directories grow during the day and then drop in size during the night backup, load on servers tends to be higher on Monday, etc. These are all expected variation of our data and we need a way to take them into account when looking for abnormal variation.

We'll start by looking for the seasonal cycle in our data. This step is often skipped as the cycles are known in advance and are part of the business plans. Once the cycle is known, we can proceed to remove it from our data set.

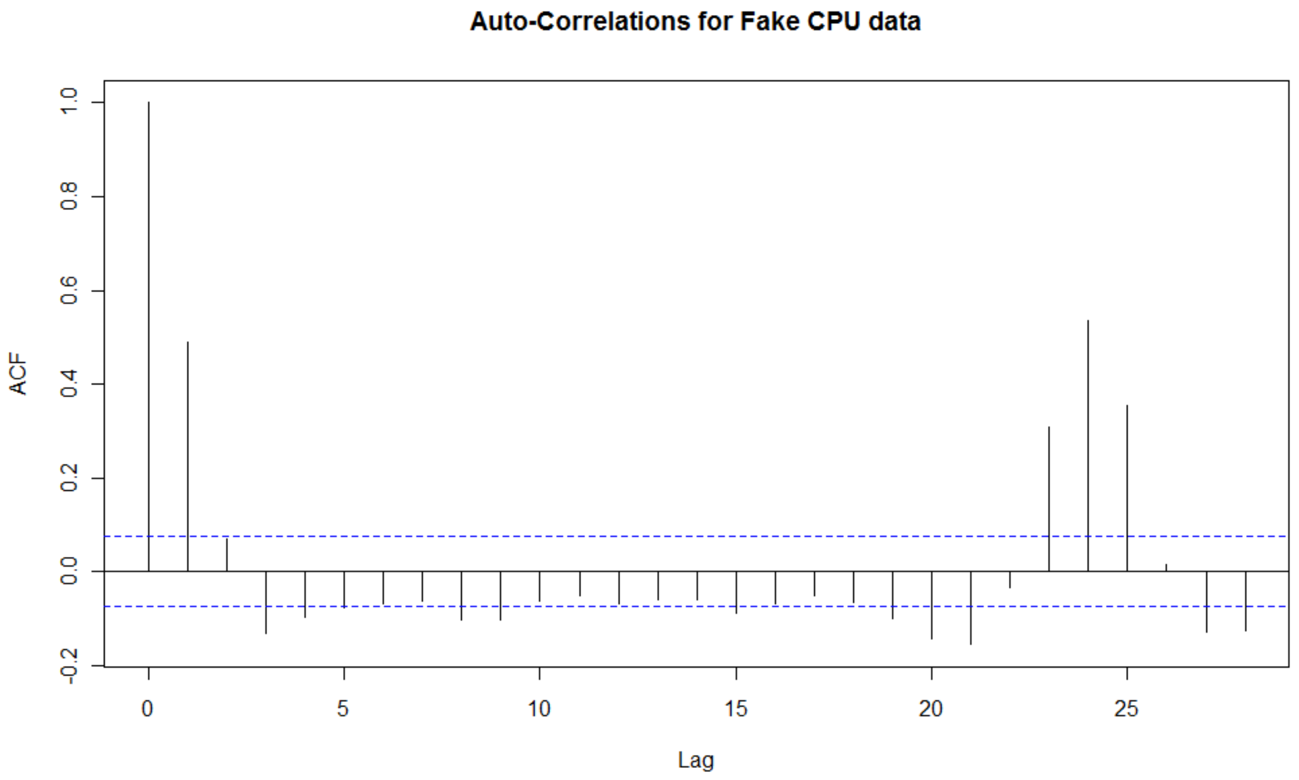
To discover seasonal cycles in our data we will use a graph called a Correlogram. This graph shows the strength of a correlation between data points at different intervals.

The idea is that just like since can test for correlation between two different data sets (for example - height and salary) using statistical tools such as Pearson's correlation coefficient [2]. We can use the same tools to check the correlation between each measurement and the measurement taken immediate after it. That type of correlation is called autocorrelation with a lag of 1. Autocorrelation with a lag of 2 is the correlation between a measurement and the measurement taken 2 intervals later.

Note that a Correlogram is seriously affected by outliers and trends, so these must be removed before using the Correlogram to detect seasonality.

We create a Correlogram by calculating autocorrelation for all lags in a specific range (normally 1 to 20) and then plotting the results in a graph. Obviously, we don't do it ourselves; we let R do it for us.

```
#generate a correlogram for fake_cpu with no trend data
acf(fc_nt)
```

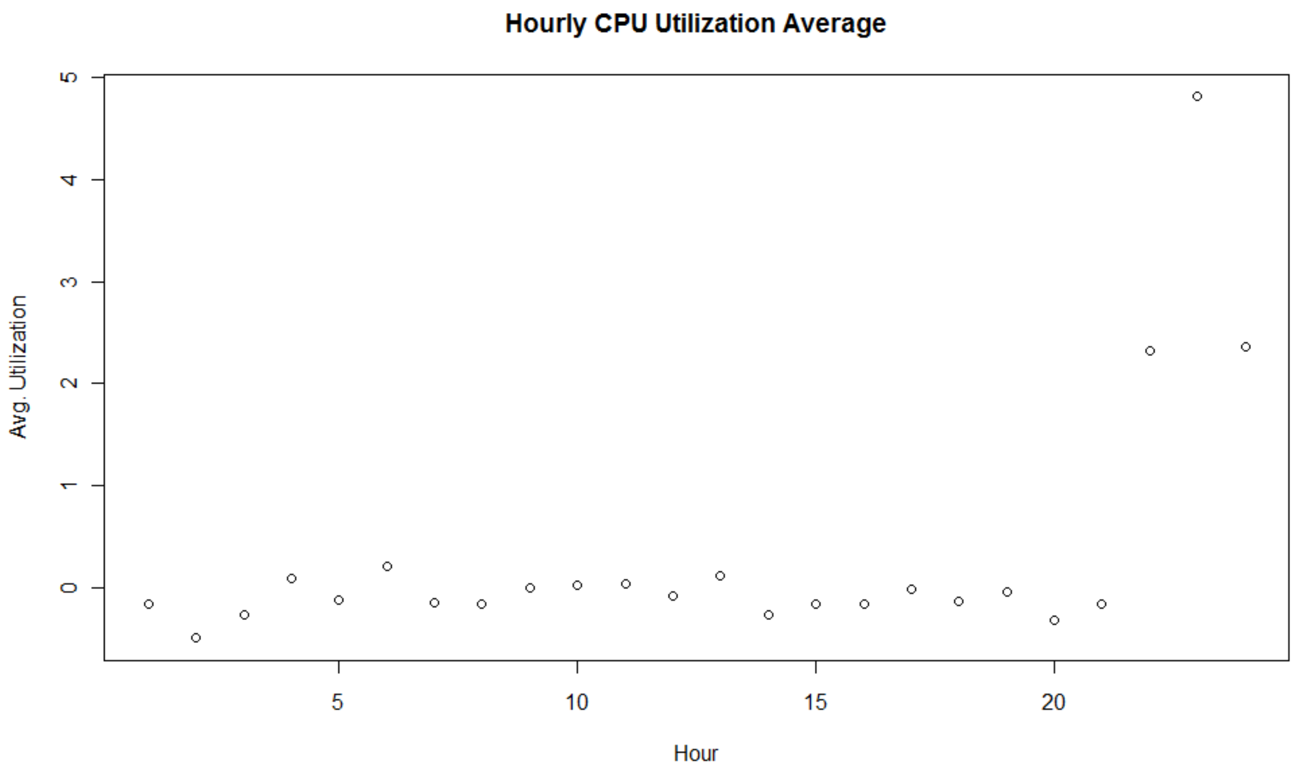


As you can see, in our data set there is a strong correlation between each point and the point measured 24 intervals later. In our case this indicates a 24 hour cycle of CPU utilization. Perhaps a heavy batch jobs that runs at the same time every night.

Note that the 24 hour cycle in the seasonality analysis is not related to the 24 hour window previously used for moving average. I could have used any window size for the moving average, while the seasonality of 24 hours stems from our analysis of the data.

Now that we discovered our cycle, we need to remove it so we can continue our analysis. First, we estimate the seasonal effect by taking the average of each hour separately.

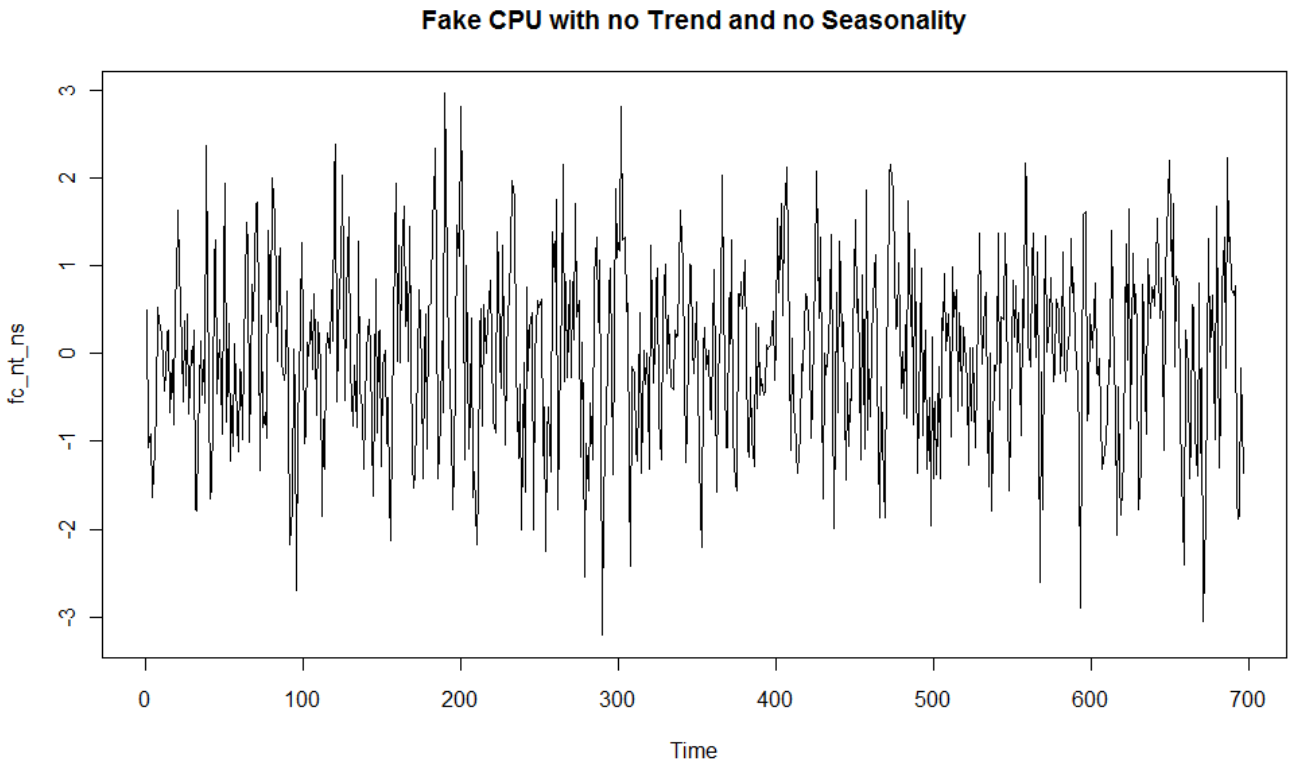
```
# We turn our data vector into a matrix
# Each column containing a specific hour
fc_nt_matrix <- matrix(fc_nt,ncol=24,byrow=T)

# Then we calculate the average on each column
fc_nt_hour_mean <- apply(fc_nt_matrix,2,mean)

# And plot the results
plot(1:24,fc_nt_hour_mean)
```



We can clearly see that the average for most hours is 0 (since we removed the trend we expect the average value to be 0) except for hours 22, 23 and 24 where the average is much higher due to the seasonal effect.

We will remove the seasonal effect by simply subtracting from each data point the corresponding hourly average.

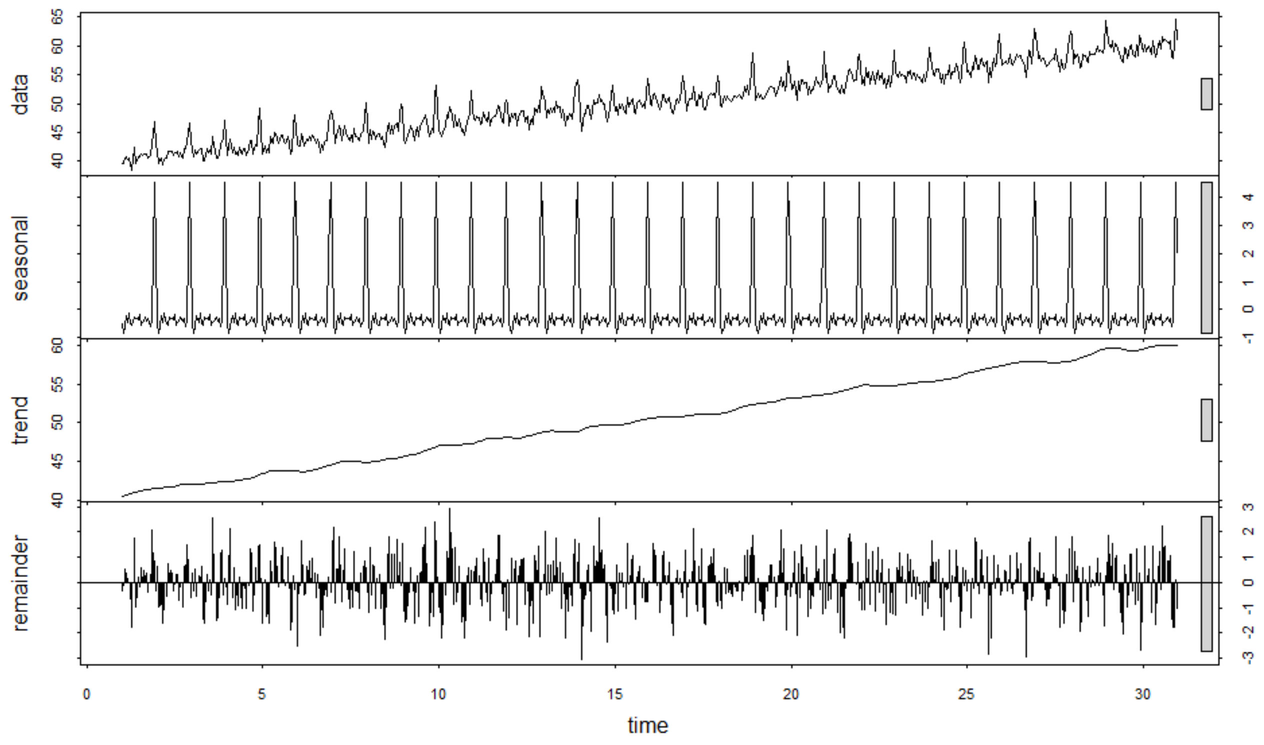
```
# The function rep() repeats the vector with hour averages
# 29 times, so we can subtract the hour averages
# from every day in our data set
fc_nt_ns <- fc_nt - rep(fc_nt_hour_mean,29)
```



Note that we no longer see the recurring "spikes" of high CPU in the graph.

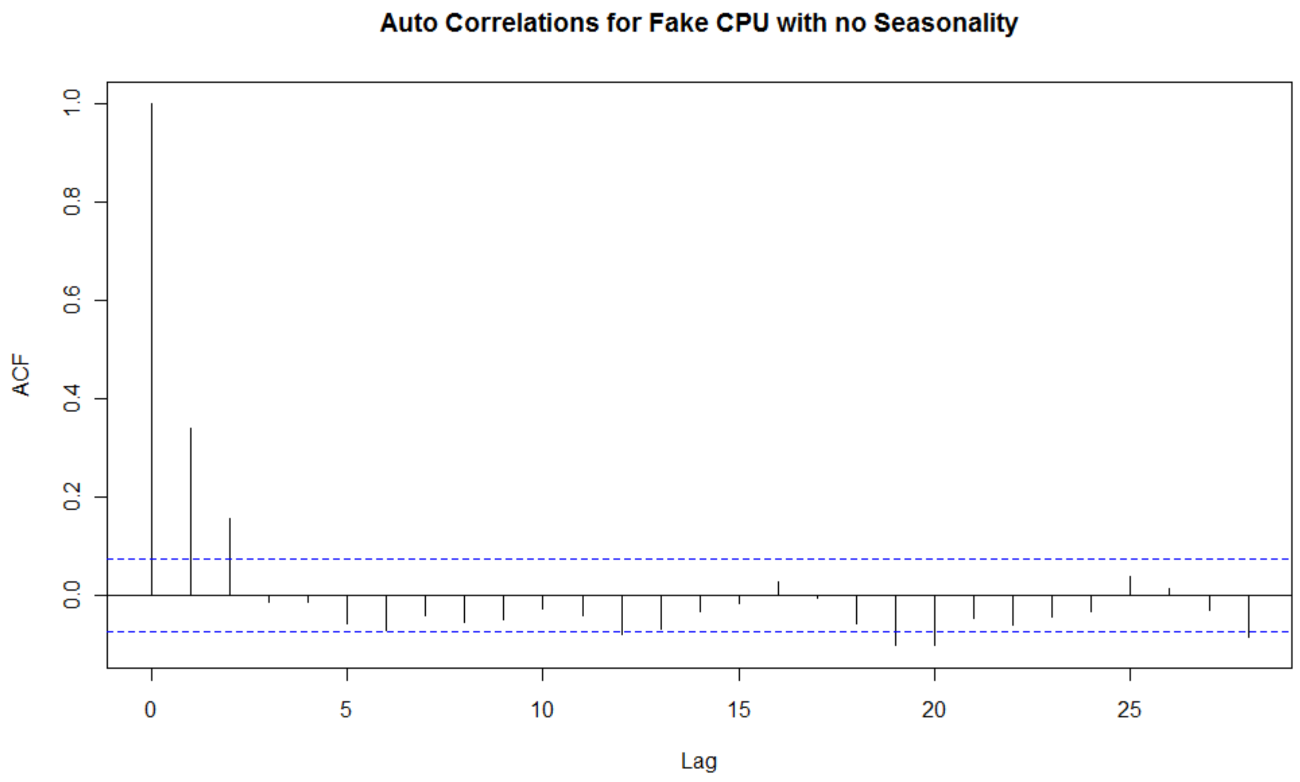
Using R we can also visualize both trend and seasonal components together. The STL() function does the work of breaking the series into components for us, and displays the components in very nice looking graph. It does require that we will know the seasonality of the series and pass it as a frequency parameter:

```
plot(stl(ts(fake_cpu,frequency=24),s.window="periodic"))
```


data
seasonal
trend
remainder
time
0
5
10
15
20
25
30
40
45
50
55
60
65
-1
0
1
2
3
4
-3
-2
-1
0
1
2
3

Auto-correlation

If we look at the Correlogram again after removing the seasonality effects from the data, we can see that it also shows a correlation between each measurement and the measurement immediately following it.



This means that the CPU in each hour is related to the CPU in the next hour. High CPU tends to remain high while low CPU tends to remain low.

Once we recognize the auto-correlation effect, we can use R to create a model that describes our data.
The graph shows strong correlation between measurements one element away and maybe two elements away (lag 1 and maybe 2), after that the correlation becomes insignificant.

This means that the model we are looking for will be of a type: Xt = a*Xt-1 + e
The current element is more or less equal to a previous element multiplied by a parameter.

We can tell R that we are looking for this one parameter, or we can let R decide on the number of appropriate parameters on its own.

```
> ar(fc_nt_ns,FALSE,1)
Coefficients:
     1
0.3404

Order selected 1  sigma^2 estimated as  0.9358

> ar(fc_nt_ns) #R chose to model our data with 3 parameters
Coefficients:
      1        2        3
 0.3290   0.0765  -0.0930

Order selected 3  sigma^2 estimated as  0.9284
```

According to R, we can model our data as:

```
Xt = 0.34Xt-1 + e
```

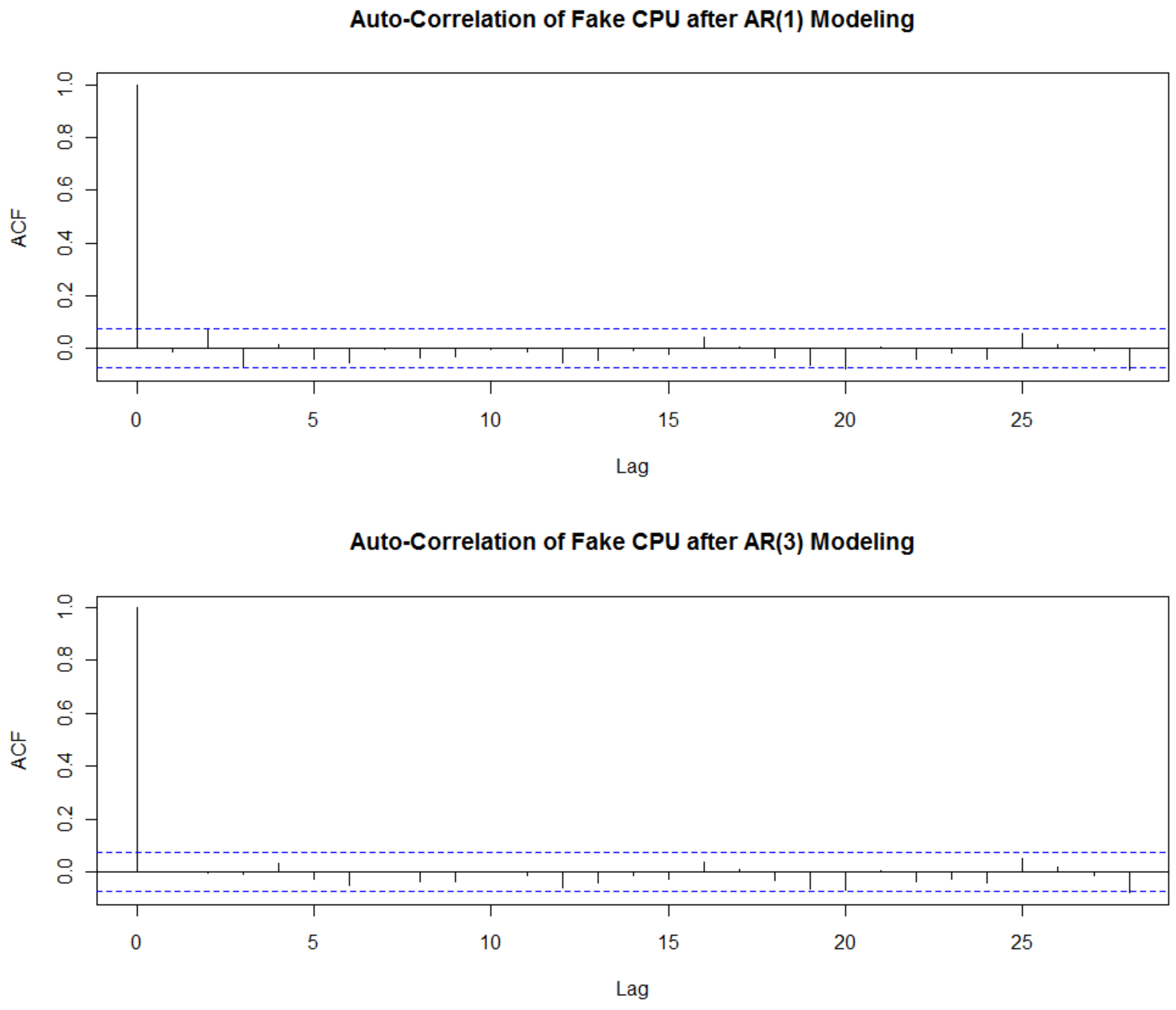
or

```
Xt = 0.33Xt-1 + 0.07Xt-2 - 0.09Xt-3 + e
```

The first model is called AR(1) and the second AR(2). These models explain how past data generates current data. Later when we deal with predictions, we'll use this model to generate future data.

When we have to choose between two models that explain our data, we always prefer the simplest model – the one with the lowest number of parameters. We can check if the model is a good fit by generating a Correlogram of the residuals (the data after removing the auto-correlation effect). The same function that created the model for us (AR) will also give us a vector of the residuals.

```
par(mfrow=c(2,1))
acf(ar(fc_nt_ns,FALSE,1)$resid[2:696])
acf(ar(fc_nt_ns,FALSE,3)$resid[4:696])
```

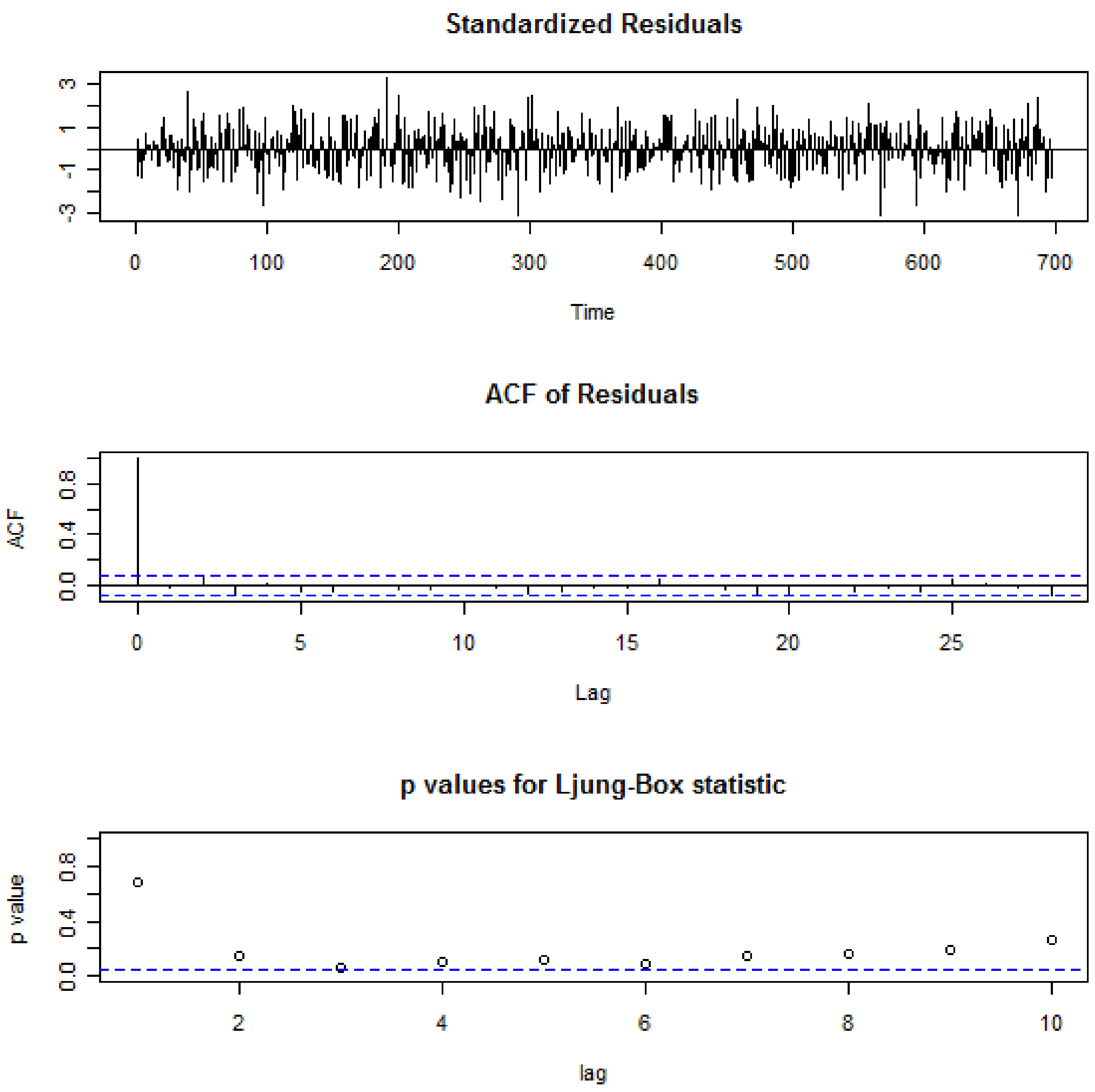


We can see that in both cases after removing the auto-correlation model the data has no auto-correlations left.

R can also generate a graph that shows multiple tests for model fit:

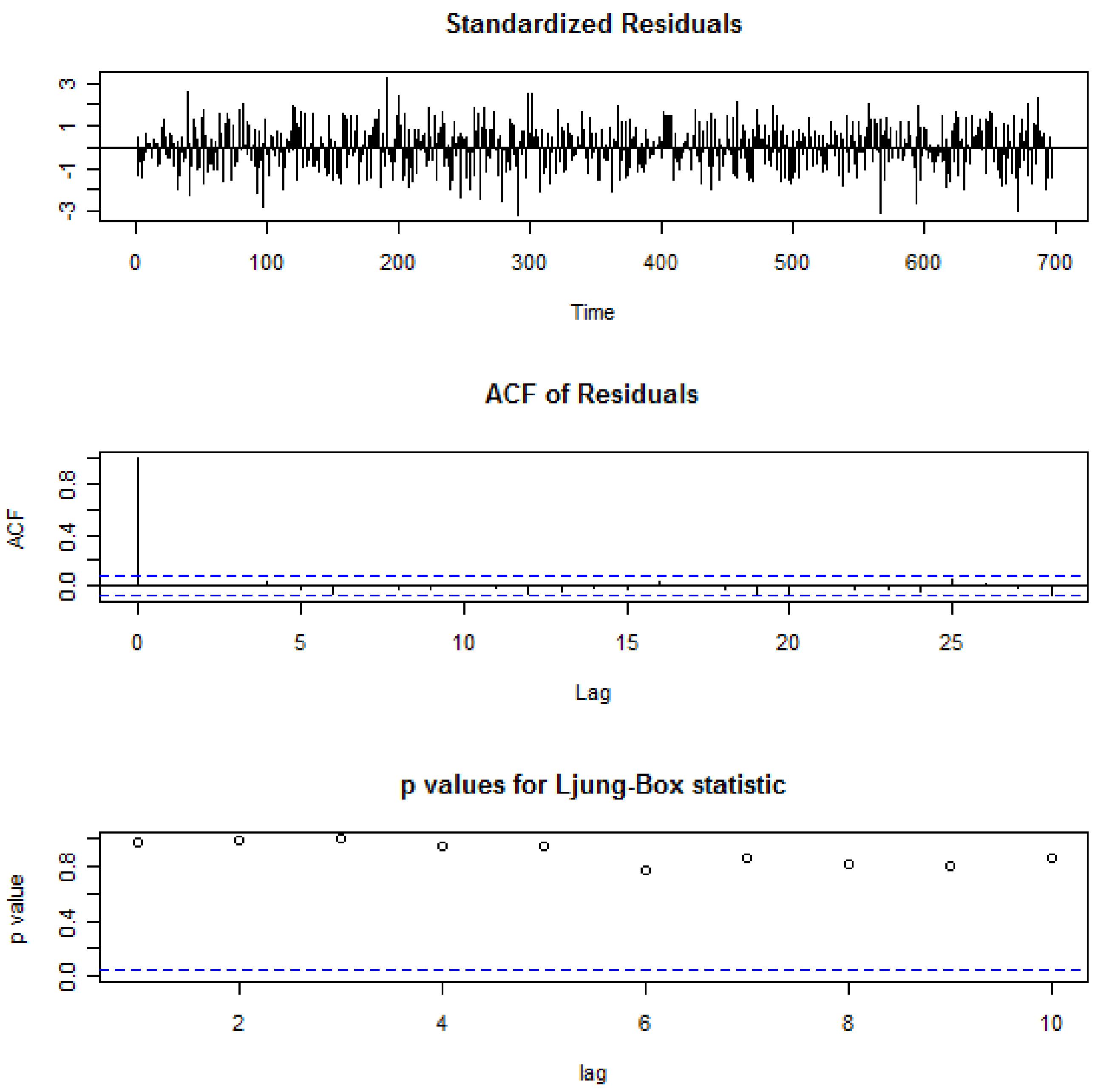
Diagnostics plot for AR(1) Model:

```
tsdiag(arima(fc_nt_ns,c(1,0,0)))
```



Diagnostics plot for AR(3) model:

```
tsdiag(arima(fc_nt_ns,c(3,0,0)))
```

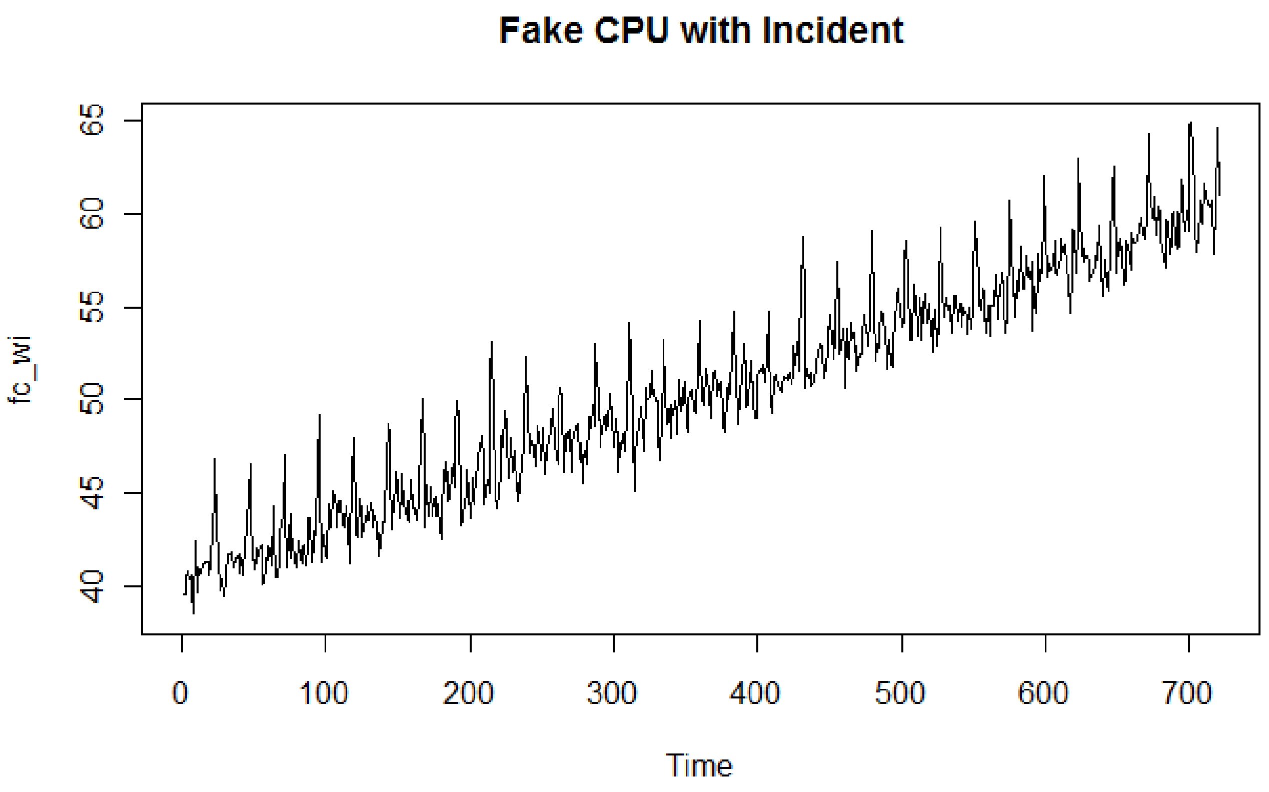


The plots show the residuals themselves, the Correlogram , and a third plot which is a visualization of a statistical test checking for lack of auto-correlation. Low values for a specific lag means that the probability they are un-correlated is low. In our first model, the test did not find enough evidence for independence between measurements; in the second model the residuals are clearly independent.

Once the residuals are independent, we are left with what appears to be random noise. This noise can't be explored with the current model, so we know our exploration and modeling work is done. We got all the time series information we can out of our data, and we can now proceed to using it.

4. Identify abnormal data using TS analysis

Below is the time-plot of the CPU data we analyzed in the previous section. I've added an incident to the data - few hours of relatively high CPU. Not 100%, but higher than we can expect. This type of change could be an indication that perhaps something is starting to go wrong.



How can we detect the incident?

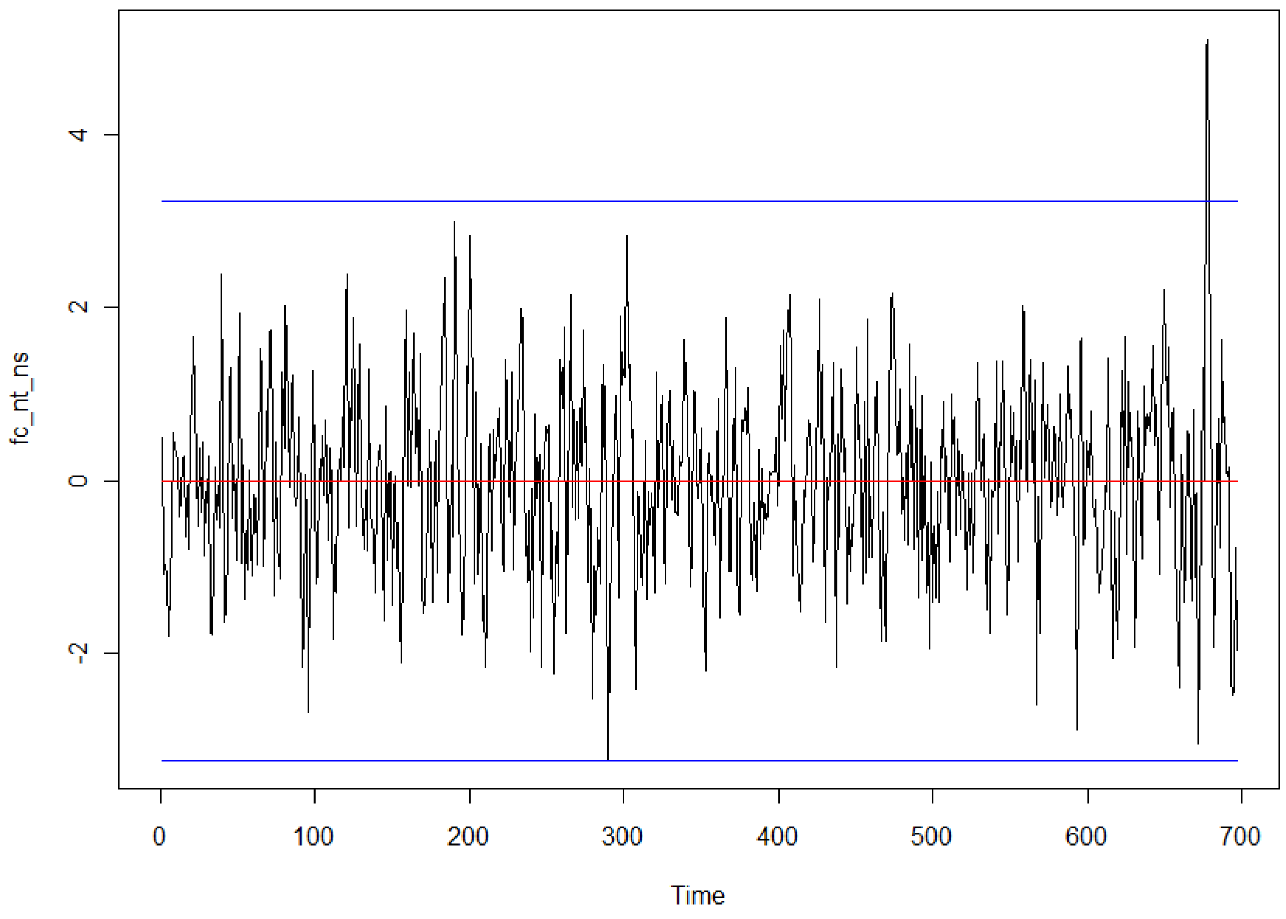
We'll define as an abnormal behavior as CPU utilization that is more than 3 standard deviations outside the average. The problem is that while the data has a trend and seasonal cycles we cannot calculate a meaningful average.

Also, it is obvious that our CPU has so much variance and noise that we can't detect an abnormal data point visually, unless it is really far outside the norm.

We start by removing the trend and seasonal effects from the series, as we did in section 3.

```
#Find Trend
ffc <- filter(fc_wi,rep(1/24,24),sides="1")

#Remove Trend
fc_nt <- fc_wi[-(1:24)] - ffc[-(1:24)]

#Remove Seasonality
fc_nt_matrix <- matrix(fc_nt,ncol=24,byrow=T)
fc_nt_hour_mean <- apply(fc_nt_matrix,2,mean)
fc_nt_ns <- fc_nt - rep(fc_nt_hour_mean,29)
```

Now we can graph the results - our CPU data with trend and seasonal cycles removed. For easier detection of anomalies, I'll add a line that indicate the average and lines placed 3 standard deviations away from average to the graph. Now the 2 hours of slightly high CPU stand-out loud and clear, even though the series still has high variance and much noise:


fc_nt_ns
4
2
0
-2
0
100
200
300
400
500
600
700
Time

5. Forecasts

"I have seen the future and it is very much like the present, only longer." --Kehlog Albran

Exponential Smoothing

Since we used moving average to estimate the trend in our data, we can use the same technique to estimate future data points.

The first predicted point will be an average of the last 24 measurements. The next predicted point will be the average of the last 23 measurements and the first predicted point. And so on. We can continue running this process to generate as many future points as we'd like, with the results getting less accurate as we continue.

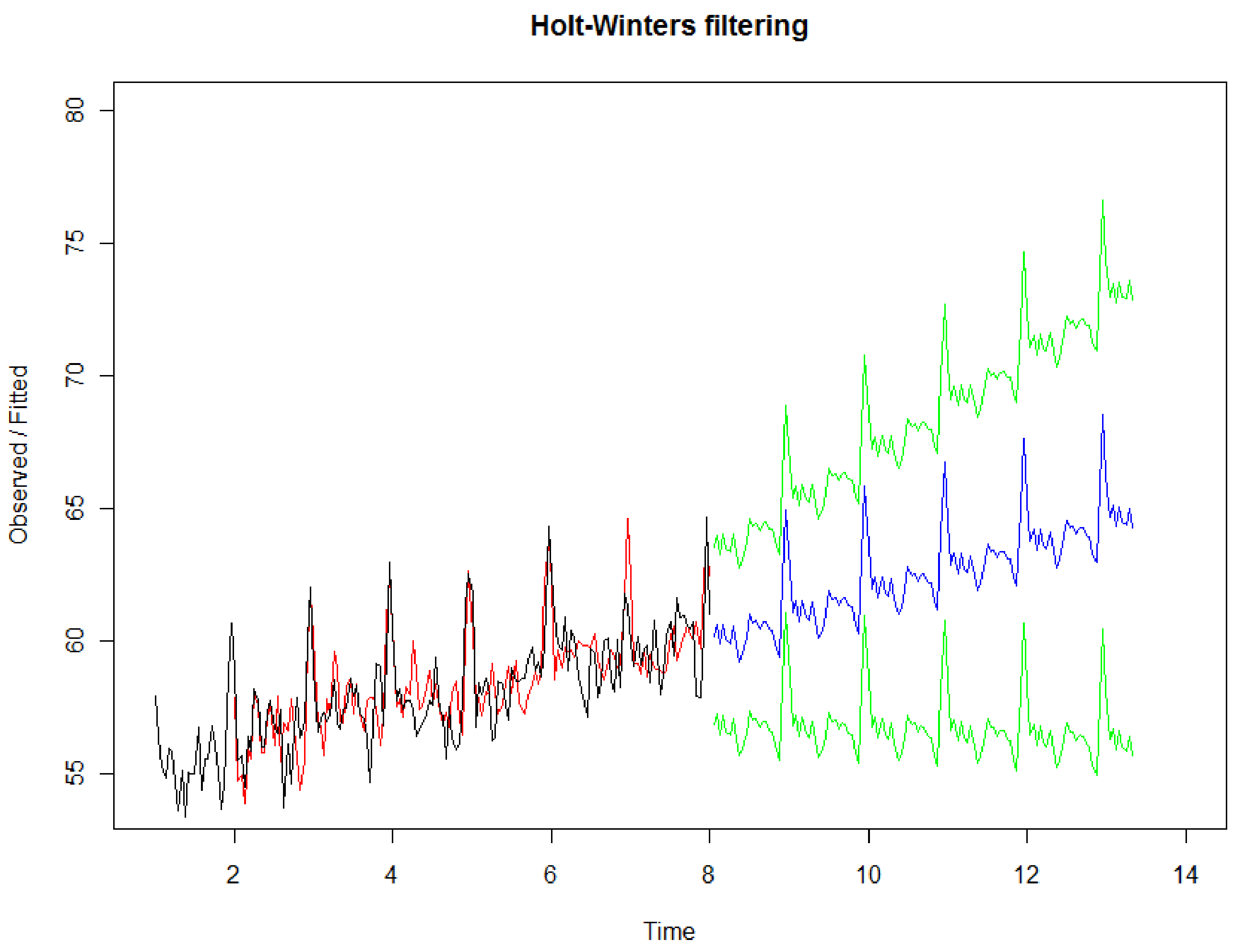
Exponential smoothing is a similar method, using weighted moving average where more recent measurements get larger weights and the older a measurement is the lower its weight, allowing old measurements to fade away from our calculations.

On top of this predicted trend we will want to add the seasonal effect. This can be done by adding the hourly averages we've calculated to the predicted data points.

R has a package for modeling our data using exponential smoothing and seasonal effects and the model can then be used to predict future data and even analyze the accuracy of our prediction.

The package is called Holt-Winters and here is an example of how it can be used to predict future CPU values of our imagined system. The function HoltWinters generates the model for the data, which can then be plotted. The function Predict generates future values based on a model. It takes as parameters the number of future data points to generate, and it can also generate upper and lower bounds of future measurements based on confidence value. In this case I generate 128 future values with 99% confidence.

```
#Plot our series and the model on it
#We use xlim and ylim parameters to leave room for future data
plot(HoltWinters(ts(fake_cpu[552:720],freq=24)),xlim=c(1,14),ylim=c(54,80))
#Add a week of future data
lines(predict(HoltWinters(ts(fake_cpu[552:720],freq=24)),n.ahead=128),col="blue")
#Upper and Lower bounds for future data
lines(predict(HoltWinters(ts(fake_cpu[552:720],freq=24)),n.ahead=128,TRUE,0.99)[,3],col="green")
lines(predict(HoltWinters(ts(fake_cpu[552:720],freq=24)),n.ahead=128,TRUE,0.99)[,2],col="green")
```



This plot shows the last week of fake CPU data (black) and how closely the model matches our data (model in red). Estimation of a week of future values is in blue. Obviously, this is only a guess - we can't know what the future will actually be. The green lines show the interval where we are 99% certain that future values will be. Note that the area is quite large - our time series has lots of variability so the future is very uncertain, but still it allows us to be quite sure that CPU utilization won't go over 80% cpu in the next week.

Auto-Correlation Model

While exponential smoothing predictions use the trend and season effects to predict future data, the AR model uses the third component, auto-correlation, to make predictions.

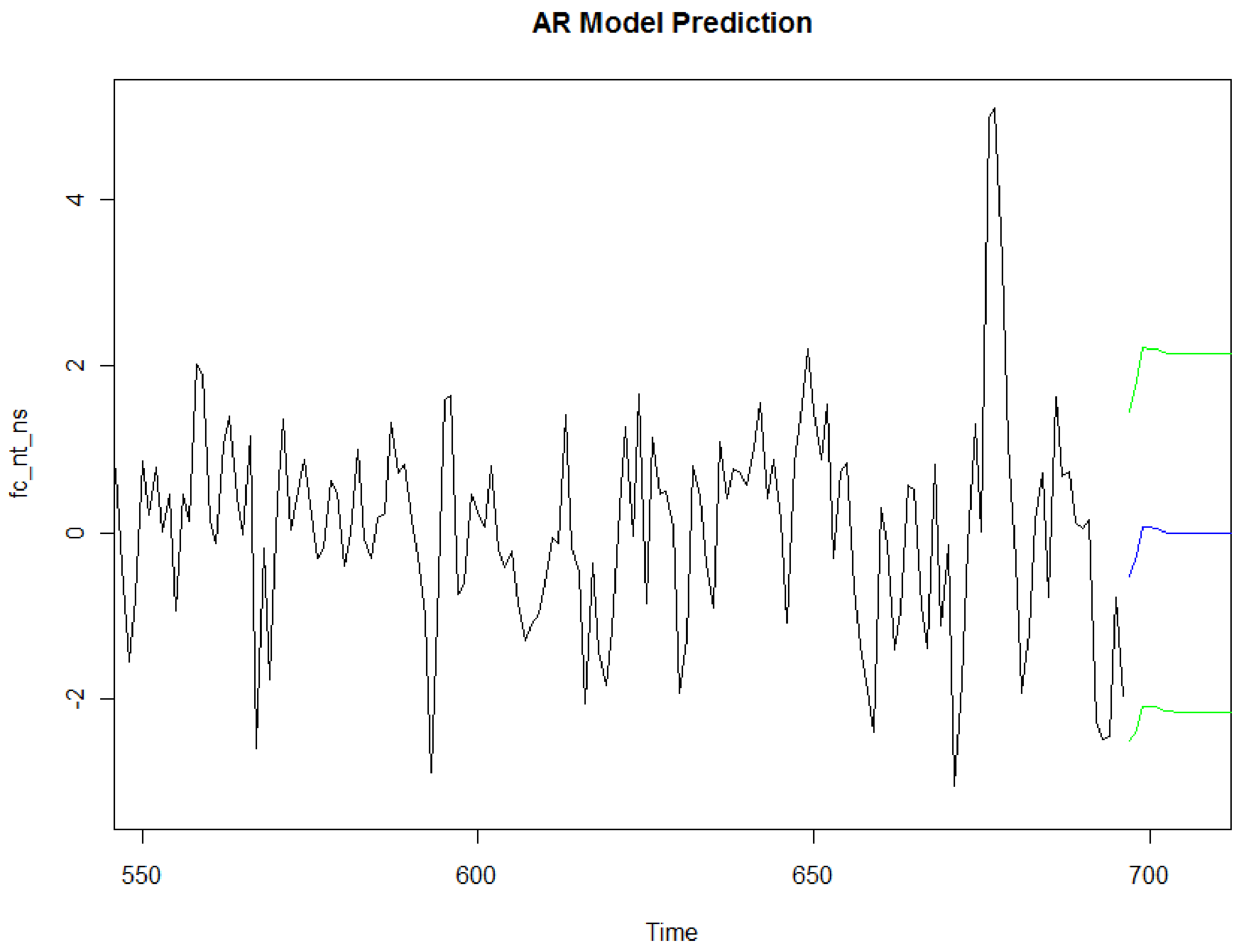
In the last part of section 3 we fitted an auto-regression model to the data. The model describes each point in our time series as a function of the previous points and we can use this same model to calculate future points.

Auto-correlation modeling happens after seasonality and trend has been removed (or never existed in the first place) and the predictions do not include the seasonality or the trend of the series.

If the series has trend or seasonality, they can be added to the predictions later. However, in such case it is probably better to just use Holt-Winters method and avoid using the auto-correlation model. As you will see soon, it gives much weaker predictions.

```
#generate the model
mod <- arima(fc_nt_ns,order=c(3,0,0))
```

```
#plot the data
plot.ts(fc_nt_ns,xlim=c(552,706),main="AR Model Prediction")

# Plot the prediction
lines((predict(mod,n.ahead=144)$pred),col="blue")

#Plot upper and lower boundaries
lines((predict(mod,n.ahead=144)$pred-2*predict(mod,n.ahead=144)$se),col="green")
lines((predict(mod,n.ahead=144)$pred+2*predict(mod,n.ahead=144)$se),col="green")
```



As you can see – other than extremely short term, the predicted data does not appear to realistically fit the data as we expect it to appear. The error interval, on the other hand looks fairly reliable. I've set the bounds at 95% confidence, and indeed it looks reasonable to expect that 95% of future measurements will not exceed the upper bound.

6. Demonstrating some of the techniques on real data sets

Redo blocks as measurement of DB activity

The number of redo blocks written to disk is tracked in table v$archived_logs and is updated on every log switch. The rate in which redo is written to disk is of interest for two reasons:

First of all, redo logs are later copied to archive and kept in backups for extended periods of time. Therefore significant variability in rate of redo log generation can impact disk space planning for log archives and backups.

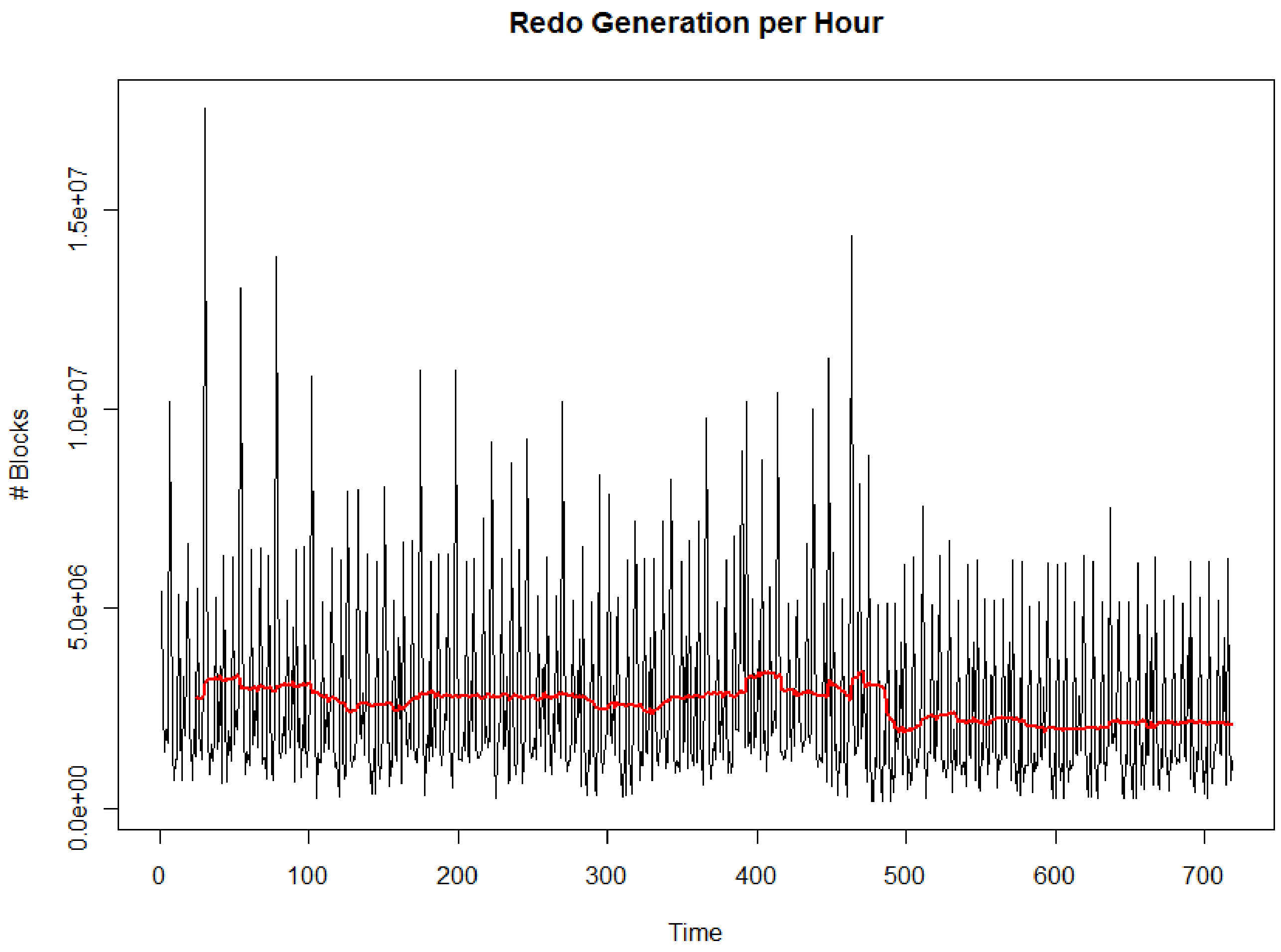
In addition, the rate of redo generation reflects the amount of DML activity on the database. In the past, detecting excessive redo generation has led to early detection and solution for problems such as infinite update loops in new code, problems in allocating new partitions and users attempting to delete 500M rows from a table in the middle of a work day.

In this example I'll analyze and model the amounts of redo generated every hour for the last 30 days on a two-node RAC system.

The data was gathered using the following query:

```
select sum(BLOCKS) BLOCKS
from v$archived_log
where FIRST_TIME >sysdate -30
group by trunc(FIRST_TIME,'HH')
order by trunc(FIRST_TIME,'HH')
```

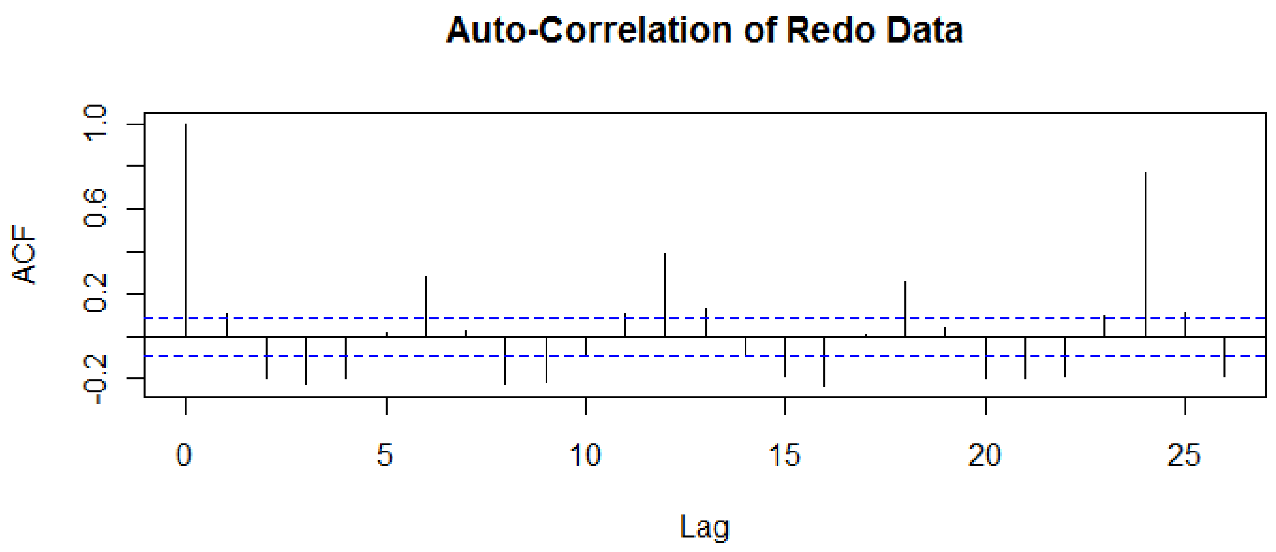
First, I'll plot the data and match a moving-average trend to it:



We can immediately see that the data is fairly noisy, has strong seasonality and no real trend. We do, however, see that there is a sudden drop in redo levels at hour 480. This is in fact a discontinuity. The series behaves differently before and after this point and analyzing this as one series does not make sense.

The point where redo levels drop is in fact Dec 20th. Our users left for the holidays and usage of the system dropped significantly. In this case, it means that the data after Dec 20th does not represent normal use of our system and should be removed entirely.

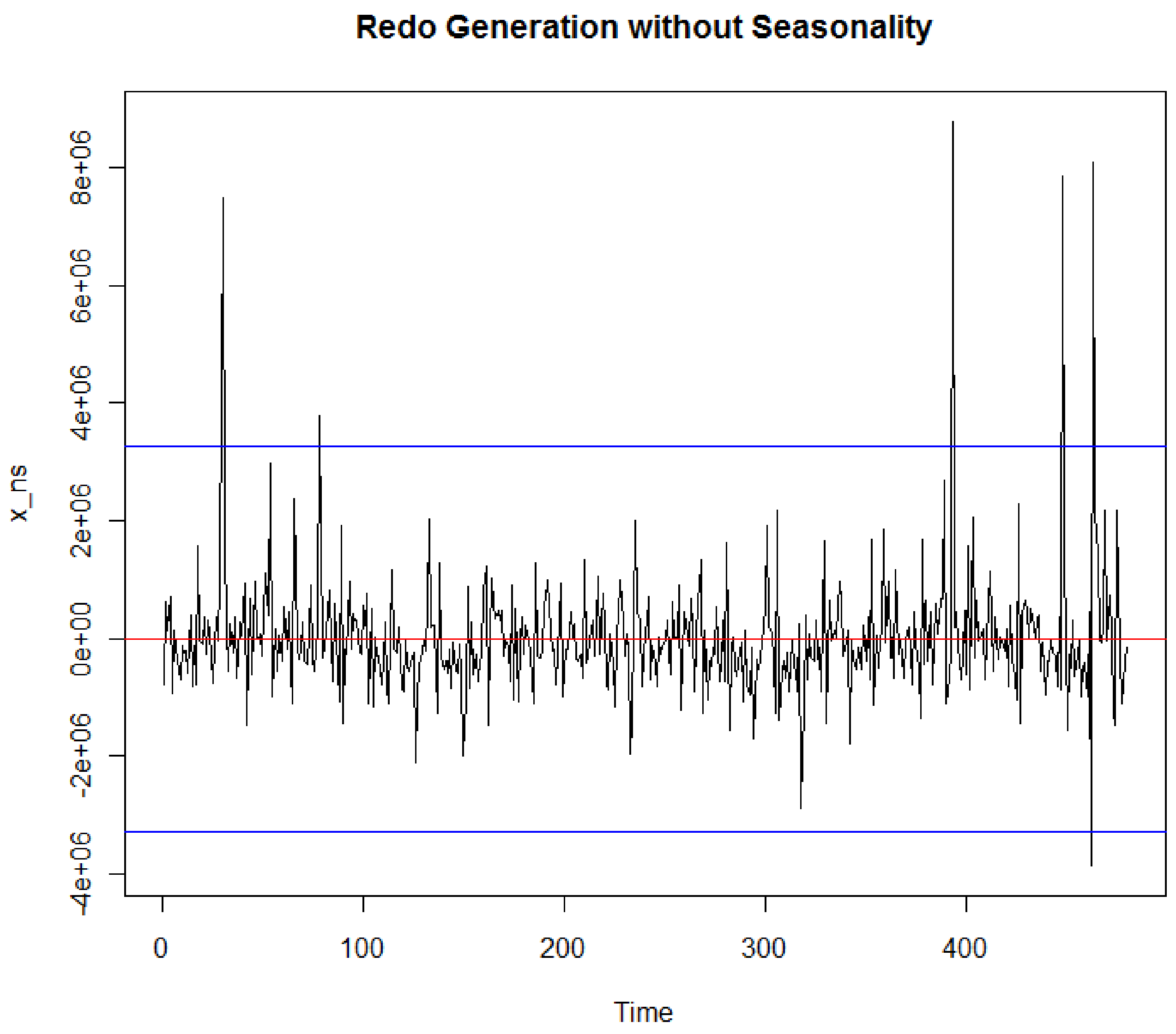
After removing the irrelevant data, we'll look for seasonality.



The Correlogram shows distinct seasonality of 6 hours. It may be of interest to investigate what process runs every 6 hours and triggers the redo generation. We have a large number of materialized views recalculating every 6 hours, it is quite possible that this is the reason.

In addition, there seems to be an even stronger auto-correlation every 24 hours. So we will use 24 hours as our season.

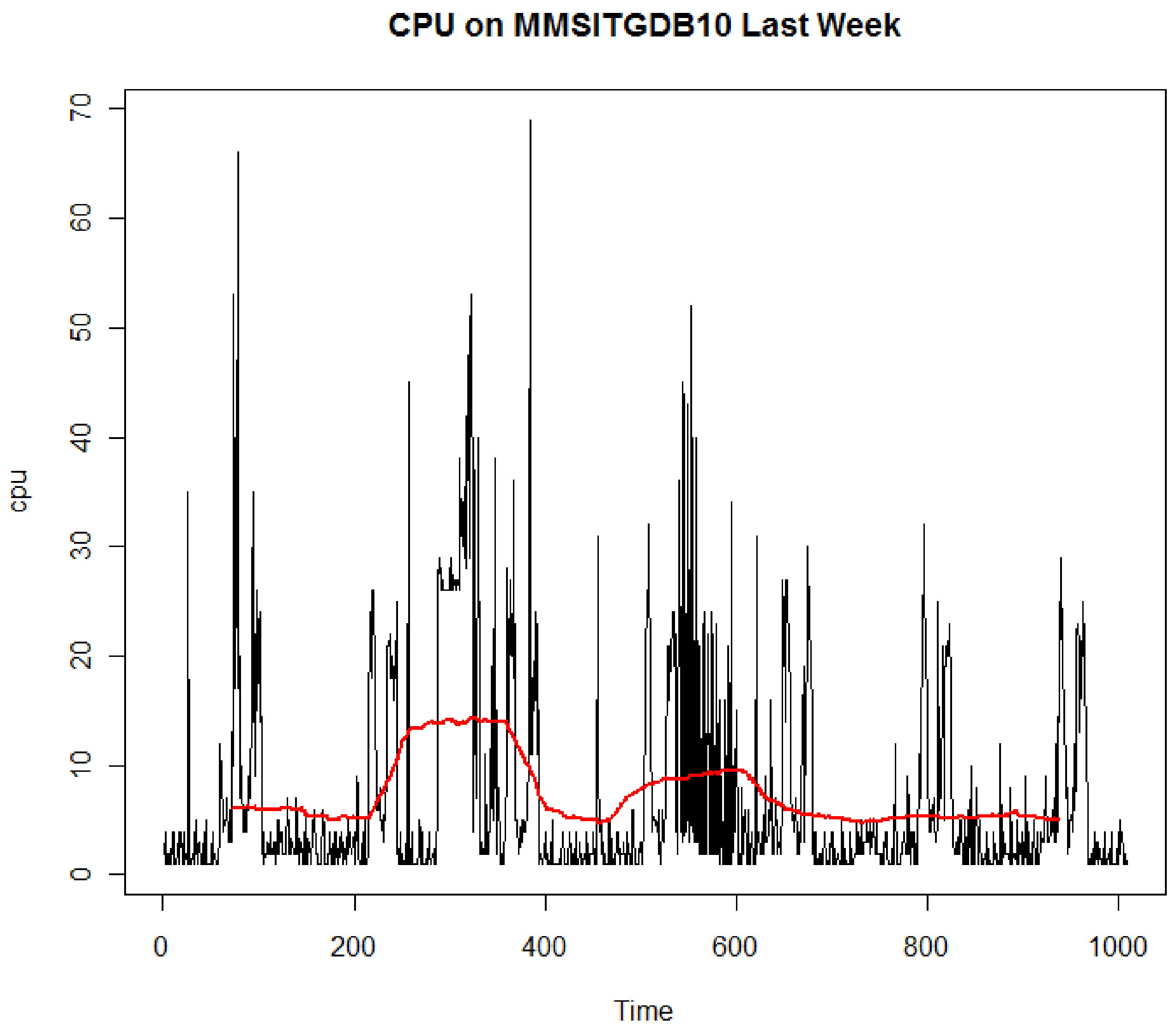
After we understood the seasonality we can remove it and look for outliers:



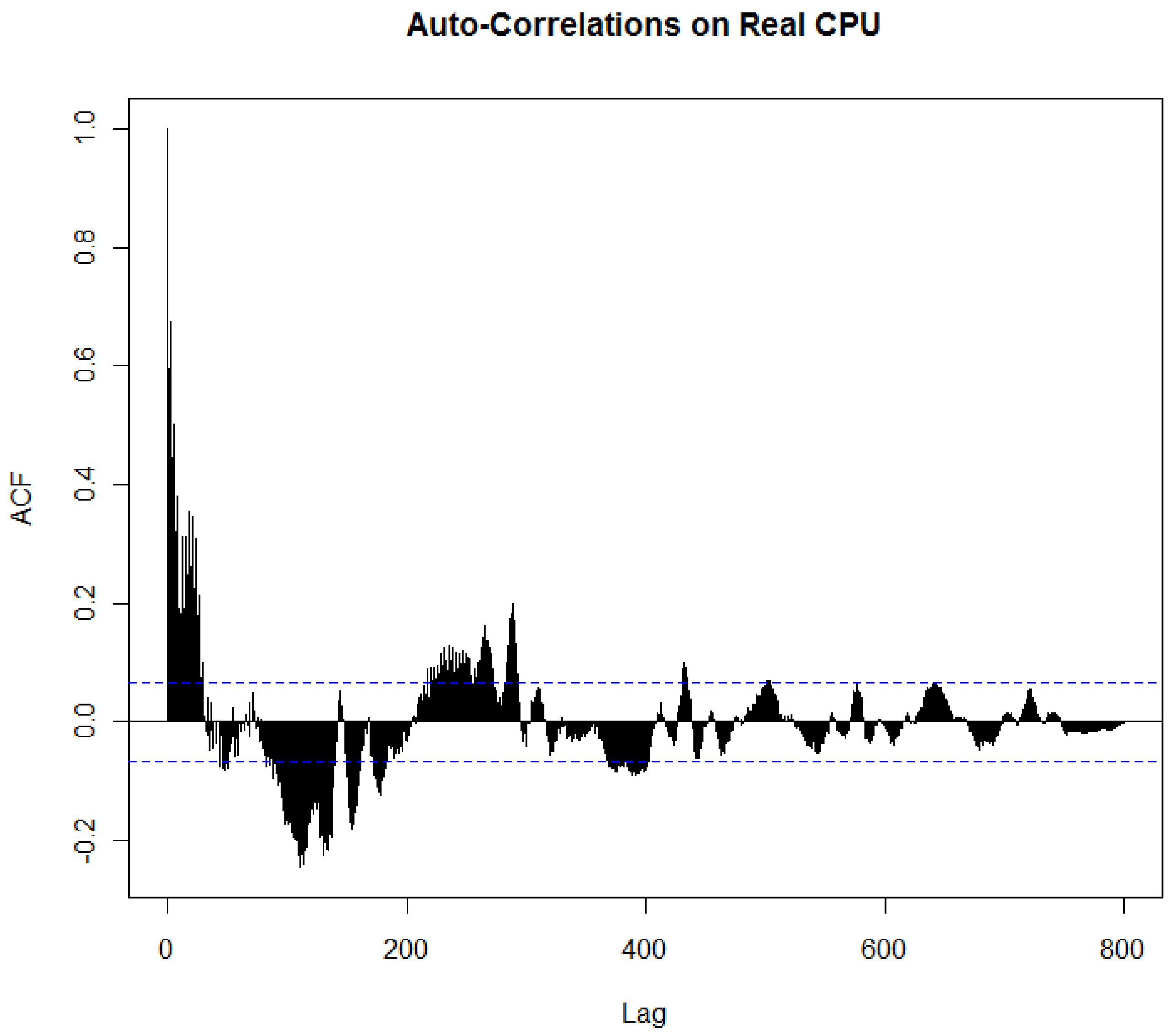
We can see clear outliers at the beginning of the month, and just before the holidays began. These indicate an unusual activity on the system, one that may be worth investigating.

CPU

We'll look at last week's CPU utilization on one of our servers, measured every 10 minutes:

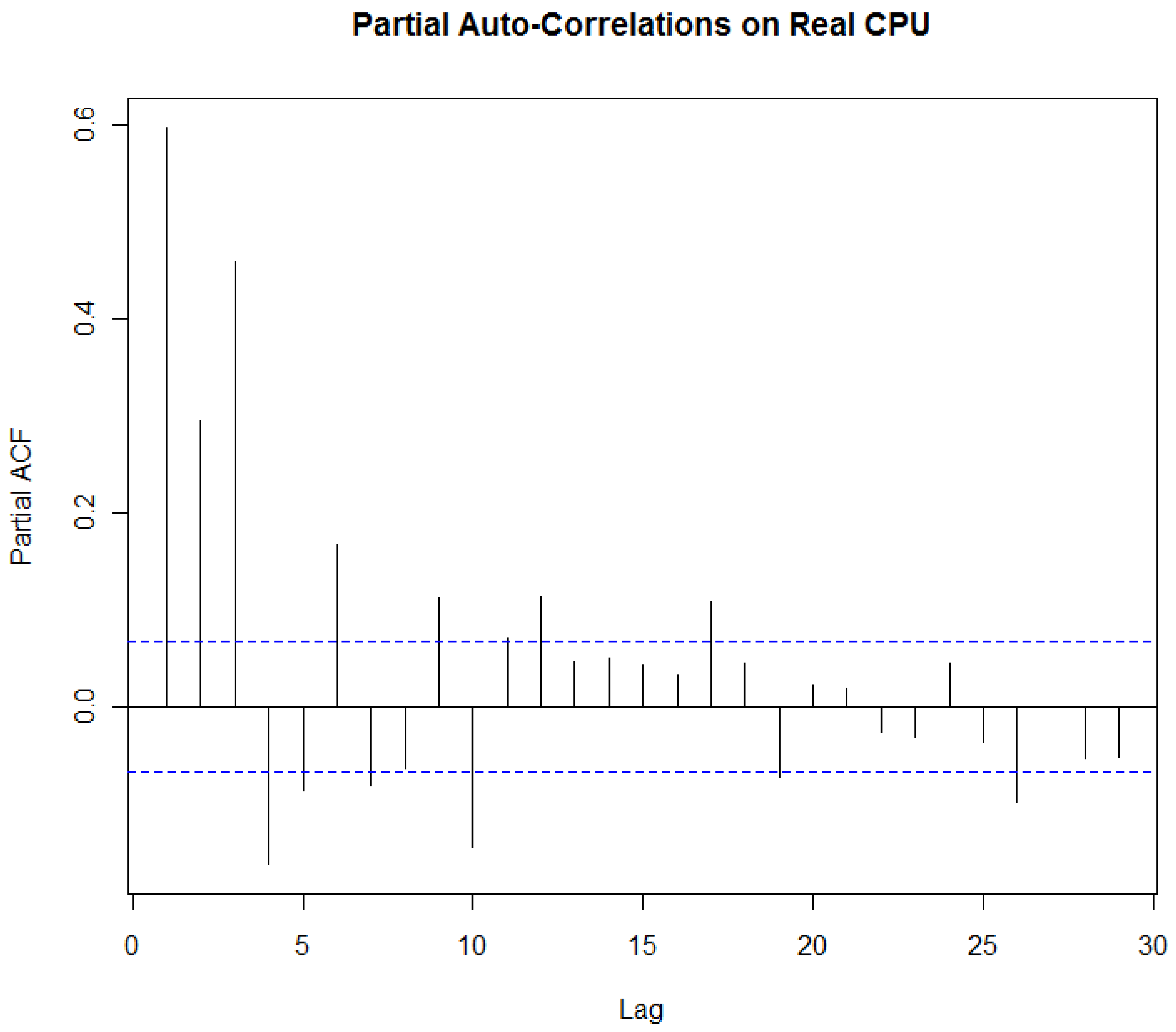


Not exactly a trend, but we do have inconsistency in average levels that may prevent us from properly analyzing the series. We'll get rid of it by subtracting the moving average from the series and look for seasonality.

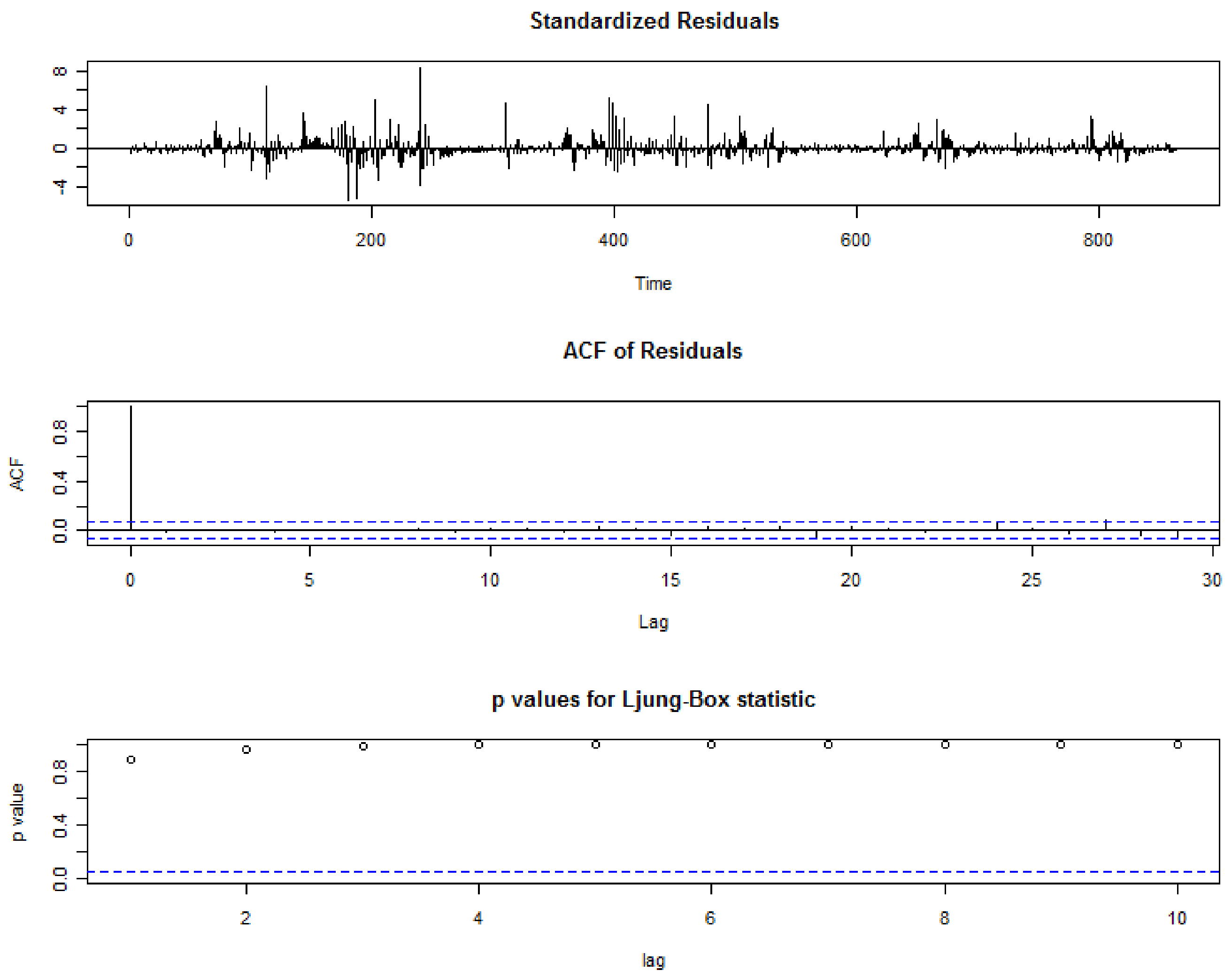


The correlogram looks different than what we encountered so far: Extremely high short term correlation and cycles that don't really look like seasonality since they are irregular. It appears that the very high short term correlation generates the cycles that only become insignificant after a very long lag.

I had to look that one up in the books [3]. The book suggested looking at another type of correlogram – one that shows "Partial Auto-Correlation" – partial auto-correlation removes the correlation that is already accounted for in lower lags from the correlation in higher lags. So the partial auto-correlation in lag 3 will not include the correlation already known in lag 2. The partial auto-correlation plot can be used to indicate what kind of auto-correlation I'm looking at and how many parameters should I expect in the model.

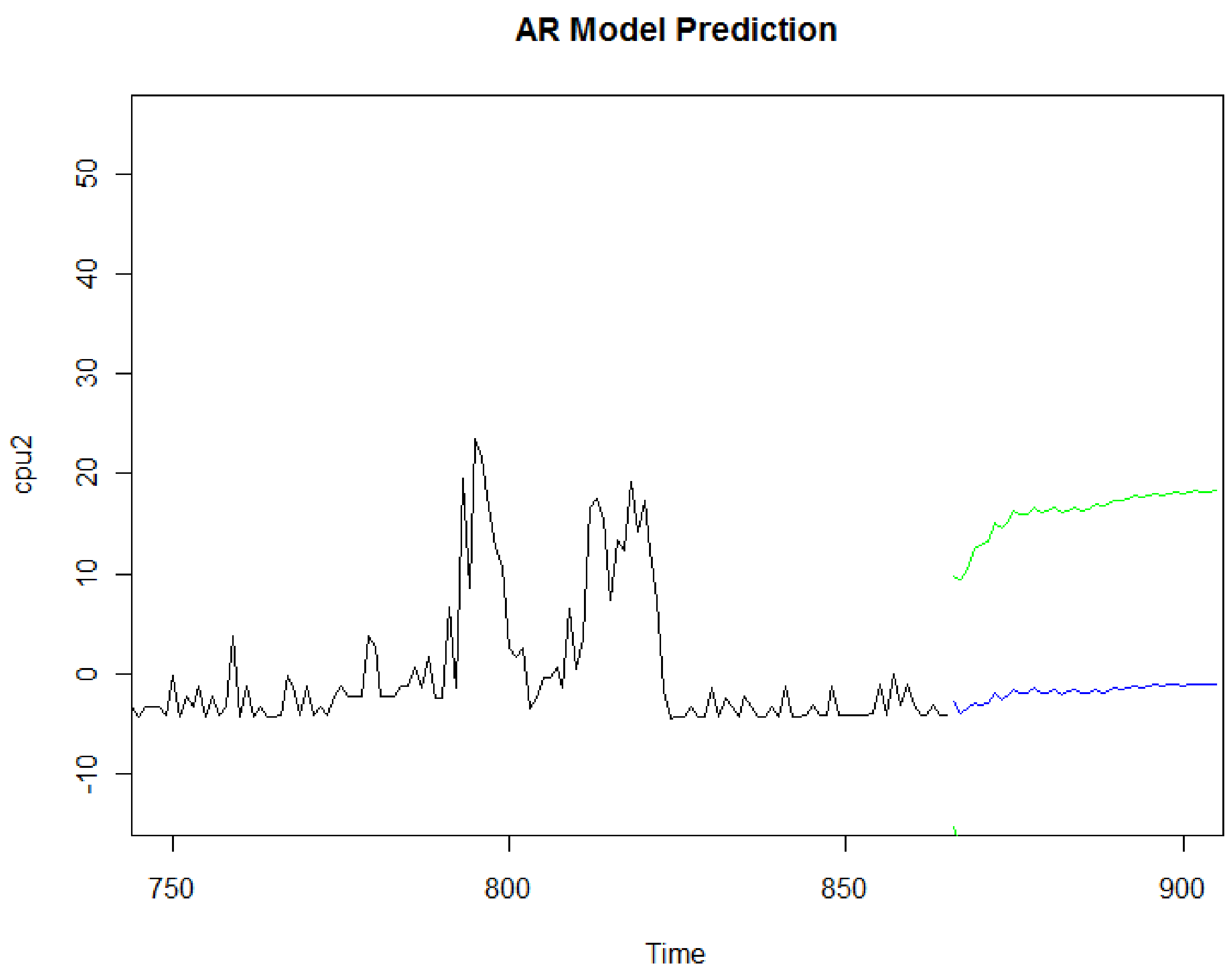


This could indicate several kinds of models, depending on when you decide that the correlation gets low enough. AR(6), AR(10) and AR(17) all seemed like good options. Testing with TSDIAG showed AR(17) to be an almost perfect fit. This is an extremely high level of auto-correlation – once the CPU goes up it takes almost 3 hours to go back to normal!



My guess is that it is a direct result of the type of web server that our system is using – the kind that when things get slow and connections time out responds by opening more and more connections, escalating small blips to full scale incidents that take a long time to recover.

Now that we have an auto-correlation model, we can try to use it for forecasts. Normally predictions based on these models are fairly weak, but with such significant correlation we may be able to get something reasonable.



The model predicts with 95% certainty that the CPU will not rise in more than 20% in the next 2 hours.

7. Conclusions and further work

The objective of this paper was to analyze performance indicators and forecast future values of same indicators using time series techniques. We looked into several techniques for modeling trend, seasonality and auto-correlations, and we used our models to make forecasts.

After applying the techniques to real-world data, it appears that if the data contains strong trend and seasonal components, using Holt-Winters method to model these components and forecast future values can give reasonably good forecasts. Certainly better forecasts than what simple methods such as linear regression can generate.

If the data contains strong auto-correlation, as we saw for CPU utilization, then it seems reasonable to fit an auto-correlation model to the data and use this model to forecast future values. Auto-correlation models, however, only generate very short term forecasts and usually with weak accuracy. In addition, using auto-correlation models is relatively difficult and require some level of guesswork and intuition. Since short term forecasts of few hours or even few days is not considered valuable in our profession, I did not find much value in using auto-correlation models to analyse database performance.

The work I did in this paper is very preliminary. The statistical field of time-series analysis contains many more methods and techniques. Of special interest, I would consider the different suggestions for different kinds smoothing of data smoothing (of which we explored only the Holt-Winters technique) that can give better predictions.

On the more practical side, I would be interested in seeing monitoring software, such as Oracle Enterprise Manager that takes into account the seasonality of the data in order to give more accurate types of alerts. I believe that this paper demonstrated that the usual "baseline" average is not adequate in many cases and can lead to either false alarms or not enough of an early warning.

8. References

1. Craig Shallahamer (2007). Forecasting Oracle Performance. Berkeley, CA: Apress.
2. http://en.wikipedia.org/wiki/Correlation
3. http://en.wikipedia.org/wiki/Box%E2%80%93Jenkins
4. Time Series Analysis with R (http://www.statoek.wiso.uni-goettingen.de/veranstaltungen/zeitreihen/sommer03/ts_r_intro.pdf)
5. Chris Chatfield (2004). The Analysis of Time Series. Boca Raton, FL: Chapman & Hall/CRC.
6. http://www.r-project.org/