# Ch1 Correlation and Autocorrelation

**A "Thin" Application of Time Series**

[**Google Trends**](https://trends.google.com/trends/) allows users to see how often a term is searched for. We downloaded a file from Google Trends containing the frequency over time for the search word "diet", which is pre-loaded in a DataFrame called diet. A first step when analyzing a time series is to visualize the data with a plot. You should be able to clearly see a gradual decrease in searches for "diet" throughout the calendar year, hitting a low around the December holidays, followed by a spike in searches around the new year as people make New Year's resolutions to lose weight.

Like many time series datasets you will be working with, the index of dates are strings and should be converted to a datetime index before plotting.

Code:

# Import pandas and plotting modules

import pandas as pd

import matplotlib.pyplot as plt

# Convert the date index to datetime

diet.index = pd.to\_datetime(diet.index)

# Plot 2012 data using slicing

diet['2012'].plot()

plt.show()

# Plot the entire time series diet and show gridlines

diet.plot(grid=True)

plt.show()

# Merging Time Series With Different Dates

Stock and bond markets in the U.S. are closed on different days. For example, although the bond market is closed on Columbus Day (around Oct 12) and Veterans Day (around Nov 11), the stock market is open on those days. One way to see the dates that the stock market is open and the bond market is closed is to convert both indexes of dates into sets and take the difference in sets.

The pandas .join() method is a convenient tool to merge the stock and bond DataFrames on dates when both markets are open.

Stock prices and 10-year US Government bond yields, which were downloaded from [**FRED**](https://fred.stlouisfed.org/), are pre-loaded in DataFrames stocksand bonds.

Code:

# Import pandas

import pandas as pd

# Convert the stock index and bond index into sets

set\_stock\_dates = set(stocks.index)

set\_bond\_dates = set(bonds.index)

# Take the difference between the sets and print

print(set\_stock\_dates - set\_bond\_dates)

# Merge stocks and bonds DataFrames using join()

# To get the intersection of dates, use the argument how='inner'.

stocks\_and\_bonds = stocks.join(bonds,how='inner')

# Correlation of Stocks and Bonds

Investors are often interested in the correlation between the returns of two different assets for asset allocation and hedging purposes. In this exercise, you'll try to answer the question of whether stocks are positively or negatively correlated with bonds. Scatter plots are also useful for visualizing the correlation between the two variables.

Keep in mind that you should compute the correlations on the percentage changes rather than the levels.

Stock prices and 10-year bond yields are combined in a DataFrame called stocks\_and\_bonds under columns SP500 and US10Y

The pandas and plotting modules have already been imported for you. For the remainder of the course, pandas is imported as pdand matplotlib.pyplot is imported as plt.

Code:

# Compute percent change using pct\_change()

returns = stocks\_and\_bonds.pct\_change()

# Compute correlation using corr()

# using the .corr() method for Series which has the syntax series1.corr(series2).

correlation = returns['SP500'].corr(returns['US10Y'])

print("Correlation of stocks and interest rates: ", correlation)

# Make scatter plot

plt.scatter(returns['SP500'],returns['US10Y'])

plt.show()

# Flying Saucers Aren't Correlated to Flying Markets

Two trending series may show a strong correlation even if they are completely unrelated. This is referred to as "spurious correlation". That's why when you look at the correlation of say, two stocks, you should look at the correlation of their returns and not their levels.

To illustrate this point, calculate the correlation between the levels of the stock market and the annual sightings of UFOs. Both of those time series have trended up over the last several decades, and the correlation of their levels is very high. Then calculate the correlation of their percent changes. This will be close to zero, since there is no relationship between those two series.

The DataFrame levels contains the levels of DJI and UFO. UFO data was downloaded from [**www.nuforc.org**](http://www.nuforc.org/).

Code:

# Compute correlation of levels

correlation1 = levels['DJI'].corr(levels['UFO'])

print("Correlation of levels: ", correlation1)

# Compute correlation of percent changes

changes = levels.pct\_change()

correlation2 = changes['DJI'].corr(changes['UFO'])

print("Correlation of changes: ", correlation2)

# Looking at a Regression's R-Squared

R-squared measures how closely the data fit the regression line, so the R-squared in a simple regression is related to the correlation between the two variables. In particular, the magnitude of the correlation is the square root of the R-squared and the sign of the correlation is the sign of the regression coefficient.

In this exercise, you will start using the statistical package statsmodels, which performs much of the statistical modeling and testing that is found in R and software packages like SAS and MATLAB.

You will take two series, x and y, compute their correlation, and then regress y on x using the function OLS() in the statsmodels.api library. Most linear regressions contain a constant term which is the intercept (the αα in the regression yt=α+βxt+ϵtyt=α+βxt+ϵt). To include a constant using the function OLS(), you need to add a column of 1's to the right hand side of the regression.

The module statsmodels.api has been imported for you as sm.

Code:

# Import the statsmodels module

import statsmodels.api as sm

# Compute correlation of x and y

correlation = x.corr(y)

print("The correlation between x and y is %4.2f" %(correlation))

# Convert the Series x to a DataFrame and name the column x

x = pd.DataFrame(x, columns=['x'])

# Add a constant to the DataFrame x

x = sm.add\_constant(x)

# Fit the regression of y on x

result = sm.OLS(y,x).fit()

# Print out the results and look at the relationship between R-squared and the correlation above

print(result.summary())

# A Popular Strategy Using Autocorrelation

One puzzling anomaly with stocks is that investors tend to overreact to news. Following large jumps, either up or down, stock prices tend to reverse. This is described as mean reversion in stock prices: prices tend to bounce back, or revert, towards previous levels after large moves, which are observed over time horizons of about a week. A more mathematical way to describe mean reversion is to say that stock returns are negatively autocorrelated.

This simple idea is actually the basis for a popular hedge fund strategy. If you're curious to learn more about this hedge fund strategy (although it's not necessary reading for anything else later in the course), see [**here**](https://www.quantopian.com/posts/enhancing-short-term-mean-reversion-strategies-1).

You'll look at the autocorrelation of weekly returns of MSFT stock from 2012 to 2017. You'll start with a DataFrame MSFT of daily prices. You should use the .resample() method to get weekly prices and then compute returns from prices. Use the pandas method .autocorr() to get the autocorrelation and show that the autocorrelation is negative. Note that the .autocorr() method only works on Series, not DataFrames (even DataFrames with one column), so you will have to select the column in the DataFrame.

Code:

# Convert the daily data to weekly data

MSFT = MSFT.resample(rule='W',how='last')

# Compute the percentage change of prices

returns = MSFT.pct\_change()

# Compute and print the autocorrelation of returns

autocorrelation = returns['Adj Close'].autocorr()

print("The autocorrelation of weekly returns is %4.2f" %(autocorrelation))

# Are Interest Rates Autocorrelated?

When you look at daily changes in interest rates, the autocorrelation is close to zero. However, if you resample the data and look at annual changes, the autocorrelation is negative. This implies that while short term changes in interest rates may be uncorrelated, long term changes in interest rates are negatively autocorrelated. A daily move up or down in interest rates is unlikely to tell you anything about interest rates tomorrow, but a move in interest rates over a year can tell you something about where interest rates are going over the next year. And this makes some economic sense: over long horizons, when interest rates go up, the economy tends to slow down, which consequently causes interest rates to fall, and vice versa.

The DataFrame daily\_data contains daily data of 10-year interest rates from 1962 to 2017.

Code:

# Compute the daily change in interest rates

daily\_data['change\_rates'] = daily\_data.diff()

# Compute and print the autocorrelation of daily changes

autocorrelation\_daily = daily\_data['change\_rates'].autocorr()

print("The autocorrelation of daily interest rate changes is %4.2f" %(autocorrelation\_daily))

# Convert the daily data to annual data

annual\_data = daily\_data['US10Y'].resample(rule='A',how='last')

# Repeat above for annual data

annual\_data['diff\_rates'] = annual\_data.diff()

autocorrelation\_annual = annual\_data['diff\_rates'].autocorr()

print("The autocorrelation of annual interest rate changes is %4.2f" %(autocorrelation\_annual))

# ch2 Some Simple Time Series

# Taxing Exercise: Compute the ACF

In the last chapter, you computed autocorrelations with one lag. Often we are interested in seeing the autocorrelation over many lags. The quarterly earnings for H&R Block (ticker symbol HRB) is plotted on the right, and you can see the extreme cyclicality of its earnings. A vast majority of its earnings occurs in the quarter that taxes are due.

You will compute the array of autocorrelations for the H&R Block quarterly earnings that is pre-loaded in the DataFrame HRB. Then, plot the autocorrelation function using the plot\_acf module. This plot shows what the autocorrelation function looks like for cyclical earnings data. The ACF at lag=0 is always one, of course. In the next exercise, you will learn about the confidence interval for the ACF, but for now, suppress the confidence interval by setting alpha=1.

Code:

# Import the acf module and the plot\_acf module from statsmodels

from statsmodels.tsa.stattools import acf

from statsmodels.graphics.tsaplots import plot\_acf

# Compute the acf array of HRB

acf\_array = acf(HRB)

print(acf\_array)

# Plot the acf function

plot\_acf(HRB,alpha=1)

plt.show()

# Are We Confident This Stock is Mean Reverting?

In the last chapter, you saw that the autocorrelation of MSFT's weekly stock returns was -0.16. That autocorrelation seems large, but is it statistically significant? In other words, can you say that there is less than a 5% chance that we would observe such a large negative autocorrelation if the true autocorrelation were really zero? And are there any autocorrelations at other lags that are significantly different from zero?

Even if the true autocorrelations were zero at all lags, in a finite sample of returns you won't see the estimate of the autocorrelations exactly zero. In fact, the standard deviation of the sample autocorrelation is 1/N−−√1/N where NN is the number of observations, so if N=100N=100, for example, the standard deviation of the ACF is 0.1, and since 95% of a normal curve is between +1.96 and -1.96 standard deviations from the mean, the 95% confidence interval is ±1.96/N−−√±1.96/N. This approximation only holds when the true autocorrelations are allzero.

You will compute the actual and approximate confidence interval for the ACF, and compare it to the lag-one autocorrelation of -0.16 from the last chapter. The weekly returns of Microsoft is pre-loaded in a DataFrame called returns.

Code:

# Import the plot\_acf module from statsmodels and sqrt from math

from statsmodels.graphics.tsaplots import plot\_acf

from math import sqrt

# Compute and print the autocorrelation of MSFT weekly returns

autocorrelation = returns['Adj Close'].autocorr()

print("The autocorrelation of weekly MSFT returns is %4.2f" %(autocorrelation))

# Find the number of observations by taking the length of the returns DataFrame

nobs = len(returns)

# Compute the approximate confidence interval

conf = 1.96/sqrt(nobs)

print("The approximate confidence interval is +/- %4.2f" %(conf))

# Plot the autocorrelation function with 95% confidence intervals and 20 lags using plot\_acf

plot\_acf(returns, alpha=0.05, lags=20)

plt.show()

# Notice that the autocorrelation with lag 1 is significantly negative, but none of the other lags are significantly different from zero

# Can't Forecast White Noise

A white noise time series is simply a sequence of uncorrelated random variables that are identically distributed. Stock returns are often modelled as white noise. Unfortunately, for white noise, we cannot forecast future observations based on the past - autocorrelations at all lags are zero.

You will generate a white noise series and plot the autocorrelation function to show that it is zero for all lags. You can use np.random.normal() to generate random returns. For a Gaussian white noise process, the mean and standard deviation describe the entire process.

Plot this white noise series to see what it looks like, and then plot the autocorrelation function.

Code:

# Import the plot\_acf module from statsmodels

from statsmodels.graphics.tsaplots import plot\_acf

# Simulate white noise returns

returns = np.random.normal(loc=0.02, scale=0.05, size=1000)

# Print out the mean and standard deviation of returns

mean = np.mean(returns)

std = np.std(returns)

print("The mean is %5.3f and the standard deviation is %5.3f" %(mean,std))

# Plot returns series

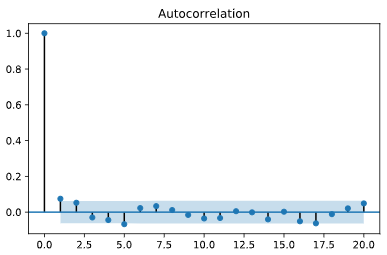
plt.plot(returns)

plt.show()

# Plot autocorrelation function of white noise returns

plot\_acf(returns, lags=20)

plt.show()



# Notice that for a white noise time series, all the autocorrelations are close to zero, so the past will not help you forecast the future.

# Generate a Random Walk

Whereas stock returns are often modelled as white noise, stock pricesclosely follow a random walk. In other words, today's price is yesterday's price plus some random noise.

You will simulate the price of a stock over time that has a starting price of 100 and every day goes up or down by a random amount. Then, plot the simulated stock price. If you hit the "Run Code" code button multiple times, you'll see several realizations.

Code:

# Generate 500 random steps with mean=0 and standard deviation=1

steps = np.random.normal(loc=0, scale=1, size=500)

# Set first element to 0 so that the first price will be the starting stock price

steps[0]=0

# Simulate stock prices, P with a starting price of 100, (Cumulate the random steps using the numpy .cumsum() method)

P = 100 + np.cumsum(steps)

# Plot the simulated stock prices

plt.plot(P)

plt.title("Simulated Random Walk")

plt.show()

**Get the Drift**

In the last exercise, you simulated stock prices that follow a random walk. You will extend this in two ways in this exercise.

* You will look at a random walk with a drift. Many time series, like stock prices, are random walks but tend to drift up over time.
* In the last exercise, the noise in the random walk was additive: random, normal changes in price were added to the last price. However, when adding noise, you could theoretically get negative prices. Now you will make the noise multiplicative: you will add one to the random, normal changes to get a total return, and multiply that by the last price.

Code:

# Generate 500 random steps

steps = np.random.normal(loc=0.001, scale=0.01, size=500) + 1

# Set first element to 1

steps[0]=1

# Simulate the stock price, P, by taking the cumulative product (Cumulate the product of the steps using the numpy .cumprod() method.)

P = 100 \* np.cumprod(steps)

# Plot the simulated stock prices

plt.plot(P)

plt.title("Simulated Random Walk with Drift")

plt.show()

# Are Stock Prices a Random Walk?

Most stock prices follow a random walk (perhaps with a drift). You will look at a time series of Amazon stock prices, pre-loaded in the DataFrame AMZN, and run the 'Augmented Dickey-Fuller Test' from the statsmodels library to show that it does indeed follow a random walk.

With the ADF test, the "null hypothesis" (the hypothesis that we either reject or fail to reject) is that the series follows a random walk. Therefore, a low p-value (say less than 5%) means we can reject the null hypothesis that the series is a random walk.

Code:

# Import the adfuller module from statsmodels

from statsmodels.tsa.stattools import adfuller

# Run the ADF test on the price series and print out the results

results = adfuller(AMZN['Adj Close'])

print(results)

(Print out the entire output, which includes the test statistic, the p-values, and the critical values for tests with 1%, 10%, and 5% levels.)

# Just print out the p-value

# results[0] is the test statistic, and results[1] is the p-value

print('The p-value of the test on prices is: ' + str(results[1]))

# we cannot reject the hypothesis that Amazon prices follow a random walk

# How About Stock Returns?

In the last exercise, you showed that Amazon stock prices, contained in the DataFrame AMZN follow a random walk. In this exercise. you will do the same thing for Amazon returns (percent change in prices) and show that the returns do not follow a random walk.

Code:

# Import the adfuller module from statsmodels

from statsmodels.tsa.stattools import adfuller

# Create a DataFrame of AMZN returns

AMZN\_ret = AMZN.pct\_change()

# Eliminate the NaN in the first row of returns

AMZN\_ret = AMZN\_ret.dropna()

# Run the ADF test on the return series and print out the p-value

results = adfuller(AMZN\_ret['Adj Close'])

print('The p-value of the test on returns is: ' + str(results[1]))

# Seasonal Adjustment During Tax Season

Many time series exhibit strong seasonal behavior. The procedure for removing the seasonal component of a time series is called seasonal adjustment. For example, most economic data published by the government is seasonally adjusted.

You saw earlier that by taking first differences of a random walk, you get a stationary white noise process. For seasonal adjustments, instead of taking first differences, you will take differences with a lag corresponding to the periodicity.

Look again at the ACF of H&R Block's quarterly earnings, pre-loaded in the DataFrame HRB, and there is a clear seasonal component. The autocorrelation is high for lags 4,8,12,16,..., because of the spike in earnings every four quarters during tax season. Apply a seasonal adjustment by taking the fourth difference (four represents the periodicity of the series). Then compute the autocorrelation of the transformed series.

Code:

# Import the plot\_acf module from statsmodels

from statsmodels.graphics.tsaplots import plot\_acf

# Seasonally adjust quarterly earnings

HRBsa = HRB.diff(4)

# Print the first 10 rows of the seasonally adjusted series

print(HRBsa.head(10))

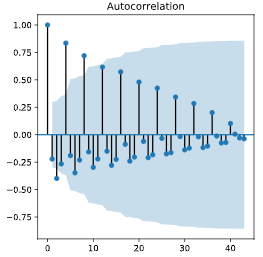
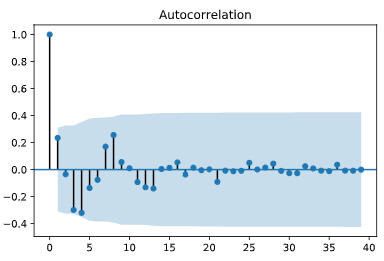
# Drop the NaN data in the first three three rows

HRBsa = HRBsa.dropna()

# Plot the autocorrelation function of the seasonally adjusted series

plot\_acf(HRBsa)

plt.show()

-🡪 

# Ch3: Autoregressive (AR) Models

# Simulate AR(1) Time Series

You will simulate and plot a few AR(1) time series, each with a different parameter, ϕϕ, using the arima\_process module in statsmodels. In this exercise, you will look at an AR(1) model with a large positive ϕϕ and a large negative ϕϕ, but feel free to play around with your own parameters.

There are a few conventions when using the arima\_processmodule that require some explanation. First, these routines were made very generally to handle both AR and MA models. We will cover MA models next, so for now, just ignore the MA part. Second, when inputting the coefficients, you must include the zero-lag coefficient of 1, and the sign of the other coefficients is opposite what we have been using (to be consistent with the time series literature in signal processing). For example, for an AR(1) process with ϕ=0.9ϕ=0.9, the array representing the AR parameters would be ar = np.array([1, -0.9])

Code:

# import the module for simulating data

from statsmodels.tsa.arima\_process import ArmaProcess

# Plot 1: AR parameter = +0.9

# Let ar1 represent an array of the AR parameters [1, −ϕ] as explained above. For now, the MA parameter array, ma1, will contain just the lag-zero coefficient of one.

plt.subplot(2,1,1)

ar1 = np.array([1, -0.9])

ma1 = np.array([1])

AR\_object1 = ArmaProcess(ar1, ma1)

simulated\_data\_1 = AR\_object1.generate\_sample(nsample=1000)

plt.plot(simulated\_data\_1)

# Plot 2: AR parameter = -0.9

plt.subplot(2,1,2)

ar2 = np.array([1, 0.9])

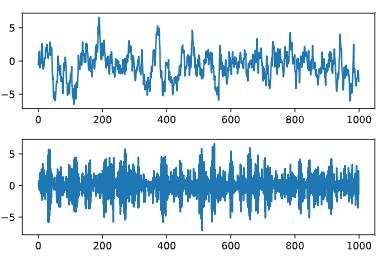
ma2 = np.array([1])

AR\_object2 = ArmaProcess(ar2, ma2)

simulated\_data\_2 = AR\_object2.generate\_sample(nsample=1000)

plt.plot(simulated\_data\_2)

plt.show()



# Compare the ACF for Several AR Time Series

The autocorrelation function decays exponentially for an AR time series at a rate of the AR parameter. For example, if the AR parameter, ϕ=+0.9ϕ=+0.9, the first-lag autocorrelation will be 0.9, the second-lag will be (0.9)2=0.81(0.9)2=0.81, the third-lag will be (0.9)3=0.729(0.9)3=0.729, etc. A smaller AR parameter will have a steeper decay, and for a negative AR parameter, say -0.9, the decay will flip signs, so the first-lag autocorrelation will be -0.9, the second-lag will be (−0.9)2=0.81(−0.9)2=0.81, the third-lag will be (−0.9)3=−0.729(−0.9)3=−0.729, etc.

The object simulated\_data\_1 is the simulated time series with an AR parameter of +0.9, simulated\_data\_2 is for an AR parameter of -0.9, and simulated\_data\_3 is for an AR parameter of 0.3

Code:

# Import the plot\_acf module from statsmodels

from statsmodels.graphics.tsaplots import plot\_acf

# Plot 1: AR parameter = +0.9

plot\_acf(simulated\_data\_1, alpha=1, lags=20)

plt.show()

# Plot 2: AR parameter = -0.9

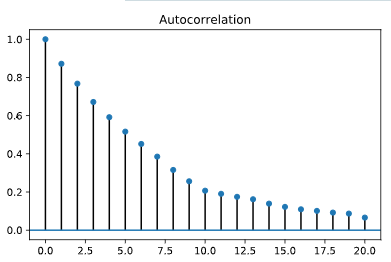
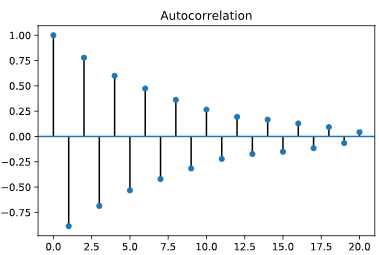
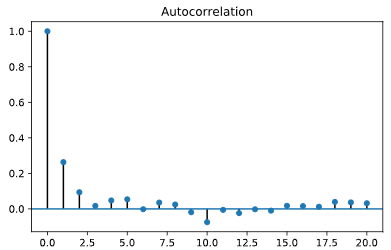
plot\_acf(simulated\_data\_2, alpha=1, lags=20)

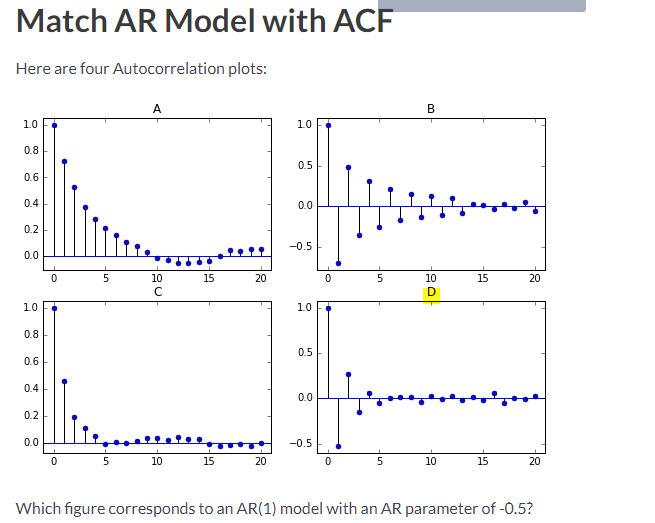
plt.show()

# Plot 3: AR parameter = +0.3

plot\_acf(simulated\_data\_3, alpha=1, lags=20)

plt.show()



# Estimating an AR Model

You will estimate the AR(1) parameter, ϕϕ, of one of the simulated series that you generated in the earlier exercise. Since the parameters are known for a simulated series, it is a good way to understand the estimation routines before applying it to real data.

For simulated\_data\_1 with a true ϕϕ of 0.9, you will print out the estimate of ϕϕ. In addition, you will also print out the entire output that is produced when you fit a time series, so you can get an idea of what other tests and summary statistics are available in statsmodels.

Code:

# Import the ARMA module from statsmodels

from statsmodels.tsa.arima\_model import ARMA

# Fit an AR(1) model to the first simulated data

mod = ARMA(simulated\_data\_1, order=(1,0))

res = mod.fit()

# Print out summary information on the fit

print(res.summary())

# Print out the estimate for the constant and for phi

print("When the true phi=0.9, the estimate of phi (and the constant) are:")

print(res.params)

# Forecasting with an AR Model

In addition to estimating the parameters of a model that you did in the last exercise, you can also do forecasting, both in-sample and out-of-sample using statsmodels. The in-sample is a forecast of the next data point using the data up to that point, and the out-of-sample forecasts any number of data points in the future. These forecasts can be made using either the predict() method if you want the forecasts in the form of a series of data, or using the plot\_predict() method if you want a plot of the forecasted data. You supply the starting point for forecasting and the ending point, which can be any number of data points after the data set ends.

For the simulated series simulated\_data\_1 with ϕ=0.9ϕ=0.9, you will plot in-sample and out-of-sample forecasts.

Code

# Import the ARMA module from statsmodels

from statsmodels.tsa.arima\_model import ARMA

# Forecast the first AR(1) model

mod = ARMA(simulated\_data\_1, order=(1,0))

res = mod.fit()

res.plot\_predict(start=990, end=1010)

# Start the forecast 10 data points before the end of the 1000 point series at 990, and end the forecast 10 data points after the end of the series at point 1010

plt.show()

# Notice how, when phi is high like here, the forecast gradually moves to the long term mean of zero, but if phi were low, it would move much quicker to the long term mean.

# Let's Forecast Interest Rates

You will now use the forecasting techniques you learned in the last exercise and apply it to real data rather than simulated data. You will revisit a dataset from the first chapter: the annual data of 10-year interest rates going back 56 years, which is in a Series called interest\_rate\_data. Being able to forecast interest rates is of enormous importance, not only for bond investors but also for individuals like new homeowners who must decide between fixed and floating rate mortgages.

You saw in the first chapter that there is some mean reversion in interest rates over long horizons. In other words, when interest rates are high, they tend to drop and when they are low, they tend to rise over time. Currently they are below long-term rates, so they are expected to rise, but an AR model attempts to quantify how much they are expected to rise.

Code

# Import the ARMA module from statsmodels

from statsmodels.tsa.arima\_model import ARMA

# Forecast interest rates using an AR(1) model

mod = ARMA(interest\_rate\_data, order=(1,0))

res = mod.fit()

# Plot the original series and the forecasted series

# Pass the arguments start=0 to start the in-sample forecast from the beginning, and choose end to be '2022' to forecast several years in the future.

res.plot\_predict(start=0,end='2022')

plt.legend(fontsize=8)

plt.show()

# Compare AR Model with Random Walk

Sometimes it is difficult to distinguish between a time series that is slightly mean reverting and a time series that does not mean revert at all, like a random walk. You will compare the ACF for the slightly mean-reverting interest rate series of the last exercise with a simulated random walk with the same number of observations.

You should notice when plotting the autocorrelation of these two series side-by-side that they look very similar.

Code

# Import the plot\_acf module from statsmodels

from statsmodels.graphics.tsaplots import plot\_acf

# Plot the interest rate series and the simulated random walk series side-by-side

fig, axes = plt.subplots(2,1)

# Plot the autocorrelation of the interest rate series in the top plot

fig = plot\_acf(interest\_rate\_data, alpha=1, lags=12, ax=axes[0])

# Plot the autocorrelation of the simulated random walk series in the bottom plot

fig = plot\_acf(simulated\_data, alpha=1, lags=12, ax=axes[1])

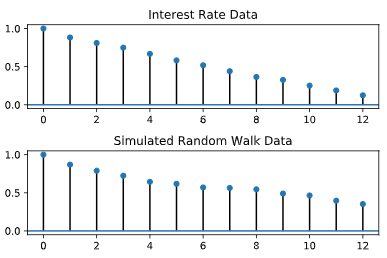
# Label axes

axes[0].set\_title("Interest Rate Data")

axes[1].set\_title("Simulated Random Walk Data")

plt.show()

# Notice the Autocorrelation functions look very similar for the two series.



# Estimate Order of Model: PACF

One useful tool to identify the order of an AR model is to look at the Partial Autocorrelation Function (PACF). In this exercise, you will simulate two time series, an AR(1) and an AR(2), and calculate the sample PACF for each. You will notice that for an AR(1), the PACF should have a significant lag-1 value, and roughly zeros after that. And for an AR(2), the sample PACF should have significant lag-1 and lag-2 values, and zeros after that.

Just like you used the plot\_acf function in earlier exercises, here you will use a function called plot\_pacf in the statsmodels module.

Code

# Import the modules for simulating data and for plotting the PACF

from statsmodels.tsa.arima\_process import ArmaProcess

from statsmodels.graphics.tsaplots import plot\_pacf

# Simulate AR(1) with phi=+0.6

# remember that the sign for the AR parameter is reversed

ma = np.array([1])

ar = np.array([1, -0.6])

AR\_object = ArmaProcess(ar, ma)

simulated\_data\_1 = AR\_object.generate\_sample(nsample=5000)

# Plot PACF for AR(1)

plot\_pacf(simulated\_data\_1, lags=20)

plt.show()

# Simulate AR(2) with phi1=+0.6, phi2=+0.3

ma = np.array([1])

ar = np.array([1, -0.6, -0.3])

AR\_object = ArmaProcess(ar, ma)

simulated\_data\_2 = AR\_object.generate\_sample(nsample=5000)

# Plot PACF for AR(2)

plot\_pacf(simulated\_data\_2, lags=20)

plt.show()

# Notice that the number of significant lags for the PACF indicate the order of the AR model

# Estimate Order of Model: Information Criteria

Another tool to identify the order of a model is to look at the Akaike Information Criterion (AIC) and the Bayesian Information Criterion (BIC). These measures compute the goodness of fit with the estimated parameters, but apply a penalty function on the number of parameters in the model. You will take the AR(2) simulated data from the last exercise, saved as simulated\_data\_2, and compute the BIC as you vary the order, p, in an AR(p) from 0 to 6.

Code

# Import the module for estimating an ARMA model

from statsmodels.tsa.arima\_model import ARMA

# Fit the data to an AR(p) for p = 0,...,6 , and save the BIC

BIC = np.zeros(7)

for p in range(7):

mod = ARMA(simulated\_data\_2, order=(p,0))

res = mod.fit()

# Save BIC for AR(p)

BIC[p] = res.bic

# Plot the BIC as a function of p

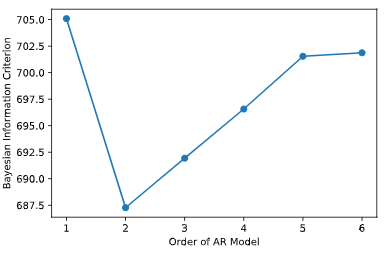
plt.plot(range(1,7), BIC[1:7], marker='o')

plt.xlabel('Order of AR Model')

plt.ylabel('Bayesian Information Criterion')

plt.show()

# For an AR(2), the BIC achieves its minimum at p=2, which is what we expect.



# Ch4 Moving Average (MA) and ARMA Models

# Simulate MA(1) Time Series

You will simulate and plot a few MA(1) time series, each with a different parameter, θθ, using the arima\_process module in statsmodels, just as you did in the last chapter for AR(1) models. You will look at an MA(1) model with a large positive θθ and a large negative θθ.

As in the last chapter, when inputting the coefficients, you must include the zero-lag coefficient of 1, but unlike the last chapter on AR models, the sign of the MA coefficients is what we would expect. For example, for an MA(1) process with θ=−0.9θ=−0.9, the array representing the MA parameters would be ma = np.array([1, -0.9])

Code

# import the module for simulating data

from statsmodels.tsa.arima\_process import ArmaProcess

# Plot 1: MA parameter = -0.9

plt.subplot(2,1,1)

ar1 = np.array([1])

ma1 = np.array([1, -0.9])

MA\_object1 = ArmaProcess(ar1, ma1)

simulated\_data\_1 = MA\_object1.generate\_sample(nsample=1000)

plt.plot(simulated\_data\_1)

# Plot 2: MA parameter = +0.9

plt.subplot(2,1,2)

ar2 = np.array([1])

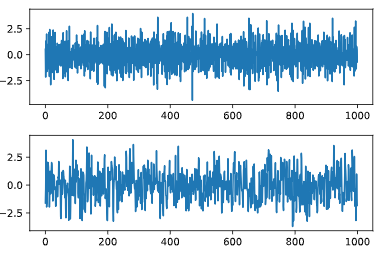
ma2 = np.array([1, 0.9])

MA\_object2 = ArmaProcess(ar2,ma2)

simulated\_data\_2 = MA\_object2.generate\_sample(nsample=1000)

plt.plot(simulated\_data\_2)

plt.show()



# Compute the ACF for Several MA Time Series

Unlike an AR(1), an MA(1) model has no autocorrelation beyond lag 1, an MA(2) model has no autocorrelation beyond lag 2, etc. The lag-1 autocorrelation for an MA(1) model is not θθ, but rather θ/(1+θ2)θ/(1+θ2). For example, if the MA parameter, θθ, is = +0.9, the first-lag autocorrelation will be 0.9/(1+(0.9)2)=0.4970.9/(1+(0.9)2)=0.497, and the autocorrelation at all other lags will be zero. If the MA parameter, θθ, is -0.9, the first-lag autocorrelation will be −0.9/(1+(−0.9)2)=−0.497−0.9/(1+(−0.9)2)=−0.497.

You will verify these autocorrelation functions for the three time series you generated in the last exercise.

Code

# Import the plot\_acf module from statsmodels

from statsmodels.graphics.tsaplots import plot\_acf

# Plot three ACF on same page for comparison using subplots

fig, axes = plt.subplots(3,1)

# Plot 1: AR parameter = -0.9

plot\_acf(simulated\_data\_1, lags=20, ax=axes[0])

axes[0].set\_title("MA Parameter -0.9")

# Plot 2: AR parameter = +0.9

plot\_acf(simulated\_data\_2, lags=20, ax=axes[1])

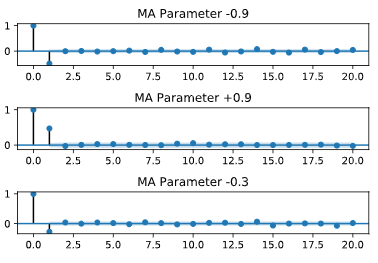
axes[1].set\_title("MA Parameter +0.9")

# Plot 3: AR parameter = -0.3

plot\_acf(simulated\_data\_3, lags=20, ax=axes[2])

axes[2].set\_title("MA Parameter -0.3")

plt.show()



# Estimating an MA Model

You will estimate the MA(1) parameter, θθ, of one of the simulated series that you generated in the earlier exercise. Since the parameters are known for a simulated series, it is a good way to understand the estimation routines before applying it to real data.

For simulated\_data\_1 with a true θθ of -0.9, you will print out the estimate of θθ. In addition, you will also print out the entire output that is produced when you fit a time series, so you can get an idea of what other tests and summary statistics are available in statsmodels.

Code

# Import the ARMA module from statsmodels

from statsmodels.tsa.arima\_model import ARMA

# Fit an MA(1) model to the first simulated data

# the order (p,q) of the model (in this case, for an MA(1)), is order=(0,1)

mod = ARMA(simulated\_data\_1, order=(0,1))

res = mod.fit()

# Print out summary information on the fit

print(res.summary())

# Print out the estimate for the constant and for theta

print("When the true theta=-0.9, the estimate of theta (and the constant) are:")

print(res.params)

# Forecasting with MA Model

As you did with AR models, you will use MA models to forecast in-sample and out-of-sample data using statsmodels.

For the simulated series simulated\_data\_1 with θ=−0.9θ=−0.9, you will plot in-sample and out-of-sample forecasts. One big difference you will see between out-of-sample forecasts with an MA(1) model and an AR(1) model is that the MA(1) forecasts more than one period in the future are simply the mean of the sample.

Code

# Import the ARMA module from statsmodels

from statsmodels.tsa.arima\_model import ARMA

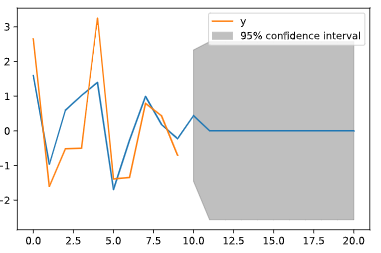
# Forecast the first MA(1) model

mod = ARMA(simulated\_data\_1, order=(0,1))

res = mod.fit()

res.plot\_predict(start=990, end=1010)

plt.show()



# High Frequency Stock Prices

Higher frequency stock data is well modelled by an MA(1) process, so it's a nice application of the models in this chapter.

The DataFrame intraday contains one day's prices (on September 1, 2017) for Sprint stock (ticker symbol "S") sampled at a frequency of one minute. The stock market is open for 6.5 hours (390 minutes), from 9:30am to 4:00pm.

Before you can analyze the time series data, you will have to clean it up a little, which you will do in this and the next two exercises. When you look at the first few rows (see the IPython Shell), you'll notice several things. First, there are no column headers.The data is not time stamped from 9:30 to 4:00, but rather goes from 0 to 390. And you will notice that the first date is the odd-looking "a1504272600". The number after the "a" is Unix time which is the number of seconds since January 1, 1970. This is how this dataset separates each day of intraday data.

If you look at the data types, you'll notice that the DATE column is an object, which here means a string. You will need to change that to numeric before you can clean up some missing data.

The source of the minute data is Google Finance (see [**here**](https://www.quantshare.com/sa-426-6-ways-to-download-free-intraday-and-tick-data-for-the-us-stock-market) on how the data was downloaded).

The datetime module has already been imported for you.

Code

# import datetime module

import datetime

# Change the first date to zero

intraday.iloc[0,0] = 0

# Change the column headers to 'DATE' and 'CLOSE'

intraday.columns = ['DATE','CLOSE']

# Examine the data types for each column

print(intraday.dtypes)

# Convert DATE column to numeric

intraday['DATE'] = pd.to\_numeric(intraday['DATE'])

# Make the `DATE` column the new index

intraday = intraday.set\_index(['DATE'])

# More Data Cleaning: Missing Data

When you print out the length of the DataFrame intraday, you will notice that a few rows are missing. There will be missing data if there are no trades in a particular one-minute interval. One way to see which rows are missing is to take the difference of two sets: the full set of every minute and the set of the DataFrame index which contains missing rows. You can fill in the missing rows with the .reindex() method, convert the index to time of day, and then plot the data.

Stocks trade at discrete one-cent increments (although a small percentage of trades occur in between the one-cent increments) rather than at continuous prices, and when you plot the data you should observe that there are long periods when the stock bounces back and forth over a one cent range. This is sometimes referred to as "bid/ask bounce".

Code

# Notice that some rows are missing

print("The length of the DataFrame is: ",len(intraday))

# Find the missing rows

print("Missing rows: ", set(range(391)) - set(intraday.index))

# Fill in the missing rows

intraday = intraday.reindex(range(391), method='ffill')

# Change the index to the intraday times

intraday.index = pd.date\_range(start='2017-08-28 9:30', end='2017-08-28 16:00', freq='1min')

# Plot the intraday time series

intraday.plot(grid=True)

plt.show()

# Missing data is common with high frequency financial time series, so good job fixing that.

# Applying an MA Model

The bouncing of the stock price between bid and ask induces a negative first order autocorrelation, but no autocorrelations at lags higher than 1. You get the same ACF pattern with an MA(1) model. Therefore, you will fit an MA(1) model to the intraday stock data from the last exercise.

The first step is to compute minute-by-minute returns from the prices in intraday, and plot the autocorrelation function. You should observe that the ACF looks like that for an MA(1) process. Then, fit the data to an MA(1), the same way you did for simulated data.

Code

# Import plot\_acf and ARMA modules from statsmodels

from statsmodels.graphics.tsaplots import plot\_acf

from statsmodels.tsa.arima\_model import ARMA

# Compute returns from prices and drop the NaN

returns = intraday.pct\_change()

returns = returns.dropna()

# Plot ACF of returns with lags up to 60 minutes

plot\_acf(returns, lags=60)

plt.show()

# Fit the data to an MA(1) model

mod = ARMA(returns, order=(0,1))

res = mod.fit()

print(res.params)

# Notice the significant negative lag-1 autocorrelation, just like for an MA(1) model.

# Equivalence of AR(1) and MA(infinity)

To better understand the relationship between MA models and AR models, you will demonstrate that an AR(1) model is equivalent to an MA(∞∞) model with the appropriate parameters.

You will simulate an MA model with parameters 0.8,0.82,0.83,…0.8,0.82,0.83,… for a large number (30) lags and show that it has the same Autocorrelation Function as an AR(1) model with ϕ=0.8ϕ=0.8.

Code

# import the modules for simulating data and plotting the ACF

from statsmodels.tsa.arima\_process import ArmaProcess

from statsmodels.graphics.tsaplots import plot\_acf

# Build a list MA parameters

ma = [0.8\*\*i for i in range(30)]

# Simulate the MA(30) model

ar = np.array([1])

AR\_object = ArmaProcess(ar, ma)

simulated\_data = AR\_object.generate\_sample(nsample=5000)

# Plot the ACF

plot\_acf(simulated\_data, lags=30)

plt.show()

# Notice that the ACF looks the same as an AR(1) with parameter 0.8

# Ch 5: Putting It All Together

# A Dog on a Leash? (Part 1)

The Heating Oil and Natural Gas prices are pre-loaded in DataFrames HO and NG. First, plot both price series, which look like random walks. Then plot the difference between the two series, which should look more like a mean reverting series (to put the two series in the same units, we multiply the heating oil prices, in $/gallon, by 7.25, which converts it to $/millionBTU, which is the same units as Natural Gas).

The data for continuous futures (each contract has to be spliced together in a continuous series as contracts expire) was obtained from **[Quandl](https://blog.quandl.com/api-for-futures-data" \t "_blank)**.

Code

# Plot the prices separately

# Make sure you multiply the HO price by 7.25 to match the units of NG

plt.subplot(2,1,1)

plt.plot(7.25\*HO, label='Heating Oil')

plt.plot(NG, label='Natural Gas')

plt.legend(loc='best', fontsize='small')

# Plot the spread

plt.subplot(2,1,2)

plt.plot(7.25\*HO-NG, label='Spread')

plt.legend(loc='best', fontsize='small')

plt.axhline(y=0, linestyle='--', color='k')

plt.show()

# Notice from the plot that when Heating Oil briefly dipped below Natural Gas, it quickly reverted back up.

# A Dog on a Leash? (Part 2)

To verify that HO and NG are cointegrated, First apply the Dickey-Fuller test to HO and NG separately to show they are random walks. Then apply the test to the difference, which should strongly reject the random walk hypothesis. The Heating Oil and Natural Gas prices are pre-loaded in DataFrames HO and NG.

Code

# Import the adfuller module from statsmodels

from statsmodels.tsa.stattools import adfuller

# Compute the ADF for HO and NG

result\_HO = adfuller(HO['Close'])

print("The p-value for the ADF test on HO is ", result\_HO[1])

result\_NG = adfuller(NG['Close'])

print("The p-value for the ADF test on NG is ", result\_NG[1])

# Compute the ADF of the spread

result\_spread = adfuller(7.25 \* HO['Close'] - NG['Close'])

print("The p-value for the ADF test on the spread is ", result\_spread[1])

# As we expected, we cannot reject the hypothesis that the individual futures are random walks, but we can reject that the spread is a random walk.

# Are Bitcoin and Ethereum Cointegrated?

Cointegration involves two steps: regressing one time series on the other to get the cointegration vector, and then perform an ADF test on the residuals of the regression. In the last example, there was no need to perform the first step since we implicitly assumed the cointegration vector was (1,−1)(1,−1). In other words, we took the difference between the two series (after doing a units conversion). Here, you will do both steps.

You will regress the value of one cryptocurrency, bitcoin (BTC), on another cryptocurrency, ethereum (ETH). If we call the regression coefficient bb, then the cointegration vector is simply (1,−b)(1,−b). Then perform the ADF test on BTC −b−b ETH. Bitcoin and Ethereum prices are pre-loaded in DataFrames BTC and ETH.

Bitcoin data are in DataFrame BTC and Ethereum data are in ETH.

Code

# Import the statsmodels module for regression and the adfuller function

import statsmodels.api as sm

from statsmodels.tsa.stattools import adfuller

# Regress BTC on ETH

ETH = sm.add\_constant(ETH)

result = sm.OLS(BTC,ETH).fit()

# Compute ADF

b = result.params[1]

adf\_stats = adfuller(BTC['Price'] - b\*ETH['Price'])

print("The p-value for the ADF test is ", adf\_stats[1])

# The data suggests that Bitcoin and Ethereum are cointegrated.

# The p-value for the ADF test is 0.0233690023235

# Is Temperature a Random Walk (with Drift)?

An ARMA model is a simplistic approach to forecasting climate changes, but it illustrates many of the topics covered in this class.

The DataFrame temp\_NY contains the average annual temperature in Central Park, NY from 1870-2016 (the data was downloaded from the NOAA [**here**](https://www.ncdc.noaa.gov/cdo-web/search)). Plot the data and test whether it follows a random walk (with drift).

Code

# Import the adfuller function from the statsmodels module

from statsmodels.tsa.stattools import adfuller

# Convert the index to a datetime object

temp\_NY.index = pd.to\_datetime(temp\_NY.index, format='%Y')

# Plot average temperatures

temp\_NY.plot()

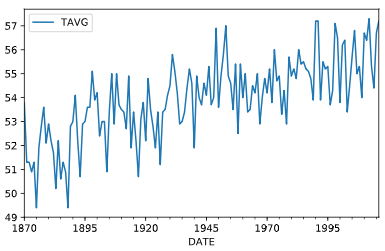
plt.show()

# Compute and print ADF p-value

result = adfuller(temp\_NY['TAVG'])

print("The p-value for the ADF test is ", result[1])

# The data seems to follow a random walk with drift.



# Getting "Warmed" Up: Look at Autocorrelations

Since the temperature series, temp\_NY, is a random walk with drift, take first differences to make it stationary. Then compute the sample ACF and PACF. This will provide some guidance on the order of the model.

Code

# Import the modules for plotting the sample ACF and PACF

from statsmodels.graphics.tsaplots import plot\_acf, plot\_pacf

# Take first difference of the temperature Series

chg\_temp = temp\_NY.diff()

chg\_temp = chg\_temp.dropna()

# Plot the ACF and PACF on the same page

fig, axes = plt.subplots(2,1)

# Plot the ACF

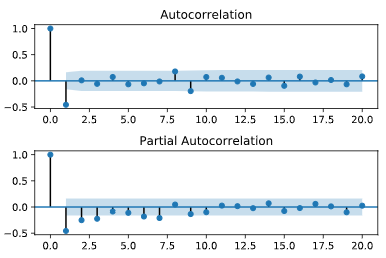
plot\_acf(chg\_temp, lags=20, ax=axes[0])

# Plot the PACF

plot\_pacf(chg\_temp, lags=20, ax=axes[1])

plt.show()

# There is no clear pattern in the ACF and PACF except the negative lag-1 autocorrelation in the ACF.



# Which ARMA Model is Best?

Fit the temperature data to an AR(1), AR(2), MA(1), and ARMA(1,1) and see which model is the best fit, using the AIC criterion.

Code

# Import the module for estimating an ARMA model

from statsmodels.tsa.arima\_model import ARMA

# Fit the data to an AR(1) model and print AIC:

mod = ARMA(chg\_temp, order=(1,0))

res = mod.fit()

print("The AIC for an AR(1) is: ", res.aic)

# Fit the data to an AR(2) model and print AIC:

mod = ARMA(chg\_temp, order=(2,0))

res = mod.fit()

print("The AIC for an AR(2) is: ", res.aic)

# Fit the data to an MA(1) model and print AIC:

mod = ARMA(chg\_temp, order=(0,1))

res = mod.fit()

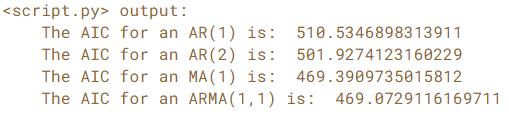
print("The AIC for an MA(1) is: ", res.aic)

# Fit the data to an ARMA(1,1) model and print AIC:

mod = ARMA(chg\_temp, order=(1,1))

res = mod.fit()

print("The AIC for an ARMA(1,1) is: ", res.aic)



# The MA(1) and ARMA(1,1) have the two lowest AIC values.

# Don't Throw Out That Winter Coat Yet

Finally, you will forecast the temperature over the next 30 years using an ARMA(1,1) model, including confidence bands around that estimate. Keep in mind that the estimate of the drift will have a much bigger impact on long range forecasts than the ARMA parameters.

Earlier, you determined that the temperature data follows a random walk and you looked at first differencing the data. You will use the ARIMA module on the temperature data, pre-loaded in the DataFrame temp\_NY, but the forecast would be the same as using the ARMA module on changes in temperature, and then using cumulative sums of these changes to get the temperature.

The data is in a DataFrame called temp\_NY.

Code

# Import the ARIMA module from statsmodels

from statsmodels.tsa.arima\_model import ARIMA

# Forecast interest rates using an AR(1) model

# The d in order(p,d,q) is one, since we first differenced once

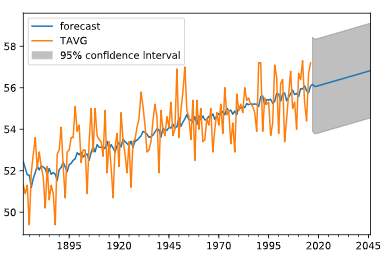
mod = ARIMA(temp\_NY, order=(1,1,1))

res = mod.fit()

# Plot the original series and the forecasted series

res.plot\_predict(start='1872-01-01', end='2046-01-01')

plt.show()



# According to the model (almost entirely due to the trend, the temperature is expected to be about 0.6 degrees higher in 30 years, but the 95% confidence interval around that is over 5 degrees.