Stats Bootcamp Day 10 - Time Series

Anastasia Bernat; inspired by Bowei Kang (University of Chicago) & Rick Presman (Duke University)

09/26/21

What is a time series?

Primarily, a time series is a set of data taken sequentially in time, usually at equal time intervals. x_t at times $t = 1, 2, ...N$, where N is the length of the time series.

It is different than non-temporal data because each data point has an order and is, typically, related to the data points before and after by some underlying process.

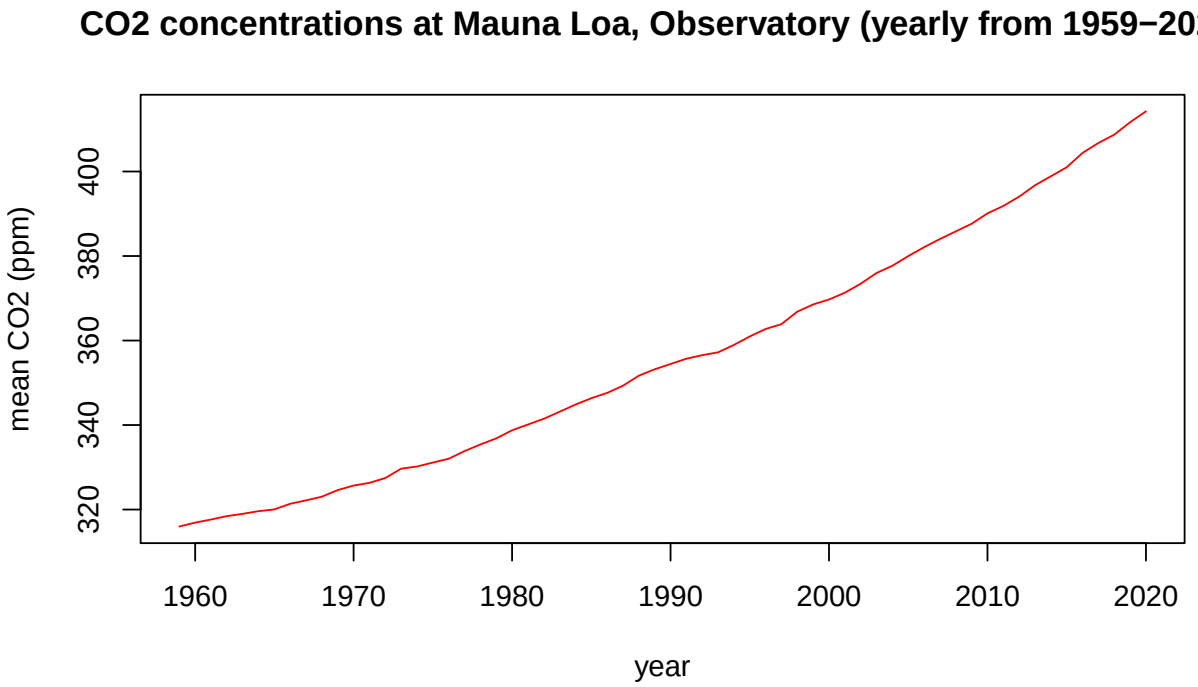
Questions often asked (via examples)

Take a look at these examples and answer the common questions asked by analysts when working with time series data.

The data below was collected by the National Oceanic and Atmospheric Administration (NOAA), specifically through the Scripps Institution of Oceanography (SIO). The data are comprised of CO2 concentration, readings, measured in parts per million (ppm) readings, from March 1958 through April 1974. To look more into it, copy and paste the URL into your browser.

You'll also notice that data is being read in a way you haven't seen before - using a URL instead of a CSV file. This is another great way to read and collect data and it can get as intricate as you want. If you're curious, look up "web scraping in R". And be sure to have an internet connection; otherwise, the code will throw an error!

```
title = "CO2 concentrations at Mauna Loa, Observatory (yearly from 1959-2020)"

# read the data
URL = "ftp://aftp.cmdl.noaa.gov/products/trends/co2/co2_annmean_mlo.txt"
CO2_yr = read.csv(url(URL),
                  skip=56, # when you open the link, you'll see a lot of commentary
                           # in the beginning, then the data. This skips that to the line
                           # you want to start at
                  sep="", header=TRUE,
                  col.names=c("year", "mean_CO2_ppm", "uncertainty"))

# plot
      # note: notice the type argument below.
      # there are various "types" that you can choose from,
      # try replacing type="l" with type="o"
plot(CO2_yr$year, CO2_yr$mean_CO2_ppm, main=title,
     xlab="year", ylab="mean CO2 (ppm)", col="red", type = "l")
```

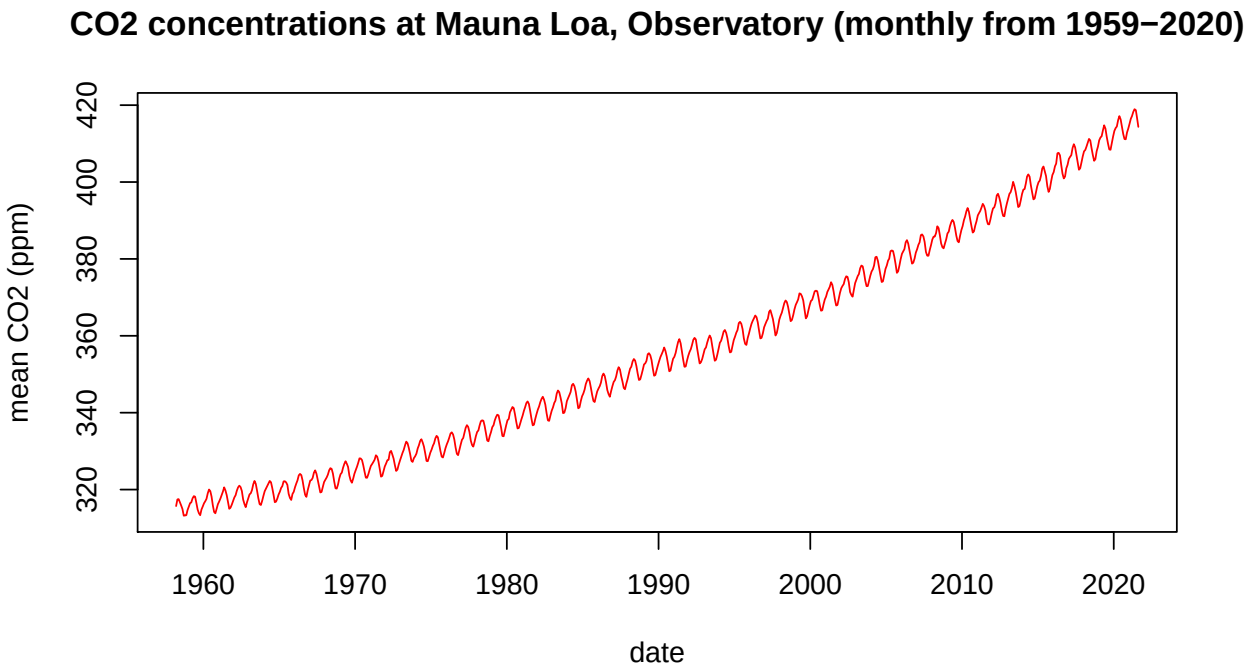


Q: What information can we get from this series? Are there any trends or patterns?

A: There's a positive upwards trend. There also aren't any noticeable dips or rises that stand out, so it's pretty smooth.

Different data resolutions tell very different stories. Consider this dataset also collected from NOAA SIO, but instead of yearly data it is monthly data:

```
title = "CO2 concentrations at Mauna Loa, Observatory (monthly from 1959-2020)"

# read the data
URL = "https://scrippsco2.ucsd.edu/assets/data/atmospheric/stations/in_situ_co2/monthly/monthly_in_situ
CO2_mon = read.csv(url(URL), skip=56)

# clean the data
colnames(CO2_mon) =c("year", "month", "Excel_date", "date",
                     "CO2_ppm1", "CO2_ppm2", "CO2_ppm3", "CO2_ppm4", "CO2_ppm5")

CO2_mon_filtered = CO2_mon[CO2_mon$CO2_ppm1 !=-99.99,] # remove NA [CO2] values designated as -99.99

# plot
plot(CO2_mon_filtered$date, CO2_mon_filtered$CO2_ppm1, main=title,
     xlab="date", ylab="mean CO2 (ppm)", col="red", type="l")
```



Q: Is there a constant variance over time? What patterns do you notice at this new resolution?

A: There is a constant variance overtime, it's seasonal, dipping in the summer (crops absorb and store atmospheric CO2) and peaking in the winter (crops dying and releasing CO2 stored back into the atmosphere).

Time series data also often exhibits cyclical behavior, but sometimes cycles are unique - the series are not perfectly predictable or strictly periodic.

Consider this economic data on new housing authorizations ('building permits') issued by communities in the USA. This data was sourced from the US Census Bureau & U.S. Department of Housing and Urban Development. It spans from 1960 to 2021, and it has permit numbers for each month of the year.

```
title = "New Housing Authorizations in the USA (monthly from 1960-2021)"
# source: https://fred.stlouisfed.org/series/PERMIT
# source #2: https://www.princeton.edu/~mwatson/papers/isb201119.pdf

# read and clean the data
permits = read.csv("permit_data.csv")
head(permits)
```

```
##         DATE PERMIT
## 1 1960-01-01   1092
## 2 1960-02-01   1088
## 3 1960-03-01    955
## 4 1960-04-01   1016
## 5 1960-05-01   1052
## 6 1960-06-01    958
```

One thing you have to do before plotting this is change the class of the date column from a **character** to a **Date type**, using the `as.Date()` function. This is a common step during time series, but it can also be streamlined using particular time series libraries.

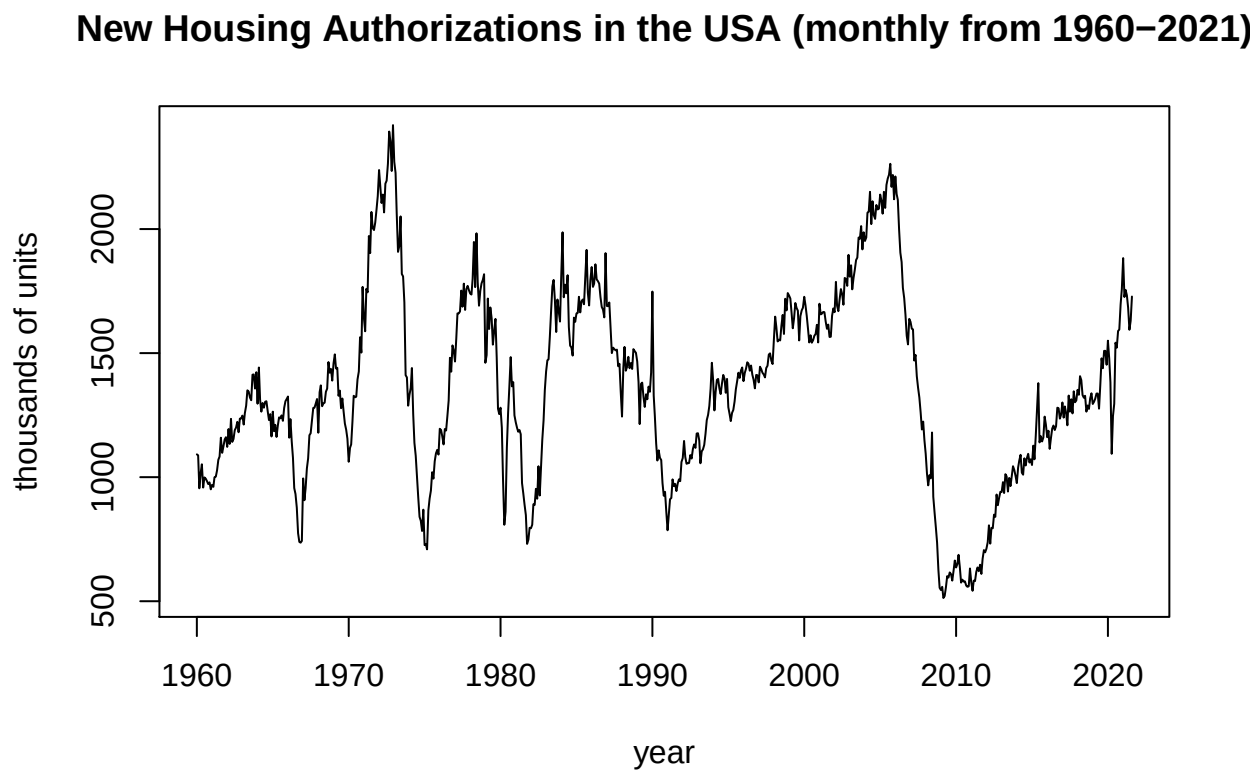
`as.Date()` takes two important arguments, 1) a vector of values you want to convert to a Date type and 2) the format the values were written in. Here is a helpful table of the conversion symbols you will need: https://www.statmethods.net/input/dates.html. And here is a quick example of the function in action using today's date and the `format()` function to show various conversion symbols.

```r
# for example, print today's date
today <- Sys.Date()
today # date

format(today, format="%B %d %Y") # fully unabbreviated month, day, and year
format(today, format="%b %d %y") # abbreviated month, day, and year
format(today, format="%A") # weekday
```

```
## [1] "2021-09-26"
## [1] "September 26 2021"
## [1] "Sep 26 21"
## [1] "Sunday"
```

Now let's get back to the data and do the necessary conversions and then plot.

```r
title = "New Housing Authorizations in the USA (monthly from 1960-2021)"
# source: https://fred.stlouisfed.org/series/PERMIT
# source #2: https://www.princeton.edu/~mwatson/papers/isb201119.pdf

# read and clean the data
permits = read.csv("permit_data.csv")
colnames(permits) = tolower(colnames(permits))
permits$date = as.Date(permits$date, "%Y-%m-%d") # convert character column to a Date type

# plot
plot(permits$date, permits$permit, main = title,
     xlab="year",
     ylab="thousands of units",
     type="l")
```



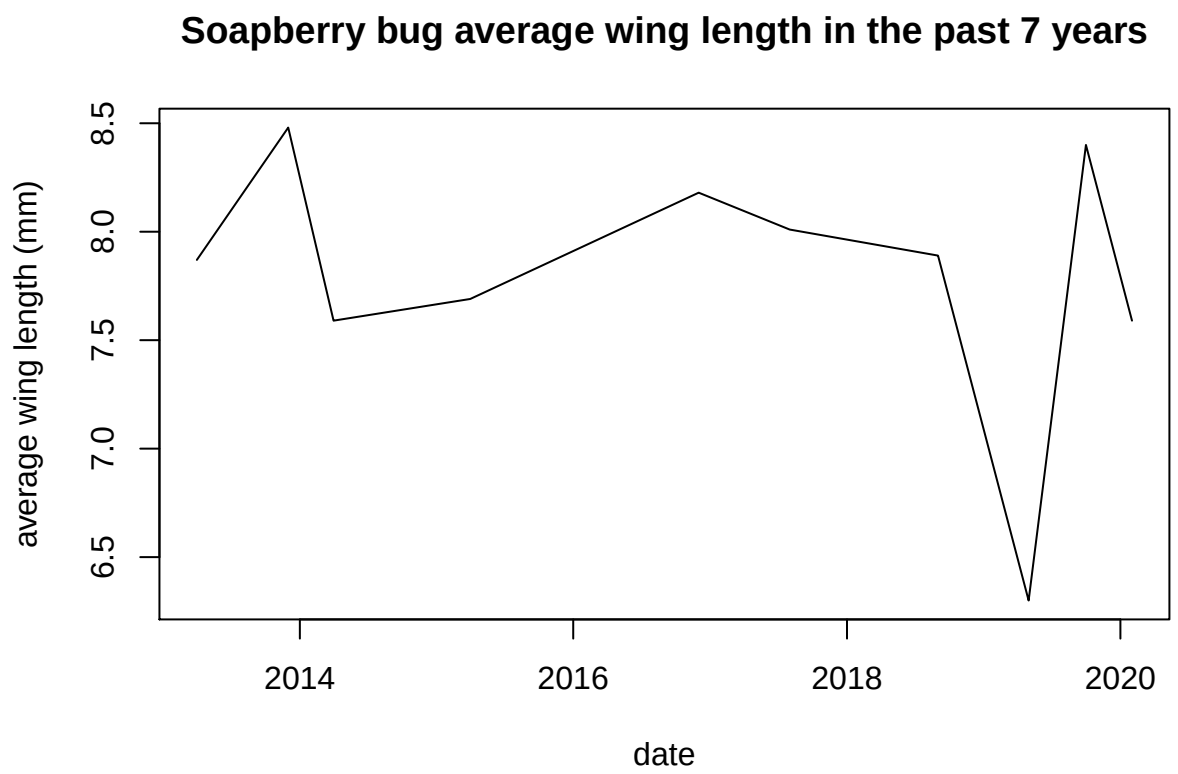
This plot has characteristics that are typical of many economic time series. That being said,

Q: What characteristics can you identify? Is there a long-run cycle or period unrelated to seasonality factors? Which local minima/maxima would you point out?

A: There are periods of high activity, with a local maxima around 1972? And there are periods of low activity with a local minima around 2010. And from 1970 to 1990 these dips and peaks were shaped similarly until after 1990, the cycle changed and it became a wider cycle with gradual increasing in permit authorizations and then a sharp dip.

And what about outliers? Consider this final dataset. This is ecology data on an insect species from Florida called soapberry bugs. The data averages the wing length of soapberry bug populations across time. The data is more of a 'short time series', and it isn't ideally spaced across equal time intervals, but it can give a sense of the very realistic data you might come across if you're studying the biological sciences.

```
title = "Soapberry bug average wing length in the past 7 years"

date = c("April 2013 01", "December 2013 01", "April 2014 01", "April 2015 01", "December 2016 01",
         "August 2017 01", "September 2018 01", "May 2019 01", "October 2019 01", "February 2020 01")
wing_length_avg = c(7.87, 8.48, 7.59, 7.69, 8.18, 8.01, 7.89, 6.3, 8.40, 7.59)
wing_length = data.frame(date, wing_length_avg)
wing_length$date = as.Date(wing_length$date, "%b %Y %d")

# plot
plot(wing_length$date, wing_length$wing_length_avg, main = title,
     xlab="date",
     ylab="average wing length (mm)",
     type="l")
```



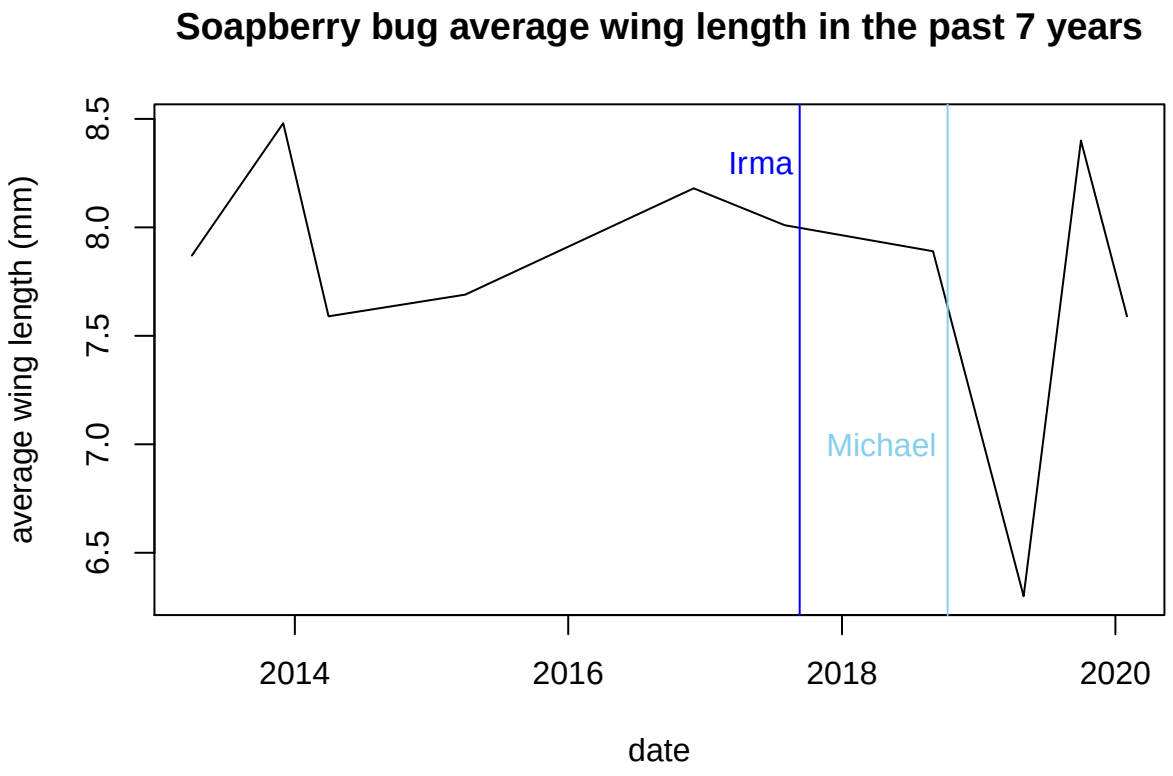
Q: Are there any outliers or abnormal observations? Are there any abrupt changes to either the level of the series or the variance? And what would be your best guess as to why the outlier(s) exist(s)?

A: Between 2019 and 2020 there's a drop in soapberry bug average wing length. It's possible that some event changed the population drastically (e.g. genetic drift, environmental degradation). Keep reading to find

out...

If you're stumped, consider this. Florida is subject to hurricanes. What would happen if you superimpose those major hurricane events on the plot? Run this:

```
# events
hurricanes = c("Irma", "Michael")
date = c("Sep 2017 10", "Oct 2018 10")
FL_hurr_events = data.frame(date, hurricanes)
FL_hurr_events$date =  as.Date(date, "%b %Y %d")

# plot
plot(wing_length$date, wing_length$wing_length_avg, main = title,
     xlab="date",
     ylab="average wing length (mm)",
     type="l")
abline(v = FL_hurr_events$date, col=c("blue","skyblue"))
text(FL_hurr_events$date, c(8.3,7), labels=FL_hurr_events$hurricanes,
     adj=1.10,
     col=c("blue","skyblue"))
```



Two hurricanes, Irma (southern Florida, category 5) and Michael (northern Florida, category 5) ran through the state in September 2017 and Ocotber 2018, respectively. These back-to-back major hurricanes could have wiped out longer-winged soapberry bugs across Florida? Or maybe because the same sites are sampled each collection season, it could be that the longer-winged soapberry bugs flew away from their previously colonized areas to colonize new areas and left shorter-winged bugs behind? It's unclear, but there's some great time-sensitive research on insect re-colonization after a hurricane in this talk: https://www.youtube.com/watch?v=VMLkkxEQJ3I&t=15480s

Now that you've seen a variety of examples, you might consider that time series have a certain noticeable, general characteristic. Usually, it is useful to think of time series as realizations of **stochastic processes**, and this raises the question of how to represent the patterns you notice (e.g. cyclicality, variability, trends) in

stochastic processes. And you might wonder how you can predict future anomalies based on past information?

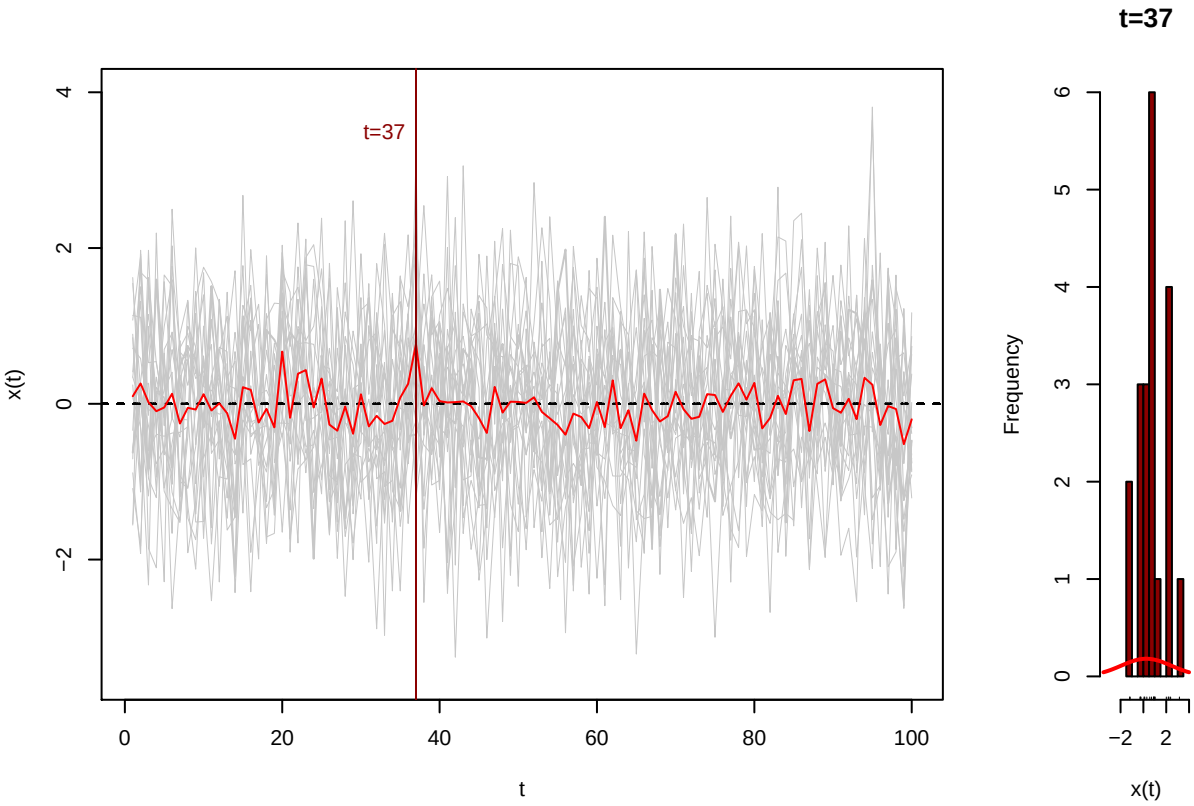
What makes time series special?

1) **consider sequence of random variables & models their joint distribution**
2) **it is impossible to make multiple observations at one time point**
3) **conventional statistical procedures may be inappropriate in this case**

1 We are dealing with a sequence of random variables and want to model some aspects of their joint distribution. We are interested in a joint distribution because when we analyze time-series we are interested in *the whole series that evolves over time* (aka. its joint distribution). So instead of describing a process which can only evolve in one way, in a stochastic or random process there is some indeterminacy: even if the initial condition (or starting point) is known, there are several (often infinitely many) directions in which the process may evolve.

Run this code below to visualize all the various random processes, or pathways, that can evolve. Don't worry if you don't understand the code - focus on seeing those pathways and how, at each time point, there are multiple possible outcomes.

```
set.seed(1)
m = 100 # length of time series
p = 20 # number of processes
x = rnorm(m, 0, 1)

xdat <- data.frame(matrix(NA,
                          nrow = p, # each row is a process
                          ncol = m)) # each column is an observation in the time series

layout(matrix(c(1,1,1,1,2), nrow = 1, ncol = 5, byrow = TRUE))
# layout() function can display multiple plots in a grid-like formation

plot(x, type='l', col='grey78', lwd=1, xlab="t", ylab="x(t)", ylim=c(-3.5,4))
for (process in 1:p) {
  x = rnorm(m, 0, 1)
  lines(1:m, x, col='grey78', lwd=0.5)
  xdat[process,] = x # populate the xdata
  abline(h=0, lty=2)
}
lines(1:m, apply(xdat, 2, mean) , col="red")
abline(v=37, col="darkred")
text(37, 3.5, "t=37", col="darkred", adj=1.25)

x = unlist(xdat[,37])
hist(x, xlim=c(-3.5,4),
     main="t=37",
     xlab="x(t)",
     col="darkred", breaks=10)
rug(x) # "rug plot": displays a single variable as marks along an axis
curve(dnorm(x, mean(x), sd(x)), lwd=2, col="red", add=TRUE) # normal probability density function
```



2 Although we could vary the length of an observed time series, it is often impossible to make multiple observations at a given time point in series. For instance, if you're recording the temperature as it evolves over a day, you cannot measure two temperatures at one time point.

3 Additionally, statistical practices that you've learned in this bootcamp do not necessarily apply in time series. Many statistical methods assume independence of observations, because the probability of independent events happening simultaneously can be easily calculated as a product of their individual probabilities: $P(A \cap B) = P(A) \cdot P(B)$. This is extensively applied e.g. in maximum likelihood estimation of parameters. But if your events are not independent, the above formula is not valid anymore. So, intuitively, the present and future will depend on the past. And there are many different mechanisms underlying this, including memory, inertia or momentum in strict or broad senses as well as the phenomena of growth or decay/decline.

Also, regressions aim to quantiy the specific impacts of specific underlying indpenednt variables, of the form:

$y = b_1x_1 + b_2x_2 + ... + b_nx_n$

But time series modeling allows us to replicate every element of the process by decomposing the mathematical process into a combination of **signals** (e.g. year-on-year growth, seasonal variability, etc.) and **noise** (random probabilistic processes), without necessarily knowing the underlying cuases for each.

So let's start with important concepts surrounding time series.

What are the fundamental concepts of time series data and analysis?

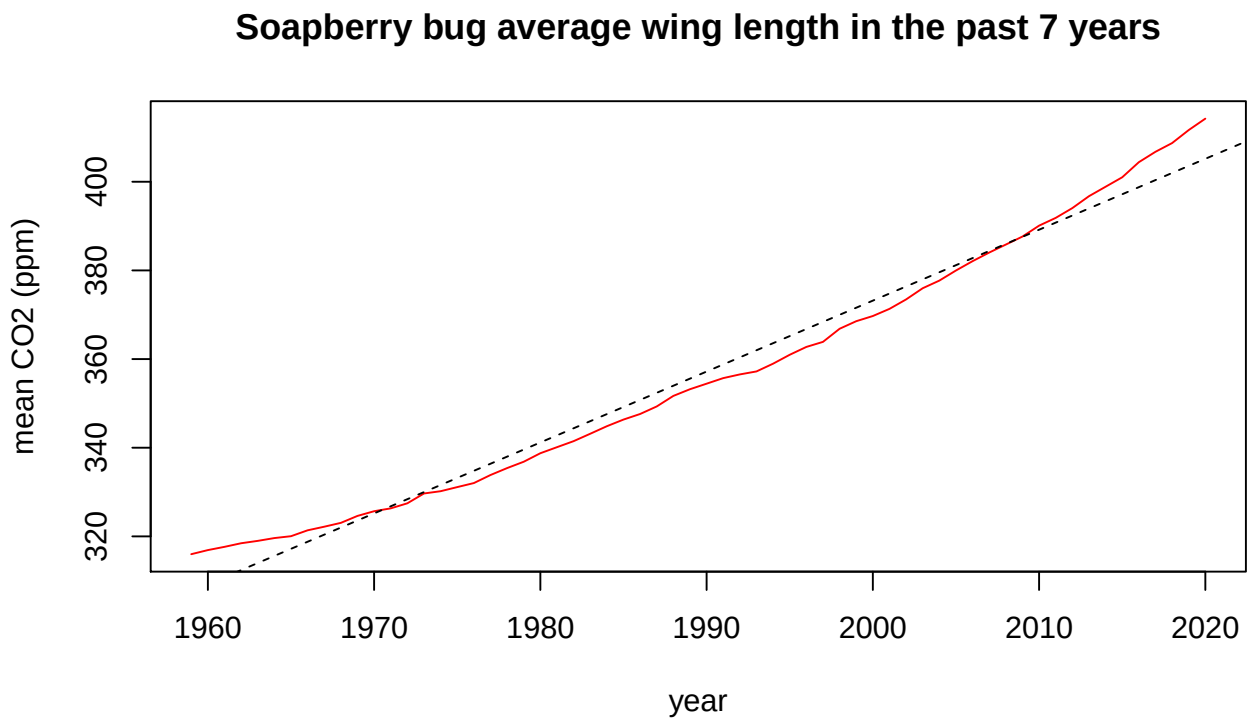
There are a variety of concepts: **autoregression, autocorrelation, serial correlation, stationarity, exogeneity, weak dependence, trending, seasonality, structural breaks, and stability**. But the main ones covered in this lesson will be stationarity, non-stationarity, and autocorrelation. And these concepts really revolve around three statistical terms, loosely defined based on what you noticed in the examples earlier:

- **variance/standard deviation** = amplitude of the data points

- **mean** = the central value of the data points
- **covariance** = frequency of the data points

In many cases, we can describe parts of time series processes in terms of a randomized variable with statistical moments, though an important feature of many time series processes is that their mean and/or variance changes through time. In the case of the yearly CO2 concentrations data, you can see that the mean is varying, but the oscillation around the mean (the standard deviation) looks constant. In other words, there's an upwards **trend**, the black dotted regression line below:

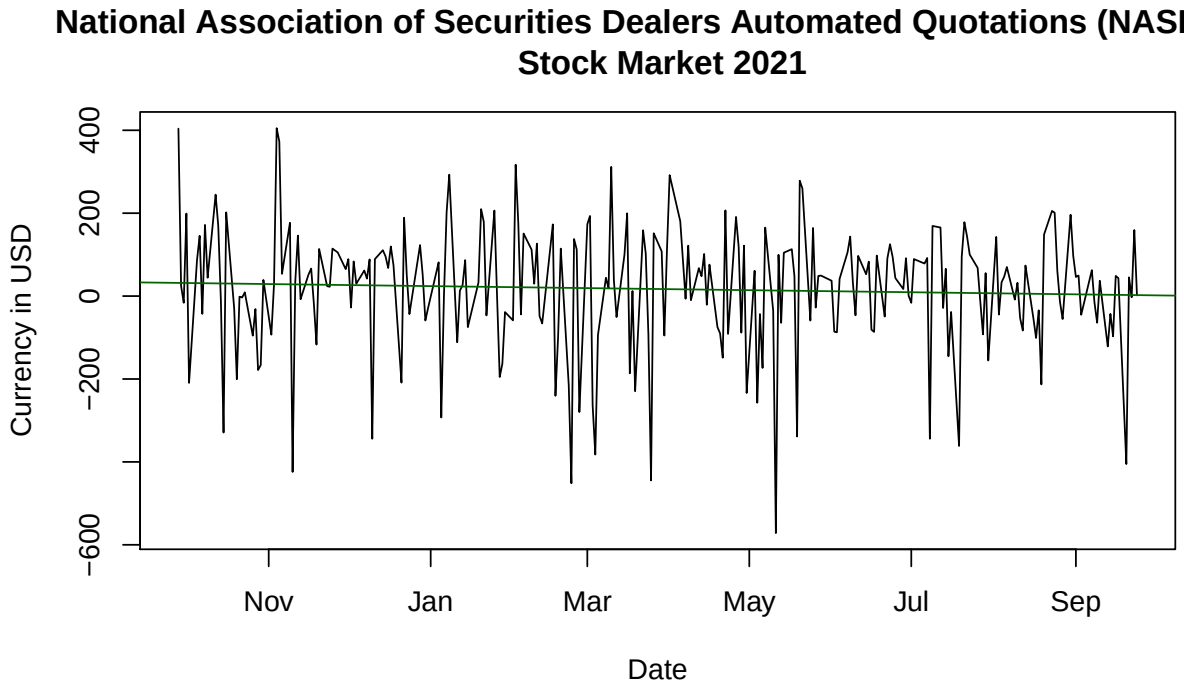
```
plot(CO2_yr$year, CO2_yr$mean_CO2_ppm, main=title,
     xlab="year", ylab="mean CO2 (ppm)", col="red", type = "l")
m = lm(mean_CO2_ppm ~ year, data=CO2_yr)
abline(m, lty=2)
```



The trend moves above the average, then regresses back towards it, often overshooting up and down.

If the mean is constant, but there's a changing standard deviation/variance, then you have "volatility". For example, this stock market data has a mean essentially around 0, but there are periods in the year where there are large fluctuations and small fluctuations.

```
nasdaq = read.csv("NASDAQ.csv")
nasdaq$Date = as.Date(nasdaq$Date, "%Y-%m-%d")
detrend = diff(nasdaq$Open) # I'm detrending here ("regressing out"), don't worry about it for now
nasdaq = nasdaq[-1,]
nasdaq$diff = detrend

plot(nasdaq$Date, nasdaq$diff, type="l",
     main="National Association of Securities Dealers Automated Quotations (NASDAQ)\n Stock Market 2021
     xlab="Date",
     ylab="Currency in USD")
m = lm(diff ~ Date, data=nasdaq)
abline(m, col="darkgreen") # mean is essentially at 0 now
```

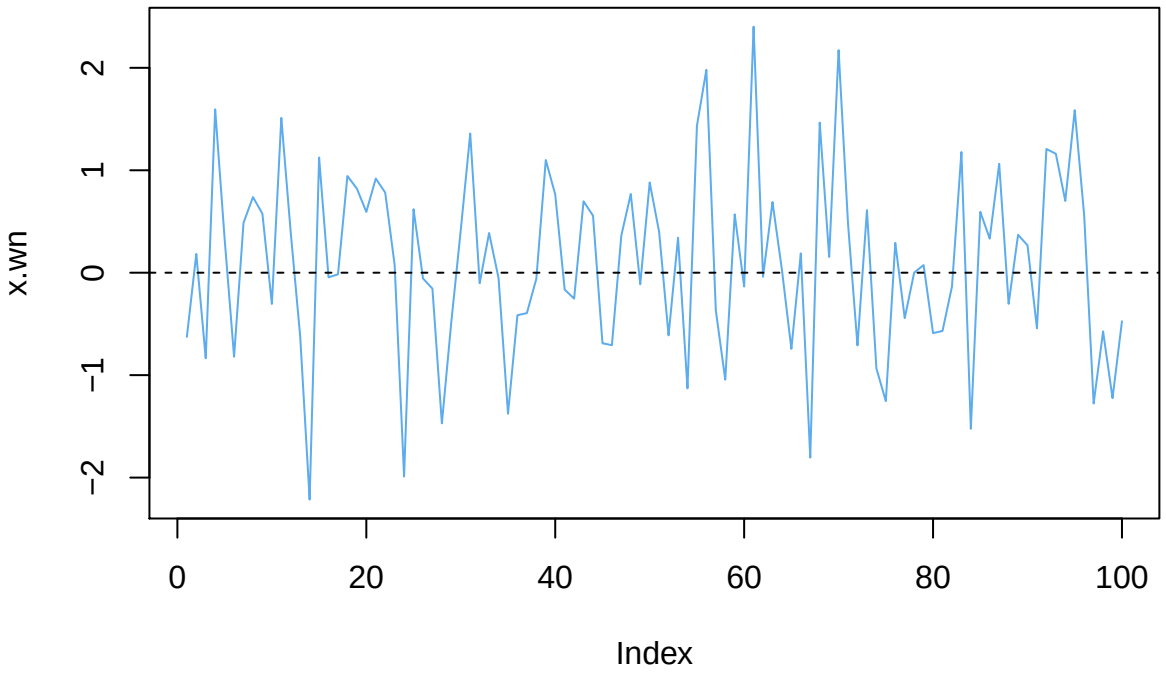


But what about noise? I mentioned it earlier but didn't explain it. White noise has a constant mean equal to 0 ($E_{e_t} = 0$), finite constant variance ($Var(e_t) = \sigma^2$), and zero correlation ($Cov(e_t, e_s)$ for all $s \neq t$). So you are left with the following,

$y_t = e_t$

where your random variable equals a random process. It can be simulated as follows,

```
# Gaussian White noise
set.seed(1)
m = 100 # length of time series
x.wn = rnorm(m, 0, 1)
plot(x.wn, type='l', col='steelblue2')
abline(h=0, lty=2)
```



Now that you've got a sense of how these terms are considered in a either a tangible or simulated way, let's jump into those three fundamental concepts and begin setting up the data for analysis and time series modeling. Here is a summary of those concepts and let's start with stationarity:

- **Stationarity**: a stochastic process where any trends or patterns in the time series are independent with time. Notably, when a time series is at stationarity, it is equivalent to a linear regression. Why? Because a linear regression assumes that the data points are independent of each other.
- **Non-Stationary**: a stochastic process where the trends in the series are dependent in time, whether in the mean or statistical deviation of the data points.
- **Autocorrelation**: "memory"; the degree to which time series values in period t are related to time series values in periods $t+1, t+2, t+3$... etc. For this, we can test to see how long certain values last in a system.

What is stationarity?

Stationarity occurs when a stochastic process, specifically its mean and variance, does not change over time. For our purposes, it is a flat-looking time series, without a trend and no periodic fluctuations. In other words, the pattern that a time series follows is independent with time.

There is strong stationarity where at every finite dimensionality are independent with time, and there is weak stationarity where the mean, variance, and covariance are broadly kept constant over time. In most cases, it is enough to focus on weak form.

So let's go through the process of checking for stationarity in your data as well as knowing how to deal with non-stationarity down the line. Let's begin through code and go through these first guiding steps highlighted in bold:

1. create a ts and xts object and plot it 2. check stationarity using ACF plots 3. handle non-stationary time series (which requires model selection and model fitting) 4. model checking 5. forecasting

1. create a ts and xts object and plot it

First import needed libraries:

```r
# import libraries

library(zoo)          # data manipulation
library(tidyverse)    # data manipulation

library(tseries)      # time series analysis
library(xts)          # time series analysis
library(forecast)     # model forecasting
library(fma)          # model forecasting
```

Now, let's simulate some data first to get an idea of how time series data can be generated. Let's create a time series (ts) object with (1) values: $2t + e_t$, where $e_t \sim N(0, 20)$, $t = 1, 2, \cdots, 120$; (2) time: from Jan-2019 to Dec-2028. Name this object "tseries".

Use `?ts` to check documents.

```r
set.seed(100)
l = 120 # length of time series
k = 2
# generate a sequence storing values
t = 2* seq(from = 1, to = l, by = 1)  + rnorm(l, mean = 0, sd = 20)
# add time labels to value sequence and create ts object with name
tseries = ts(t, start = c(2019, 1), frequency = 12)
# print out your ts object
tseries
```

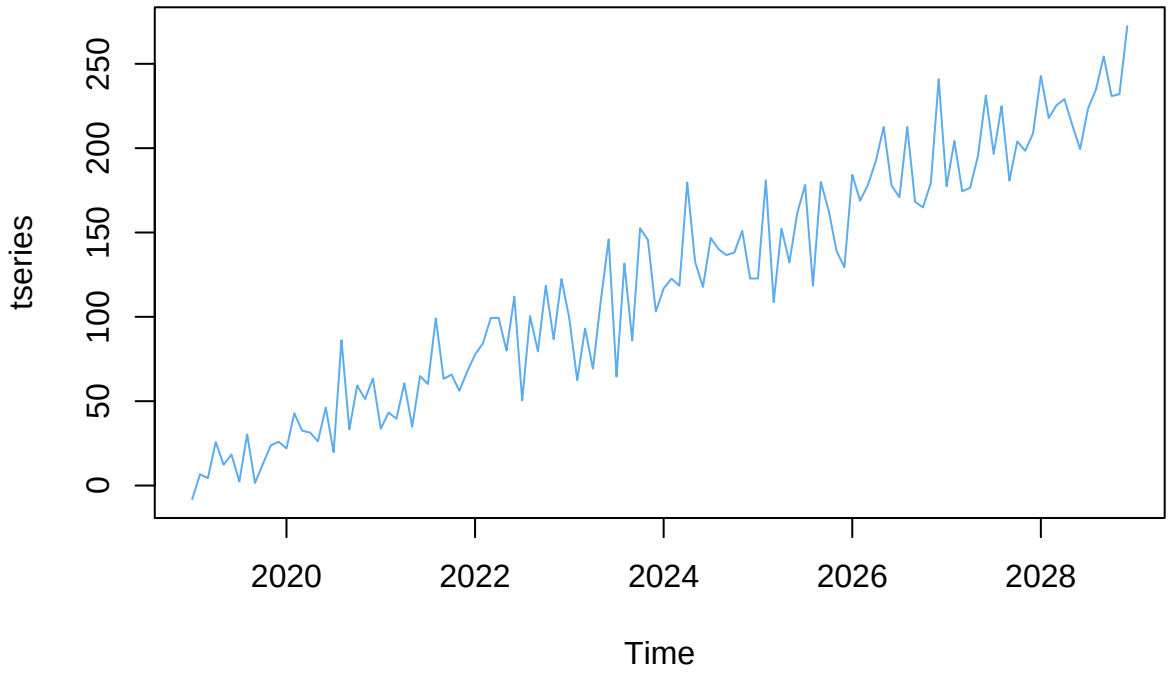
```
##             Jan        Feb        Mar        Apr        May        Jun
## 2019  -8.043847   6.630623   4.421658  25.735696  12.339425  18.372602
## 2020  21.967321  42.796810  32.467590  31.413666  26.222915  46.217125
## 2021  33.712418  43.230989  39.595569  60.618891  34.845411  64.941520
## 2022  77.658154  84.346466  99.308047  99.404040  79.967415 112.064070
## 2023  98.875581  62.426882  93.058756  69.228041 109.577297 145.949314
## 2024 116.760085 122.623119 118.422329 179.639179 132.596683 117.739500
## 2025 122.751614 180.970435 108.758080 152.254994 132.249433 161.410790
## 2026 184.191632 168.841899 178.327357 192.347242 212.543515 177.924594
## 2027 177.350084 204.270397 174.426337 176.519305 195.341533 231.262274
## 2028 242.802029 217.851324 225.451870 229.092025 213.709323 199.415698
##             Jul        Aug        Sep        Oct        Nov        Dec
## 2019   2.364186  30.290654   1.494811  12.802757  23.797723  25.925489
## 2020  19.723716  86.205936  33.238200  59.281212  51.239226  63.468092
## 2021  60.177729  99.147512  63.241408  65.776130  56.199714  67.564115
## 2022  50.464487 100.457348  79.554333 118.444619  86.731193 122.381315
## 2023  64.561490 131.609283  86.023488 152.497448 145.625975 103.222962
## 2024 146.759885 140.033832 136.601661 138.150202 150.978065 122.712887
## 2025 178.169037 118.511905 179.936445 163.000085 139.093014 129.375769
## 2026 170.857554 212.566029 168.140852 164.848575 179.394071 240.913655
## 2027 196.617053 224.857513 180.840126 203.993882 198.471654 208.614070
## 2028 223.380491 234.567721 254.362400 230.888526 231.949180 272.303814
```

Q: What does the `rnorm(120, mean = 0, sd = 20)` represent? in the code above?

A: That represents random noise.

Plot the ts object `tseries` you created using `plot.ts` or `plot`. The base R `plot` function recognizes when the first argument is a ts object and actually calls the `plot.ts` function under the hood.

```
plot(tseries, col='steelblue2', lty=1)
```



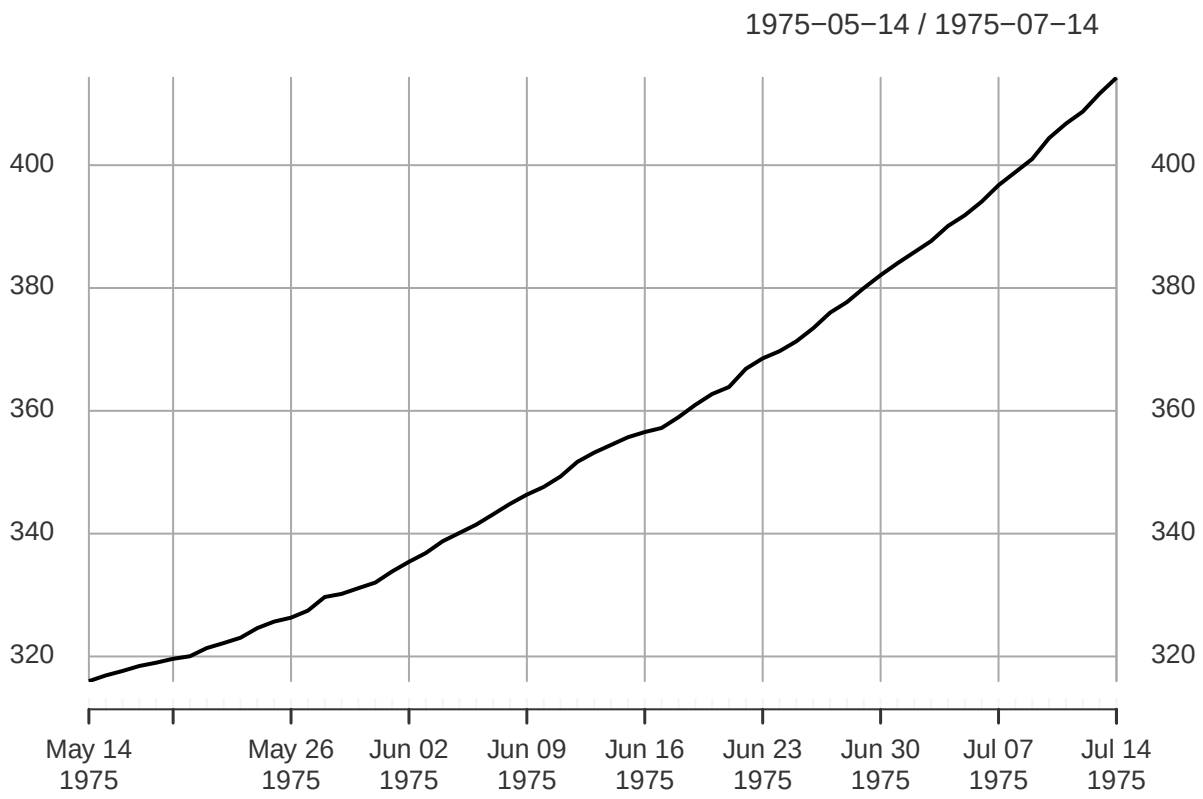
Q: What characteristics do you notice about this plot? Describe its mean and variance. And what happens if you change the k factor in the equation above from 2 to 20? Likewise, if you change the sd from 20 to 2?

A: Its mean is changing (upward positive change) and the variance is fairly constant. If you increase the k factor, the variance drops, likewise if you decrease the standard deviation.

Q: What about the data we plotted earlier? Can they be read in or converted into ts objects? They can but it's better to use the ***xts*** *function. Try for yourself. Run* ***?xts*** *and read its documentation. Try converting any dataset read in previously and then print and plot the ts object. This will be a bit more complicated and will take more trial and error. Hint: the best approach may be to actually read each column in individually into the* ***xts*** *function. And if you get an error that says "order.by requires an appropriate time-based object" that means that your time column is not a date object - so be sure to convert that too!*

A:

```
xts.sample = xts(CO2_yr$mean_CO2_ppm, as.Date(CO2_yr$year))
plot(xts.sample, main=" ")
```



Now that you've got some practice making ts and xts obects, let's check for stationarity in the data we've visualized earlier.

2. check stationarity using ACF plots

To check for stationarity, you can generate an **auto-correlation function (ACF) plot**. But what does that mean? One of the major observations notable to a time series is that the data points are not independent - an event that happened earlier at time $t-1$ is dependent on an event at time t. In turn, the ACF plot identifies temporal dependence in the data. Autocorrelation measures the linear relationship between the lagged values of a time series.

$R_s = Corr(x_t, x_{t+s})$ for lag s.

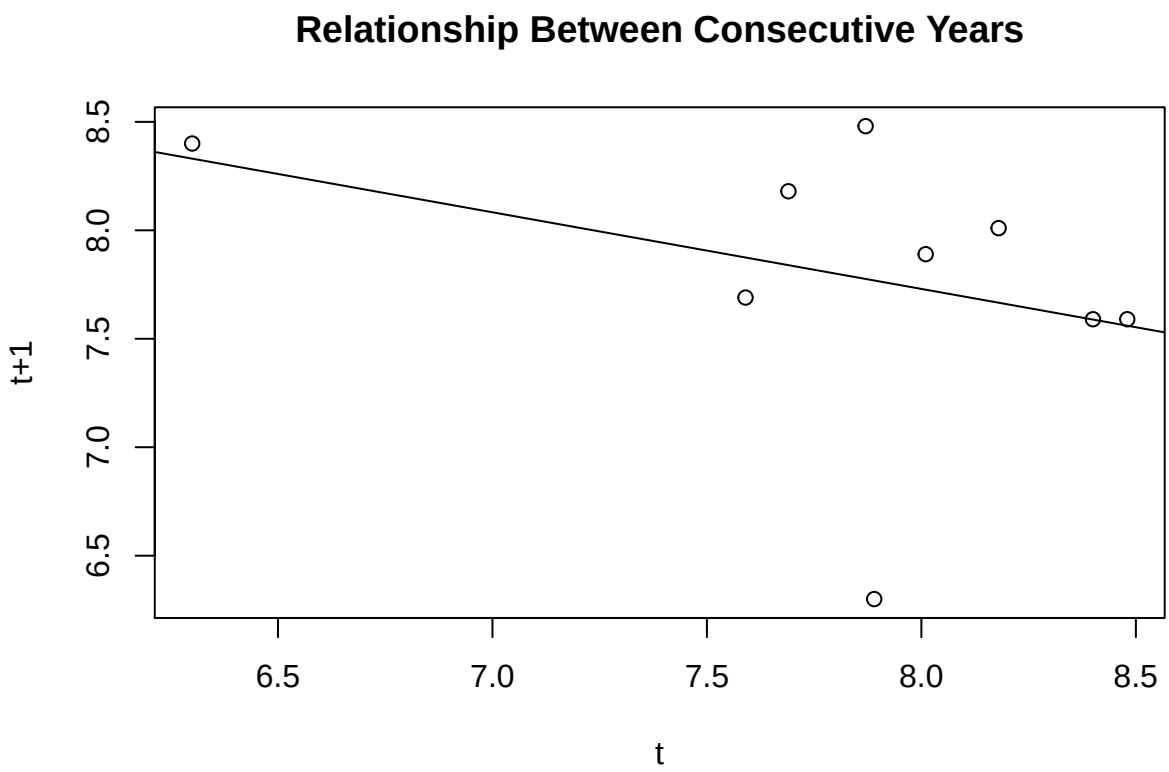
So how can we measure that dependency? Using correlations between time $t-1$ and time t. A quick way to anticipate how strong the correlations might be is plotting consecutive time points. Let's do that using the soapberry bug wing data first:

```
v = wing_length$wing_length_avg # extract the variable of interest

d = cbind(v[1:length(v)-1], # create a dataset with a column of t values and t+1 values
          v[2:length(v)])
colnames(d) = c("t", "t+1")
print(cor(d))
```

```
##              t        t+1
## t    1.0000000 -0.3509409
## t+1 -0.3509409  1.0000000
```

```
plot(d, ylab="t+1",
     main="Relationship Between Consecutive Years")
m = lm(`t+1` ~ t, data=as.data.frame(d)) # not significant
```

```
abline(m)
```

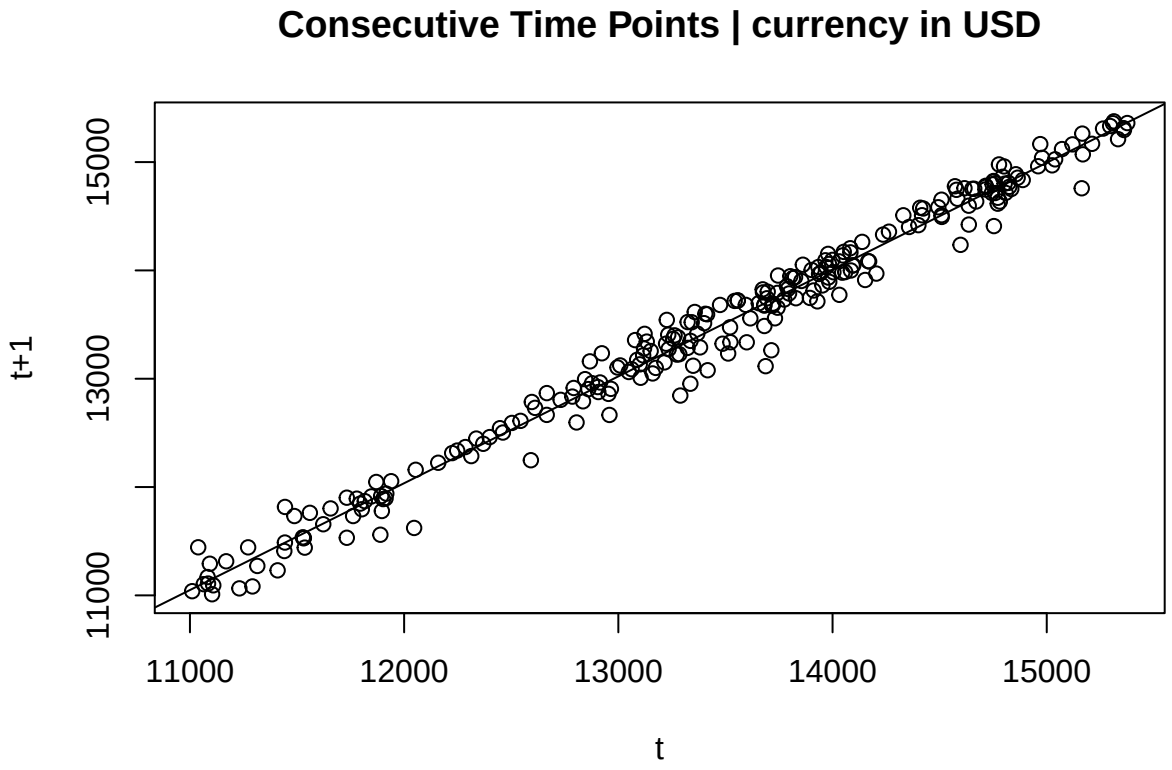


The correlation of -0.35 between consecutive years is not that strong, so let's consider another dataset. But before we do that, let's functionalize this visualization process:

```
plot_consecutive_time_points = function(v, var){
  # v is the variable given as a vector
  # var is the variable name given as a character
  d = cbind(v[1:length(v)-1],
            v[2:length(v)])
  colnames(d) = c("t", "t1")
  print(cor(d))
  plot(d, ylab="t+1",
       main=paste("Consecutive Time Points |", var))
  m = lm(t1 ~ t, data=as.data.frame(d))
  print(summary(m))
  abline(m)
}
```

```
plot_consecutive_time_points(nasdaq$Open, "currency in USD")
```

```
##            t        t1
## t  1.0000000 0.9913609
## t1 0.9913609 1.0000000
```



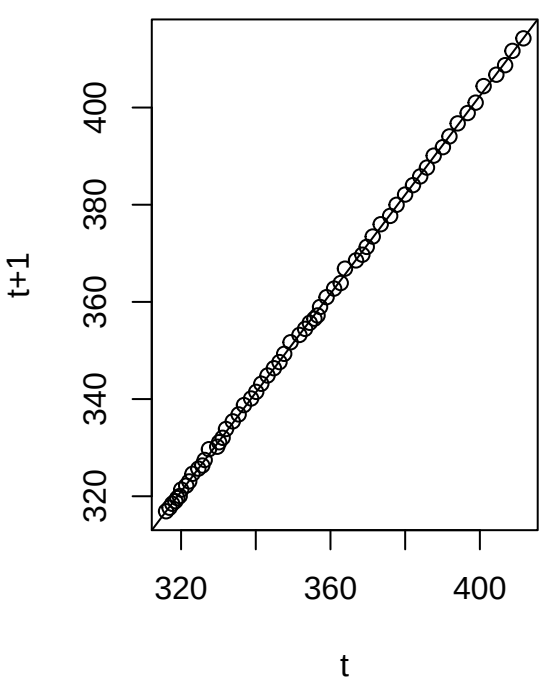

```
##
## Call:
## lm(formula = t1 ~ t, data = as.data.frame(d))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -583.91  -67.94   23.08   91.67  356.50
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 2.005e+02  1.118e+02   1.793   0.0742 .
## t           9.862e-01  8.286e-03 119.027   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 149.1 on 248 degrees of freedom
## Multiple R-squared:  0.9828, Adjusted R-squared:  0.9827
## F-statistic: 1.417e+04 on 1 and 248 DF,  p-value: < 2.2e-16
```

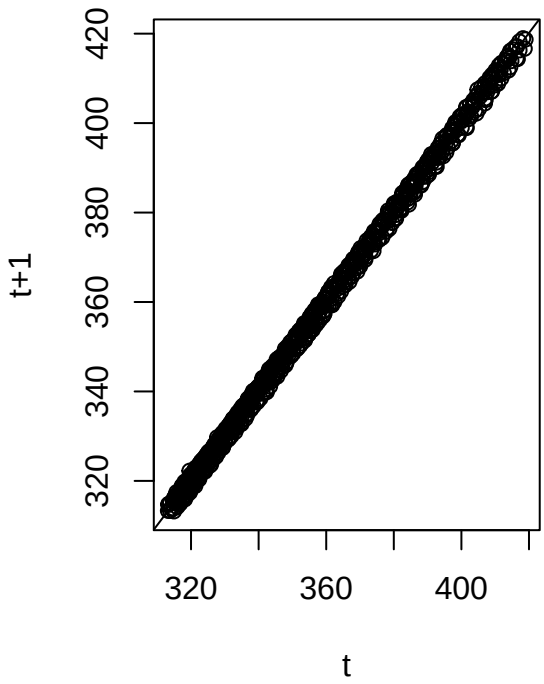
This is much stronger!

Q: Now you try on the other datasets and report what you find.

A:

```
# this is open ended but here are the CO2 concentration examples
par(mfrow=c(1,2))
plot_consecutive_time_points(CO2_yr$mean_CO2_ppm, "[CO2] ppm")
plot_consecutive_time_points(CO2_mon_filtered$CO2_ppm1, "[CO2] ppm")
```



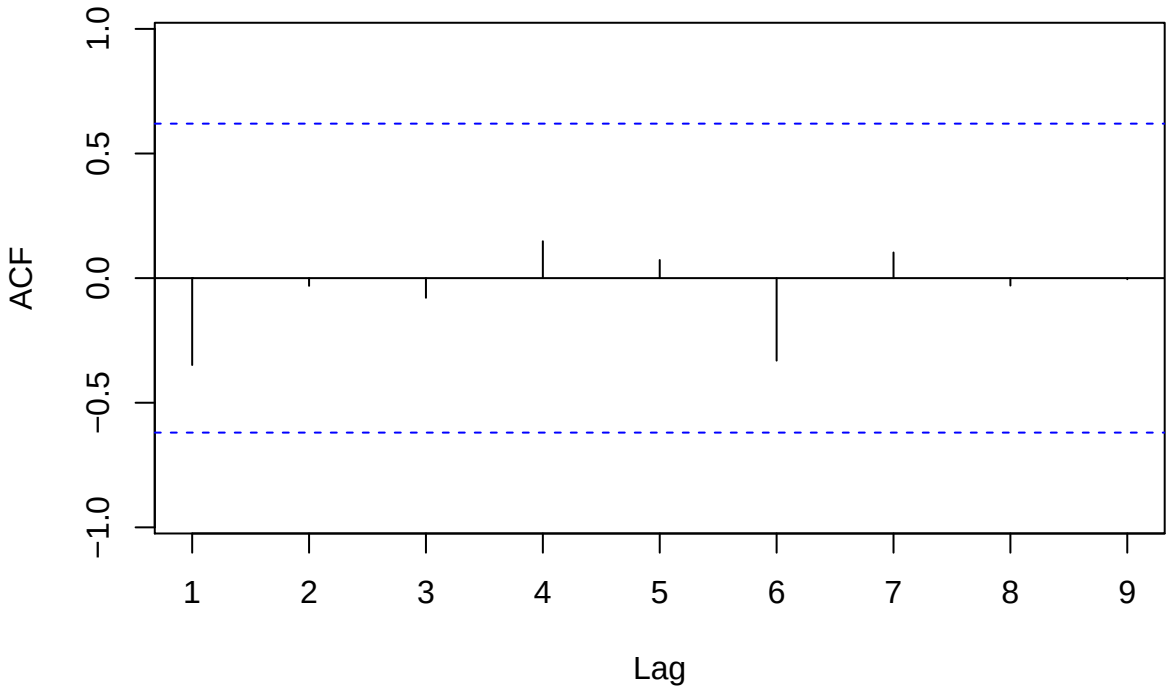



```
##            t        t1
## t  1.0000000 0.9998794
## t1 0.9998794 1.0000000
##
## Call:
## lm(formula = t1 ~ t, data = as.data.frame(d))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.96040 -0.26896 -0.03427  0.27728  1.19924
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -4.701517   0.733755  -6.407  2.7e-08 ***
## t            1.017759   0.002058 494.554  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4522 on 59 degrees of freedom
## Multiple R-squared:  0.9998, Adjusted R-squared:  0.9998
## F-statistic: 2.446e+05 on 1 and 59 DF,  p-value: < 2.2e-16
##
##            t        t1
## t  1.0000000 0.9990892
## t1 0.9990892 1.0000000
##
## Call:
## lm(formula = t1 ~ t, data = as.data.frame(d))
##
## Residuals:
```

```
##     Min      1Q  Median      3Q     Max
## -2.7571 -1.1367  0.4262  1.0168  2.5631
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.0007009  0.5565249  -0.001    0.999
## t            1.0003680  0.0015559 642.934   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.259 on 754 degrees of freedom
## Multiple R-squared:  0.9982, Adjusted R-squared:  0.9982
## F-statistic: 4.134e+05 on 1 and 754 DF,  p-value: < 2.2e-16
```

Now that you've got a hang of these consecutive time point plots, let's run ACF plots first on the soapberry wing data and then on the NASDAQ data. To do so, use the `Acf` function on the variable of interest:

```
Acf(wing_length$wing_length_avg, lag.max = 15, main="")
```



How do we read this plot? The black vertical lines, or spikes, measure the correlation between lags.But how do we know if these correlations are significant? Well, that's what the two horizontal, blue, dashed lines are for. Those represent the significance threshold, where only the spikes that exceed those dashed lines are considered significant. We expect 95% of the spikes in the ACF to lie within $\pm 2/\sqrt{(T)}$ where T is the length of the time series. In this case, the acf plots below the thresholds will be equal to

```
+ 2/sqrt(length(wing_length$wing_length_avg))
```

```
## [1] 0.6324555
```

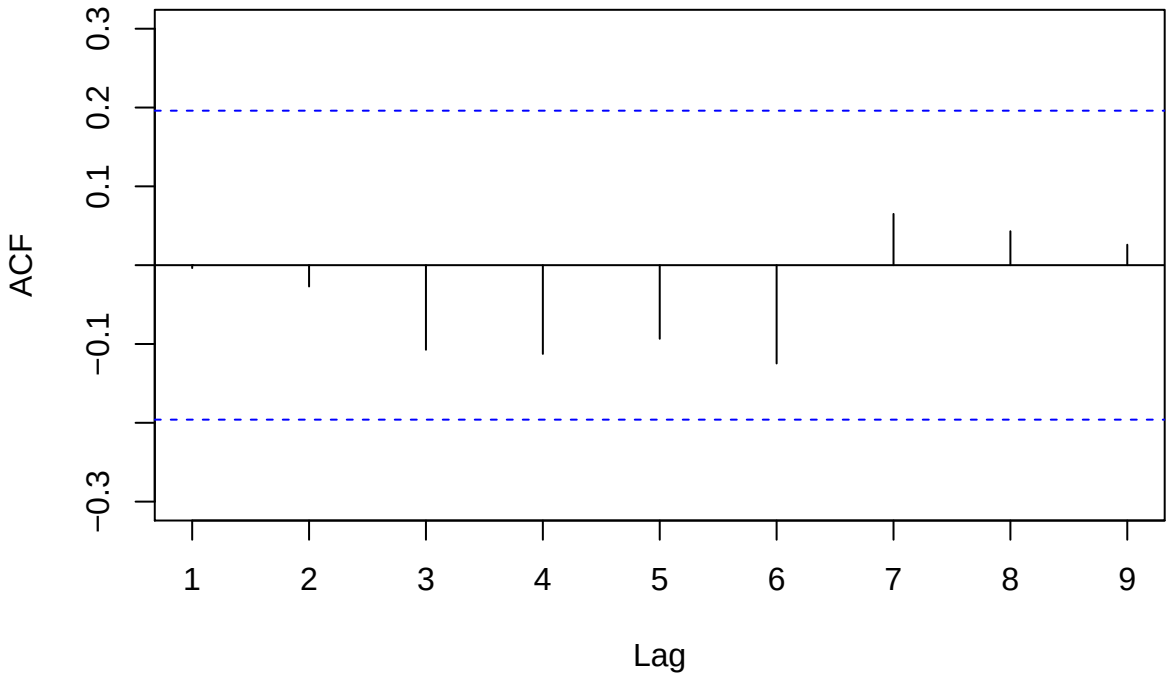
```
- 2/sqrt(length(wing_length$wing_length_avg))
```

```
## [1] -0.6324555
```

If one or more large spikes are outside these bounds, or if substantially more than 5% of spikes are outside these bounds, then the series is probably not white noise, meaning there is some pattern. What would the

ACF look like from purely simulated white noise data? Just like this:

```
set.seed(1)
m = 100 # length of time series
x.wn = rnorm(m, 0, 1)
Acf(x.wn, lag.max = 9, main='') # lag up to 9
```

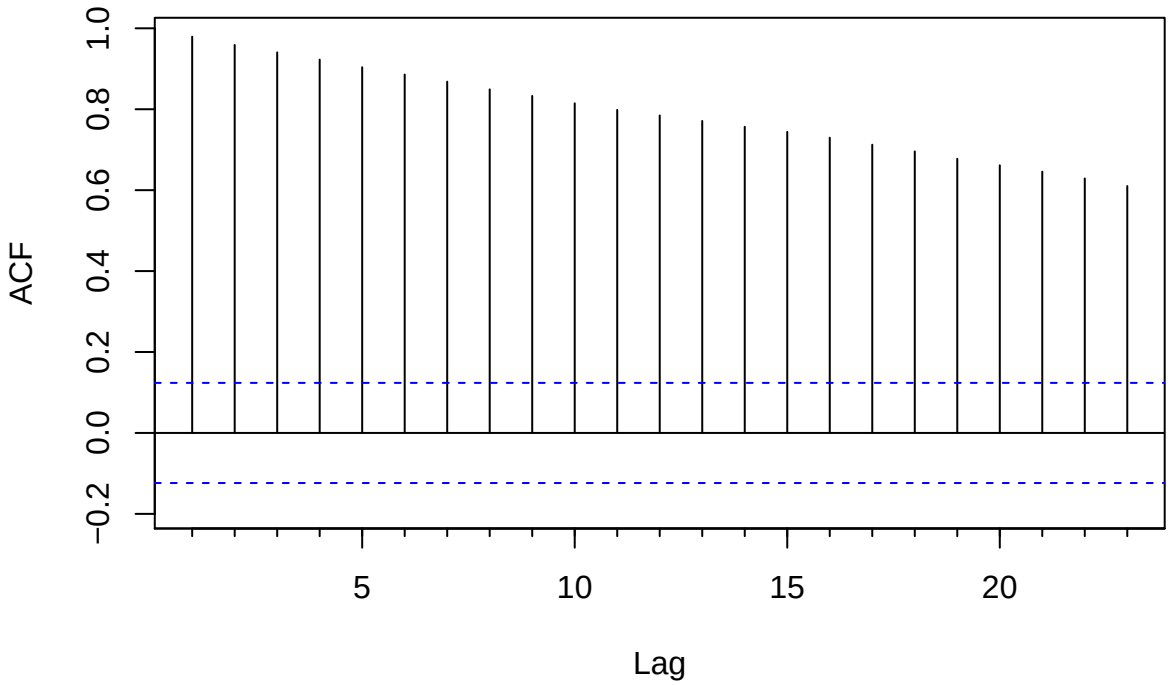


So you can see that this ACF is not much different from the soapberry bug average wing length data.

Thus, in summary, these ACF plots describe how well the present value of the series is related with its past values. Since, none are significant here, then no present value of the time series can be obtained using previous values of the same time series. But we already had an intuition that would be the case from the consecutive time point plots.

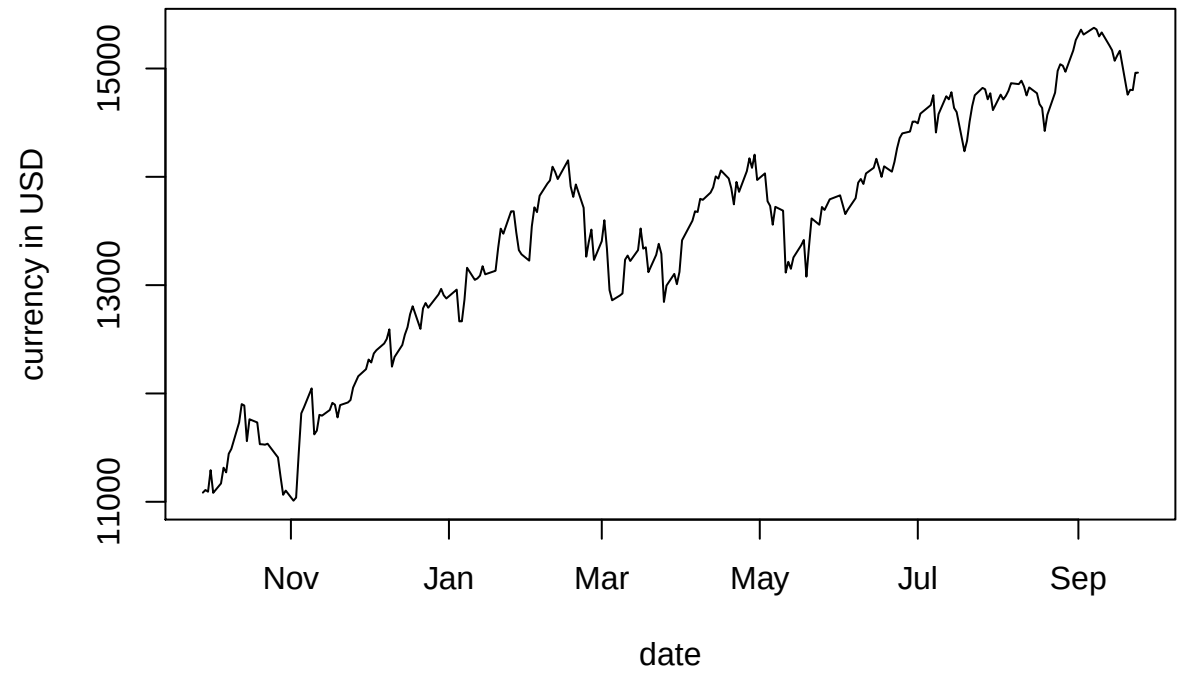
So let's move on to the NASDAQ data. *Go ahead and try plotting it yourself:*

```
Acf(nasdaq$Open, main="")
```



Wow! Look at that. All of the spikes are outside these bounds, making each significant. If we remember, time points were taken during the weekdays of each month. Let's plot it again, but this time not de-trended, just to see what's happening:

```
plot(nasdaq$Date, nasdaq$Open, type="l", xlab="date", ylab="currency in USD")
```



That's interesting, there seems to be a positive upwards trend. And that is reflected in the ACF plot because all the spikes are positive, so there is a positive changing mean. In addition, the slow decay of the autocorrelation function suggests the data follow a **long-memory process**. The duration of shocks is relatively persistent and influence the data several observations ahead, up to 23 to be exact. Thus, this is clearly reflects the smooth trending pattern in the data.

Before moving on, I also want to note that there are statistical tests that can further streamline the process of determining whether the data is stationary or not. One is the **Augmented Dickey-Fuller (ADF) test**. You can use the `adf.test` function and when you run it you'll notice spits out 3 lines of output:

```
adf.test(nasdaq$Open)
```

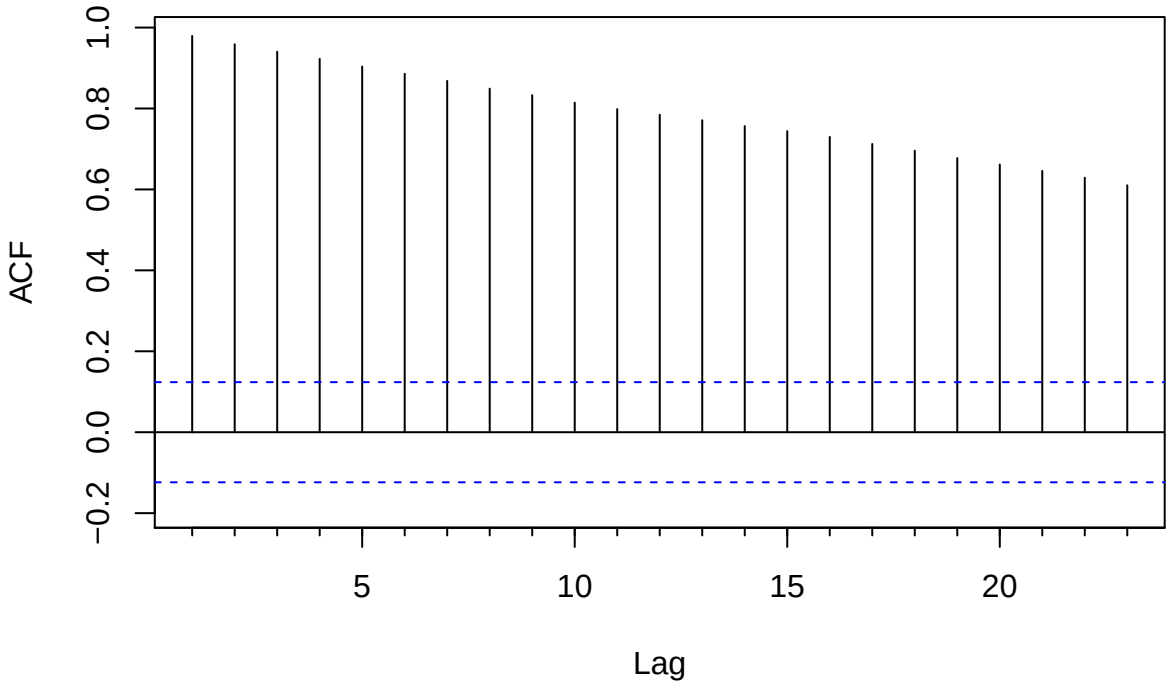
```
##
##  Augmented Dickey-Fuller Test
##
## data:  nasdaq$Open
## Dickey-Fuller = -2.7541, Lag order = 6, p-value = 0.2583
## alternative hypothesis: stationary
```

What's important to notice on the last line of output is that it states `alternative hypothesis: stationary`, meaning that the null hypothesis is non-stationary and since the p value is 0.26, the null hypothesis is *not* rejected. Thus, the data is non-stationary. So let's combine these plotting devices and tests into a common function and then move on:

```
check_stationarity = function(dx) {
  #  dx is time series data
  print(adf.test(dx))
  Acf(dx, main='') # ACF
}
```

```
check_stationarity(nasdaq$Open)
```

```
##
##  Augmented Dickey-Fuller Test
##
## data:  dx
## Dickey-Fuller = -2.7541, Lag order = 6, p-value = 0.2583
## alternative hypothesis: stationary
```

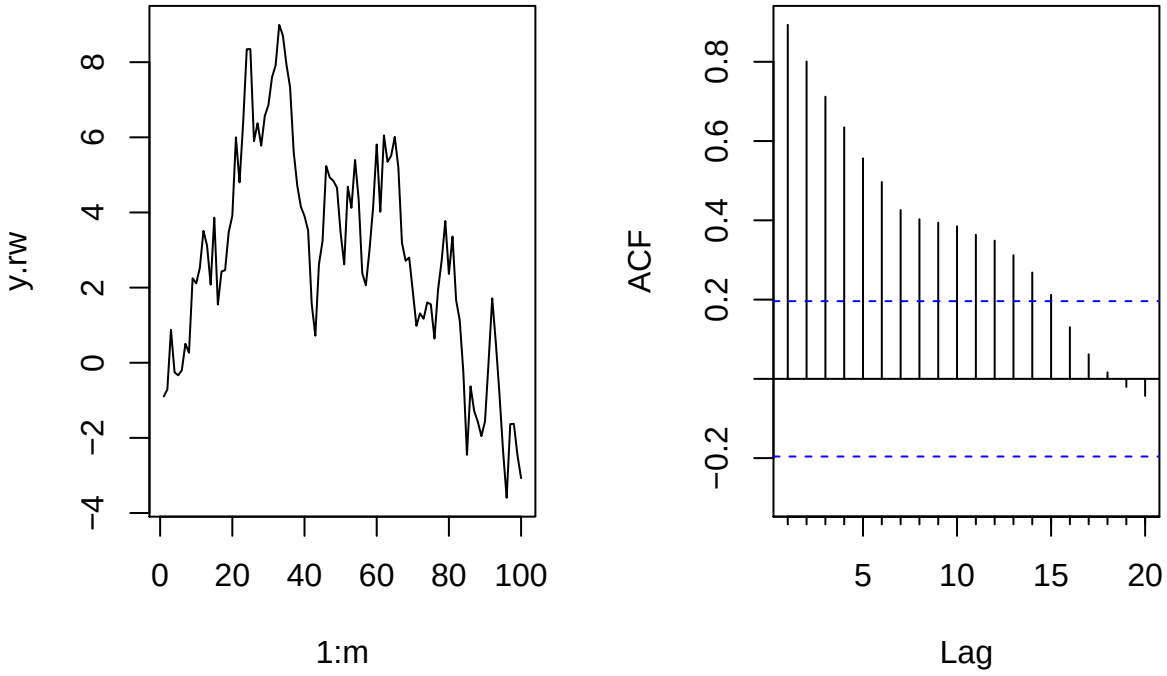


Now that you've observed stationarity and non-stationarity in real data. Let's wrap up this data one last simulation that can help you better categorize your data, aside from white noise vs. non-stationarity (trend

or drift), so you can have a wider intuition of potential time dependencies.

One last stochastic process that I haven't mentioned yet is the **random walk**. It is a random process, that follows a path that consists of a succession of random steps. Let's simulate it and plot it's ACF plot quickly:

```
set.seed(2)
m = 100 # number of time points

random_walk = function(sd=1, m=100) cumsum(rnorm(m, 0, sd))
y.rw = random_walk(1, m)

par(mfrow=c(1,2))
plot(1:m, y.rw, type="l")
Acf(y.rw, main='')
```



First, we construct a random walk function that simulates the random walk model. It takes the standard deviation (sd) and the max length of the time series (m). The function use `rnorm()` to generate random normal variable, and then use `cumsum()` to get the random walk. This is different than a vector of random numbers because the process used to generate the series forces dependence from one-time step to the next. This dependence provides some consistency from step-to-step rather than the large jumps that a series of independent, random numbers provides.

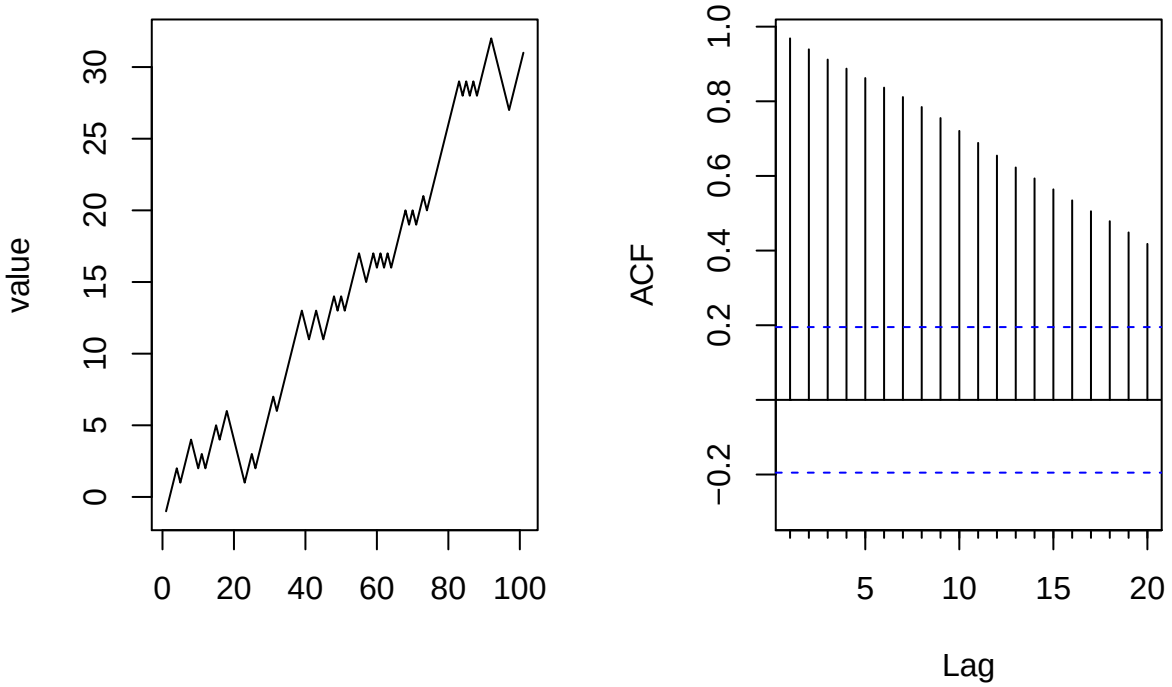
Given the way that this random walk is constructed, the ACF plot shows that the data are not stationary because the random walk is dependent on time up to 15 lags. This is not surprising because all random walk processes are non-stationary.

Go ahead and try it yourself. Try to recreate this random walk process and the plot its ACF plot:

1. Start with a random number of either -1 or 1.
2. Randomly select a -1 or 1 and add it to the observation from the previous time step. (Hint: this will require a for loop and some thoughtful indexing!)
3. Repeat step 2 for as long as you like.

```
# this problem is intense! but rewarding:
set.seed(1)
random_walk = c(-1)
```

```r
for (i in 1:100){
  rvar = rnorm(1, mean=0, sd=1)
  if (rvar < 0.5) {
    movement = 1
    random_walk[i + 1] = random_walk[i] + movement
  }
  else {
    movement = -1
    random_walk[i + 1] = random_walk[i] + movement
  }
}

par(mfrow=c(1,2))
plot(random_walk, type="l", xlab="", ylab="value")
Acf(random_walk, main="")
```



Now that we've analyzed a series of stochastic processes to determine if they are stationary or not, let's revisit where we are in the time series analysis process. We have done 1 and 2, so let's move on to 3, 4, and 5:

1. create a ts and xts object and plot it
2. check stationarity using ACF plots **3. handle non-stationary time series (which requires model selection and model fitting) 4. model checking 5. forecasting**

3. handle non-stationary time series

Now that your data is telling you its non-stationary, what do you do? Intuitively, you'll want to try to model it so you can see if you can more precisely define the patterns you see and then try to forecast.

The way you do that is by **'regressing out'** the patterns occurring in the time series until you turn it into stationarity. There are several ways 'regress out' these patterns, for example

1. Differencing: differencing can be used to remove the linear trend because it is equivalent to applying a linear regression on time. If there is a non-linear trend, differencing (order > 1) also works

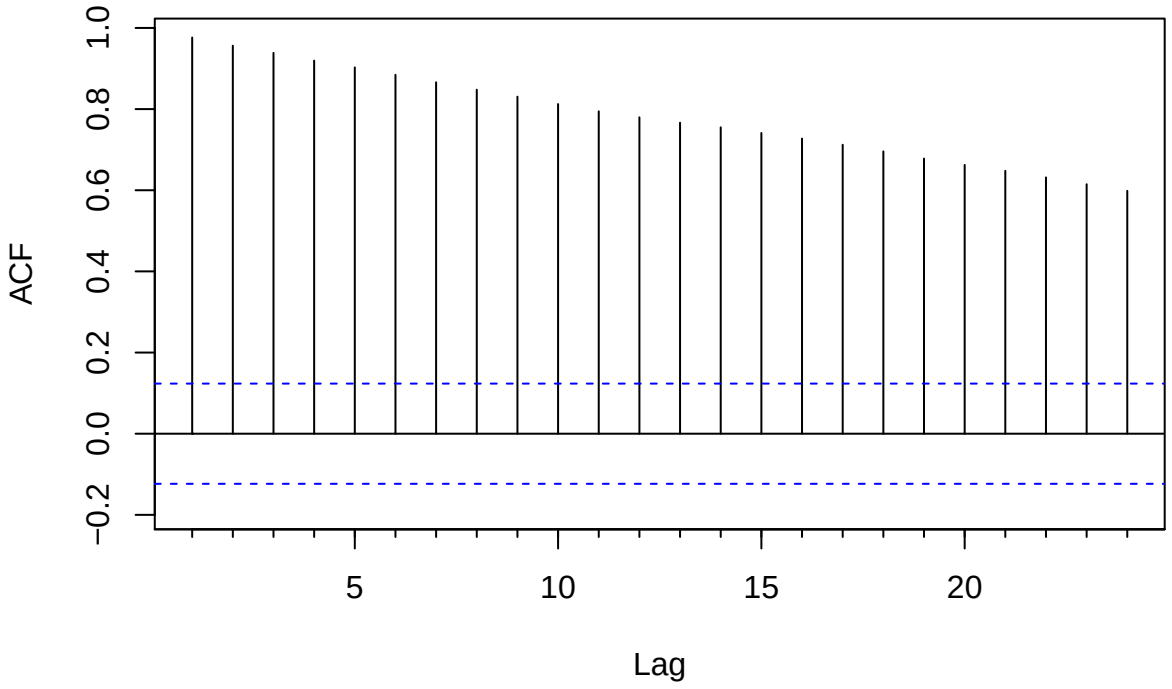
2. Detrending with regression: if you already have the trend shape in mind, e.g. $a_{t_2} + bt$, you can detrend using that equation.
3. Transformation: You can also transform the data in order to stablize variance or linearize the time trend. This can be done using log, square-root, inverse, inverse square-root, and Box-Cox transformations.

Then, once you've removed the non-linear time trend or confounding effect, you wnat to check stationarity again.

So let's go back to that NASDAQ data. I already was hinting that I did some detrending, but let's go step-by-step and follow through what I did now that you know how useful it is. So let's start clean, rereading the data and checking for stationarity

```
nasdaq = read.csv("NASDAQ.csv")
nasdaq$Date = as.Date(nasdaq$Date, "%Y-%m-%d")
check_stationarity(nasdaq$Open)
```

```
##
##  Augmented Dickey-Fuller Test
##
## data:  dx
## Dickey-Fuller = -2.8031, Lag order = 6, p-value = 0.2377
## alternative hypothesis: stationary
```



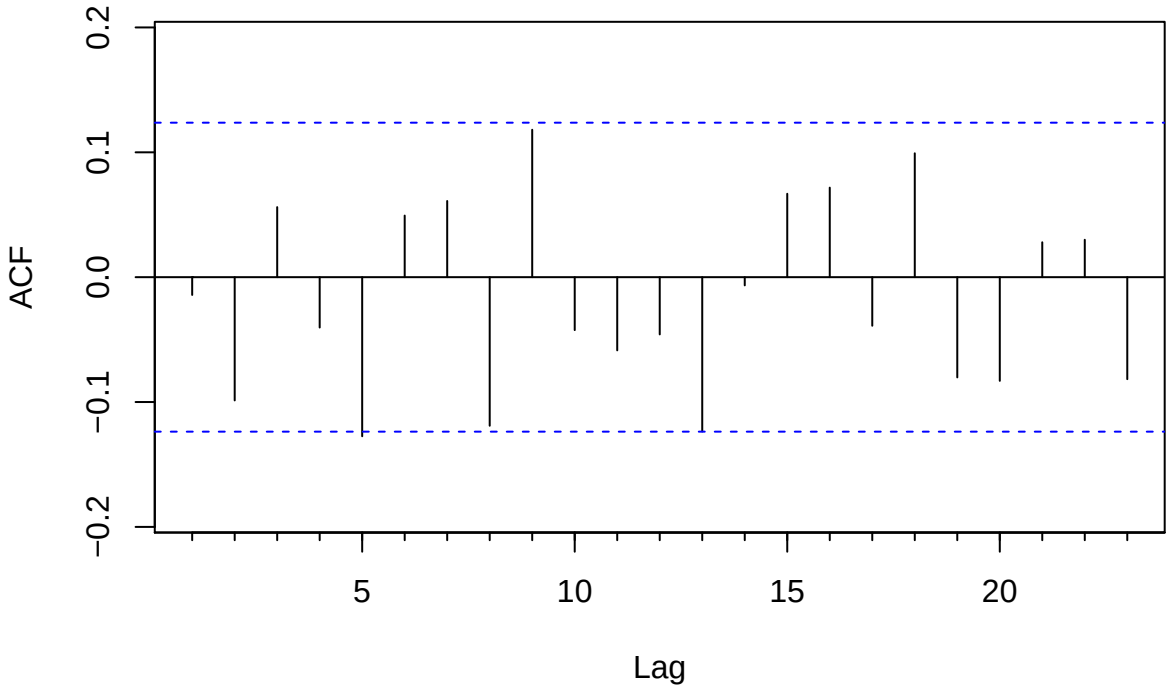
Now let's use the `diff` function to take the difference between data points, sequentially, and then check for stationarity

```
detrend = diff(nasdaq$Open)
nasdaq_dt = nasdaq[-1,]
nasdaq_dt$diff = detrend
check_stationarity(nasdaq_dt$dif)
```

```
## Warning in adf.test(dx): p-value smaller than printed p-value
```

```
##
##  Augmented Dickey-Fuller Test
```

```
##
## data:  dx
## Dickey-Fuller = -6.1416, Lag order = 6, p-value = 0.01
## alternative hypothesis: stationary
```



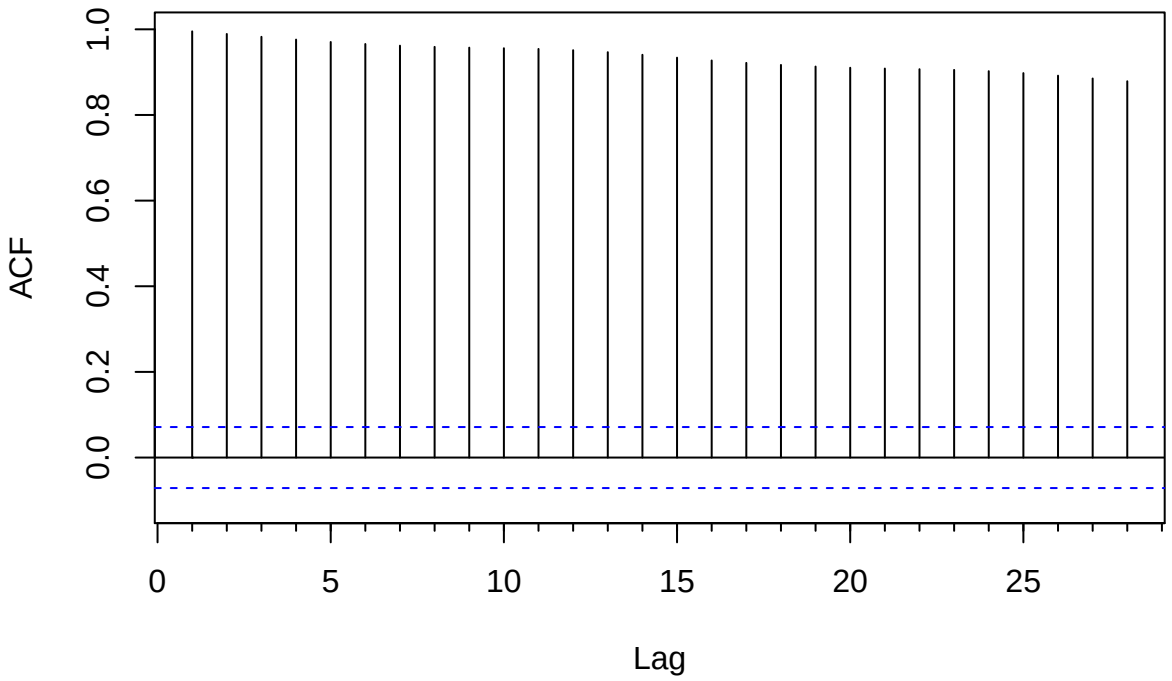
Great! Detrending clearly got us to stationarity, which implies that this NASDAQ data only has a significant positive trend. If we didn't get to stationarity, then we may still have some volatility that's significant and we would need to detrend for that as well. So then let's try a dataset that may raise that situation.

Let's go back to our climate data on CO2 concentrations monthly and check for stationarity.

```
check_stationarity(CO2_mon_filtered$CO2_ppm1)
```

```
## Warning in adf.test(dx): p-value greater than printed p-value
```

```
##
##  Augmented Dickey-Fuller Test
##
## data:  dx
## Dickey-Fuller = -0.27916, Lag order = 9, p-value = 0.99
## alternative hypothesis: stationary
```

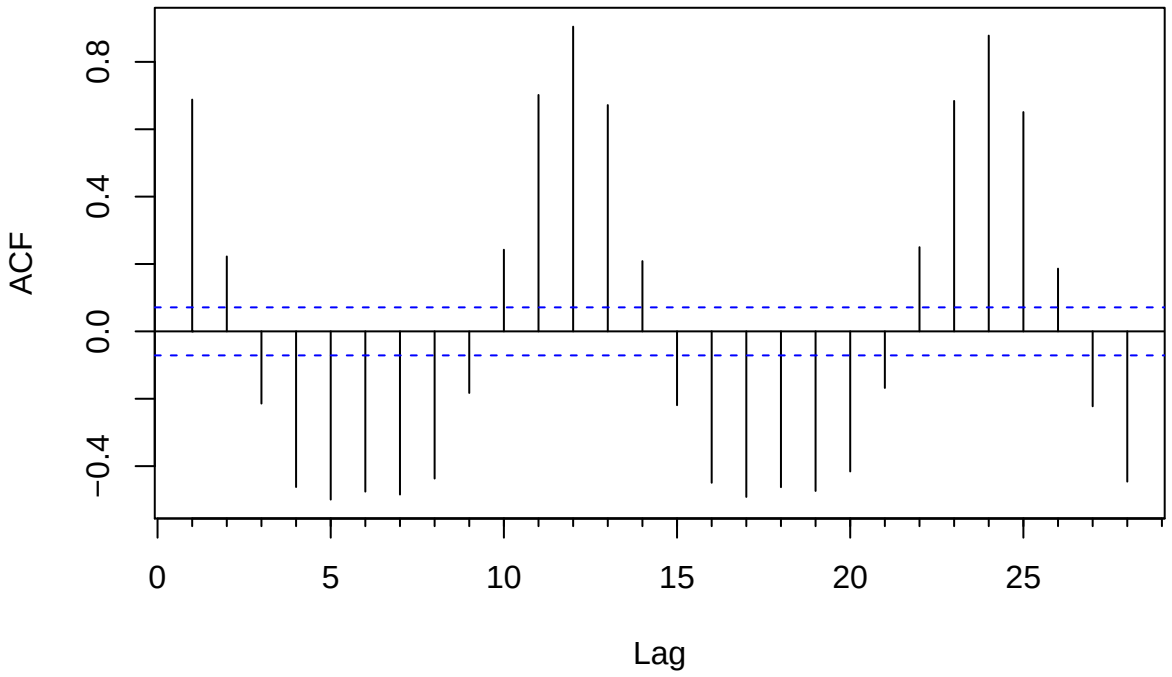


Ok great, this is clearly non-stationary, so let's start by detrending out using `diff`. **Go ahead and try that**.

```
detrend = diff(CO2_mon_filtered$CO2_ppm1)
CO2_mon_fdt = CO2_mon_filtered[-1,]
CO2_mon_fdt$diff = detrend
check_stationarity(CO2_mon_fdt$diff)
```

```
## Warning in adf.test(dx): p-value smaller than printed p-value
```

```
##
##  Augmented Dickey-Fuller Test
##
## data:  dx
## Dickey-Fuller = -30.444, Lag order = 9, p-value = 0.01
## alternative hypothesis: stationary
```



So it looks like the linear detrend was good enough to get rid of that really strong upwards trend and reach stationarity. But it seems like we have that seasonal trend still persisting. So now we need to **dedrift** (desesonalize). (Note: you can also fit a quadratic/polynomial function to the data and detrend that way, but let's move forward for now).

A simple way to correct for a seasonal component is to use differencing as well. If there is a seasonal component at the level of each season, then we can remove it on an observation today by subtracting the value from last season. In the case of the CO2 concentration dataset, it looks like we have a seasonal component each year showing swings from summer to winter. We can subtract the monthly minimum CO2 concentration from the same month last year to correct for seasonality. Then check for stationarity.

```
# make sure start at a year that has month 1
X = CO2_mon_filtered$CO2_ppm1[9:nrow(CO2_mon_filtered)]

deseasonalize = c()
months_in_yr = 12
for (i in months_in_yr:length(X)) {
  value = X[i] - X[i - months_in_yr]
  deseasonalize = c(deseasonalize, value)
}


length(CO2_mon_filtered$date[9:nrow(CO2_mon_filtered)])
```

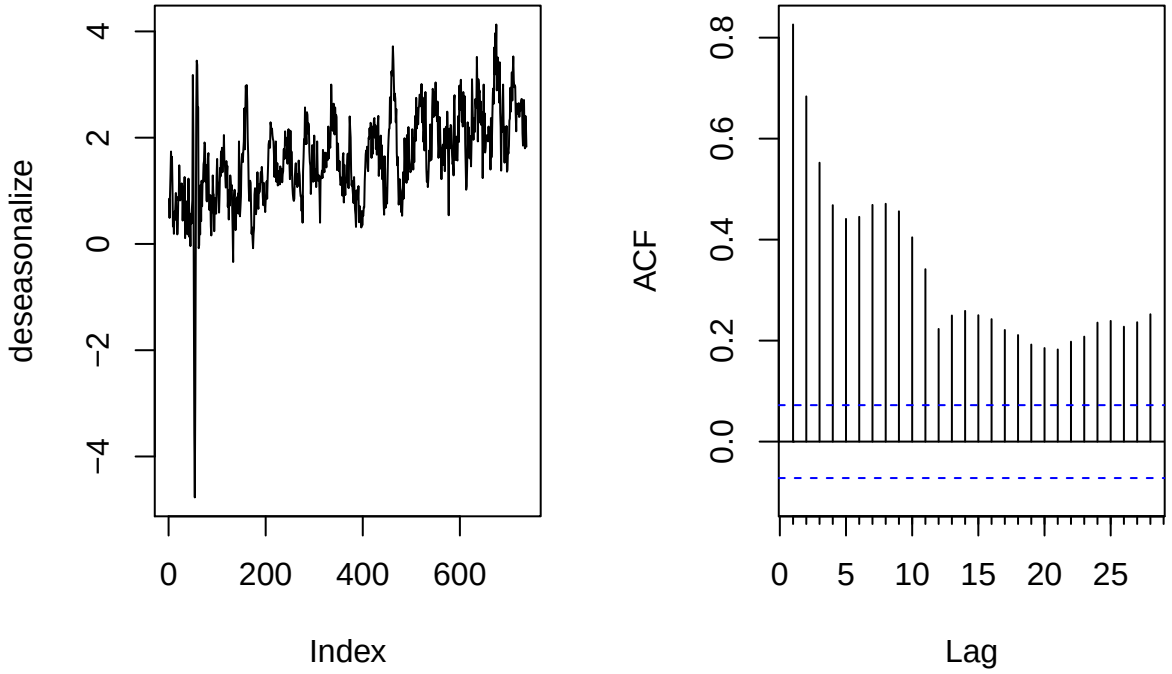
```
## [1] 749
```

```
par(mfrow=c(1,2))
plot(deseasonalize, type="l")
check_stationarity(deseasonalize)
```

```
## Warning in adf.test(dx): p-value smaller than printed p-value
```

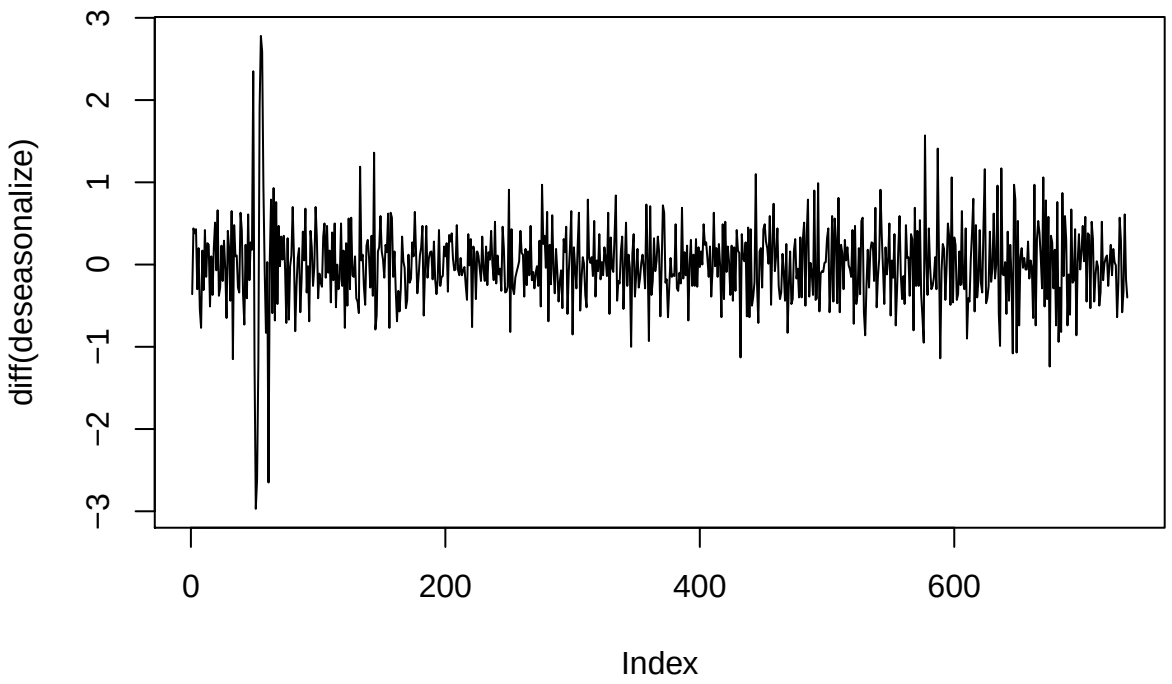
```
##
##  Augmented Dickey-Fuller Test
##
```

```
## data:  dx
## Dickey-Fuller = -6.7577, Lag order = 9, p-value = 0.01
## alternative hypothesis: stationary
```



That looks a lot better, although I'm not sure why there are outliers like that. We can then detrend like we did earlier on the deseaonalized data and get a truly stationary data, expect for a few spikes. *Go ahead and try that. Use* ***`diff`*** *on* ***`deseasonalize`*** *and plot it and check for stationarity.*

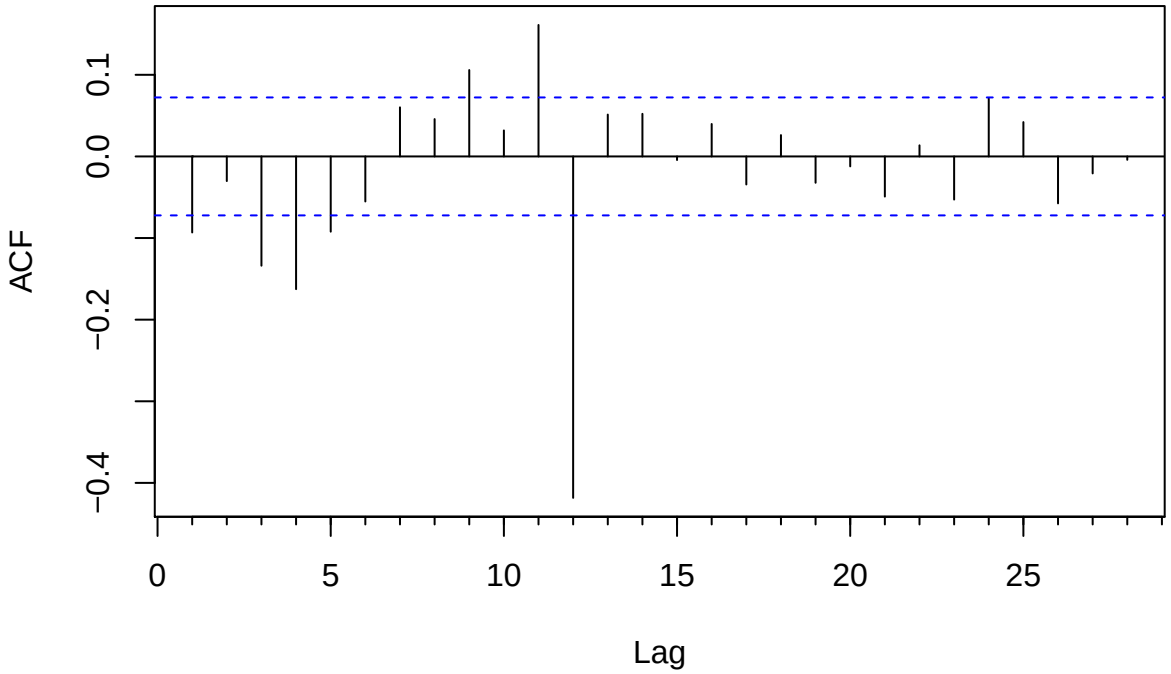
```
plot(diff(deseasonalize), type="l")
```




```
check_stationarity(diff(deseasonalize))
```

```
## Warning in adf.test(dx): p-value smaller than printed p-value
```

```
##
##  Augmented Dickey-Fuller Test
##
## data:  dx
## Dickey-Fuller = -10.704, Lag order = 9, p-value = 0.01
## alternative hypothesis: stationary
```



Now that we've learned how to detrend or dedrift the data, you might be wondering how to generally describe, in a model form, these autocorrelation function plots (this persisting "memory" in the lags) that we've see. So let's talk about that next:

4. model checking

Autoregressive (AR) models are the prototypical time series model. The AR(1) model combines data from "yesterday" and random noise to produce the next data point:

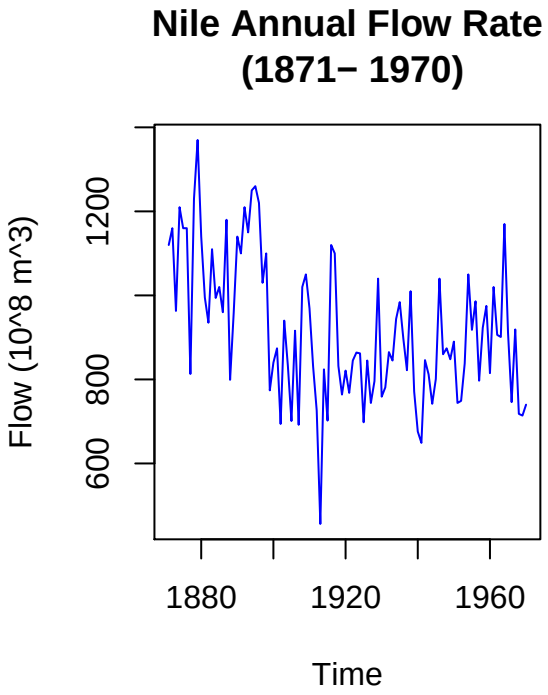
$y_t = \beta_0 + \beta_1 y_{t-1} + \epsilon_t$

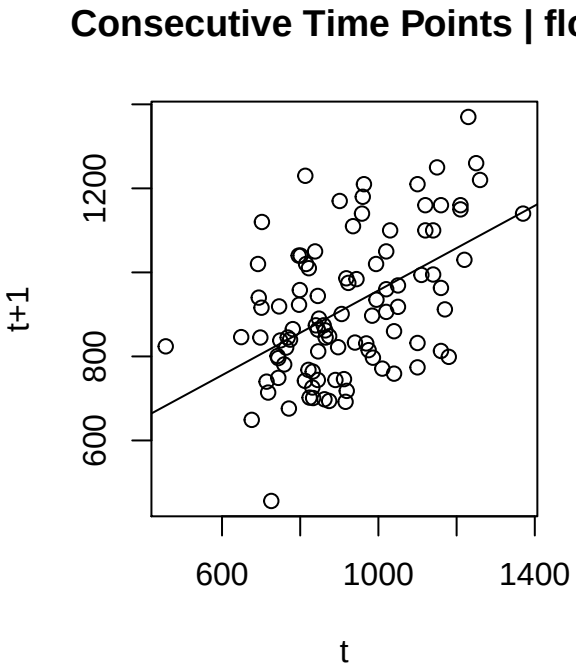
So let's work through a new dataset, the R base dataset `Nile`. Let's plot it and use our time series functions from before to test for stationarity and observe correlations between time points:

```
data(Nile)

par(mfrow=c(1,2))
plot(Nile,
     col="blue",
     xlab="Time",
     ylab= "Flow (10^8 m^3)", main="Nile Annual Flow Rate\n (1871- 1970)")

plot_consecutive_time_points(Nile, "flow")
```

```
##            t        t1
## t  1.0000000 0.5050531
```

```
## t1 0.5050531 1.0000000
```

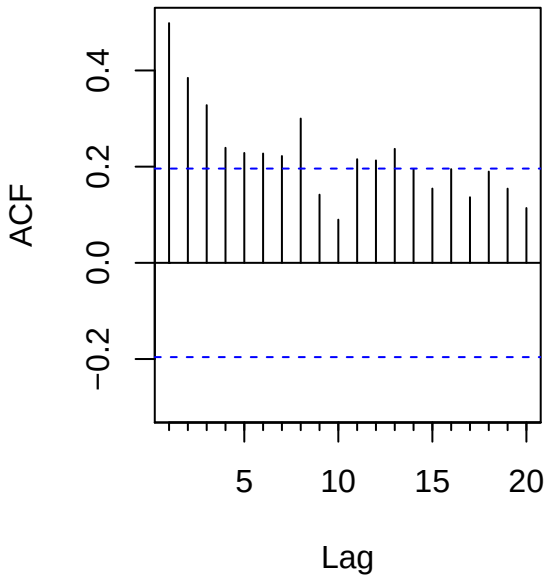





```
##
## Call:
## lm(formula = t1 ~ t, data = as.data.frame(d))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -362.90 -109.40  -13.30   99.65  367.22
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 452.76675   81.94024   5.526 2.76e-07 ***
## t             0.50432    0.08751   5.763 9.76e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 146.5 on 97 degrees of freedom
## Multiple R-squared:  0.2551, Adjusted R-squared:  0.2474
## F-statistic: 33.22 on 1 and 97 DF,  p-value: 9.76e-08
```

```
check_stationarity(Nile)
```

```
##
##  Augmented Dickey-Fuller Test
##
## data:  dx
## Dickey-Fuller = -3.3657, Lag order = 4, p-value = 0.0642
## alternative hypothesis: stationary
```



How would you describe the autocorrelations? (e.g., are they positive, how do they change as the lag increases, etc.)

They are positive, there seems to be some initial memory in the system and then the lags are insignificant.

And on how many lags do you think the current value of the flow in the Nile depends? Why?

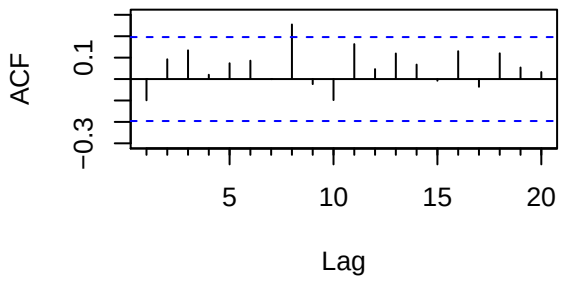
3 lags. . . an AR(3). First three or so lags have very high correlation from the previous lag. But the lags are significant until 8. It may be a number between 3 and 8 that best describes the model fit.

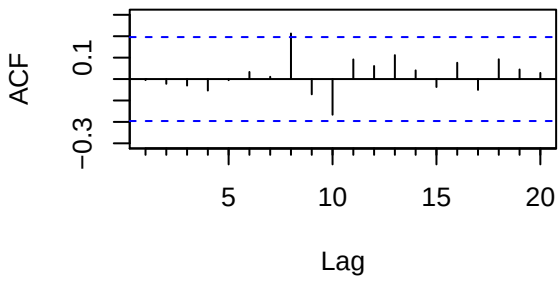
So it's these observations on the number of lags, or "memory", persisting in the system that describes an AR model. So we are going to fit a series of AR(p) models and check whether the model captures the dependence structure of the noise terms. Our final aim will then be to check that the residuals of the fitted model exhibits white noise because we do not want patterns in the residuals.

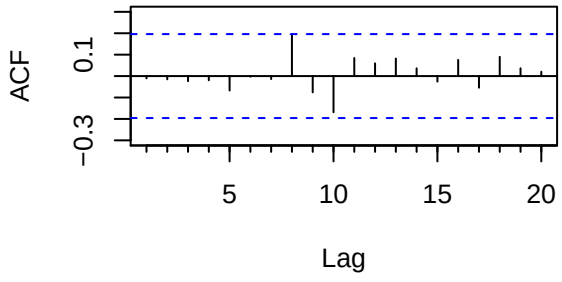
So let's start by identifying a model that appears to be most appropriate using the `arima` function to fit the model. The two key parameters we need are `x`, which is our dataset, and `order` which tells us how many lags to use. For now, only toggle the first parameter to choose an appropriate AR(p) model (i.e., you will have `order=c(x,0,0)`, where `x` is replaced by an appropriate number of lags for your model). (Also, `order=c(x,y,z)` refers to x as the AR order, y as the differencing, and z as the MA order).

I started the first model for you, now you try a couple more:

```
nile <- as.vector(Nile)

par(mfrow=c(2,2))
# Example AR(1) model
mod.ar1 <- arima(x=nile, order=c(1,0,0))
Acf(residuals(mod.ar1), main='')

# Now try fitting your own models and examine the Acf of the residuals
mod.ar3 <- arima(x=nile, order=c(3,0,0))
Acf(residuals(mod.ar3), main='')

mod.ar5 <- arima(x=nile, order=c(5,0,0))
Acf(residuals(mod.ar5), main='') # AR(5) looks okay! But is it too much?
```



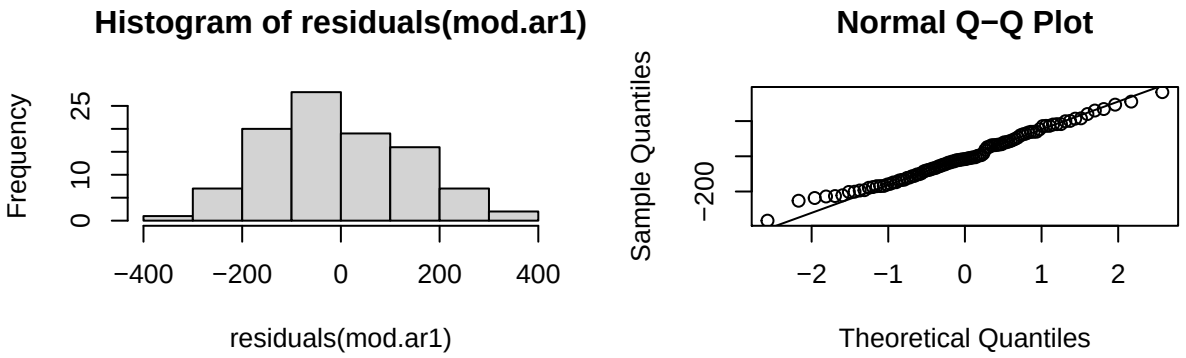




Now let's do some model diagnostics with the model that you chose. Consider, for example, a histogram of the residuals. **What do you expect it to look like?** (Hint: Suppose we assume that the white noise is normally distributed). Also, check out the QQ plot for the residuals. I wrote the code `mod.ar1` below as a reference:

A: I expect it to look like a normal distribution, since we assume that the white noise is normally distributed. We don't want patterns in our model's residuals.

```
par(mfrow=c(2,2))
# Make a histogram of the residuals from your model
hist(residuals(mod.ar1))

# Make a QQ plot for the residuals from your model
qqnorm(residuals(mod.ar1))
qqline(residuals(mod.ar1))

# Make a histogram of the residuals from your model
# [write your code here]

# Make a QQ plot for the residuals from your model
# [write your code here]
```



Great! The more normally distributed the residuals, the better the model. Let's quickly end with forecasting.

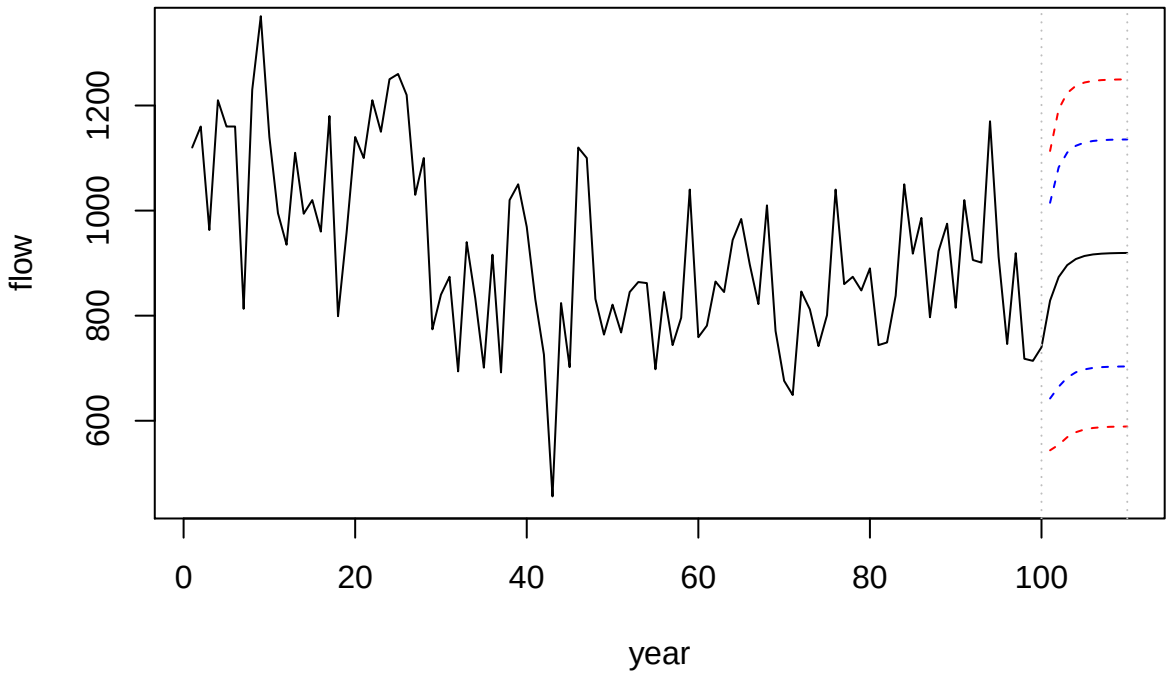
5. forecasting

Finally, let's end with some forecasting with the model that we chose. In any case where we generate a model, we do it with the intent of having predictive power. So let's see if we can predict the flow of the Nile River 10 period forward. The way to do that is to use the `forecast` function which takes in your model and the number of time points forward. Then, convert that forecast object into a data.frame object before merging it into one large total dataframe. Finally, plot the forecasted points and their 95% and 80% confidence intervals.

```
length(Nile)
```

```
## [1] 100
```

```
# Forecast data forward 10 periods
n_periods = 10
n_obs = length(Nile)
first_period = 1 + n_obs
last_period = n_obs + n_periods

forecasted.data <- data.frame(forecast(mod.ar1, h=n_periods))
total.data = data.frame(year = c(1:last_period),
                        flow = c(Nile, forecasted.data$Point.Forecast))

head(total.data)

plot(total.data, type="l", ylim=c(450, 1350))
lines(c(first_period:last_period), forecasted.data$Lo.95, col="red", type="l", lty=2)
lines(c(first_period:last_period), forecasted.data$Hi.95, col="red", type="l", lty=2)
lines(c(first_period:last_period), forecasted.data$Lo.80, col="blue", type="l", lty=2)
lines(c(first_period:last_period), forecasted.data$Hi.80, col="blue", type="l", lty=2)
abline(v=n_obs, col="grey", lty=3)
abline(v=last_period, col="grey", lty=3)
```

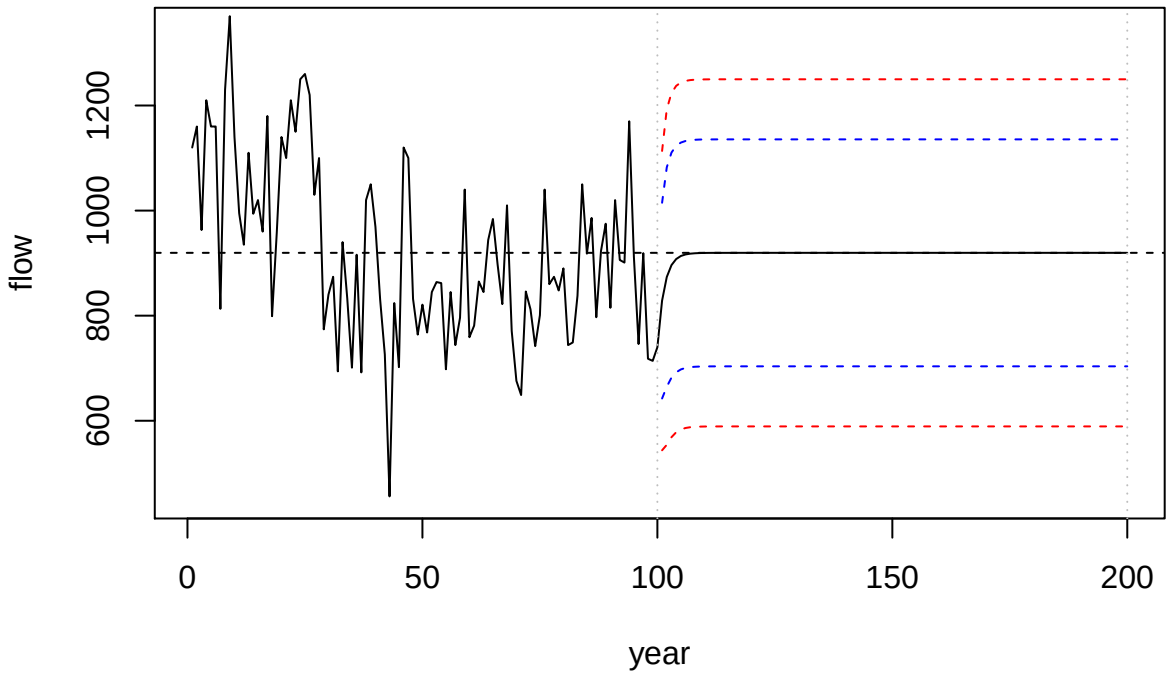



```
##   year flow
## 1    1 1120
## 2    2 1160
## 3    3  963
## 4    4 1210
## 5    5 1160
## 6    6 1160
```

Q: What value does the point forecast converge to if you forecast many quarters into the future?

A:

```
# Forecast data forward 10 periods
n_periods = 100
n_obs = length(Nile)
first_period = 1 + n_obs
last_period = n_obs + n_periods

forecasted.data <- data.frame(forecast(mod.ar1, h=n_periods))
total.data = data.frame(year = c(1:last_period),
                        flow = c(Nile, forecasted.data$Point.Forecast))

head(total.data)

plot(total.data, type="l", ylim=c(450, 1350))
lines(c(first_period:last_period), forecasted.data$Lo.95, col="red", type="l", lty=2)
lines(c(first_period:last_period), forecasted.data$Hi.95, col="red", type="l", lty=2)
lines(c(first_period:last_period), forecasted.data$Lo.80, col="blue", type="l", lty=2)
lines(c(first_period:last_period), forecasted.data$Hi.80, col="blue", type="l", lty=2)
abline(v=n_obs, col="grey", lty=3)
abline(v=last_period, col="grey", lty=3)
abline(h=total.data[nrow(total.data),2], lty=2)
```




```
total.data[nrow(total.data),2]
```

```
##   year flow
## 1    1 1120
## 2    2 1160
## 3    3  963
## 4    4 1210
## 5    5 1160
## 6    6 1160
## [1] 919.5685
```

It converges to 919.5685

*Q: And Are you satisfied with the model we fit? Would you choose another model? Why?**

A: Only for about 5 periods or so. It can't capture the fluctuations in the model after a certain point because it converges to 919.5685, so it would need a more dynamic model.

Summary

And that's it! In summary, time series modeling is a complex but fruitful analysis to apply in a variety of datasets. Time-dependencies can arise in various fashions in your models so consider this a gentle introduction to the sort of wide scope in which time can be considered, modele, and applied. Thank you for following and best of luck with your time series data!