Chapter 6: Trends and Time

- Describing trends in non-temporal data
- The many flavors of regression
- Plotting regression coefficients
- Working with temporal data
- Calculate a mean or correlation in circular time (clock time)
- Plotting time-series data
- Detecting autocorrelation
- Plotting monthly patterns
- Plotting seasonal adjustment on the fly
- Decomposing time series into components
- Using spectral analysis to identify periodicity
- Plotting survival curves
- Evaluating quality with control charts
- Identifying possible breakpoints in a time series
- Exploring relationships between time series: cross-correlation
- Basic forecasting

Trends are of fundamental interest throughout business intelligence work, but can be tricky to deal with statistically. And dealing with temporal data presents its own set of unique challenges. R lessens that burden considerably—as long as your data are in the right form.

Describing trends in non-temporal data

If you're in the data exploration phase, and want to get a loose sense of the trend in the data, starting with a loess curve is probably the best first step. Quantile regression provides tools for evaluating trends in data with increasing or decreasing variance, or any data with a monotonic trend regardless of variance. Ordinary least squares is probably the *last* tool you want to use in the exploratory phase, though it can sometimes be what you start with when doing model building.

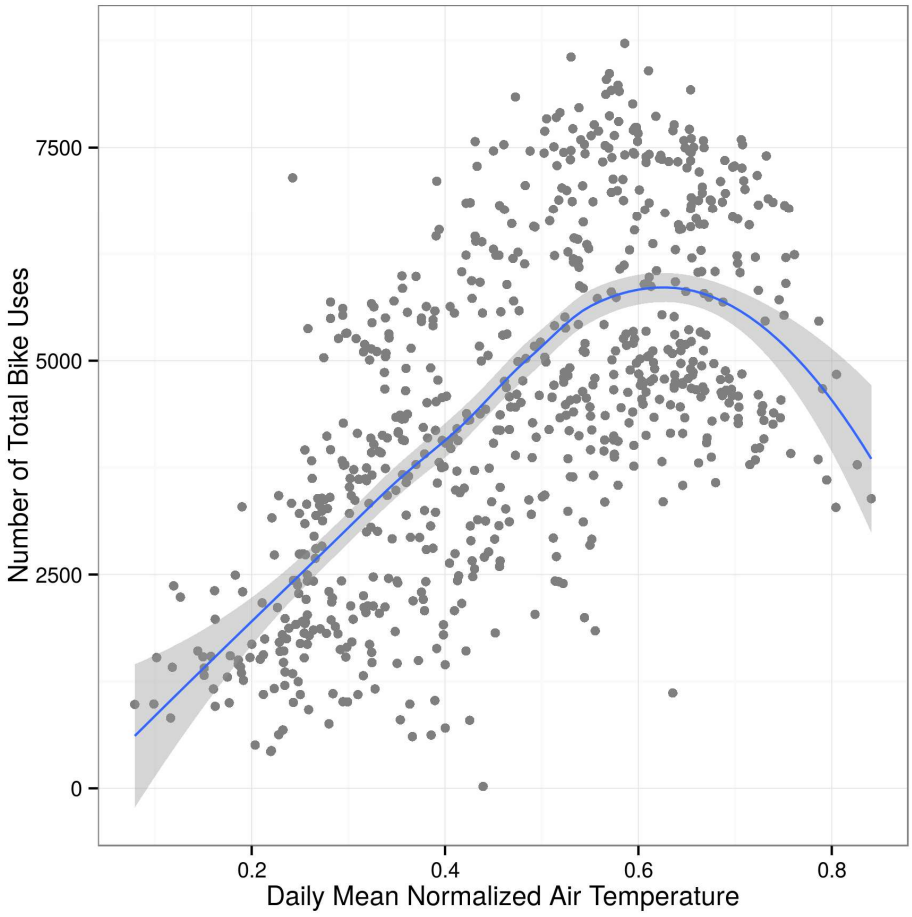
We'll continue to use the bike share data, and plot the results with `ggplot2`. We'll also need the `quantreg` package in order to plot the quantile regression trend lines, and the `mgcv` package to do the same with GAMs:

```
require(ggplot2)
require(quantreg)
require(mgcv)
```

Smoothed trends

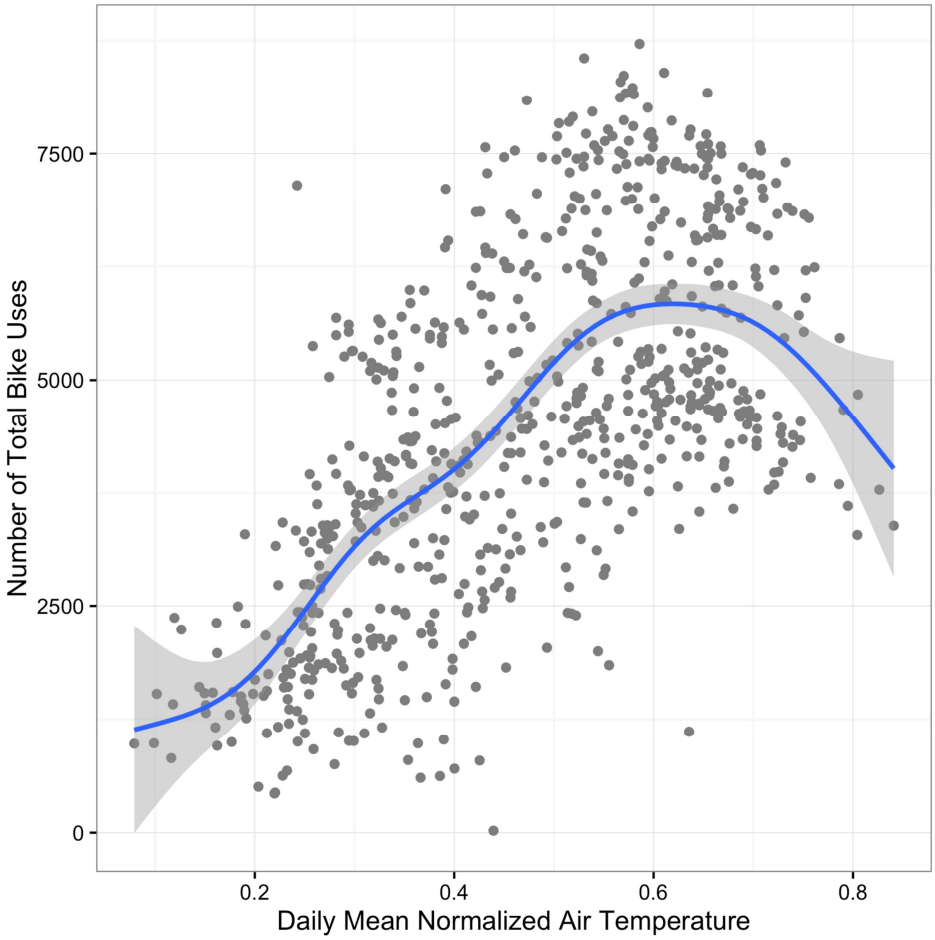
To see a smoothed average trend, use `method="loess"` in the `geom_smooth` function:

```
ggplot(bike_share_daily, aes(x=atemp, y=cnt)) +
  xlab("Daily Mean Normalized Air Temperature") +
  ylab("Number of Total Bike Uses") +
  geom_point(col="gray50") +
  geom_smooth(method="loess") +
  theme_bw()
```



ggplot2 automatically smooths via a GAM if you have > 1,000 points. If you have fewer, and want to see the trend a GAM can plot, you'll need to change the the geom_smooth method *and* include the option, written just like it is here: `formula = y ~ s(x)`. Don't replace the y and x with the variable names; you'll get an error. Other GAM options can be included in the geom_smooth call inside the formula (e.g., `formula = y ~ s(x), k=10`).

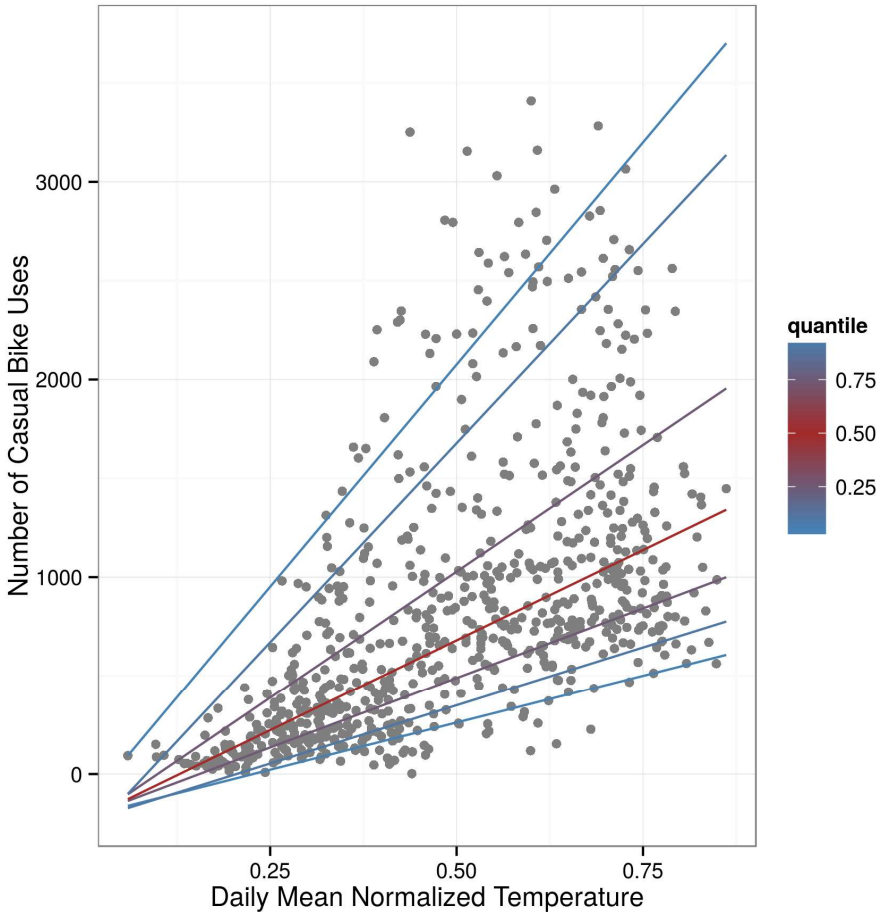
```
ggplot(bike_share_daily, aes(x=atemp, y=cnt)) +
  xlab("Daily Mean Normalized Air Temperature") +
  ylab("Number of Total Bike Uses") +
  geom_point(col="gray50") +
  geom_smooth(method="gam", formula=y~s(x)) +
  theme_bw()
```



Quantile trends

To show trends based on quantiles, use the stat_quantile option, and choose which quantiles you want to show, either specifically (as below) or sequentially (e.g., `quantiles = seq(0.05, 0.95,`

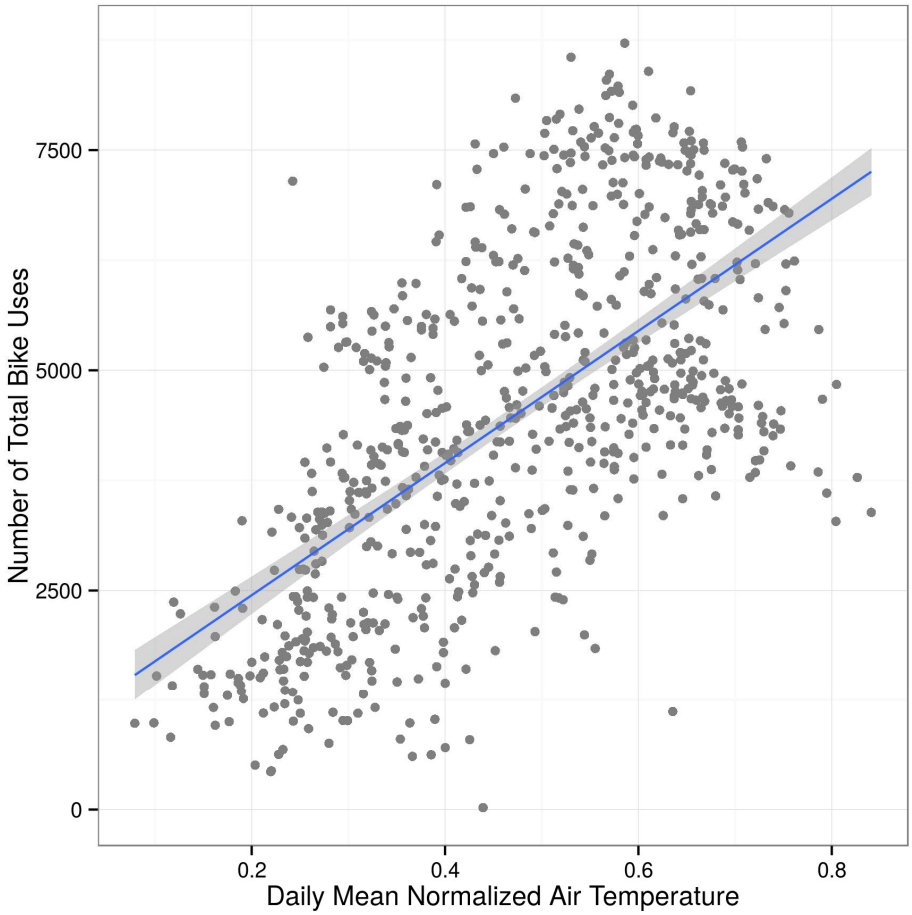
```
by=0.05):


ggplot(bike_share_daily, aes(x=temp, y=casual)) +
  xlab("Daily Mean Normalized Temperature") +
  ylab("Number of Casual Bike Uses") +
  geom_point(col="gray50") +
  stat_quantile(aes(color = ..quantile..), quantiles = c(0.05, 0.1, 0.25,
    0.5, 0.75, 0.9, .95)) +
  scale_color_gradient2(midpoint=0.5, low="steelblue", mid="brown",
    high="steelblue ") +
  theme_bw()
```



Simple linear trends

Finally, to get a simple linear trend, use the `lm` method with `geom_smooth`:

```
ggplot(bike_share_daily, aes(x=atemp, y=cnt)) +
  xlab("Daily Mean Normalized Air Temperature") +
  ylab("Number of Total Bike Uses") +
  geom_point(col="gray50") +
  geom_smooth(method="lm") +
  theme_bw()
```



If you're exploring trends, it's good habit to *not* start with an OLS/linear trend, as in real data exploration you usually don't have any *a priori* knowledge about the functional relationship or trend. Only using this approach after you've tried the others helps prevent starting your thinking into a (linear) rut.

Segmented linear trends

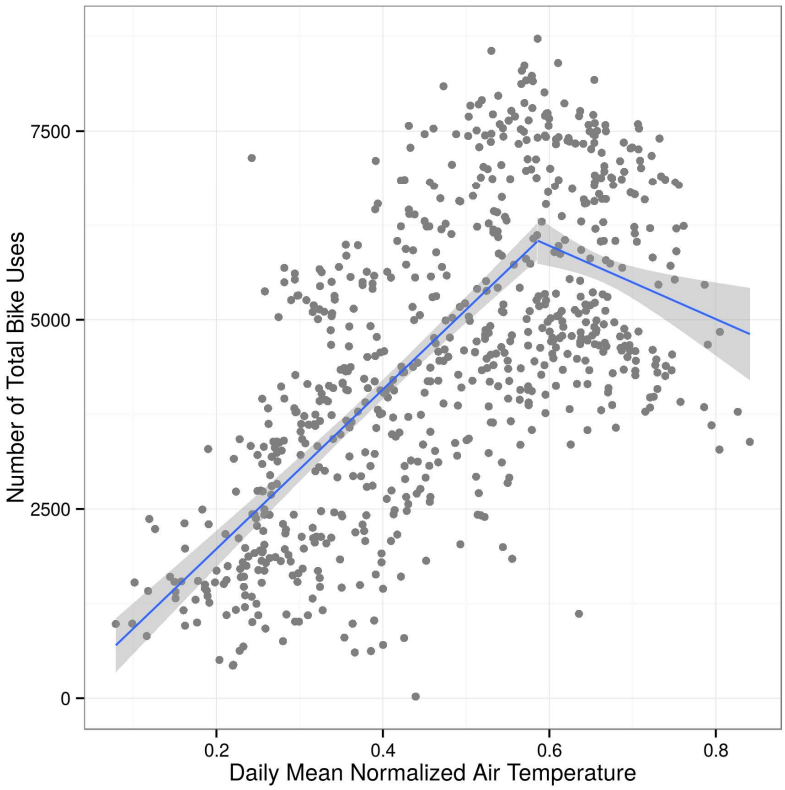
Sometimes you do want to see a linear trend, but know (or suspect) that there are changes in the trend. We'll explore ways to evaluate changepoints later, but for now we'll just quickly plot

a segmented linear trend line. We'll allow the `segmented` package to choose the optimal breakpoint for us, after we specify approximately where we think it is with the `psi` option:

```
require(segmented)

bike_segment = segmented(lm(cnt~atemp, data=bike_share_daily), ~atemp, psi=0.6)
bike_segment$psi
```

```
           Initial     Est.     St.Err
psi1.atemp     0.6 0.585762 0.01352518
```

```
psi = bike_segment$psi[2]

ggplot(bike_share_daily, aes(x=atemp, y=cnt, group = atemp > psi)) +
  xlab("Daily Mean Normalized Air Temperature") +
  ylab("Number of Total Bike Uses") +
  geom_point(col="gray50") +
  geom_smooth(method="lm") +
  theme_bw()
```



The `psi` option is pretty robust to starting values when you have a large sample size and there's at least a hint of a real break in the trend. For example, you can run the above code using a `psi` of 0.2—where there's clearly no change in trend—and the breakpoint estimate is exactly the same. However, with fewer data points and/or a lot of variance, the choice of `psi` could change the results, so try a few different values.

The many flavors of regression

There are many ways to model trends in data, but the majority of cases can be addressed with a few types. Still, there are more flavors of regression than a Ben & Jerry's warehouse after a tornado.

The coefficients, of course, are effect sizes.

Response type	Response error distribution	Predictor type	Model type	Code**
Continuous	Normal	Continuous	Regression	`lm(y ~ x1 + x2 ... + xn)`
Continuous	Normal	Categorical	ANOVA	`lm(y ~ x1 + x2 ... + xn)`
Continuous	Normal	Mixed	ANCOVA	`lm(y ~ x1 + x2 ... + xn)`
Binary	Logistic	Mixed	Logistic regression	`glm(y ~ x1 + x2 ... + xn, family=binomial(link=logit))`
Ordinal	Logistic	Mixed	Proportional odds logistic regression	`polr(y ~ x1 + x2 ... + xn, Hess=TRUE)`
Nominal	Logistic	Mixed	Multinomial regression	`multinom(y ~ x1 + x2 ... + xn)`
Count	Poisson	Mixed	Poisson regression	`glm(y ~ x1 + x2 ... + xn, family=poisson(link=log))`
	Overdispersed Poisson	Mixed	Negative binomial regression	`glm.nb(y ~ x1 + x2 ... + xn)`
Rate	Poisson	Mixed	Poisson regression	`glm(y ~ x1 + x2 ... + xn, offset=log(t), family=poisson(link=log))`
	Overdispersed Poisson	Mixed	Negative binomial regression	`glm.nb(y ~ x1 + x2 ... + xn, offset=log(t))`
Counts with lots of 0s	Overdispersed Poisson	Mixed	Zero-inflated Poisson regression	`require (pscl); zeroinfl(y ~ x1 + x2 ... + xn, dist="poisson")`

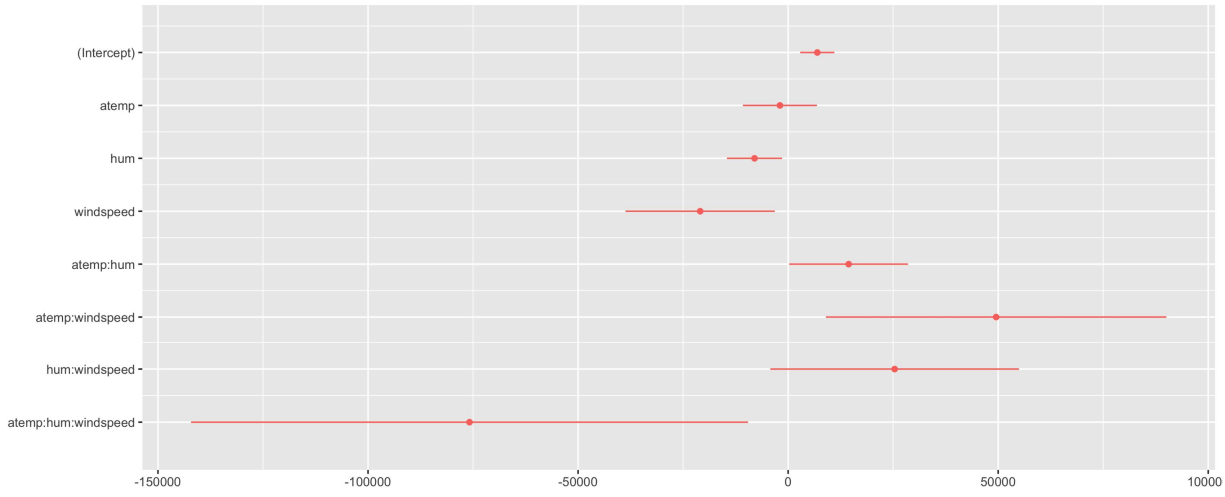
Response type	Response error distribution	Predictor type	Model type	Code**
Counts where 0 is not possible and mean < 5	Overdispersed Poisson	Mixed	Zero-truncated Poisson regression	`require (gamlss.tr); gamlss(y ~ x1 + x2 ... + xn, family=PO)`
Any	Same as `glm`	Smoothed	GAM	'library(mgcv); gam(y ~ x1 + x2 ... + xn, family=...)

** change + to * to create interactions

Plotting regression coefficients

The `dotwhisker` package has a simple function to graph regression results using `ggplot2`. `dwplot` shows the coefficients and their confidence intervals (95% by default):

```
require(dotwhisker)

# Create a model object
bad_model = lm(cnt ~ atemp * hum * windspeed, data=bike_share_daily)

# Plot coefficients and CIs
dwplot(bad_model)
```



Working with temporal data

Date and time formats are the bane of coders everywhere. No matter which coding language you use, you'll eventually run into the need to convert dates or times and end up wanting to punch your screen.

Time series analysis requires specific data types or structures—we'll explore different types of those structures below, and the CRAN Time Series task view[49] is worth spending time exploring if you need to evaluate time series in any depth.

The informal consensus is that in R, `as.Date` is best for dates and `as.POSIXct` is best for times because these are the simplest ways to work with each data type. The `lubridate` package provides a more intuitive wrapper to `as.POSIXct` and is very useful for date conversions between types.

It's useful to have a cheat-sheet of the major date and time formats; you can get a complete list with `?strptime` but the major ones are as follows:

Major date formats

Date step	Symbol	Format	Alternate symbols	Alternate format
Format Date (ISO)	`%F`	Y-m-d	`%x`, `%D`	y/m/d, m/d/y
Year	`%Y`	4-digit year (2014)	`%y`	2-digit year (14)
Month	`%m`	Decimal month (10)	`%b`, `%B`	Abbreviated month (Oct.), Full month (October)
Day	`%d`	Decimal date (1-31)	`%e`	Decimal date (01-31)
Weekday	`%w`	Decimal weekday (0-6, starts with Sunday)	`%a`, `%A`, `%u`	Abbreviated weekday (Sun), Full weekday (Sunday), Decimal with Monday start (7)
Weekyear (ISO)	`%V`	Sunday start (0-53)	`%U`, `%W`	Sunday start (USA), Monday start (UK)
Day of the year	`%j`	Decimal day of the year (1-366)		

[49] http://cran.r-project.org/web/views/TimeSeries.html

Major time formats

Time step	Symbol	Format
Date and time	`%c`	a b e H:M:S Y
Time	`%T`	H:M:S
Time (short)	`%R`	H:M
Hours (24 hour clock)	`%H`	
Hours (12 hour clock)	`%I`	
Minutes	`%M`	
Seconds	`%S`	
AM or PM	`%p`	
Offset from GMT	`%z`	
Time zone	`%Z`	

There just isn't the space to cover all (or even a small subset) of date/time conversions and temporal math, so we'll just illustrate a few very common date conversion needs in this recipe.

In addition to the `base` function `as.Date`, we'll use the `lubridate` package in this recipe.

```
require(lubridate)
```

The ISO standard for a date format is 4-digit year, 2-digit month, and 2-digit day, separated by hyphens. We'll often aim to convert to that format, but as we'll see, it's not always necessary as long as you specify the format type in the conversion process.

Perhaps the most common date conversion is a character to a date. If it's not in a clear *year-month-day* format, you'll need to specify the format it's in so that R can properly convert it to a date (a table of the major formats appears below).

```
india_natl_holidays = c("Jan 26, 2014", "Apr 13, 2014", "Aug 15, 2014",
  "Oct 02, 2014", "Dec 25, 2014")

india_natl_holidays = as.Date(india_natl_holidays, format="%b %d, %Y")

str(india_natl_holidays)

Date[1:5], format: "2014-01-26" "2014-04-13" "2014-08-15"
"2014-10-02" "2014-12-25"
```

Another common date conversion need is to calculate age based on today's date (or any reference date):

```
birthdays = c("20000101", "19991201", "19760704", "20140314")

ymd(birthdays)

"2000-01-01 UTC" "1999-12-01 UTC" "1976-07-04 UTC" "2014-03-14 UTC"

age = ymd(today()) - ymd(birthdays)

round(age/dyears(1), 1) # dyears is a lubridate function, not a typo!

14.9 15.0 38.4  0.7

age = ymd(20141201) - ymd(birthdays)

round(age/dyears(1), 0)

15 15 38  1
```

The basic thing to remember is that even when you don't have a real day value (e.g., your data is binned into months), many functions in R generally expect one if you want to use date functions. See the next recipe for an example of this issue in action.

Calculate a mean or correlation in circular time (clock time)

Sometimes you want to know the average time at which something happens, or the relationship between times, but if the data occur near the end of a time period, the use of `mean()` will give you incorrect results. The `psych` package comes to the rescue with the `circadian.mean` and `circadian.cor` functions:

```
require(psych)

wake_time = c(7.0, 7.5, 6.5, 6.25, 7.0, 7.25, 7.25)

sleep_time = c(23.5, 0.5, 23.75, 0.25, 23.25, 23.5, 0.75)

wake_sleep = data.frame(wake_time, sleep_time)

circadian.mean(wake_sleep$sleep_time)

23.92808

circadian.cor(wake_sleep)
```

```
           wake_time sleep_time
wake_time  1.0000000  0.1524331
sleep_time 0.1524331  1.0000000
```

Plotting time-series data

While having dates in the dataset can allow easy plotting via `ggplot2`, there are a few plotting functions that provide one-line insight into time series data. They may not be graphically elegant, but they provide a great way to understand your data more thoroughly.

We'll use a dataset of 10 years of monthly births in the UK (2003-2012), acquired from the EU's *EuroStat* program.

```
require(eurostat)

demo_fmonth = get_eurostat('demo_fmonth')
```

R can be a little particular about date values, and if your data aren't in a format it recognizes, it can choke. Case in point is using `ggplot2` to plot this data, which contains a year field and a month field but not a day field—which makes sense, seeing as how the data are monthly values. But you'll have to tie yourself in knots to make those two work together as time elements in a ggplot, so it's actually easier if you just gave each month a fake day (i.e., the 1st), mush 'em together as a character via the `paste` function, and then specify that you want the final result to be in `Date` format.

```
demo_fmonth$date = as.Date(paste(substr(demo_fmonth$time, 1, 4),
  substr(demo_fmonth$month, 2, 3), "01", sep="-"))
```

Now we can use `dplyr` to filter and arrange the data…

```
require(dplyr)

UK_births = filter(demo_fmonth, geo == "UK" & month != 'TOTAL' &
  month != 'UNK' & date >= '2003-01-01' & date <= '2012-12-01')

UK_births = arrange(UK_births, date)
```

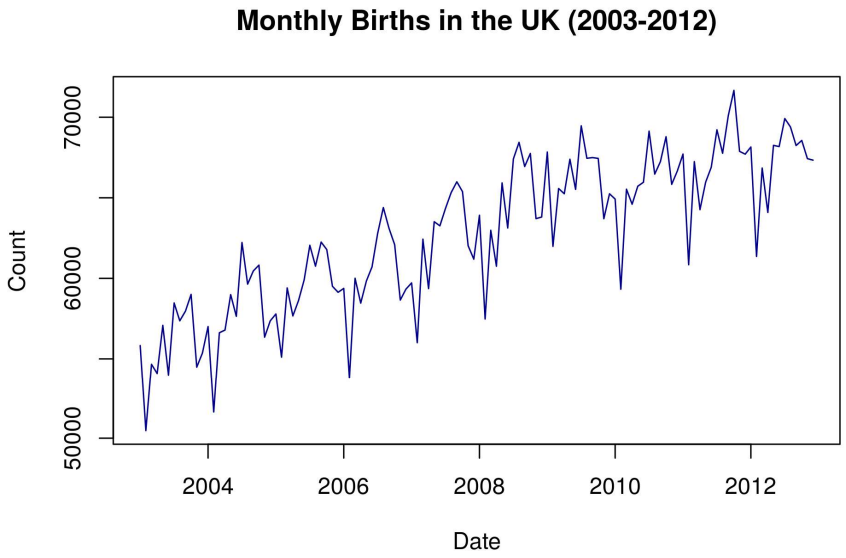
…and then we can convert the data into a `ts` (time series) object:

```
UK_births_ts = ts(UK_births$values, start=c(2003, 1), frequency=12)
```

The `frequency` option is related to how many time steps make up a "whole" unit in your data; for example, if your data is hourly and you expect seasonality on the daily level, frequency equals 24. Obviously, the above setting tells R that the data are monthly and we expect annual "seasonality."

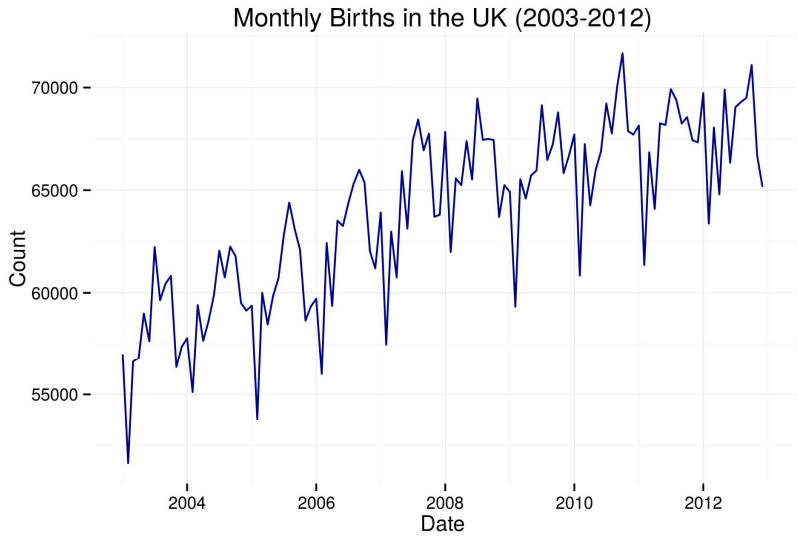
Once we have a `ts` object, we can plot it; if all you need to do is look at it, you can simply use `plot(my_ts_object)`. But if you want to use the plot for more than just your own edification, you'll need to add some decoration manually:

```
plot(UK_births_ts, xlab="Date", ylab="Count", main="Monthly Births in the UK
  (2003-2012)", col="darkblue")
```



And we can use `ggplot2` to make it prettier:

```
ggplot(UK_births, aes(x=date, y=values)) +
  geom_line(col="darkblue") +
  ylab("Count") +
  theme_minimal()
```



Detecting autocorrelation

Autocorrelation can be present in non-temporal data as well as in time series; the `acf` function works the same for either type of data.

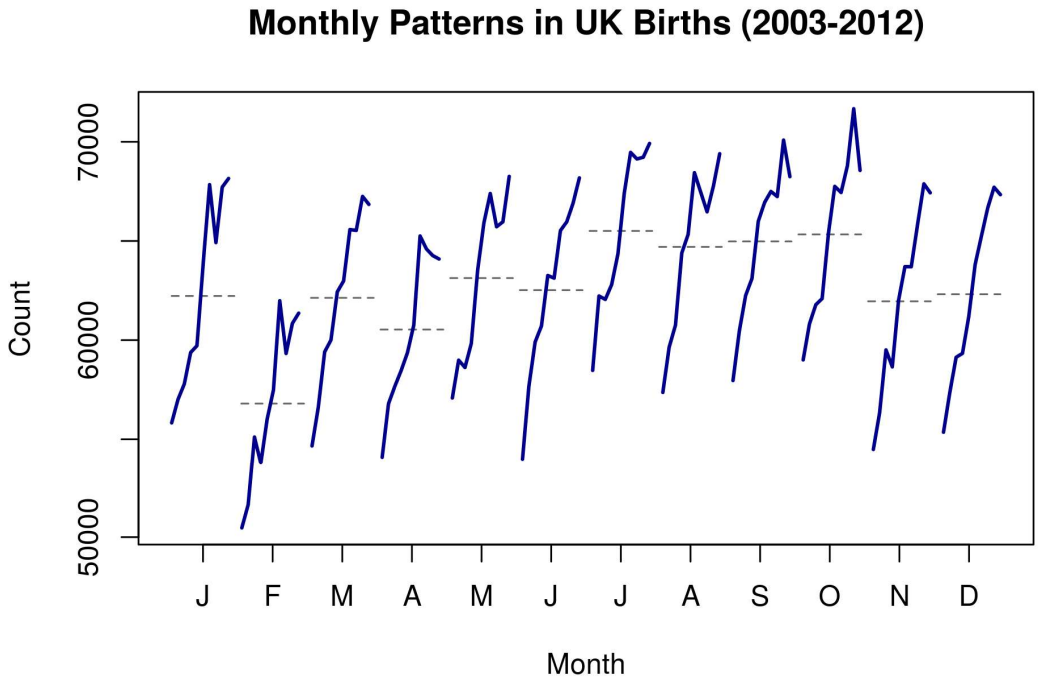
```
acf(UK_births_ts)
```

What's often under-appreciated is that autocorrelated data, temporal or not, will make standard confidence intervals *too small*. In other words, if you want to plot a trend from autocorrelated data and use the default confidence intervals, you'll underestimate the actual uncertainty associated with that trend.

Plotting monthly and seasonal patterns

If your data are monthly (or if you aggregate them to months), the `monthplot` function in the `base` installation breaks out each month's data and plots them separately, along with a horizontal line for each month's mean (the default) or median (`base = median`) value over the time span. A few manual tweaks help show the information in this plot more clearly:

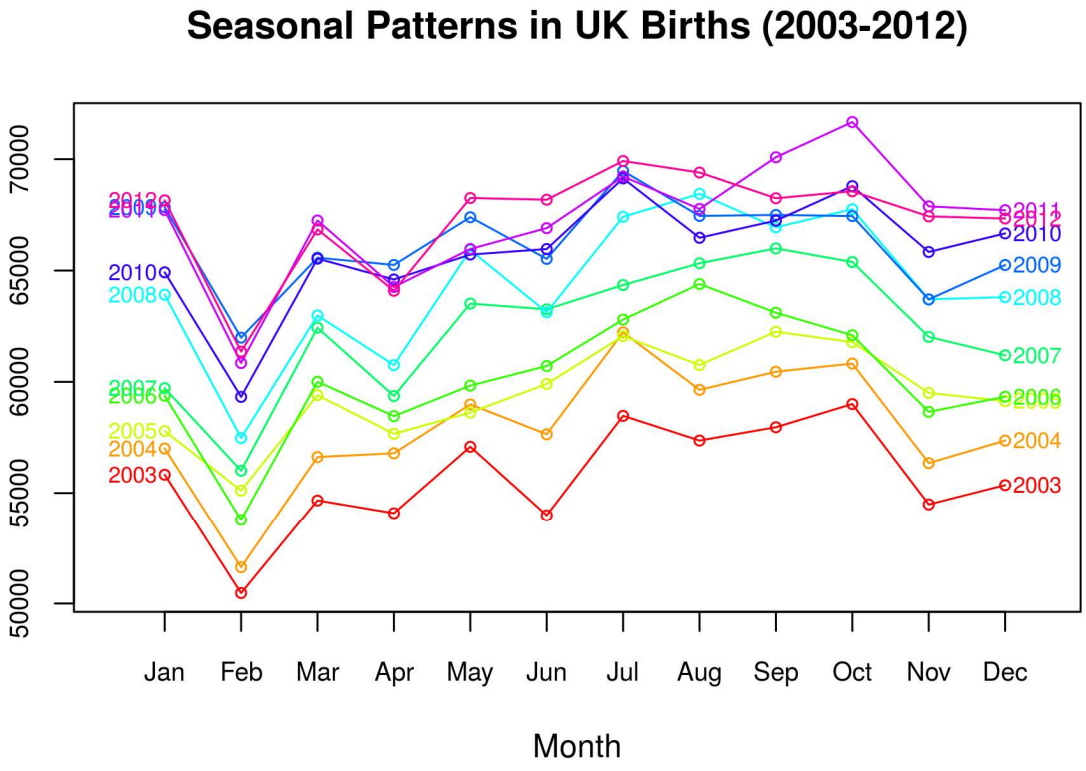
```
monthplot(UK_births_ts, main="Monthly Patterns in UK Births (2003-2012)",
  xlab="Month", ylab="Count", col="darkblue", lwd=2, lty.base=2, lwd.base=1,
  col.base="gray40")
```



You can plot out the seasonal, trend, or random components in a `monthplot` with the `choice` option, e.g., `choice = "seasonal"`).

The `forecast` package contains another useful plot function that shows each year's trend as a separate line, allowing you to discern whether there's a seasonal or monthly effect at a glance:

```
require(forecast)

seasonplot(UK_births_ts, main="Seasonal Trends in UK Births (2003-2012)",
  col=rainbow(10), year.labels=TRUE, year.labels.left=TRUE, cex=0.7,
  cex.axis=0.8)
```

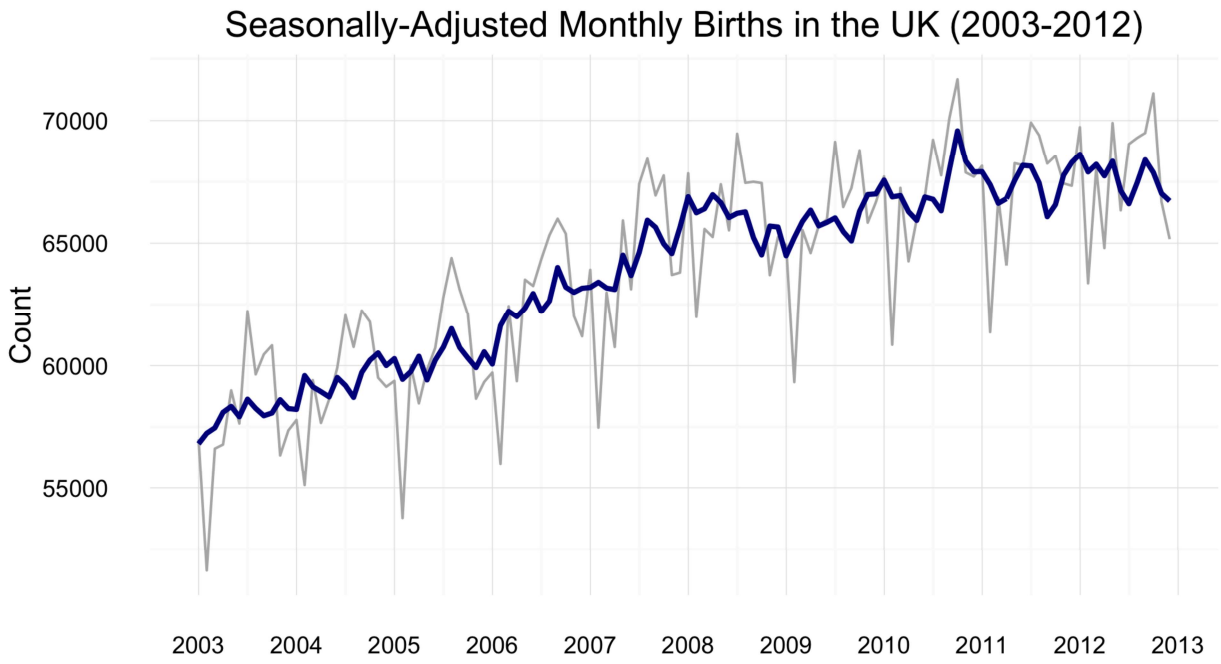


Plotting seasonal adjustment on the fly

The ggseas package provides an easy way to plot seasonally-adjusted data in ggplot2. There are a variety of ways to adjust for seasonality, so review its GitHub site[50] for more details and options that come with this package.

[50] https://github.com/ellisp/ggseas

```
require(ggseas)

# Convert the time series object to a dataframe for stat_seas
UK_births_ts_df = tsdf(UK_births_ts)

# Plot seasonally-adjusted data over actual data
ggplot(UK_births_ts_df, aes(x=x, y=y)) +
  ggtitle("Seasonally-Adjusted Monthly Births in the UK (2003-2012)") +
  geom_line(color="gray70") +
  stat_seas(color="darkblue", size=1) +
  scale_x_continuous(breaks=seq(2003, 2013, 1)) +
  ylab("Count") +
  xlab("") +
  theme_minimal()
```



Decomposing time series into components

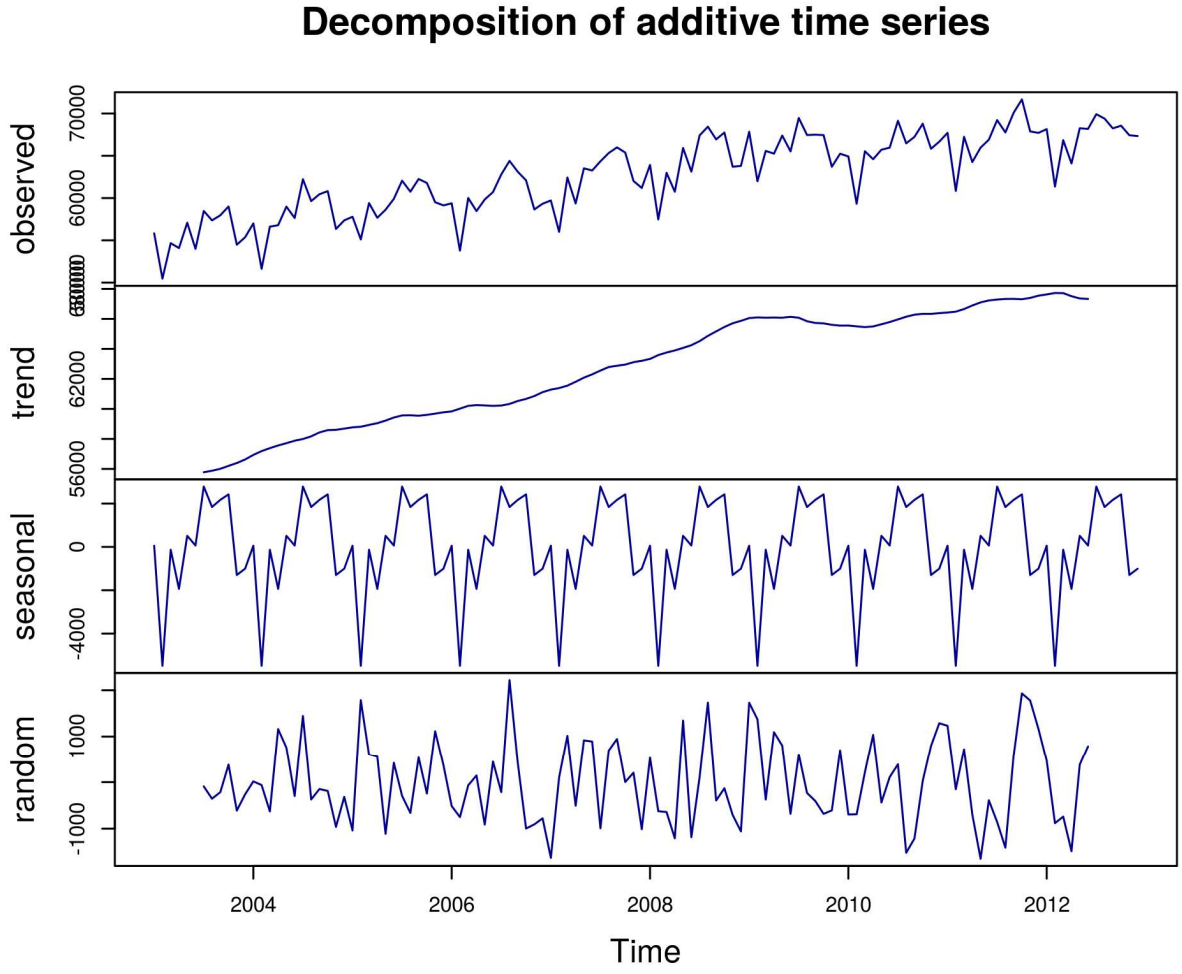
If you need to explore a time series in more depth, you'll probably want to decompose it into seasonal, trend, and random/remainder components.

Additive decomposition

We'll continue with the UK monthly births data in its time series format. Most functions we'll use in this recipe are in the `base` installation's `stats` package, but one is in the `forecast` package, so make sure it's loaded with the data.

The `decompose` function uses a moving average for simplicity:

```
plot(decompose(UK_births_ts), col="darkblue")
```



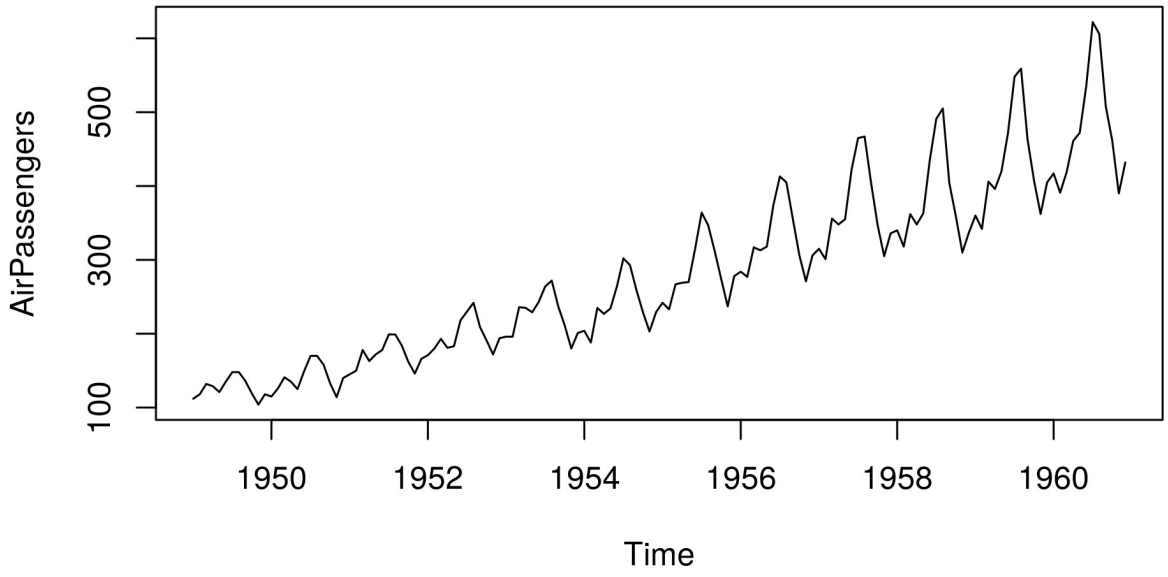
Decomposing when the variance changes

Of course, simple time series decomposition only works correctly if the variance is more or less stable. When it's not—as in the `AirPassengers` dataset, below—you have to transform the values before you decompose the series into its components.

There are two ways to do this: by using `type = "multiplicative"` in the `decompse` function as above, or by using the `BoxCox.lambda` function in the `forecast` package to determine an optimal transformation.

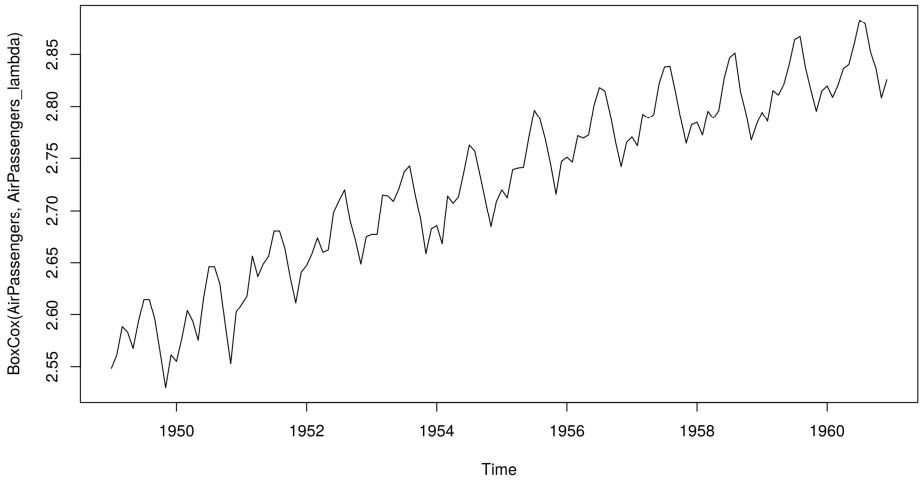
Look at the data to see the change in variance over time:

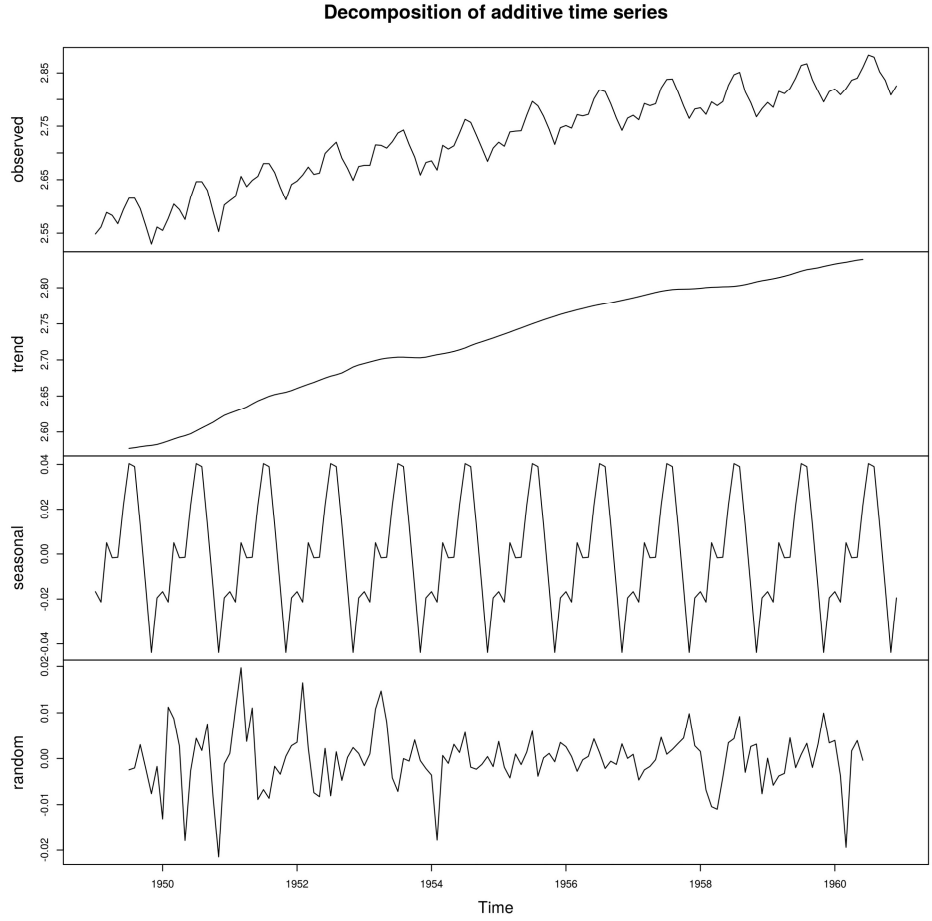
```
data(AirPassengers)

plot(AirPassengers)
```



Use the `BoxCox.lambda` function and view the results:

```
require(forecast)

# Calculate the Box-Cox transformation
AirPassengers_lambda = BoxCox.lambda(AirPassengers)

# Plot the Box-Cox transformation results
plot(BoxCox(AirPassengers, AirPassengers_lambda))

# Plot the transformed decomposition
plot(decompose(BoxCox(AirPassengers, AirPassengers_lambda)))
```


BoxCox(AirPassengers, AirPassengers_lambda)
2.55
2.60
2.65
2.70
2.75
2.80
2.85
1950
1952
1954
1956
1958
1960
Time


Decomposition of additive time series
observed
trend
seasonal
random
Time

Using spectral analysis to identify periodicity

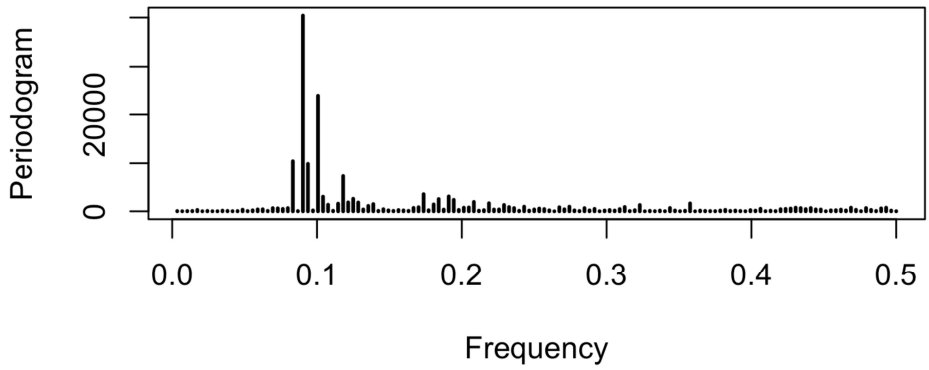
Spectral analysis provides a simple way to evaluate whether there are any cyclical patterns in your time series. The `TSA` package's `periodogram` function creates a periodogram, where the highest peak(s) in the plot represent(s) possible cycling times. To do spectral analysis, you need to ensure the data are detrended first.

We'll use the built-in annual sunspot dataset in this recipe, as sunspots have a well-known 11-year cycle.

```
data(sunspot.year)
```

To find the cycle times, look for a clear peak (or peaks) in the plot.

```
sunny = TSA::periodogram(diff(sunspot.year))
```



You can also extract the `$freq` and `$spec` values from the spectrum object into a data frame as `x` and `y`, respectively, to get the exact frequency where the spectrum is maximized, or if you want to create a prettier plot with `ggplot2`.

```
sunny_df = data.frame(freq = sunny$freq, spec = sunny$spec)

sunny_df[which.max(sunny_df[,2]),]
```

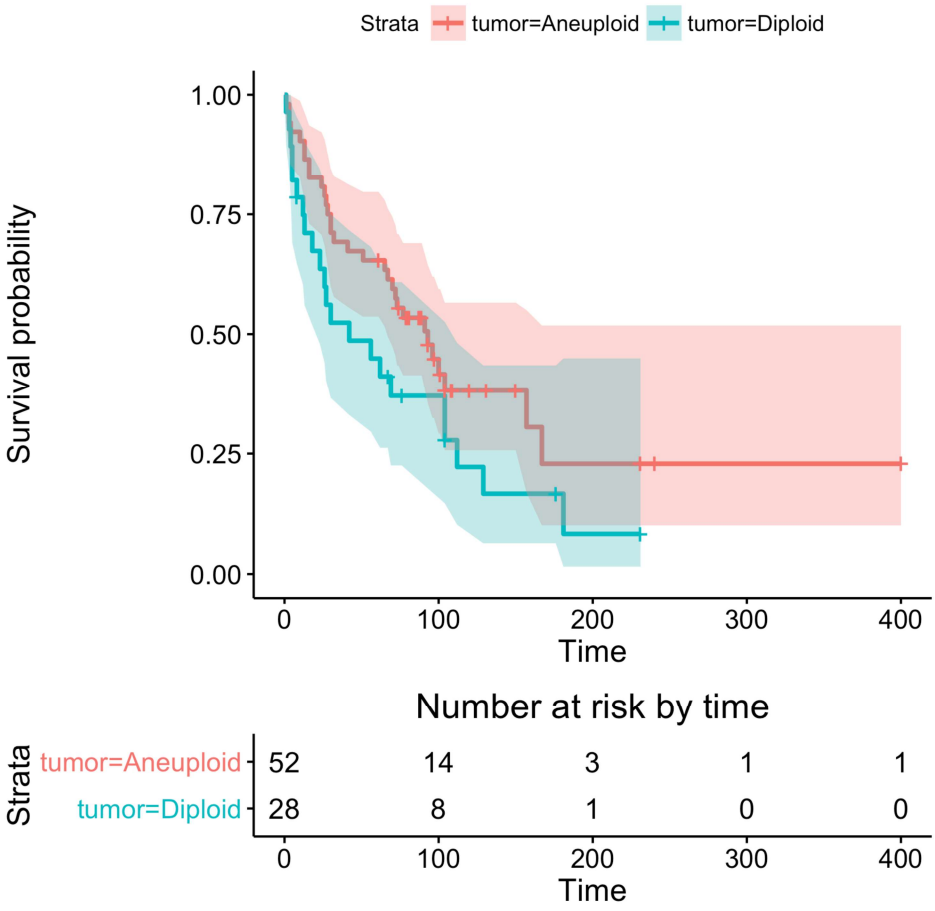
```
         freq     spec
26 0.09027778 40400.47
```

Frequency is the reciprocal of the time period. This data is annual, and since the the highest spectral peak is at a frequency of 0.09, it suggests that there is a ~11 year cycle (1/0.09=11).

Plotting survival curves

The `survminer` package makes a beautiful all-in-one survival plot and risk table. We'll use the `tongue` cancer data from the `KMsurv` package to demonstrate.

```
require(survival)
require(survminer)

# Get data and add names of cancer type
data(tongue, package = "KMsurv")

tongue$tumor = ifelse(tongue$type==1, "Aneuploid",  "Diploid")

# Create survival model
tongue_survival = survfit(Surv(time = time, event = delta) ~ tumor,
  data = tongue)

# Plot the curves and table
ggsurvplot(tongue_survival, risk.table = TRUE, conf.int = TRUE)
```



Evaluating quality with control charts

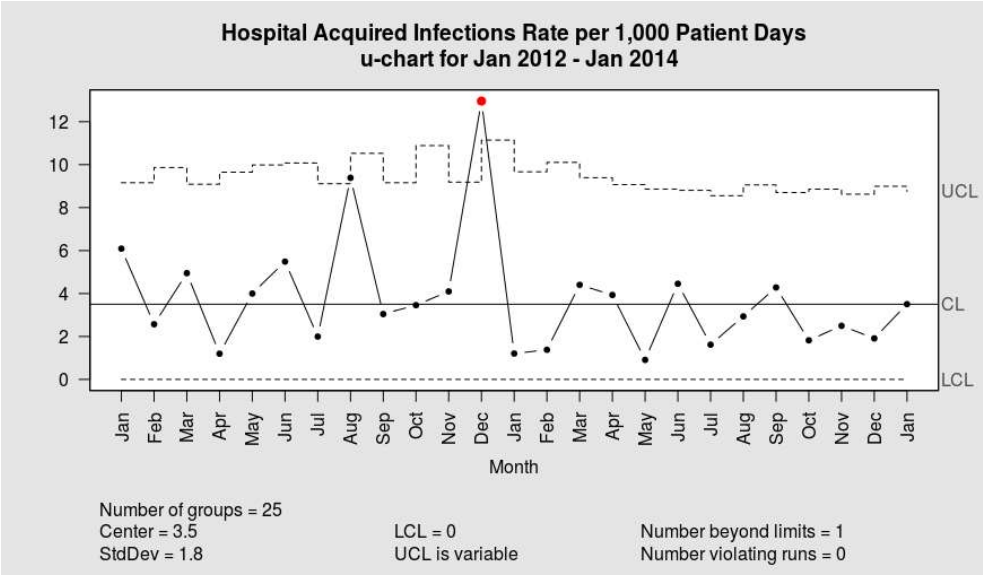
Companies with quality control processes or activities often use control charts (aka "Shewhart charts") to evaluate whether a **stable** process is moving "out of control" over time and thus requires intervention. The qcc package provides a way to plot all major control chart types simply and quickly.

As an example, we'll create some fake data on hospital-acquired-infections (HAI) for a 25-month period to illustrate the creation of a u-chart, a control chart that plots changes in a rate over time:

```
require(qcc)

infections = c(6, 2, 5, 1, 3, 4, 2, 6, 3, 2, 4, 7, 1, 1, 4, 4, 1, 5, 2, 3, 5,
  2, 3, 2, 4)

patient_days = c(985, 778, 1010, 834, 750, 729, 1002, 639, 985, 578, 976, 540,
  829, 723, 908, 1017, 1097, 1122, 1234, 1022, 1167, 1098, 1201, 1045, 1141)

# These are just labels for qcc, not real dates
month_name = month.abb[c(1:12, 1:12, 1:1)]
```

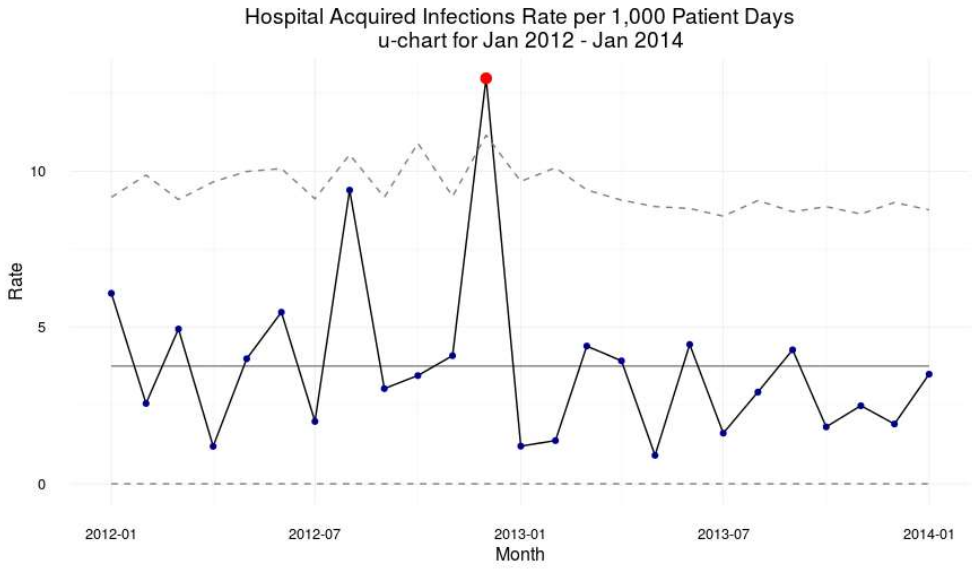
We'll use 1,000 patient days for the rate baseline. The qcc function takes care of everything:

```
infection_control = qcc(infections, sizes=patient_days/1000, type="u",
  labels=month_name, axes.las=2, xlab="Month", ylab="", digits=2,
  title="Hospital Acquired Infections Rate per 1,000 Patient Days\n
  u-chart for Jan 2012 - Jan 2014")
```



The control-chart default plot in `qcc` is fairly ugly. You can remove the components of the resulting R object for use elsewhere, such as with `ggplot2`:

```
# Create a real date and move limits and rate into a data frame
ic_qcc = data.frame(Month = seq(as.Date("2012-01-01"),
  as.Date("2014-01-01"), "months"), infection_control$limits,
  Rate = (infections / patient_days)*1000)

# Create a factor for the "special causes" points
ic_qcc$Violations = factor(ifelse(row.names(ic_qcc)
  == infection_control$violations$beyond.limits, "Violation", NA))

# Plot a cleaner control chart
ggplot(ic_qcc, aes(x=Month, y=Rate)) +
  geom_line(aes(y=mean(ic_qcc$Rate)), color="gray50") +
  geom_line() +
  geom_point(color="darkblue") +
  geom_point(data=filter(ic_qcc, Violations=="Violation"),
    color="red", size=3) +
  geom_line(aes(y=LCL), linetype="dashed", color="gray50") +
  geom_line(aes(y=UCL), linetype="dashed", color="gray50") +
  xlab("Month") +
  ylab("Rate") +
  ggtitle("Hospital Acquired Infections Rate per 1,000 Patient Days
    u-chart for Jan 2012 - Jan 2014") +
  theme_minimal()
```



Many people use control charts incorrectly on trending and/or autocorrelated data. When you need something like a control chart on these types of data, use a GAM instead. Morton et al.'s book Statistical Methods for Hospital Monitoring with R[51] has R code and some good advice on this issue.

Identifying possible breakpoints in a time series

The `strucchange` package provides a simple way to identify changes in the statistical structure of a time series.

We'll look at total monthly United States CO_2 emissions between February 2001 and June 2015 (data from the US Energy Information Administration[52]). Load the `strucchange` package, download and filter the data, and convert the data to a time series object:

```
require(strucchange)
US_co2 = read.table("http://www.eia.gov/totalenergy/data/browser/
  csv.cfm?tbl=T12.01", sep=",", header=T)

US_co2 = filter(US_co2, Column_Order == 14 & YYYYMM >= 200102 &
  substr(YYYYMM, 5, 7) != 13)

US_co2_ts = ts(US_co2[,3], freq=12, start=c(2001,2))
```

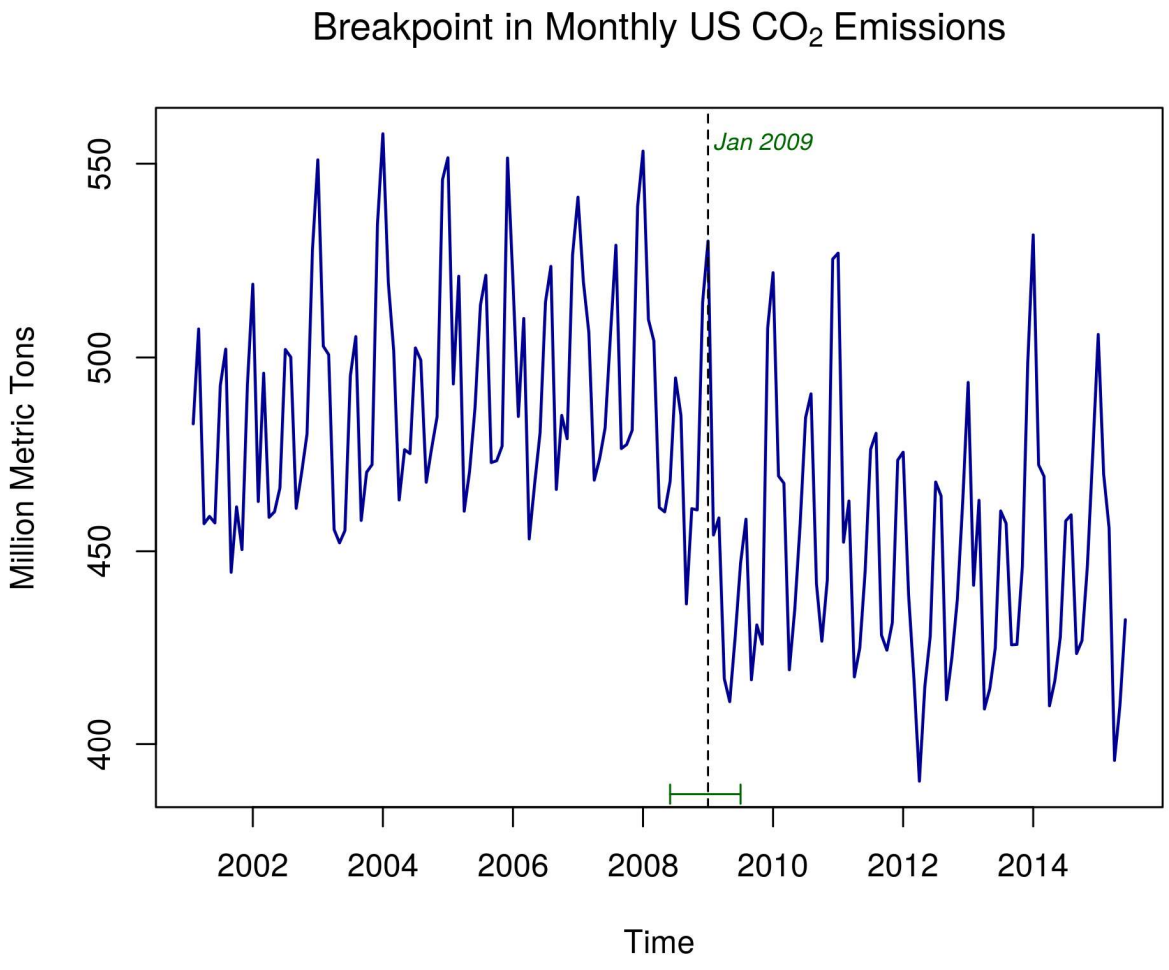
`strucchange` will pick the optimal number of breakpoints as determined by BIC, which can be subsequently plotted over the time series. As there does not seem to be a clear indication of generally trending rises or declines in mean tendency, we can use ~ 1 in the formula in place of a dependent variable to assess whether there is a step change (or changes) in the time series:

```
# Obtain breakpoints estimate
breakpoints(US_co2_ts ~ 1)

     Optimal 2-segment partition:
Call:
breakpoints.formula(formula = us_co2_ts ~ 1)
Breakpoints at observation number:
96
Corresponding to breakdates:
2009(1)
```

[51] https://www.amazon.com/Statistical-Methods-Hospital-Monitoring-R/dp/1118596307

[52] http://www.eia.gov/beta/MER/index.cfm?tbl=T12.01#/?f=M&start=200101&end=201506&charted=0-1-13

```
# Plot the time series
plot(US_co2_ts, main=expression(paste("Breakpoint in Monthly US"
  ~CO[2]~Emissions)), ylab="Million Metric Tons", col="darkblue", lwd=1.5)

# Plot the line at the optimal breakpoint
lines(breakpoints(US_co2_ts ~ 1), col="darkgreen")

# Plot a 90% confidence interval
lines(confint(breakpoints(US_co2_ts ~ 1), level=0.90), col="darkgreen")

# Add breakpoint location text
text(2008.8, 555, "Jan 2009", cex=0.75, col="darkgreen", pos=4, font=3)
```



Exploring relationships between time series: cross-correlation

Cross-correlation is extremely useful for helping identify leading and lagging indicators. The `ccf` function makes this easy.

We'll use data from the US Bureau of Labor Statistics to demonstrate cross-correlation.

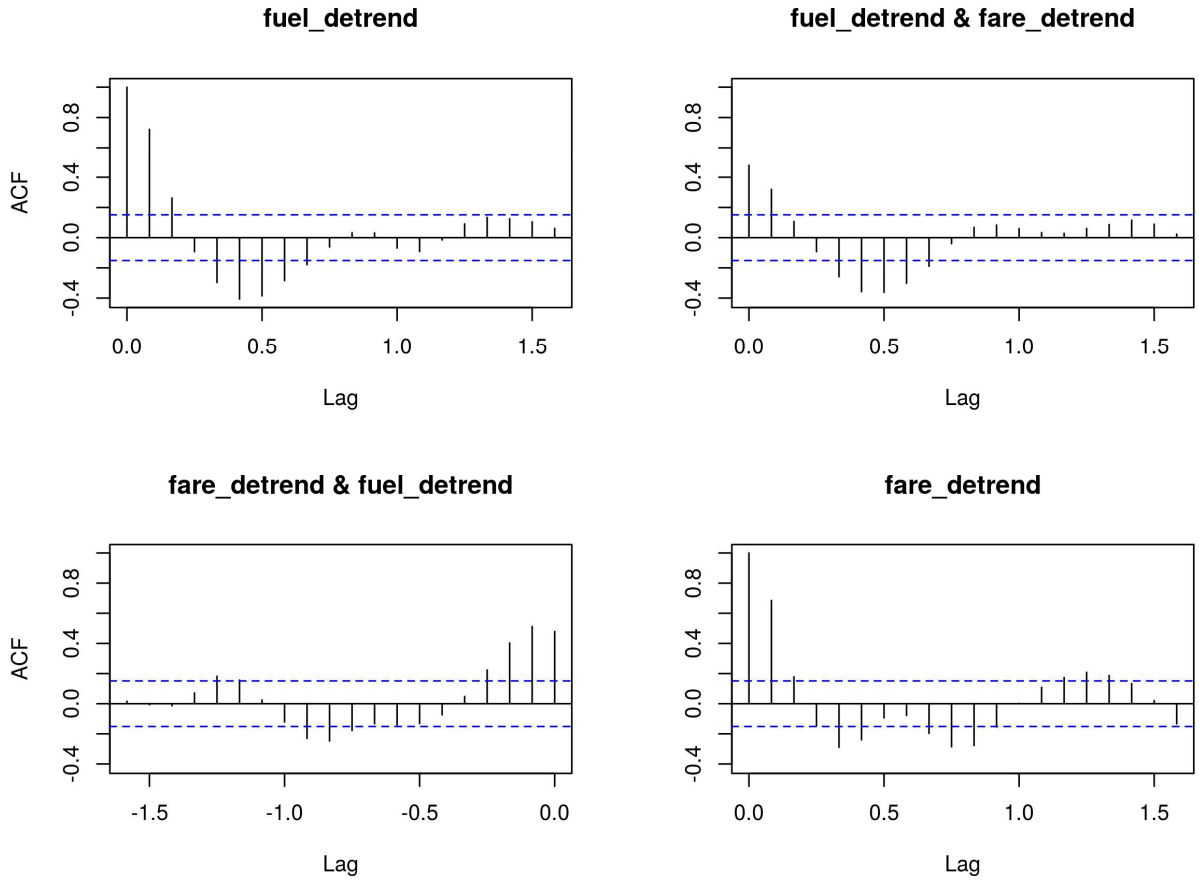
```
# Load dplyr to subset the BLS data
require(dplyr)

# Download CPI time series data from US BLS
CPI_xport = read.table("http://download.bls.gov/pub/time.series/cu/
  cu.data.14.USTransportation", header=T, sep="\t", strip.white=T)

# CPI for motor fuel, 2000-2014
fuel = filter(CPI_xport, series_id == "CUUR0000SETB" &
  year >= 2000 & year <= 2014 & period != "M13")

# CPI for airline fare, 2000-2014
fare = filter(CPI_xport, series_id == "CUUR0000SETG01" &
  year >= 2000 & year <= 2014 & period != "M13")
```

Since `ccf` assumes that the time series is already detrended, we'll do that first.

```
# Create time series
fuel_ts = ts(fuel$value, start=c(2000, 1), freq=12)

fare_ts = ts(fare$value, start=c(2000, 1), freq=12)

# CCF *requires* detrended data
# Get detrended values of each time series
fuel_detrend = na.omit(decompose(fuel_ts)$random)

fare_detrend = na.omit(decompose(fare_ts)$random)
```

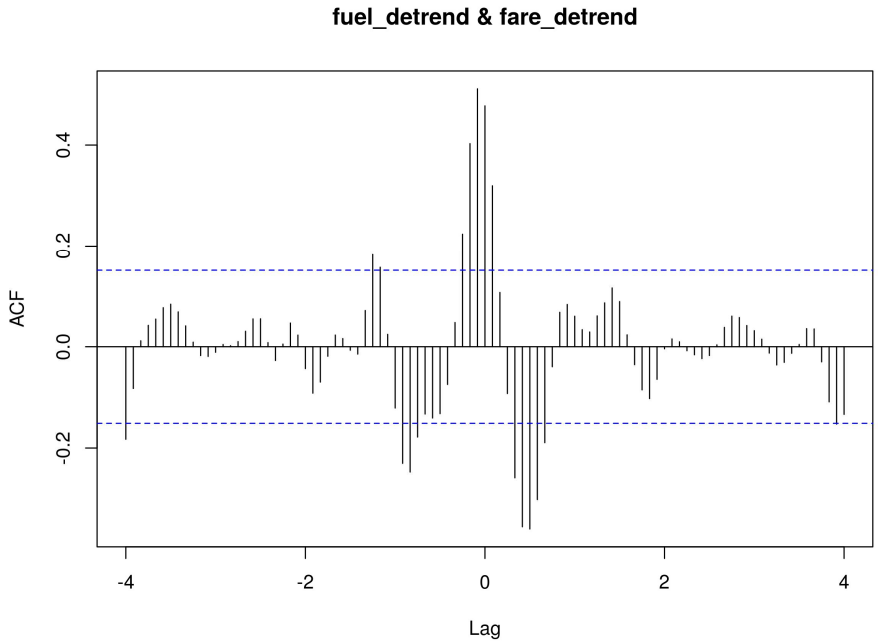
An ACF plot shows the pattern of each variable separately and together for an 18 month time span:

```
acf(ts.union(fuel_detrend, fare_detrend))
```



You can combine these views into a single plot with the `ccf` function:

```
# Calculate CCF with +/- 4 year span
ccfvalues = ccf(fuel_detrend, fare_detrend, lag.max=48)

ccfvalues
```



We can see that the maximum occurs just to the left of zero, we can infer that fuel costs lead airline fares slightly. Since it's hard to read values off a plot, we can use a small function from nvogen on Stack Overflow[53] that can find the maximum correlation value:

```
# Function to find CCF
Max_CCF = function(a, b) {
  d = ccf(a, b, plot = FALSE, lag.max = length(a)-5)
  cor = d$acf[,,1]
  abscor = abs(d$acf[,,1])
  lag = d$lag[,,1]
  res = data.frame(cor,lag)
  absres = data.frame(abscor,lag)
  absres_max = res[which.max(absres$abscor),]
  return(absres_max)
}

# Determine maximum CCF value
Max_CCF(fuel_detrend, fare_detrend)
          cor         lag
163 0.5126287 -0.08333333
```

The maximum cross-correlation value is 0.51, which occurs between airline fares (at time 0) and fuel prices in the prior month (i.e., as 1/12 = 0.08333333).

[53] http://stackoverflow.com/a/20133091

Basic forecasting

We saw forecasting as a section within the *Chapter 1* example. For the sake of keeping ideas together, we'll replay that forecasting example here.

An excellent companion to *doing* forecasting is *understanding* it: for that, I recommend Rob Hyndman's excellent online text, *Forecasting: Principles and Practice*[54].

```
require(forecast)
require(dplyr)

### Data download and prep, same as in Chapter 1 ###
download.file("http://archive.ics.uci.edu/ml/machine-learning-databases
  /00235/household_power_consumption.zip", destfile =
  "~/BIWR/Chapter1/Data/household_power_consumption.zip")
unzip("~/BIWR/Chapter1/Data/household_power_consumption.zip", exdir="Data")
power = read.table("~/BIWR/Chapter1/Data/household_power_consumption.txt",
  sep=";", header=T, na.strings=c("?",""), stringsAsFactors=FALSE)

power$Date = as.Date(power$Date, format="%d/%m/%Y")
power$Month = format(power$Date,"%Y-%m")
power$Month = as.Date(paste0(power$Month, "-01"))

power$Global_active_power_locf = na.locf(power$Global_active_power)

power_group = group_by(power, Month)
power_monthly = summarize(power_group,
  Total_Use_kWh = sum(Global_active_power_locf)/60)
power_monthly = power_monthly[2:47,]

total_use_ts = ts(power_monthly$Total_Use_kWh, start=c(2007,1), frequency=12)
```

The main function of the `forecast` package is `forecast`; you can also be more specific by calling `forecast.ets` or `auto.arima` for explicitly forecasting the time series based on the exponential smoothing model family or ARMIA models, respectively.

Once you have a time series object, forecasting is a breeze:

```
# Automatically obtain the forecast for the next 6 months
total_use_fc = forecast(total_use_ts, h=6)
```

[54] https://www.otexts.org/fpp

```
# View the forecast model results
summary(total_use_fc)
plot(total_use_fc)
```

```
Console ~/BIWR/Chapter1/
> summary(total_use_fc)

Forecast method: ETS(A,N,A)

Model Information:
ETS(A,N,A)

Call:
 ets(y = object, lambda = lambda)

  Smoothing parameters:
    alpha = 0.0613
    gamma = 3e-04

  Initial states:
    l = 829.9356
    s=257.3091 165.2708 45.671 -82.8744 -375.8192 -255.0943
           -144.7483 -20.4648 -23.0432 97.3446 88.7408 247.708

  sigma:  72.9475

     AIC     AICc      BIC
598.7736 612.3220 624.3746

Error measures:
                    ME     RMSE      MAE       MPE     MAPE     MASE        ACF1
Training set -12.79661 72.94752 54.94647 -3.944092 9.466253 0.653263 -0.09482945

Forecasts:
         Point Forecast    Lo 80     Hi 80    Lo 95     Hi 95
Nov 2010       959.1288 865.6428 1052.6148 816.1543 1102.1033
Dec 2010      1051.1827 957.5214 1144.8439 907.9402 1194.4252
Jan 2011      1041.6224 947.7862 1135.4586 898.1124 1185.1324
Feb 2011       882.5943 788.5836  976.6051 738.8173 1026.3714
Mar 2011       891.2219 797.0369  985.4069 747.1783 1035.2655
Apr 2011       770.7830 676.4241  865.1420 626.4734  915.0926
>
```