The New EDA

Brad Luen

2017-07-21

Contents

Chapter 1

What's EDA?

In S320/520 we spent most of our time estimating and testing hypotheses and models. In S470/670 we'll spend most of our time coming up with hypotheses and models for data. In practice this means we'll draw a lot of graphs and we won't have to calculate any P-values.

Chapter 2

One quantitative variable

2.1 Learning ggplot2

READ: Cleveland pp. 16–31.

2.1.1 Univariate plots: Choral singers again

In S320/520 we learned the basic plots of one quantitative variable:

- ECDF plots
- Histograms/density
- Boxplots
- Normal QQ plots

We'll learn how to redraw all these plots using the `ggplot2` package because it's a more complete "grammar" of graphics, but mostly because it produces much more attractive graphs than base R.

If you've taken my S320/520 class you'll remember we use a data set containing the heights (in inches) of members of the New York Choral Society. That data is in the `lattice` package in R, so load that:

```
library(lattice)
```

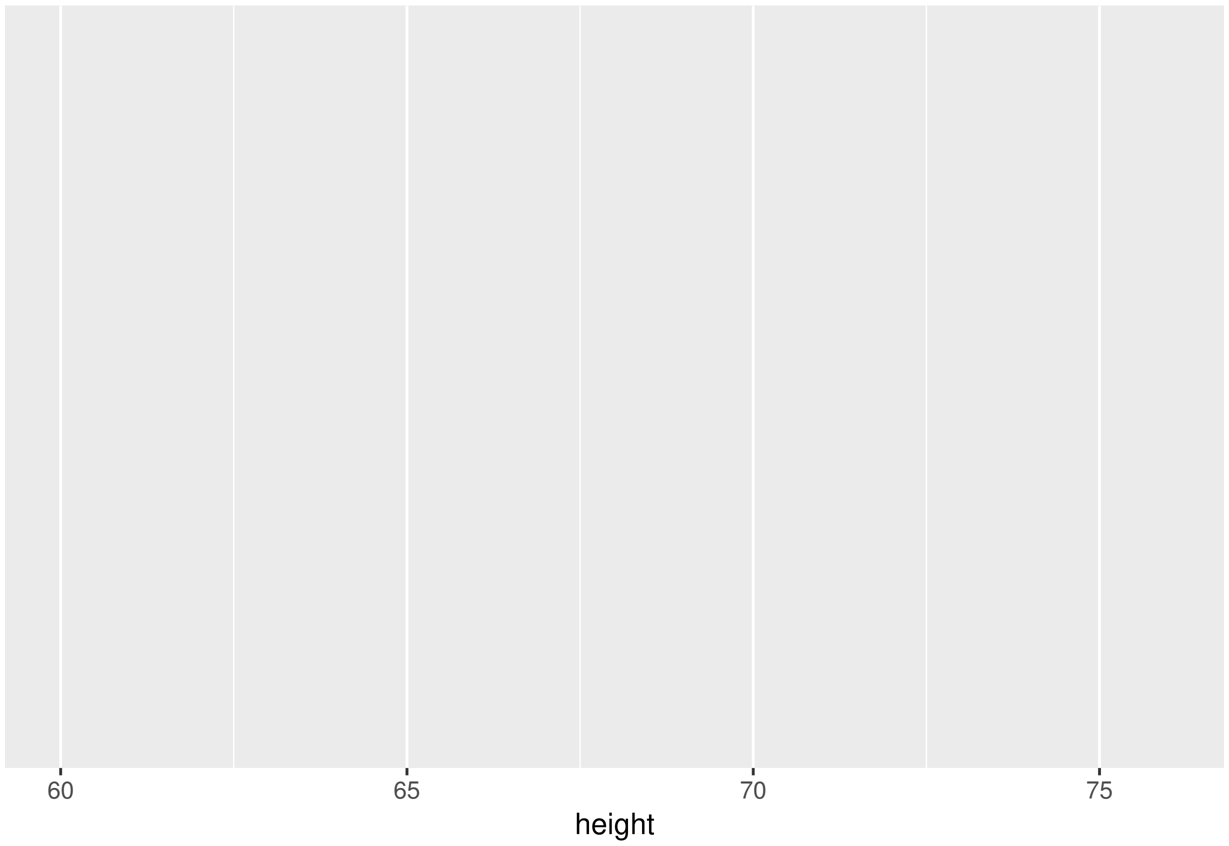
We'll need the `ggplot2` library, of course. If you have't yet installed it, then do so: `install.packages("ggplot2")`. Then load it:

```
library(ggplot2)
```

2.1.2 ggplot syntax

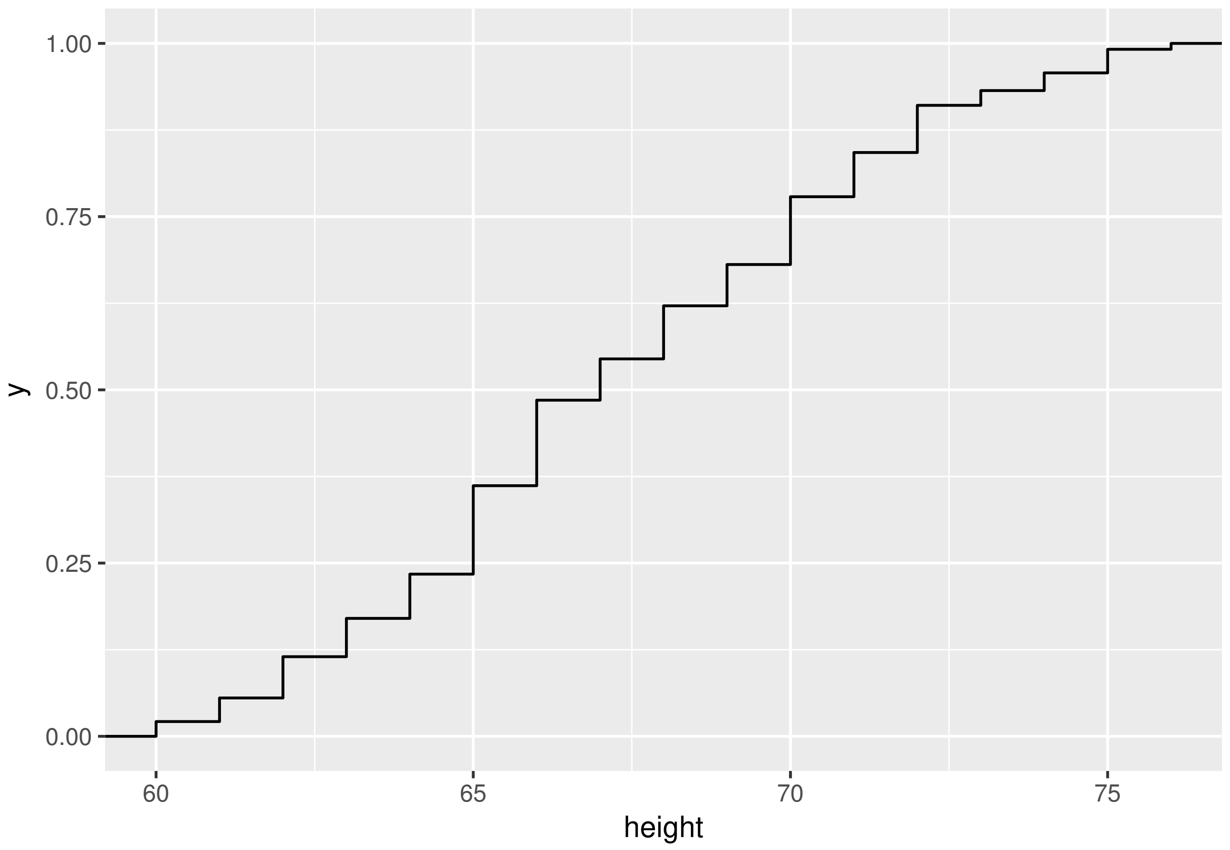
The first thing to get used to with `ggplot` syntax is the `aes` (short for aesthetic) function. Use this when you have a data frame and want to tell R what kind of variables everything is (e.g. x, y.) For example, suppose we want to draw a graph using `height` from the `singer` data frame as our x-variable. We use the following syntax:

```
ggplot(singer, aes(x = height))
```



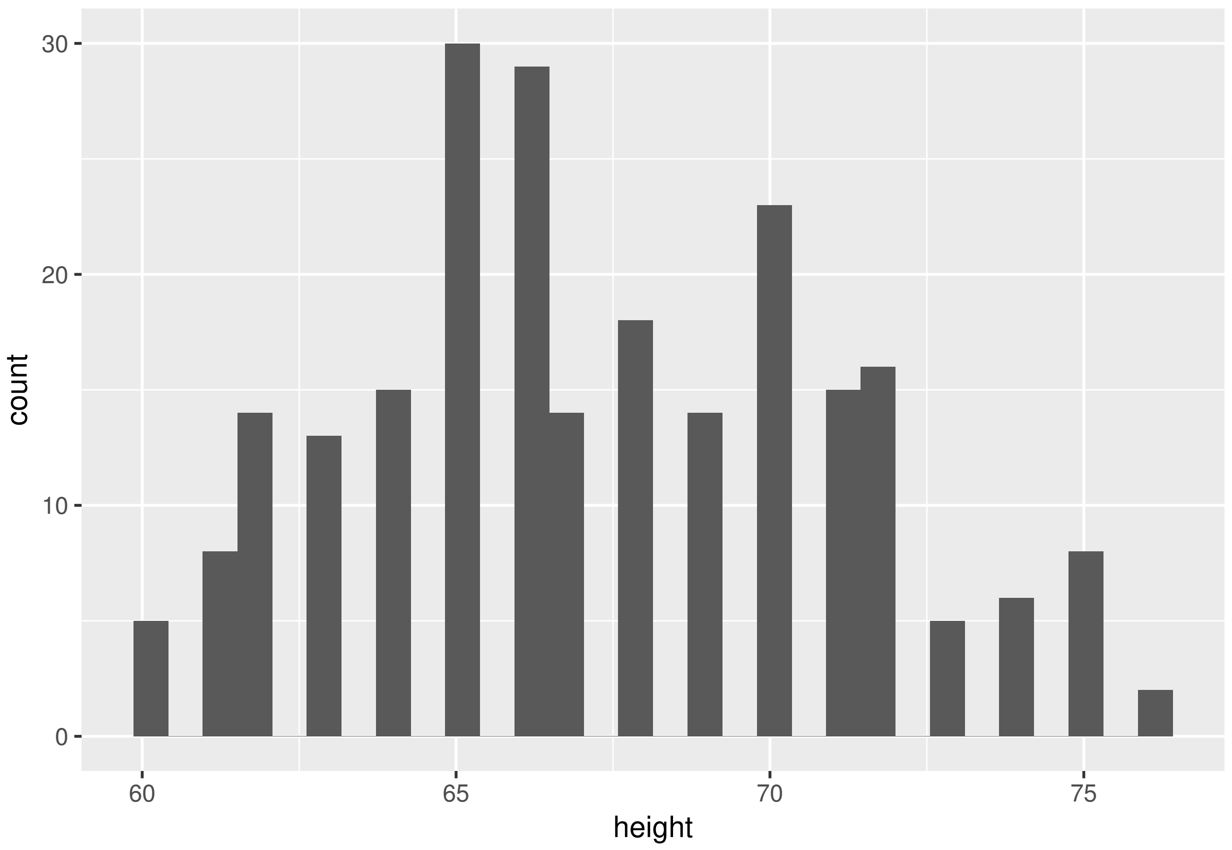
Now this doesn't actually draw a graph because we haven't specified what kind of graph we want to draw. Let's draw an *ECDF plot*. Recall that the ECDF at x is the proportion of the data less than or equal to x. All we need to do is add `+ stat_ecdf()` to our previous command:

```r
ggplot(singer, aes(x = height)) + stat_ecdf()
```



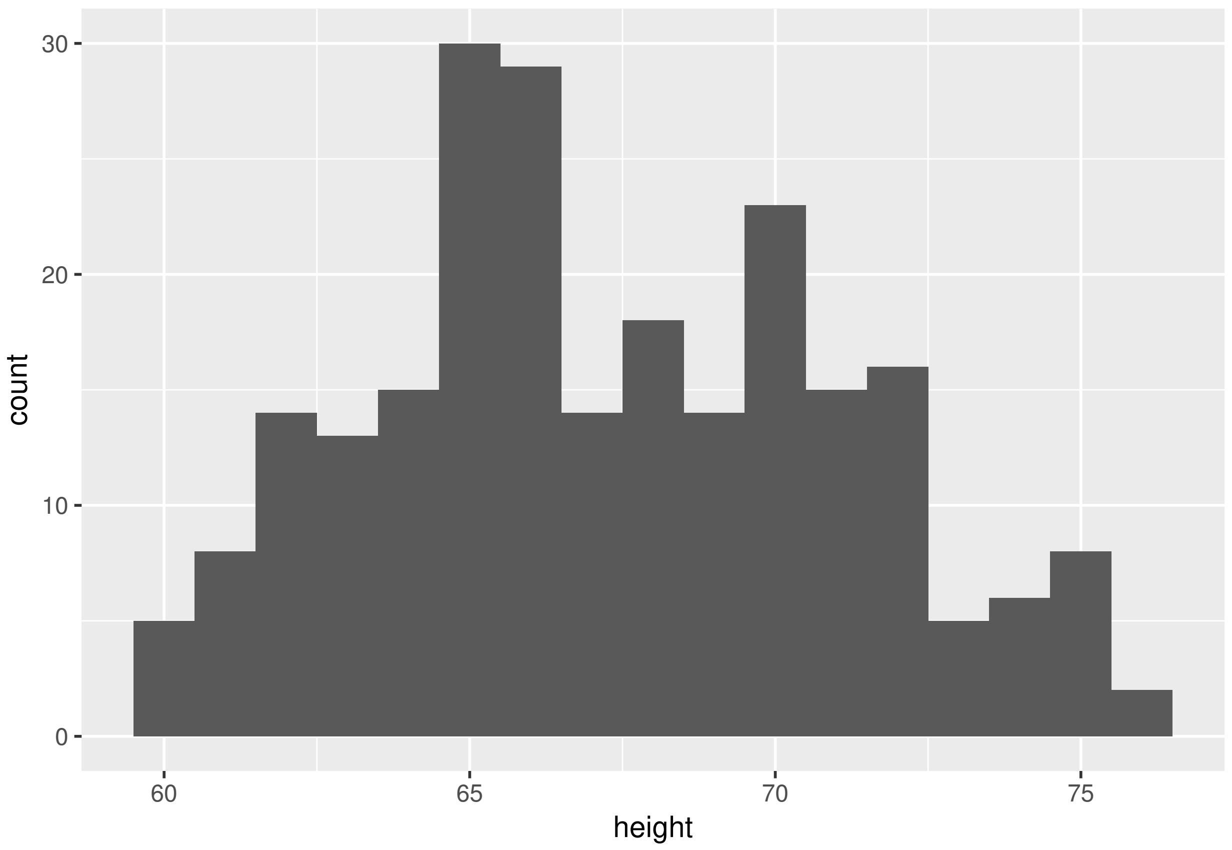
The CDF is not an object for which many people have intuition, so it's generally more useful to do a *histogram* or a *density plot*. Usually the density plot is a better estimate of the PDF, while the histogram gives you a clearer idea of the variability in the data. For a histogram:

```r
ggplot(singer, aes(x = height)) + geom_histogram()
```



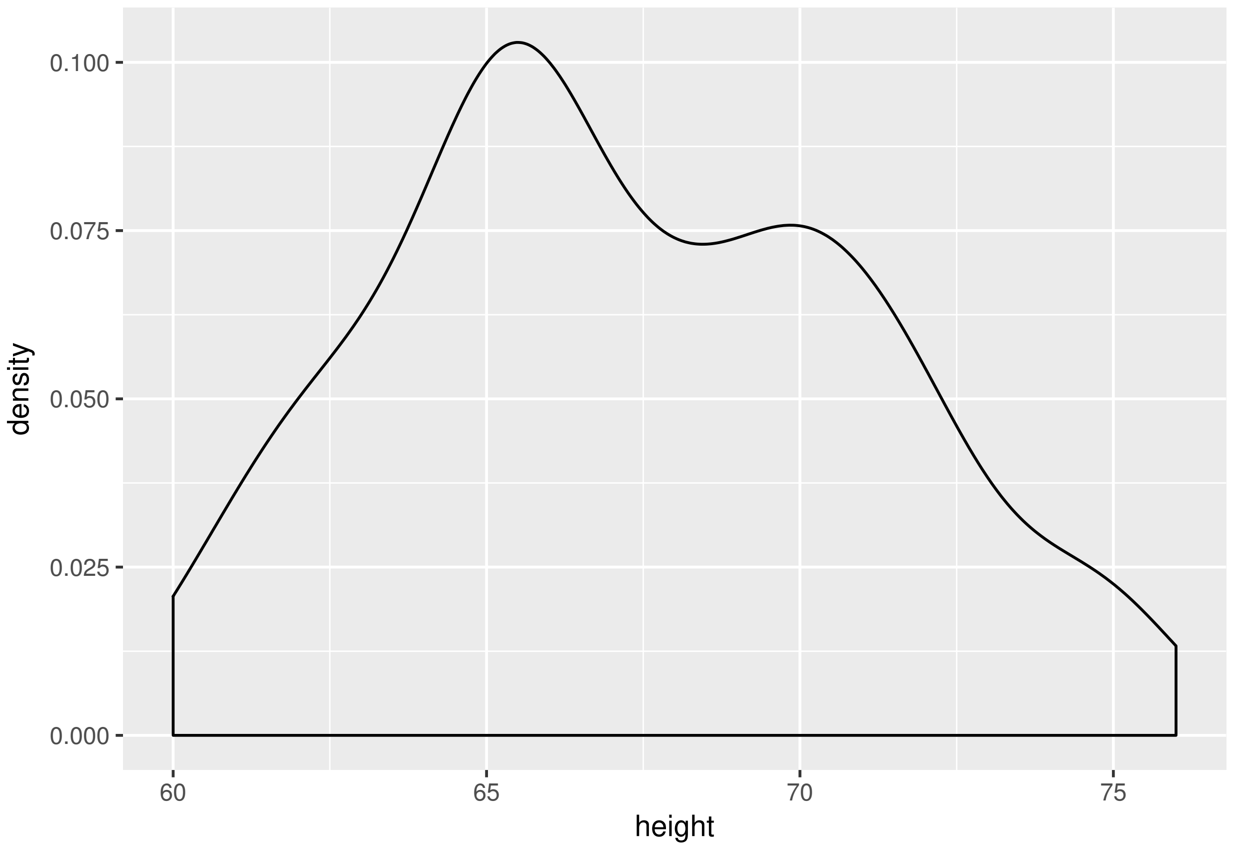
This is ugly. Since we know the heights are measured to the nearest inch, let's specify bins of width 1 inch:

```
ggplot(singer, aes(x = height)) + geom_histogram(binwidth = 1)
```



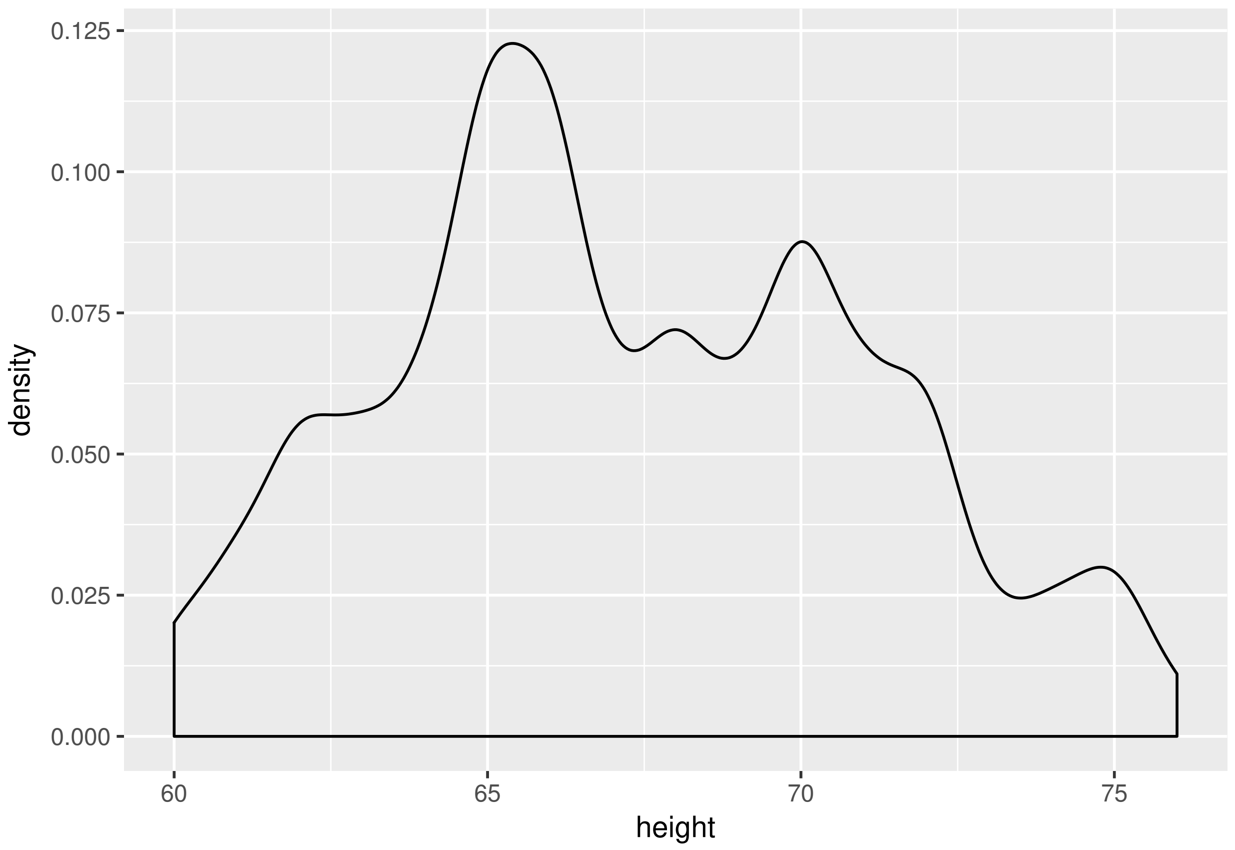
If you prefer using a density plot to estimate the PDF:

```r
ggplot(singer, aes(x = height)) + geom_density()
```



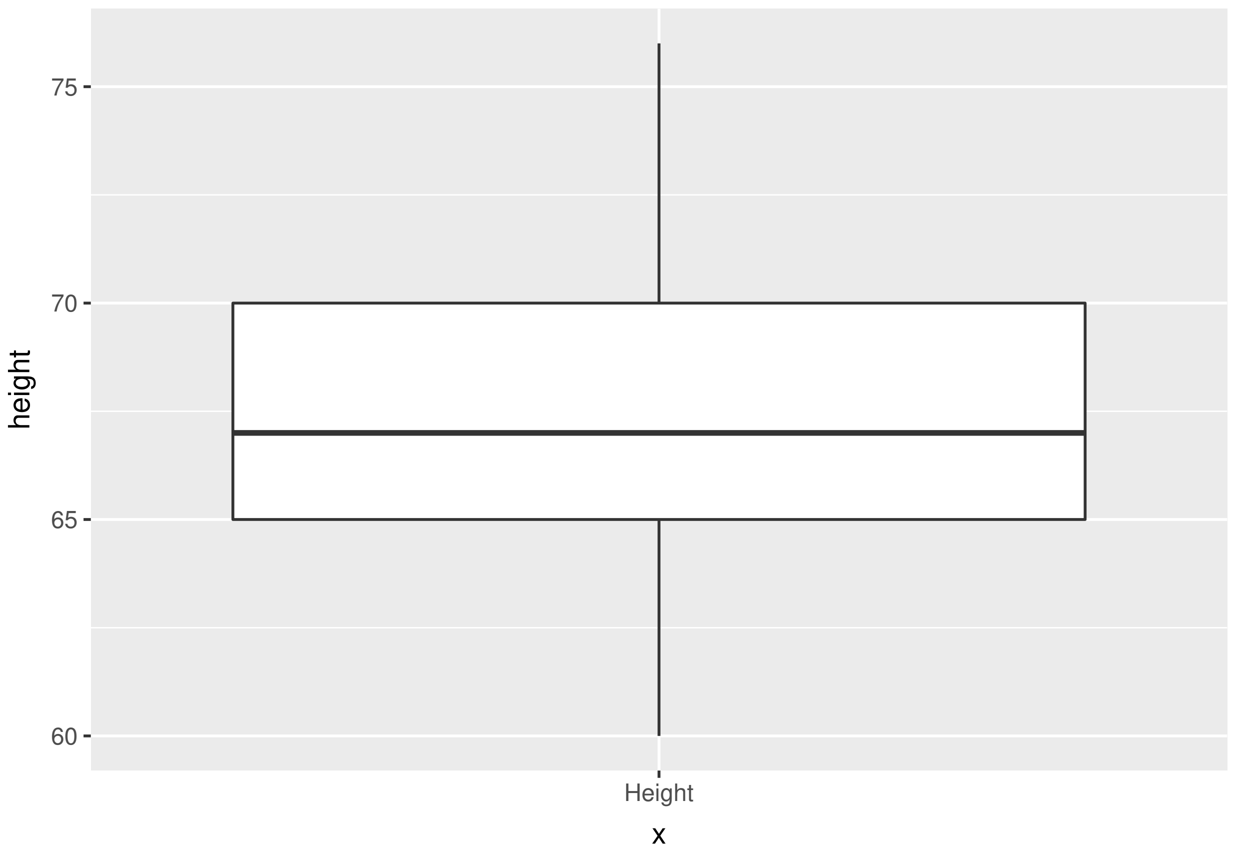
If you don't like your density plot, use `adjust` as an argument within `geom_density()` to change the bandwidth. An adjust value of less than 1 makes the plot less smooth, while a value of 1 makes it smoother:

```
ggplot(singer, aes(x = height)) + geom_density(adjust = 0.5)
```



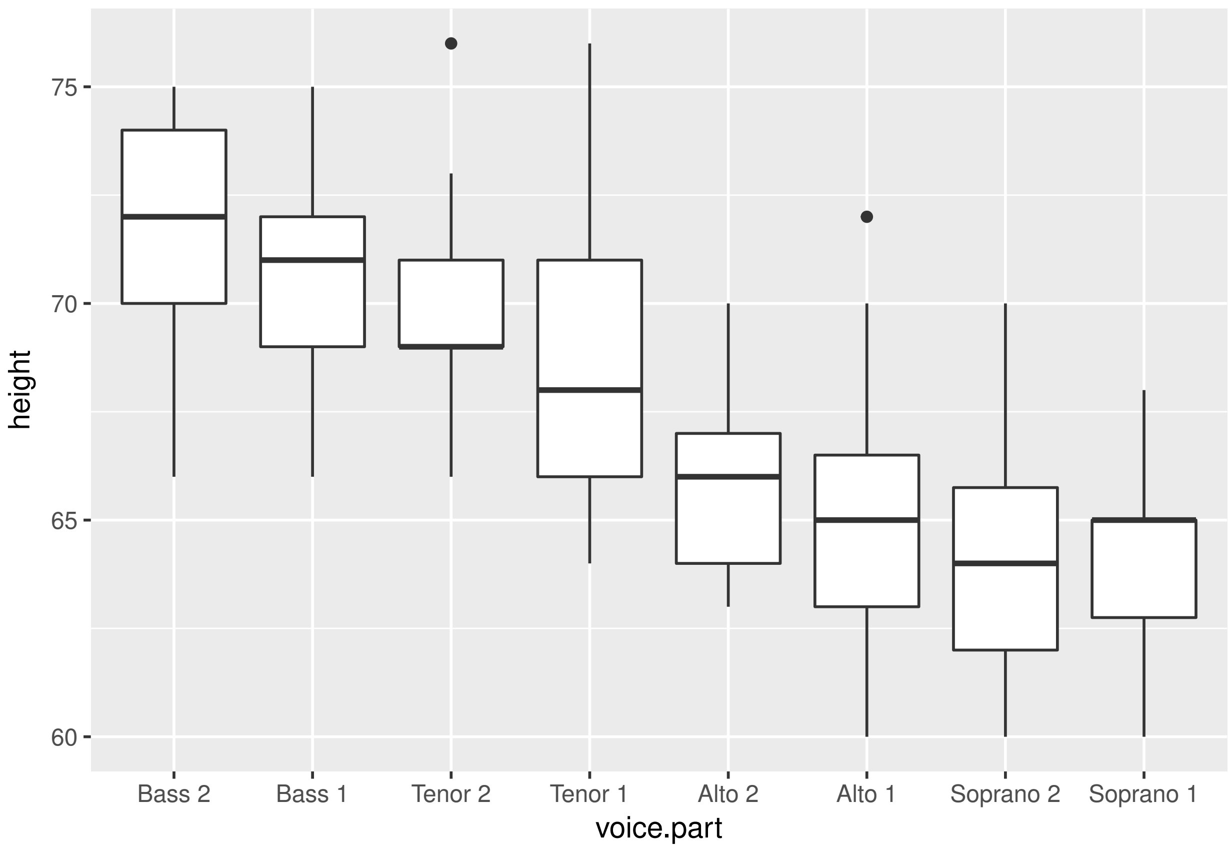
Recall that a *boxplot* gives the five-number summary for a set of data: minimum excluding outliers, first quartile, median, third quartile, and maximum excluding outliers. Drawing a single boxplot is a bit trickier because it requires both an x- and a y-variable. Here's a workaround:

```
ggplot(singer, aes(x = "Height", y = height)) + geom_boxplot()
```



Note that drawing one boxplot by itself is usually not very useful. It's much better to draw a number of boxplots to compare them. We can draw a boxplot of heights for each voice part:

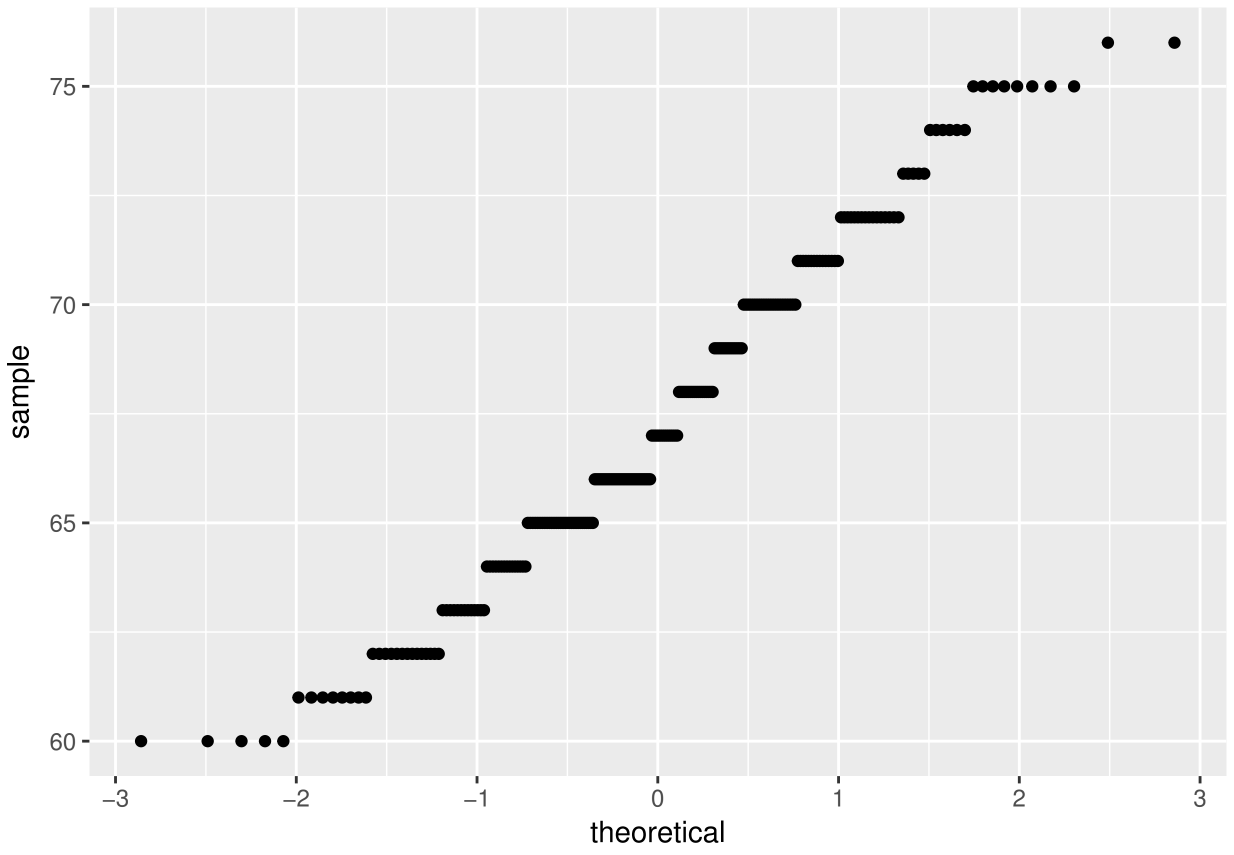
```
ggplot(singer, aes(x = voice.part, y = height)) + geom_boxplot()
```



2.1.3 Normal and uniform QQ plots

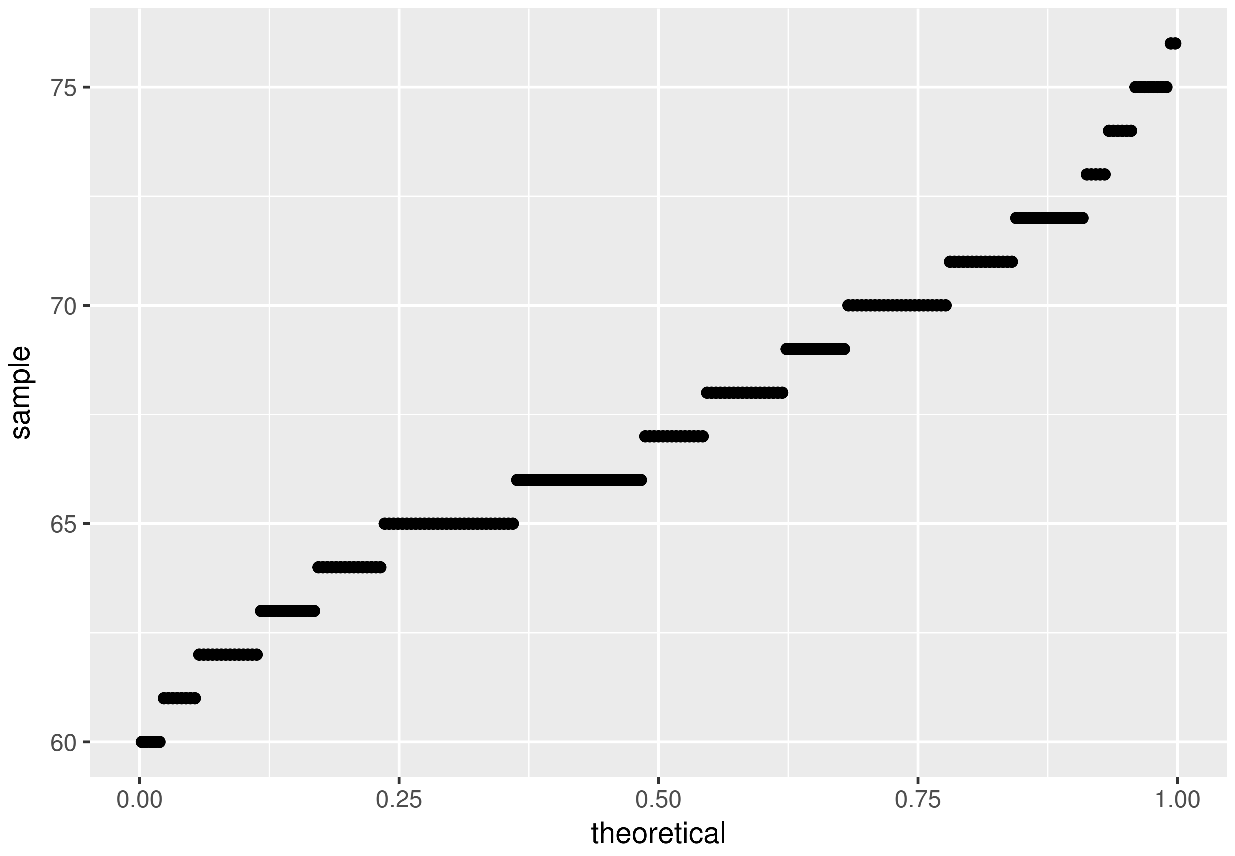
A normal QQ plot is a graph of the quantiles of a data set against the quantiles of a standard normal distribution. Here it's not as simple as setting `height` to be x or y: we need to use the heights as a *sample* from which we estimate quantiles.

```
ggplot(singer, aes(sample = height)) + stat_qq()
```



But there's no reason why the reference distribution has to be normal. In fact, it may be preferable to use a uniform as the reference.

```r
ggplot(singer, aes(sample = height)) + stat_qq(distribution = qunif)
```

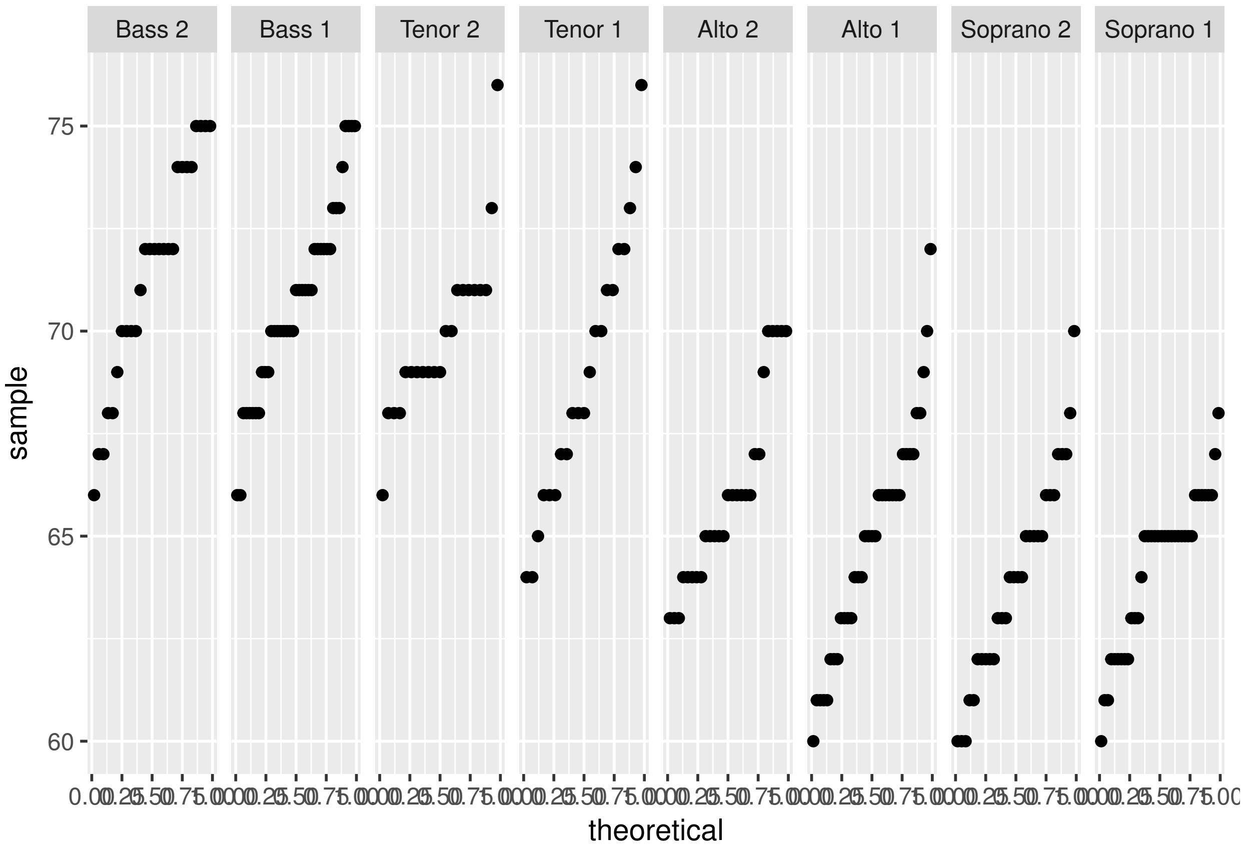


Note that this uniform QQ plot is *almost* the same as the ECDF graph with its axes flipped.

2.1.4 Faceting

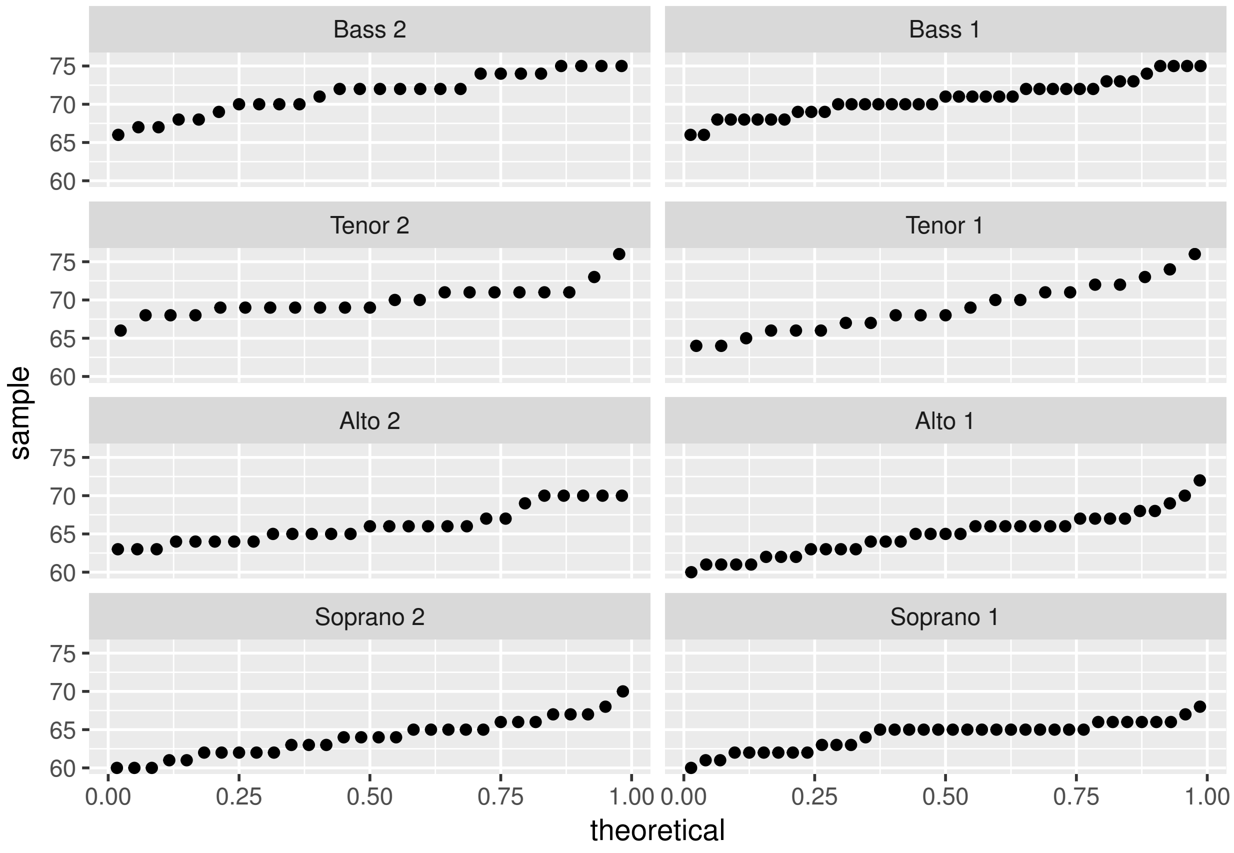
Faceting essentially means breaking up your data into subsets, then plotting those subsets separately. Suppose we want to draw a uniform QQ plot of singer heights for each voice part. We can use `facet_grid`:

```
ggplot(singer, aes(sample = height)) + stat_qq(distribution = qunif) + facet_grid(~voice.part)
```



Because we have eight different voice parts, the graph is cramped. It'll look better if we draw a grid (say 4-by-2) of graphs. `facet_wrap` lets us draw this:

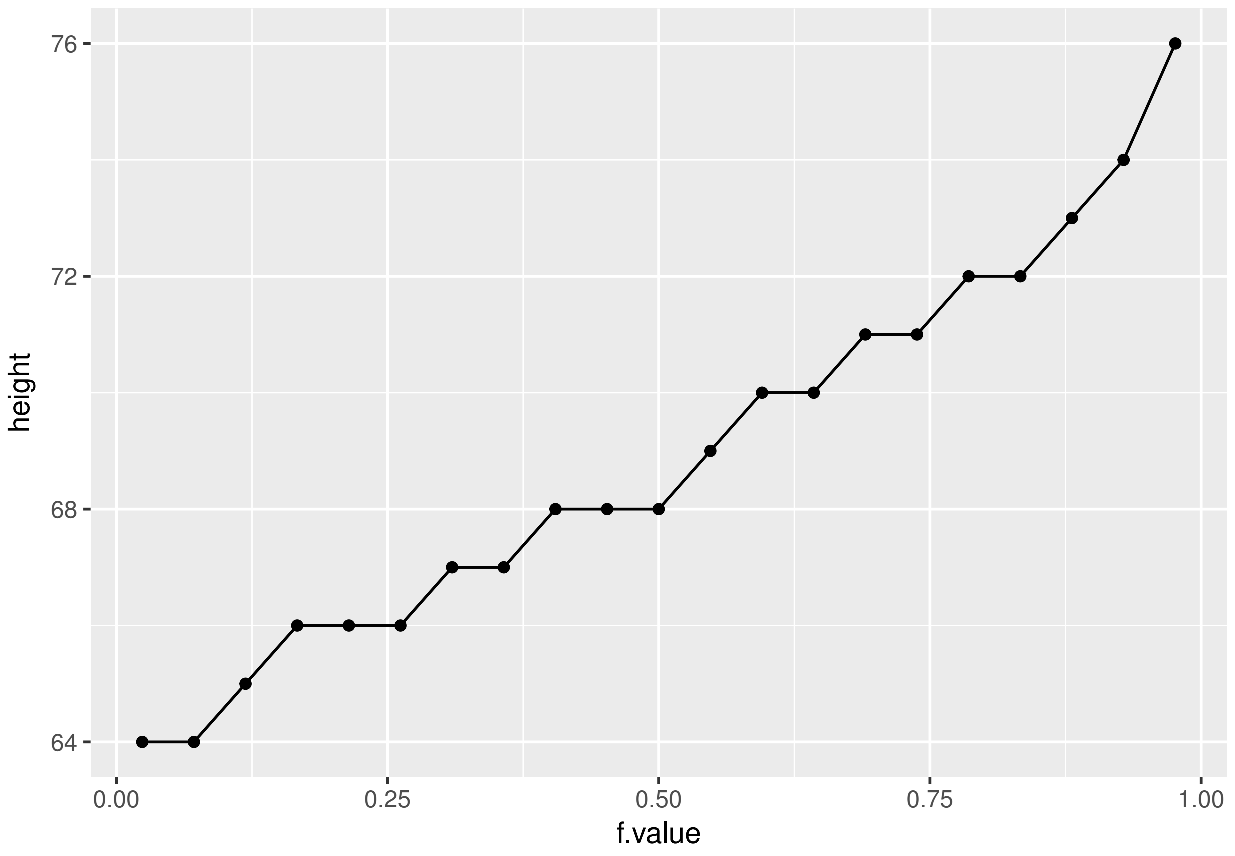
```
ggplot(singer, aes(sample = height)) + stat_qq(distribution = qunif) + facet_wrap(~voice.part,
    ncol = 2)
```



2.1.5 Constructing a uniform QQ plot manually

Here we reproduce Cleveland figure 2.2. This requires first sorting the Tenor 1 heights, then finding their corresponding quantiles ("f-values".)

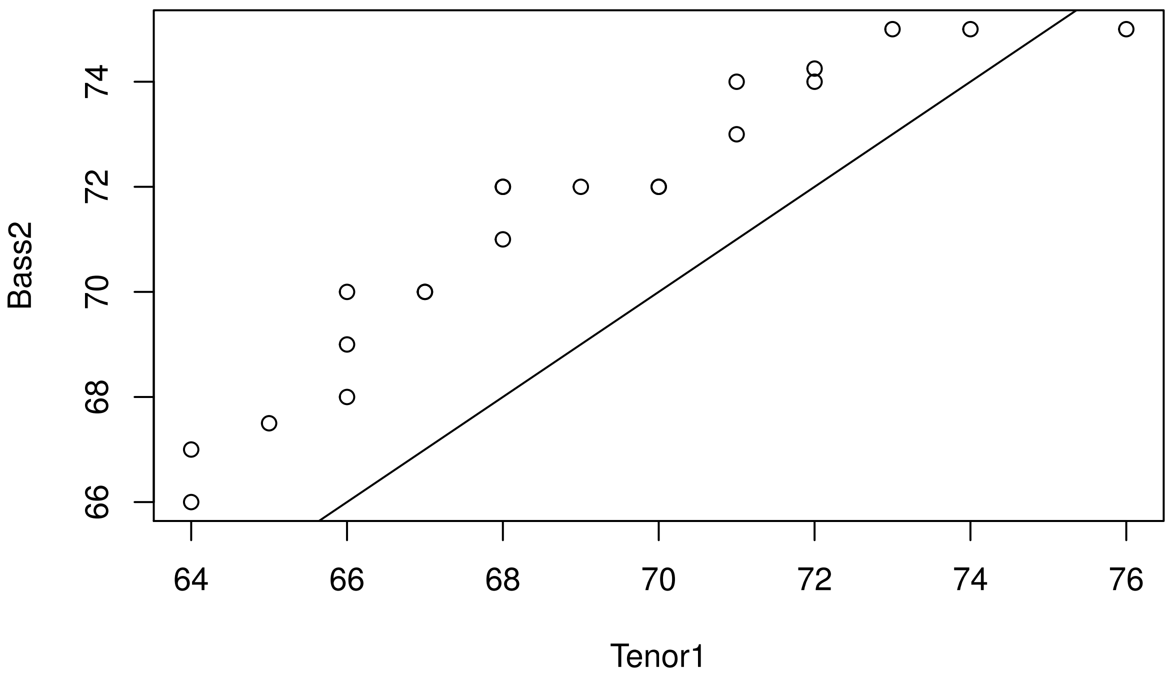
```
Tenor1 = sort(singer$height[singer$voice.part == "Tenor 1"])
nTenor1 = length(Tenor1)
f.value = (0.5:(nTenor1 - 0.5))/nTenor1
Tenor1.df = data.frame(f.value, height = Tenor1)
ggplot(Tenor1.df, aes(x = f.value, y = height)) + geom_line() + geom_point()
```



2.1.6 Two-sample QQ plots

Our QQ plots so far have plotted quantiles of one sample (at a time) against quantiles of a reference distribution. The `qqplot()` function in base R plots quantiles of one variable against quantiles of another. (Note that it does not require the data sets to be sorted.)

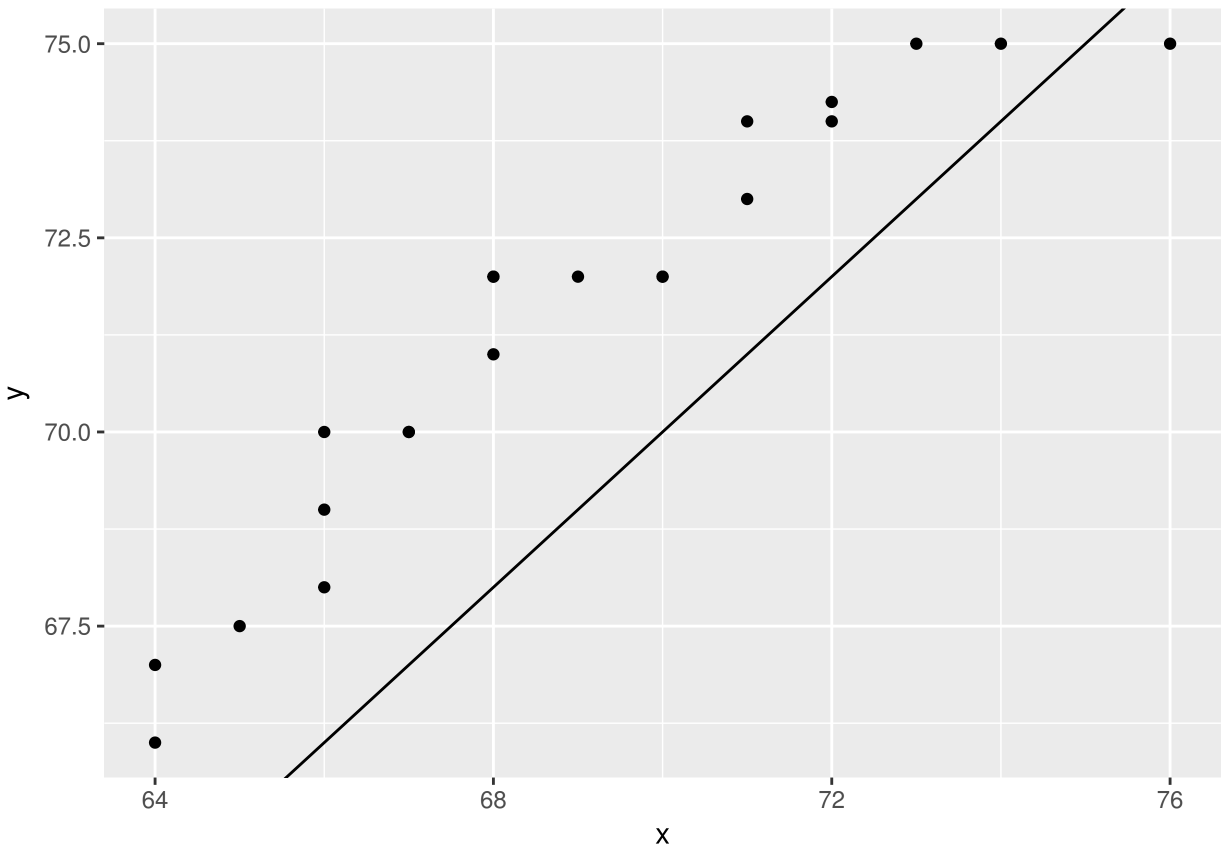
```
Tenor1 = singer$height[singer$voice.part == "Tenor 1"]
Bass2 = singer$height[singer$voice.part == "Bass 2"]
qqplot(Tenor1, Bass2)
abline(0, 1)
```



The points are well-approximated by a straight line *parallel* to the line $y = x$. This suggests the Tenor 1 and Bass 2 distributions might differ by an *additive shift**: they have similar shapes and spreads, but different centers. As the points are almost all above the line by 2–3 units, Bass 2 singers are on average 2–3 inches taller than Tenor 1 singers.

We can feed the quantiles calculated by `qqplot()` into a data frame, which we can then feed into ggplot. We'll represent the quantiles as points (`geom_point()`) then add a diagonal line (`geom_abline()`):

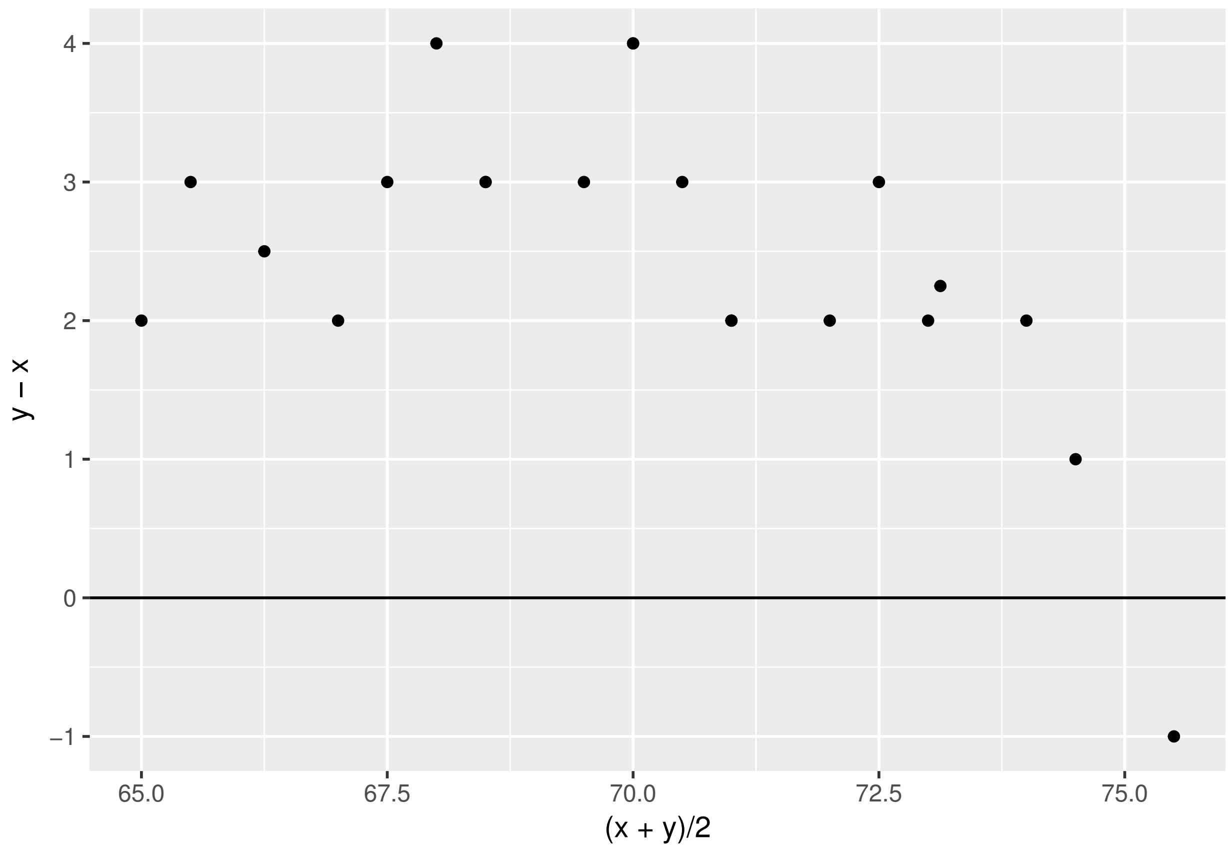
```
Tenor1 = singer$height[singer$voice.part == "Tenor 1"]
Bass2 = singer$height[singer$voice.part == "Bass 2"]
qq.df = as.data.frame(qqplot(Tenor1, Bass2, plot.it = FALSE))
ggplot(qq.df, aes(x = x, y = y)) + geom_point() + geom_abline()
```



2.1.7 Extra: Tukey mean-difference plots

Humans can more accurately compare to a horizontal line than a diagonal one. The *Tukey mean-difference plot* is designed to take advantage of this. On the x-axis goes the *mean* of the two quantiles (averaging Tenor 1 and Bass 2), while on the y-axis goes the difference between them (Bass 2 minus Tenor 1.)

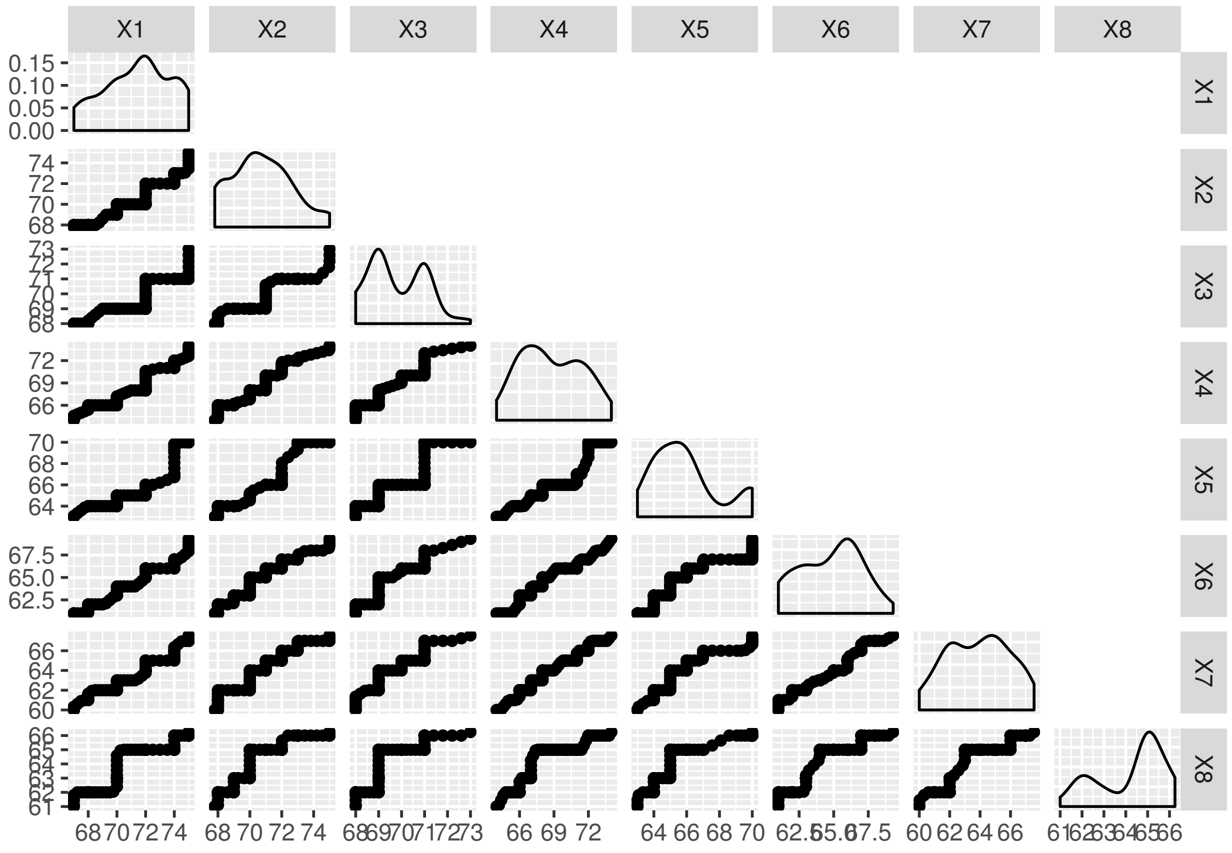
```
ggplot(qq.df, aes(x = (x + y)/2, y = y - x)) + geom_point() + geom_abline(slope = 0)
```



2.1.8 Extra: Pairwise QQ plots

If you want to do these, a good choice is to use the `ggpairs()` function in the `GGally` library. The plots on the diagonal are density plots. The graphs are a bit too busy to be very useful in my opinion.

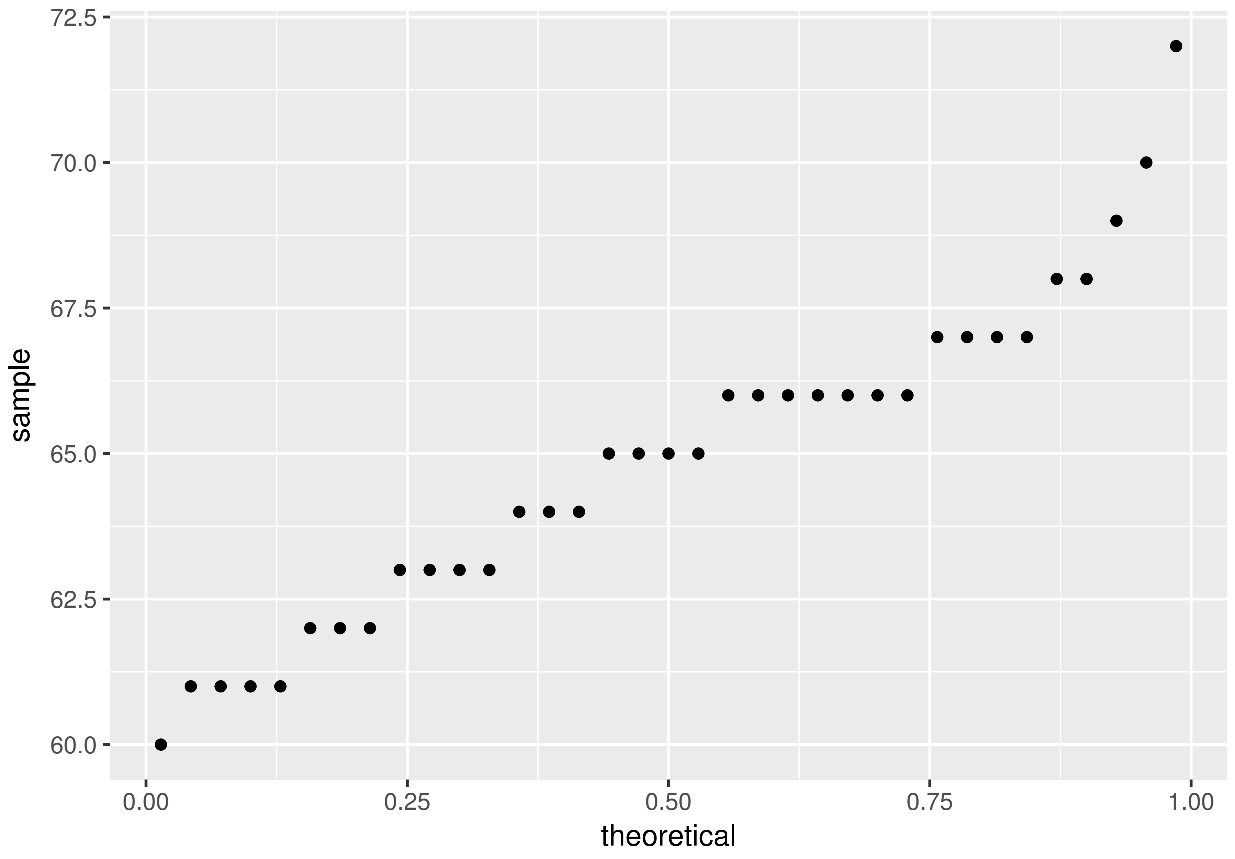
```
singer.q.rows = aggregate(height ~ voice.part, quantile, probs = seq(0.05, 0.95,
    0.01), data = singer)
singer.q = t(singer.q.rows[-1])
names(singer.q) = singer.q.rows[, 1]
singer.q = data.frame(singer.q)
singer.panel = function(x, y) {
    lines(x, y, xlim = range(singer$height), ylim = range(singer$height))
    abline(0, 1)
}
library(GGally)
ggpairs(singer.q, upper = c())
```



2.1.9 Normality revisited

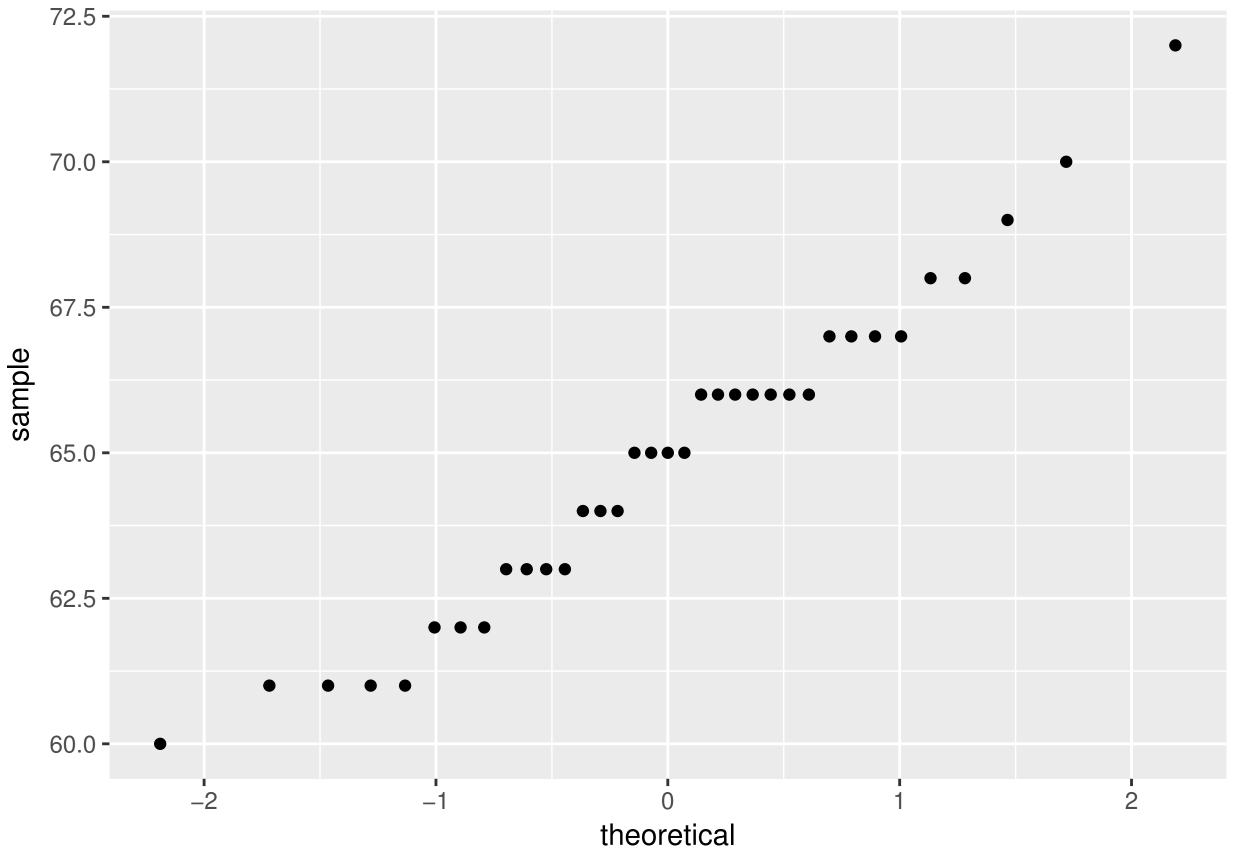
The normal QQ plot for the data as a whole doesn't look quite normal. What if we just looked at Alto 1s? First use the uniform as a reference:

```
ggplot(singer[singer$voice.part == "Alto 1", ], aes(sample = height)) + stat_qq(distribution = qunif)
```



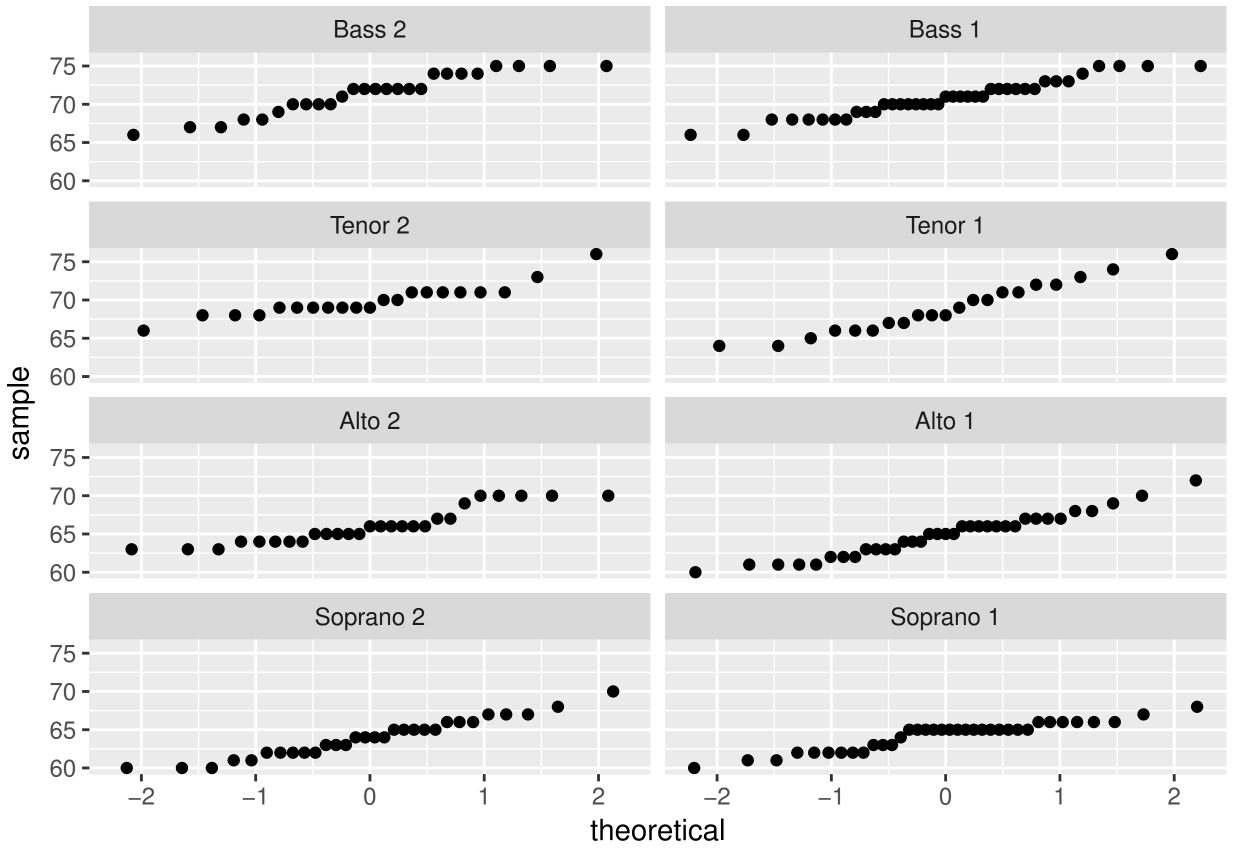
It looks like the line curves up at the end. Now let's compare to the normal:

```r
ggplot(singer[singer$voice.part == "Alto 1", ], aes(sample = height)) + stat_qq()
```



As before, we can facet the graphs to draw one for each voice part:

```
ggplot(singer, aes(sample = height)) + stat_qq() + facet_wrap(~voice.part, ncol = 2)
```



The graphs generally look straightish. It seems fair to say that as far as we can tell, each voice part's heights are reasonably well approximated by a normal distribution. (As with most normals, this is an approximation that's most accurate in the center of the distribution: there's no evidence that the tails are normal and no theoretical reason to think they should be.) The next questions to answer: What kinds of normal distributions? What are the means? Do they all have the same standard deviation, or do the spreads differ?

2.2 Fits and residuals

READ: Cleveland pp. 34–41.

Load our standard libraries:

```
library(lattice)
library(ggplot2)
```

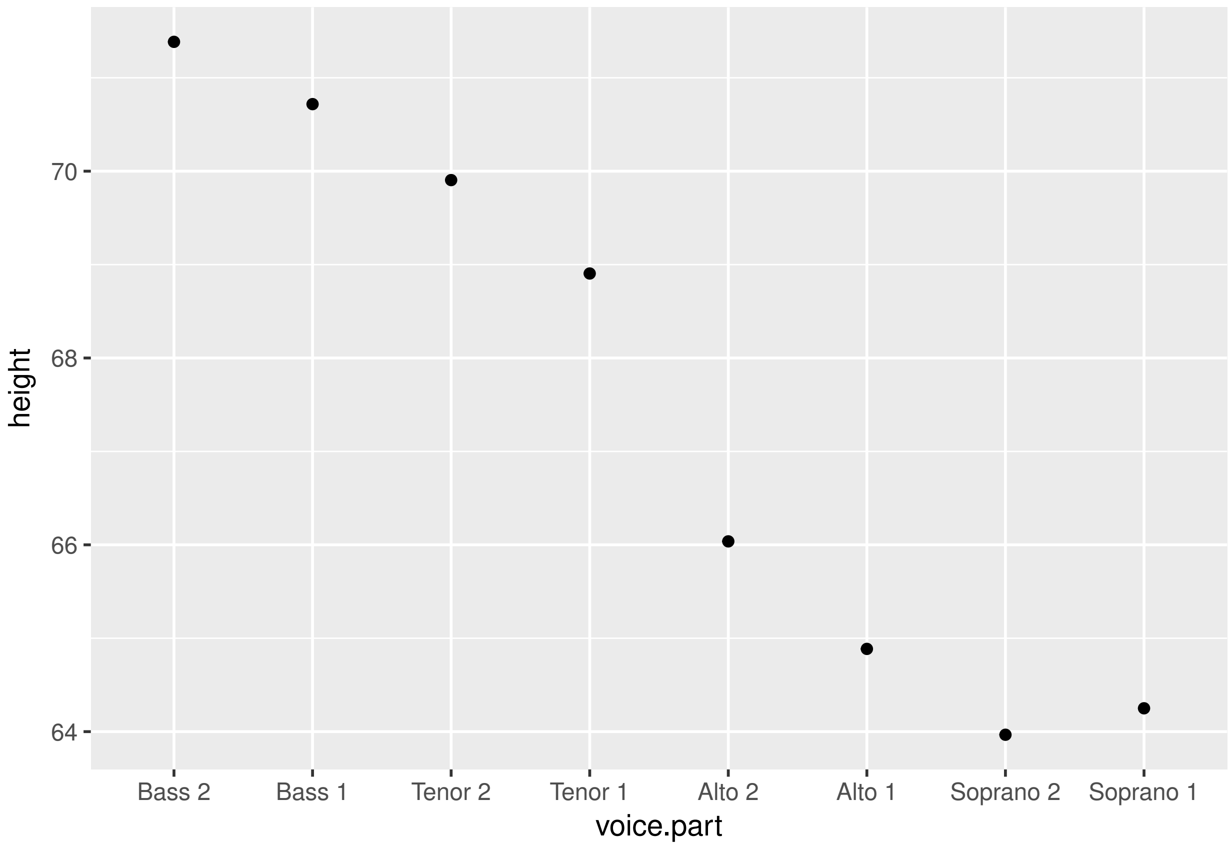
2.2.1 Aggregation and dot plots

Last time we decided that the singer heights for each vocal part were well-approximated by normal distributions. Now we ask: What are the parameters of those distributions? Firstly, what are the mean heights for Soprano 1s, Soprano 2s, etc.? As you learned a long time ago, the "best" estimate of the population mean is the sample mean. We can use the `aggregate()` function to easily find the sample mean for each voice part.

```
singer.means = aggregate(height ~ voice.part, FUN = mean, data = singer)
```

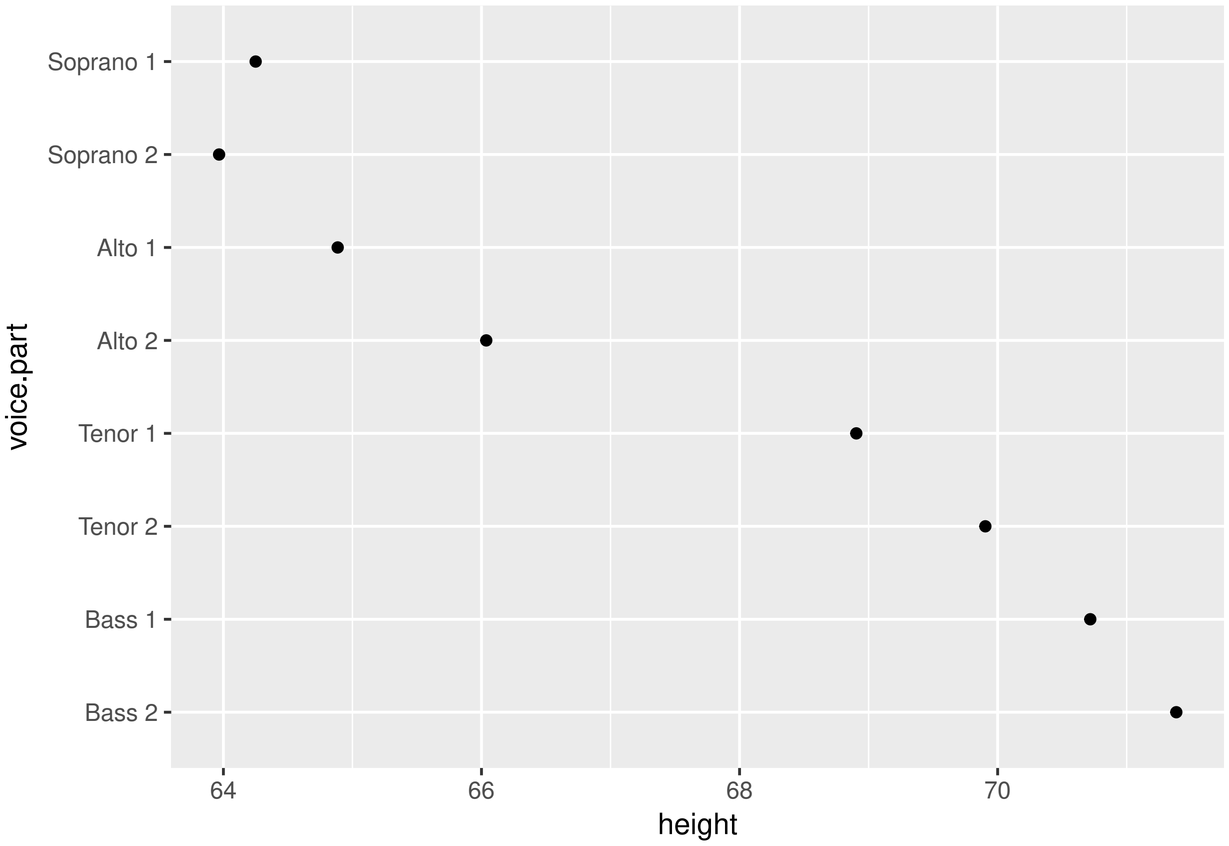
We now have one number for each of a set of categories. While this kind of data is traditionally displayed using a bar graph, a **dot plot** is arguably preferable, because they use position and not area to represent numbers. This is good because (i) humans are better at judging position than area, and (ii) for positions you don't have to start your axis at zero, while for areas you do. The `ggplot` function is `geom_point()`:

```r
ggplot(singer.means, aes(x = voice.part, y = height)) + geom_point()
```



Note that you can also draw box plots the other way around (the choice of axes is personal preference):

```r
ggplot(singer.means, aes(x = height, y = voice.part)) + geom_point()
```



2.2.2 Models and residuals

We now have a simple model for singer heights.

Singer height = Average height for their voice part + some error

If you've taken S431/631 or a similar regression course, you might recognize this as a special case of a linear model. If you haven't, well, it doesn't really matter much except we can use the `lm()` function to fit the model. The advantage of this is that `lm()` easily splits the data into **fitted values** and **residuals**:

Observed value = Fitted value + residual

Here the fitted values are just the sample averages for each voice part.

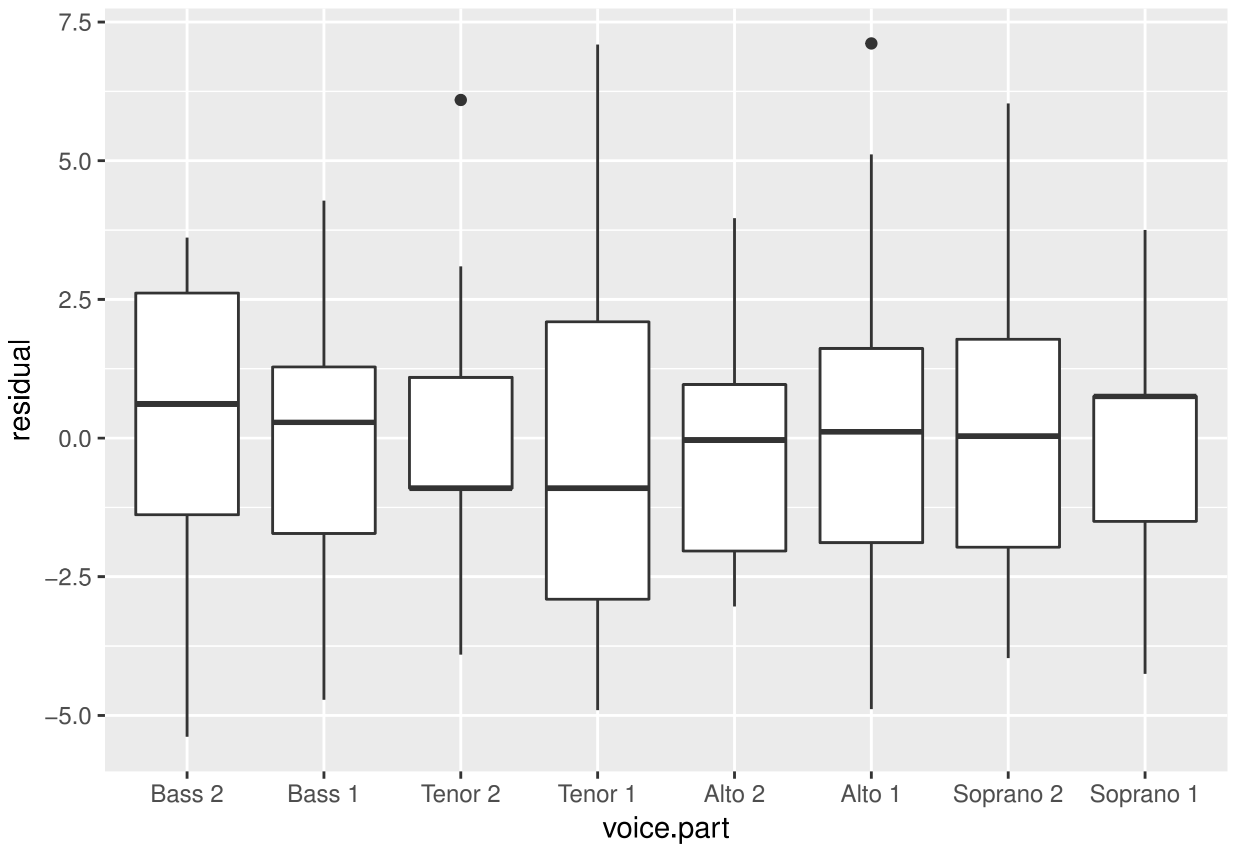
```
singer.lm = lm(height ~ voice.part, data = singer)
```

We can extract the fitted values using `fitted.values(singer.lm)` and the residuals with `residuals(singer.lm)` or `singer.lm$residuals`. For convenience, we create a data frame with two columns: the voice parts and the residuals.

```
singer.res = data.frame(voice.part = singer$voice.part, residual = residuals(singer.lm))
```

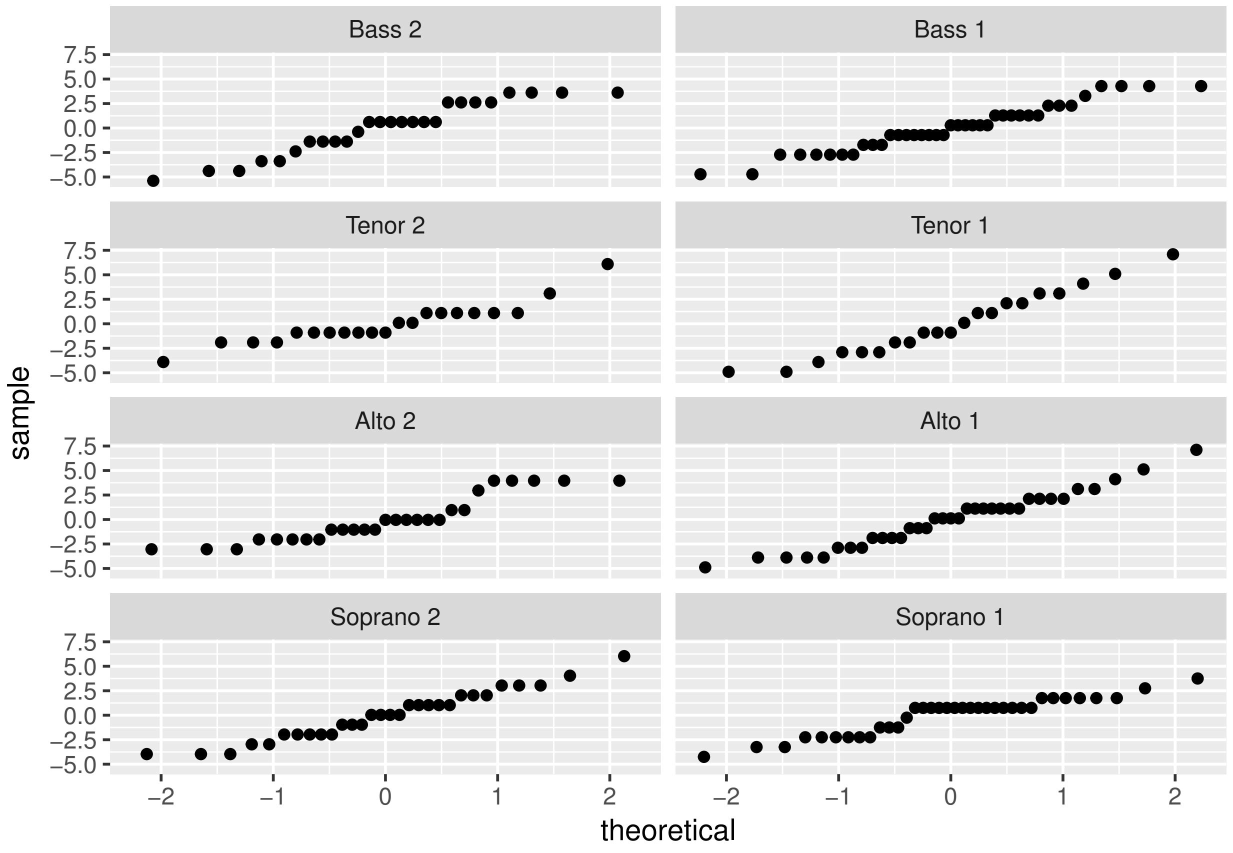
There are a few ways we can look at the residuals. Side-by-side boxplots give a broad overview:

```
ggplot(singer.res, aes(x = voice.part, y = residual)) + geom_boxplot()
```



We also want to examine normality of the residuals, broken up by voice part. We do this by faceting:

```
ggplot(singer.res, aes(sample = residual)) + stat_qq() + facet_wrap(~voice.part,
    ncol = 2)
```

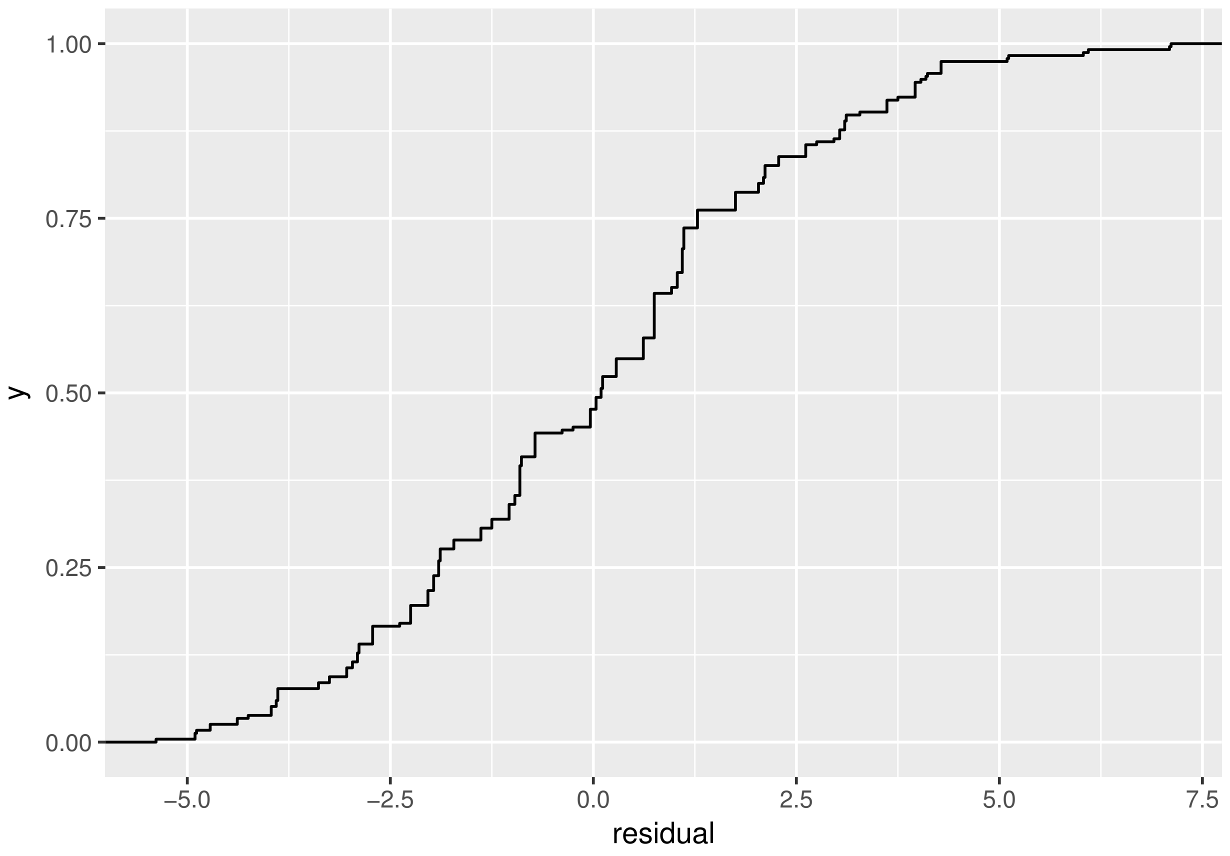


Not only do the lines look reasonably straight, the scales look similar for all eight voice parts. This suggests a model where all of the errors are normal with the *same* standard deviation. We propose this model:

Singer height = Average height for their voice part + Normal$(0, \sigma^2)$ error.

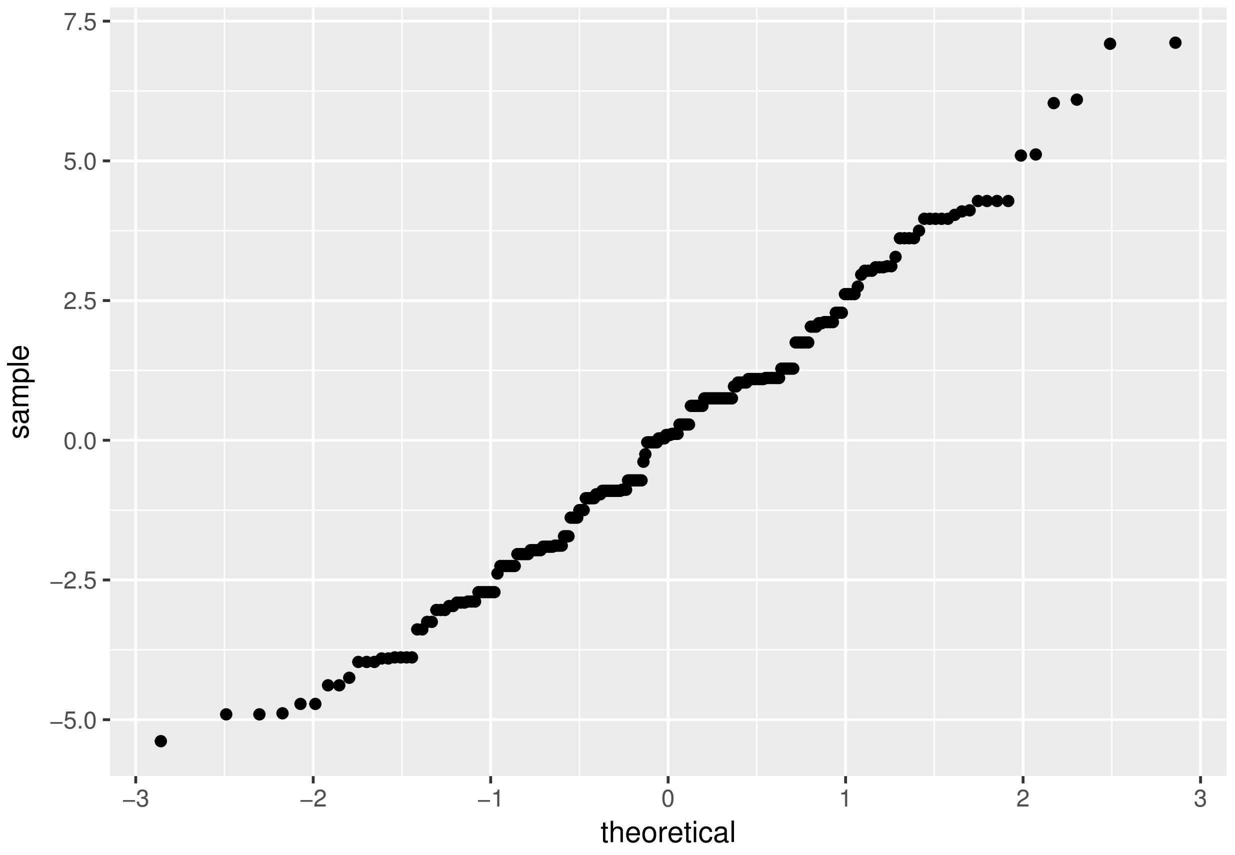
To see if this is a good fit, we **pool** the residuals and plot them. First let's try an ECDF plot:

```r
ggplot(singer.res, aes(x = residual)) + stat_ecdf()
```



A normal distribution would give this S-shape, but so would many other distributions. To check normality directly, as usual we draw a normal QQ plot.

```
ggplot(singer.res, aes(sample = residual)) + stat_qq()
```



We'll also add a line with the mean of the residuals (which should be zero) as the intercept, and the SD of the residuals as the slope.

```
round(mean(singer.res$residual), 3)
```

```
## [1] 0
```

```
round(sd(singer.res$residual), 3)
```

```
## [1] 2.465
```

Pedantic note: We should use an $n - 8$ denominator instead of $n - 1$ in the SD calculation for degrees of freedom reasons. We can get this directly from the linear model:

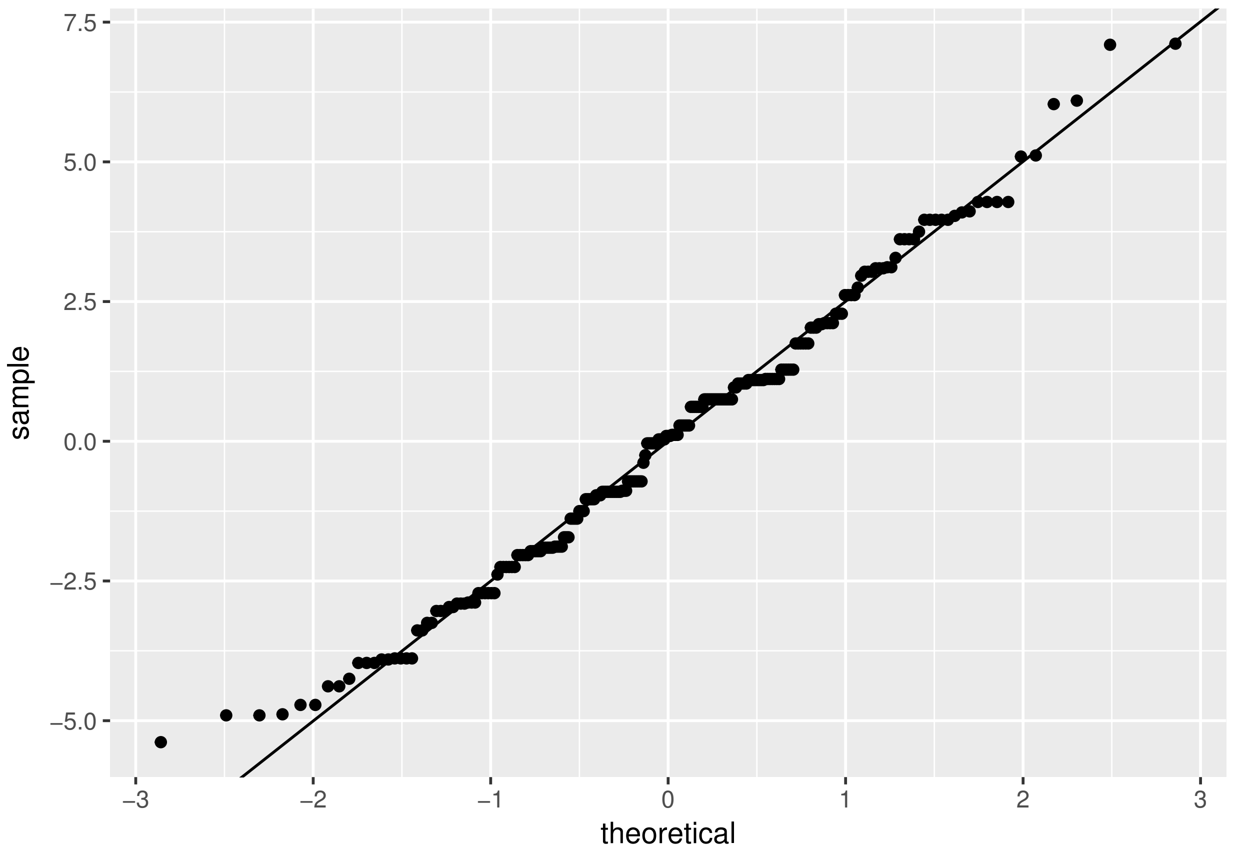
```
round(summary(singer.lm)$sigma, 3)
```

```
## [1] 2.503
```

However, the difference between this and the SD above is negligible.

Add the line:

```
ggplot(singer.res, aes(sample = residual)) + stat_qq() + geom_abline(intercept = 0,
    slope = summary(singer.lm)$sigma)
```



The straight line isn't absolutely perfect, but it's doing a pretty good job. Our model is thus

Singer height = Average height for their voice part + Normal$(0, 2.5^2)$ error.

2.2.3 Residual-fit spread

A useful thing to do (moreso when the predictor is continuous, but still useful here) is to visually compare the spread of the fitted values with the spread of the residuals. This gives an intuitive idea of how much variation is captured by the model fit and how much remains in the residuals. To do this, we want to draw two panels next to each other on the same scale: a uniform QQ plot of the fitted values (after subtracting the overall mean) and a uniform QQ plot of the residuals. We'll do this fairly manually. The first goal is to get the fitted values and the residuals, and separately sort them:

```
singer.fitted = sort(fitted.values(singer.lm)) - mean(fitted.values(singer.lm))
singer.residuals = sort(residuals(singer.lm))
```

Now we calculate the f-values (there's only one set, since there's the same number of fitted values and residuals) and stick everything into a data frame.

```
n = length(singer.residuals)
f.value = (0.5:(n - 0.5))/n
singer.fit = data.frame(f.value, Fitted = singer.fitted, Residuals = singer.residuals)
```

We now have a problem: We have a data frame with 235 observations in three columns:

f.value, Fitted, Residuals

But to use faceting in `ggplot`, we'd like a data frame with (2×235) observations in these three columns:

f.value, type, value

where “type” is a categorical variable (either “Fitted” or “Residual”) and “value” is the numerical value (either the fitted value or the residual.)

We could easily manually hammer the data into the right form, but instead we’ll learn a new tool to do this.

2.2.4 gather(): A lifesaver

We’ll use `gather()` in Hadley Wickham’s `tidyr` library, which has saved the lives of countless frustrated R users in recent years. (You may need to install `tidyr` if you don’t already have it.) `gather()`, as the names perhaps suggests, will take several variables and gathers them together, so you have more observations but fewer columns.

```r
library(tidyr)
singer.fit.long = singer.fit %>% gather(type, value, Fitted:Residuals)
```

What did we just do? Let’s illustrate with a simple example. Suppose you’re studying the votes won by Clinton and Trump in 2016 in a few Indiana counties. You enter the data:

```r
Counties = c("Hamilton", "Marion", "Monroe", "Tippecanoe", "Vermillion")
Clinton = c(2819, 212676, 34183, 27207, 2081)
Trump = c(8530, 130228, 20527, 30711, 4511)
Indiana = data.frame(Counties, Clinton, Trump)
```

Let’s see what this looks like:

```r
print(Indiana)
```

```
##     Counties Clinton  Trump
## 1   Hamilton    2819   8530
## 2     Marion  212676 130228
## 3     Monroe   34183  20527
## 4 Tippecanoe   27207  30711
## 5 Vermillion    2081   4511
```

Then you decide you want to do faceting, so you want to change the data to “long” form. That is, instead of three columns of length 5, you want 3 columns of length 10. Use `gather()`:

```r
Indiana.long = Indiana %>% gather(Candidate, Votes, Clinton:Trump)
```

Let’s see what we got:

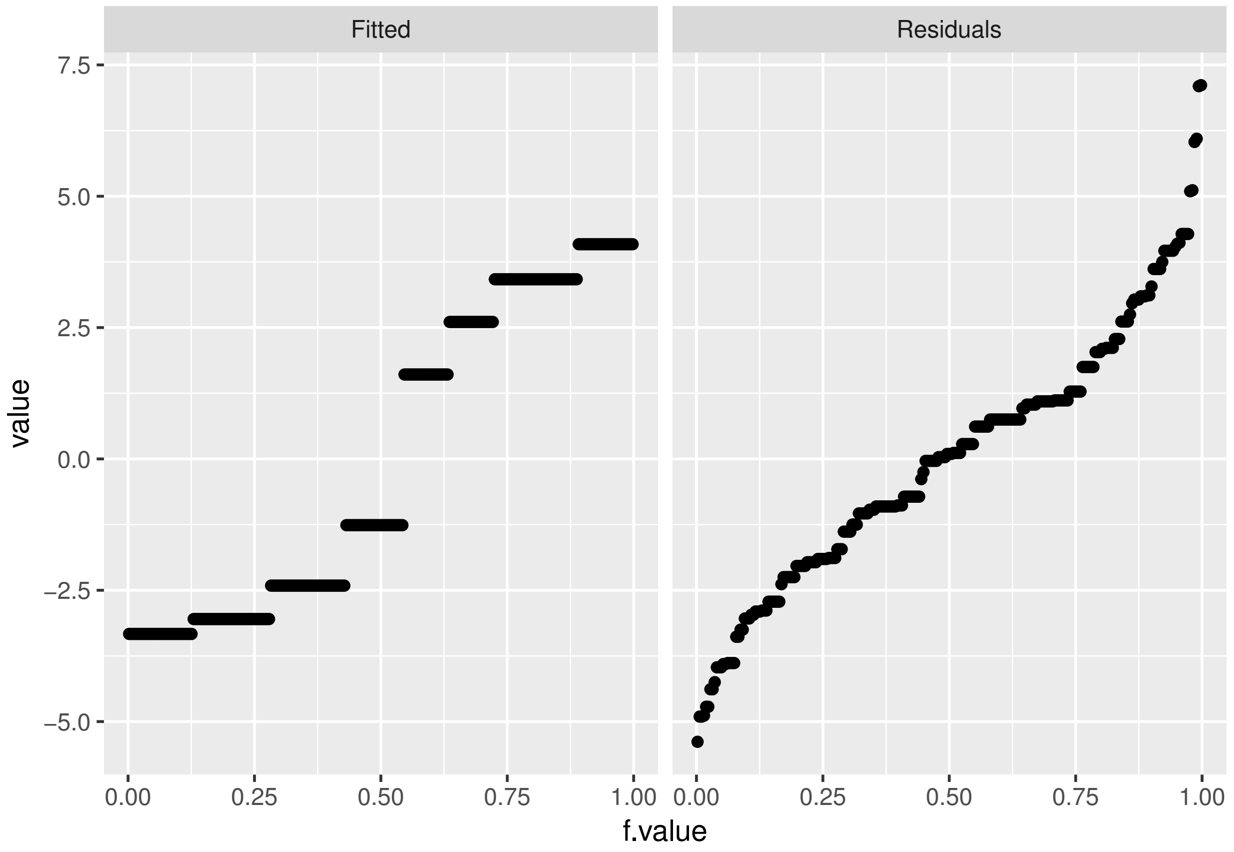
```r
print(Indiana.long)
```

```
##      Counties Candidate  Votes
## 1    Hamilton   Clinton   2819
## 2      Marion   Clinton 212676
## 3      Monroe   Clinton  34183
## 4  Tippecanoe   Clinton  27207
## 5  Vermillion   Clinton   2081
## 6    Hamilton     Trump   8530
## 7      Marion     Trump 130228
## 8      Monroe     Trump  20527
## 9  Tippecanoe     Trump  30711
## 10 Vermillion     Trump   4511
```

Now you can do faceting and all kinds of crazy stuff.

Let’s go back to the singer height residuals and draw our residual-fit plot.

```r
ggplot(singer.fit.long, aes(x = f.value, y = value)) + geom_point() + facet_wrap(~type)
```



The spread of the fitted values is broadly similar to the spread of the residuals. The model fit accounts for a decent chunk of the variation, but a decent chunk remains in the residuals. You can of course quantify this using r^2 if your tastes run that way.

2.2.5 Interlude: Tibbles

For large or complex data sets, *tibbles* are a minor improvement on data frames:

- They print more elegantly;
- They subset more predictably. e.g. When you use square brackets to subset a tibble, you always get another tibble. (With data frames you might get a vector or a data frame.)

Generally though you can use them in all the same ways as you use data frames.

Let's read in a .csv file on lengths of Billboard Hot 100 song s in 2000:

```r
billboard.raw = read.csv("https://github.com/hadley/tidy-data/raw/master/data/billboard.csv",
    stringsAsFactors = FALSE)
```

To turn this into a tibble, we use the `tbl_df()` function in the `dplyr` library.

```r
library(dplyr)
billboard = tbl_df(billboard.raw)
```

The object `billboard` displays reasonably elegantly:

```r
billboard
```

```
## # A tibble: 317 × 83
##     year     artist.inverted                                 track  time
##    <int>               <chr>                                 <chr> <chr>
## 1   2000     Destiny's Child               Independent Women Part I  3:38
## 2   2000             Santana                          Maria, Maria  4:18
## 3   2000       Savage Garden                     I Knew I Loved You  4:07
## 4   2000             Madonna                                  Music  3:45
## 5   2000 Aguilera, Christina Come On Over Baby (All I Want Is You)  3:38
## 6   2000               Janet                  Doesn't Really Matter  4:17
## 7   2000     Destiny's Child                            Say My Name  4:31
## 8   2000   Iglesias, Enrique                             Be With You  3:36
## 9   2000               Sisqo                             Incomplete  3:52
## 10  2000            Lonestar                                Amazed  4:25
## # ... with 307 more rows, and 79 more variables: genre <chr>,
## #   date.entered <chr>, date.peaked <chr>, x1st.week <int>,
## #   x2nd.week <int>, x3rd.week <int>, x4th.week <int>, x5th.week <int>,
## #   x6th.week <int>, x7th.week <int>, x8th.week <int>, x9th.week <int>,
## #   x10th.week <int>, x11th.week <int>, x12th.week <int>,
## #   x13th.week <int>, x14th.week <int>, x15th.week <int>,
## #   x16th.week <int>, x17th.week <int>, x18th.week <int>,
## #   x19th.week <int>, x20th.week <int>, x21st.week <int>,
## #   x22nd.week <int>, x23rd.week <int>, x24th.week <int>,
## #   x25th.week <int>, x26th.week <int>, x27th.week <int>,
## #   x28th.week <int>, x29th.week <int>, x30th.week <int>,
## #   x31st.week <int>, x32nd.week <int>, x33rd.week <int>,
## #   x34th.week <int>, x35th.week <int>, x36th.week <int>,
## #   x37th.week <int>, x38th.week <int>, x39th.week <int>,
## #   x40th.week <int>, x41st.week <int>, x42nd.week <int>,
## #   x43rd.week <int>, x44th.week <int>, x45th.week <int>,
## #   x46th.week <int>, x47th.week <int>, x48th.week <int>,
## #   x49th.week <int>, x50th.week <int>, x51st.week <int>,
## #   x52nd.week <int>, x53rd.week <int>, x54th.week <int>,
## #   x55th.week <int>, x56th.week <int>, x57th.week <int>,
## #   x58th.week <int>, x59th.week <int>, x60th.week <int>,
## #   x61st.week <int>, x62nd.week <int>, x63rd.week <int>,
## #   x64th.week <int>, x65th.week <int>, x66th.week <lgl>,
## #   x67th.week <lgl>, x68th.week <lgl>, x69th.week <lgl>,
## #   x70th.week <lgl>, x71st.week <lgl>, x72nd.week <lgl>,
## #   x73rd.week <lgl>, x74th.week <lgl>, x75th.week <lgl>, x76th.week <lgl>
```

2.2.6 How long are songs?

How long were the songs on the 2000 Hot 100? We see from the tibble display that the `time` column of our tibble is a character variable instead of numeric. It'll take some work to fix this.

The immediate problem is the colon. All we want are the numbers before (the minutes) and after (the seconds) the colon for each entry. The function `strsplit` splits the strings:

```
billboard.time = strsplit(billboard$time, ":")
```

`billboard.time` is now a list. Now we need to:

- "unlist" `billboard.time` by making it into a matrix;
- extract the two columns of the matrix (minutes and seconds) and make them numeric;
- calculate the length of each song in seconds;
- change this to minutes (because minutes are more intuitive than seconds);

- replace the `time` column of `billboard.time` with this new numeric vector.

```
billboard.time = matrix(unlist(billboard.time), byrow = T, ncol = 2)
billboard.mins = as.numeric(billboard.time[, 1])
billboard.secs = as.numeric(billboard.time[, 2])
billboard.time = billboard.mins * 60 + billboard.secs
billboard$time = billboard.time/60
```

Depending on what we want to do, it may be useful to have the data in long form as well as in wide form. We use `gather()`:

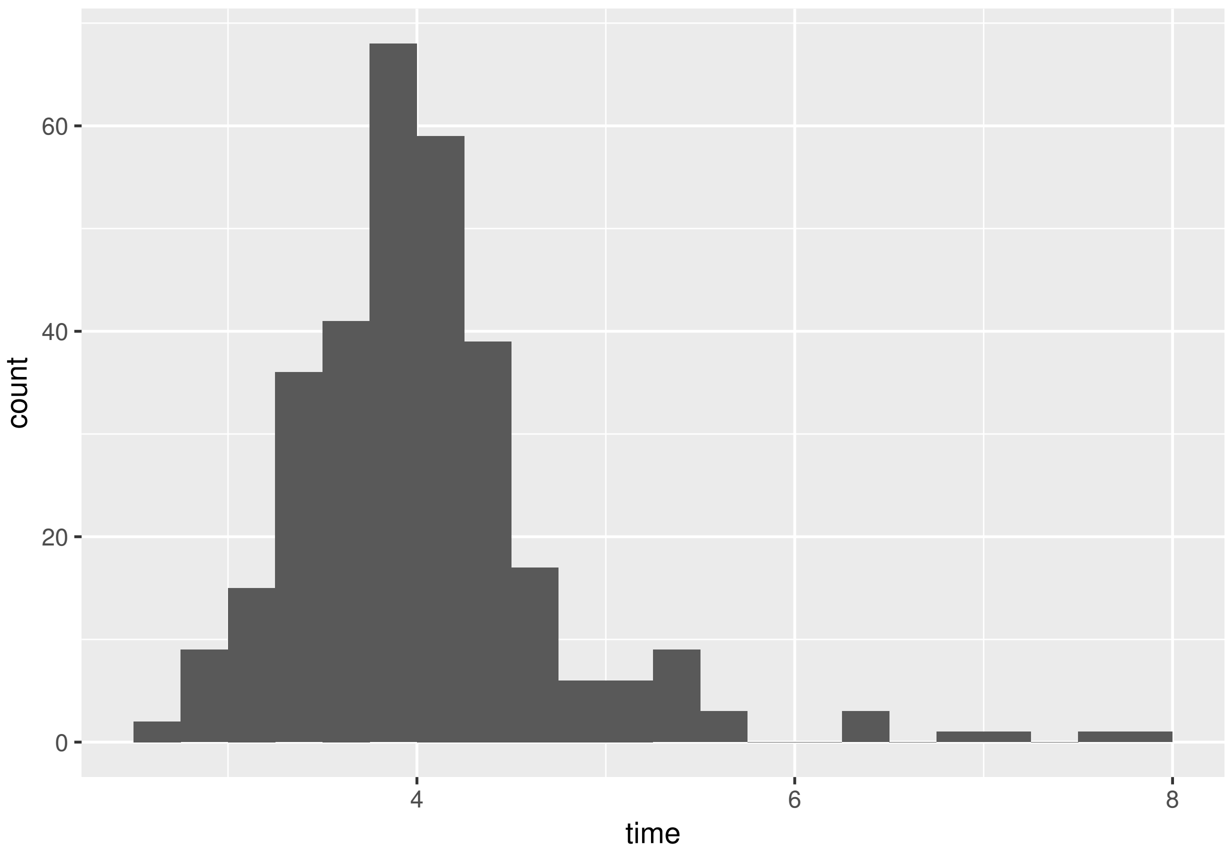
```
billboard.long <- billboard %>% gather(week, rank, x1st.week:x76th.week, na.rm = TRUE)
billboard.long
```

```
## # A tibble: 5,307 × 9
##     year     artist.inverted                                 track
## *  <int>               <chr>                                 <chr>
## 1   2000     Destiny's Child              Independent Women Part I
## 2   2000             Santana                          Maria, Maria
## 3   2000       Savage Garden                    I Knew I Loved You
## 4   2000             Madonna                                 Music
## 5   2000 Aguilera, Christina Come On Over Baby (All I Want Is You)
## 6   2000               Janet                 Doesn't Really Matter
## 7   2000     Destiny's Child                           Say My Name
## 8   2000   Iglesias, Enrique                           Be With You
## 9   2000               Sisqo                            Incomplete
## 10  2000            Lonestar                                Amazed
## # ... with 5,297 more rows, and 6 more variables: time <dbl>, genre <chr>,
## #   date.entered <chr>, date.peaked <chr>, week <chr>, rank <int>
```

Here `x1st.week:x76th.week` gathers all the columns from `x1st.week` to `x76th.week` and turns them into separate observations. Thus "Independent Women Part I" accounts for one observation in `billboard` but 28 observations in `billboard.long`, because it was on the charts for 28 weeks.

Let's first go back to the wide form data. Using songs as the unit, here's a histogram of song lengths:

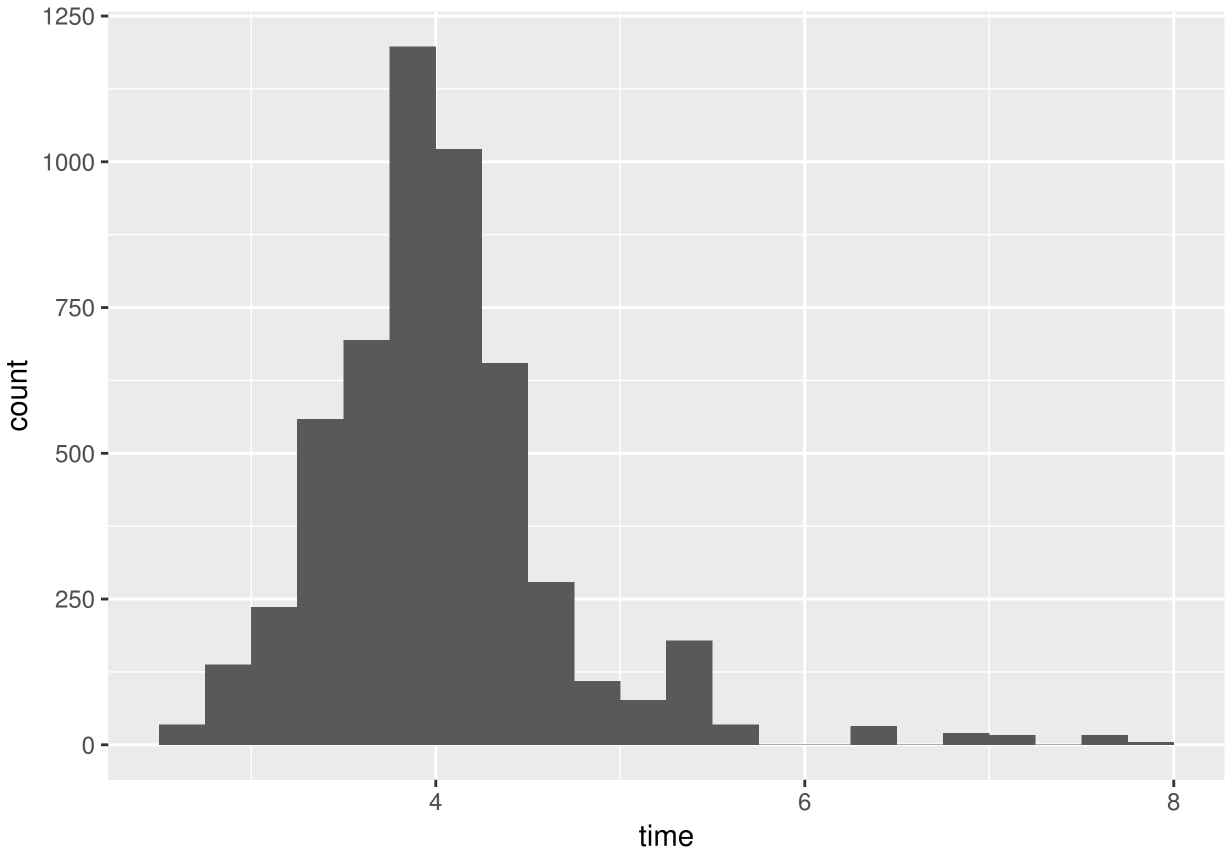
```
ggplot(billboard, aes(x = time)) + geom_histogram(breaks = seq(2.5, 8, 0.25))
```



The majority of Hot 100 songs (maybe two-thirds) are between 3:15 and 4:30.

Compare this to a histogram using song-weeks as the unit:

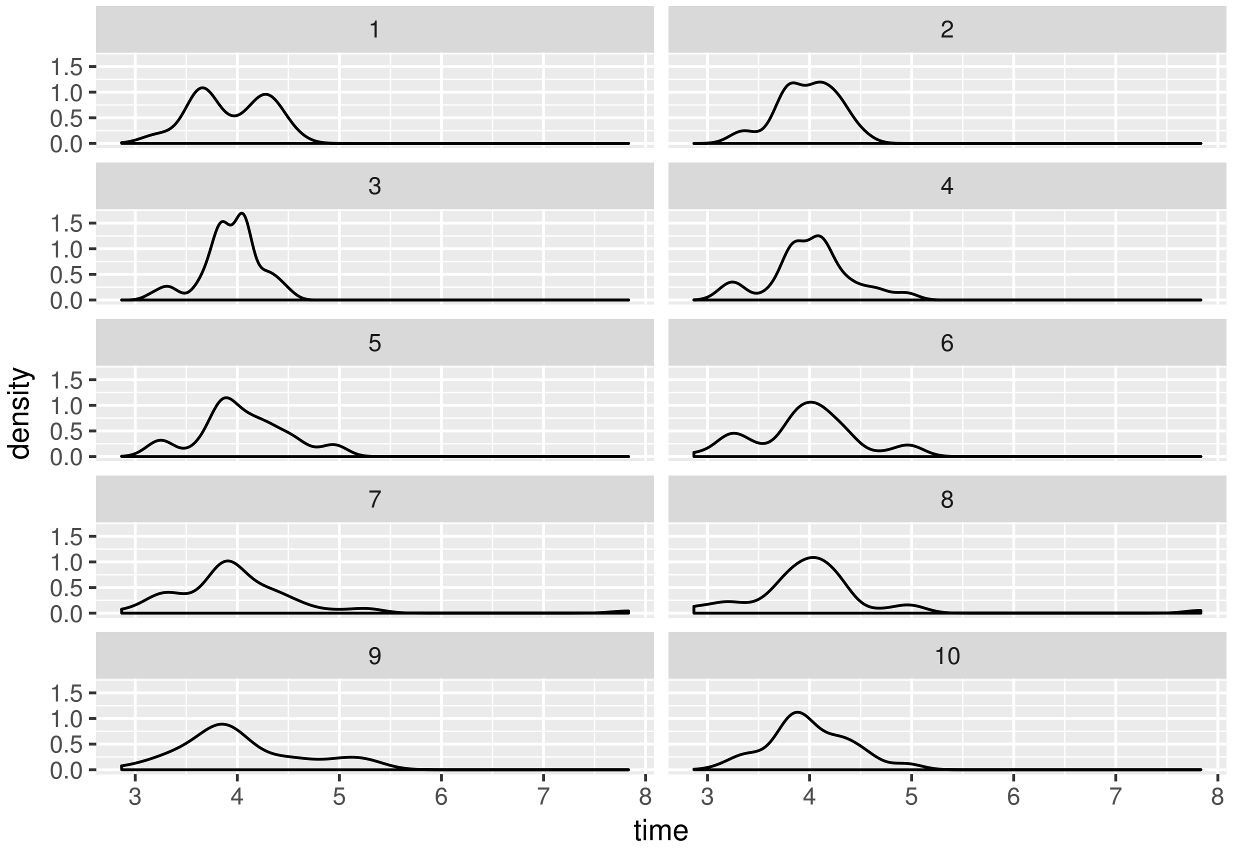
```
ggplot(billboard.long, aes(x = time)) + geom_histogram(breaks = seq(2.5, 8,
    0.25))
```



The y-axis scale has changed (since there are more observations.) However, the shape of the histogram is pretty much the same, suggesting that weighting by number of weeks on the chart doesn't make much difference.

Let's facet by chart position. To avoid drawing 100 graphs, we use `subset()` to only consider the top 10 chart positions:

```
ggplot(subset(billboard.long, rank <= 10), aes(x = time)) + geom_density() +
    facet_wrap(~rank, ncol = 2)
```



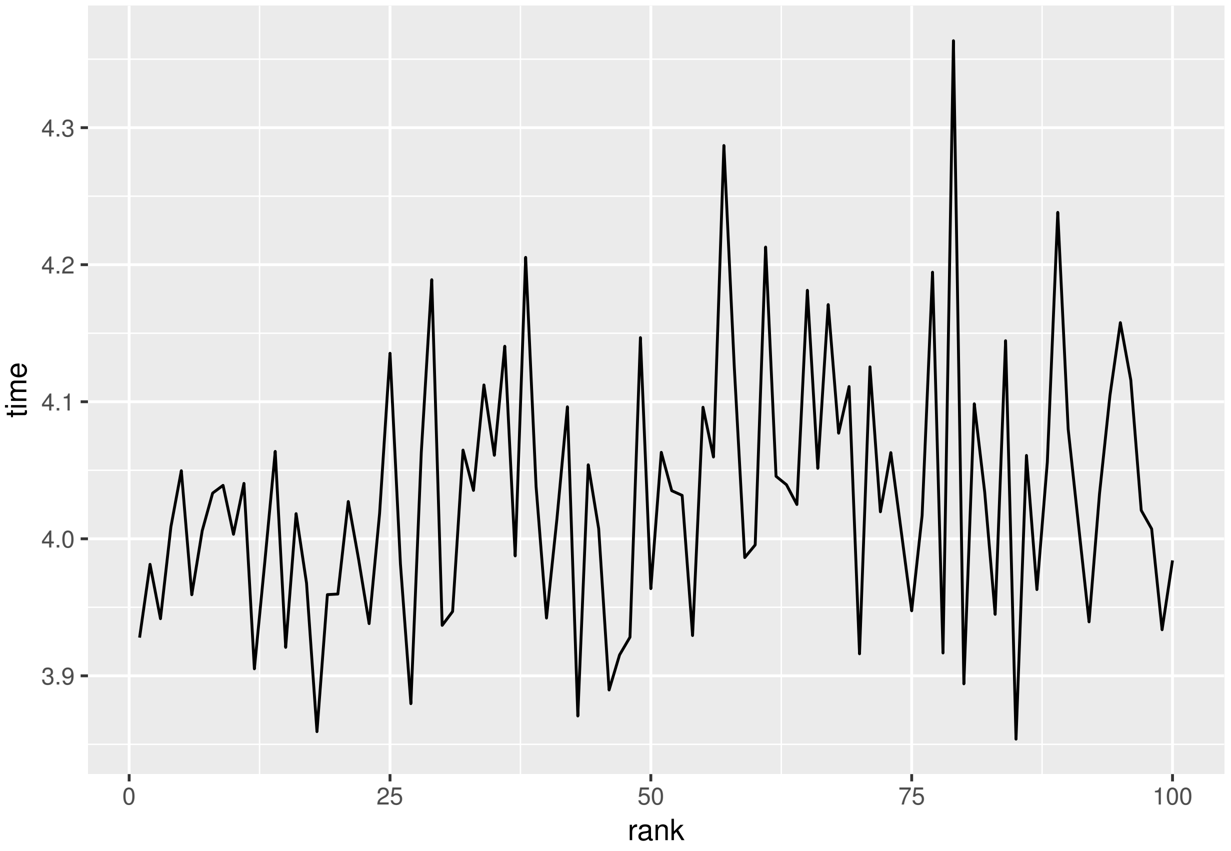
I drew density plots instead of histograms because of the small samples. It's a bit hard to tell here whether the differences are systematic or noise.

Finally, let's find mean song length by chart position.

```
time.means = aggregate(time ~ rank, FUN = mean, data = billboard.long)
```

Rank is numeric, so it makes sense to draw a **line graph** using `geom_line()`:

```
ggplot(time.means, aes(x = rank, y = time)) + geom_line()
```



There's still a lot of noise. (Later on we'll learn a bit about *smoothing*, which would help a lot here.) The one thing that seems apparent is the high ranks are noisier than low ranks. But that's just because the low ranks often have the same song appearing repeatedly (songs often stay in the top ten for ages, whilst they rarely stay in the nineties for more than a week.)

2.3 Transformations

READ: Cleveland pp. 42–67.

Load `ggplot2`:

```r
library(ggplot2)
```

We'll also use an R workspace prepared by Cleveland to use in conjunction with the book. Among other things, this contains the data sets used in the book.

```r
load("lattice.RData")
```

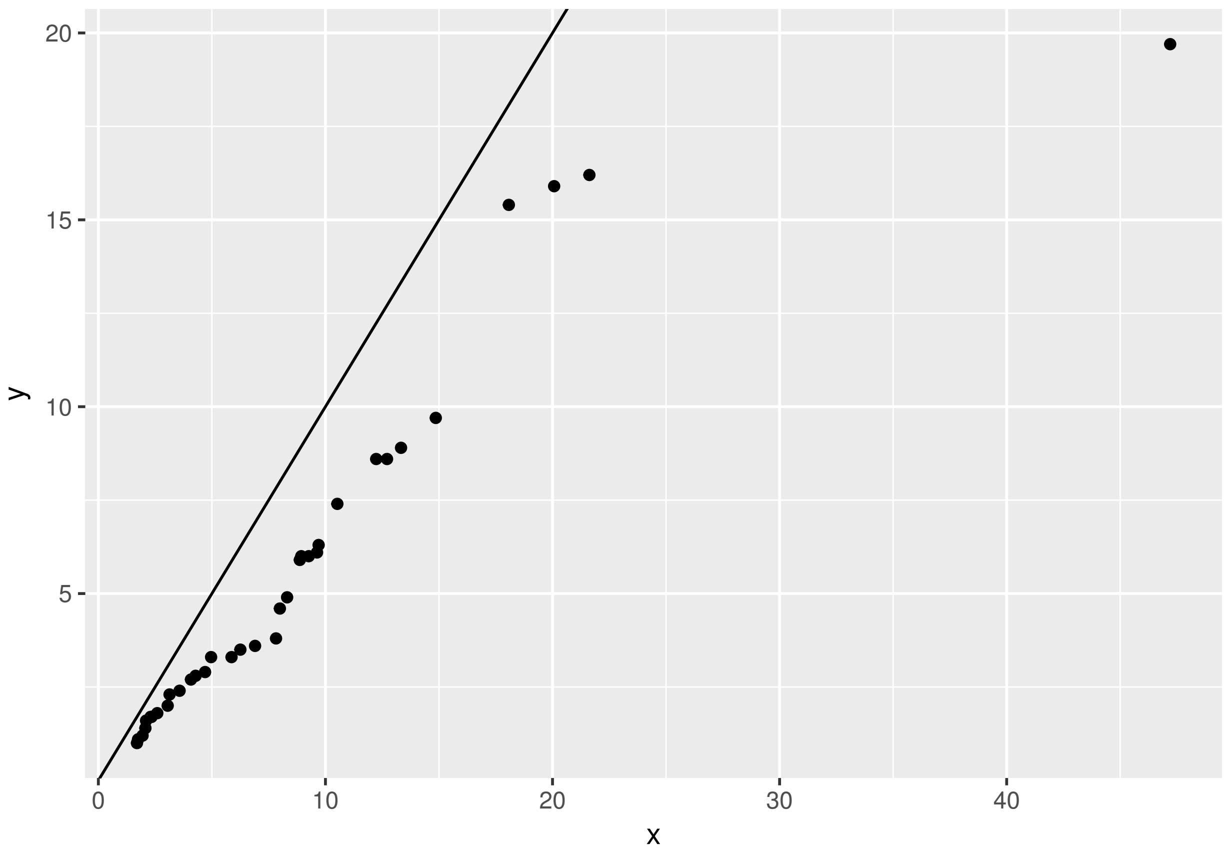
2.3.1 Log transformation

In an intro course like S320/520, you learned to use the most common transformation, the log. The main reason we gave was that it often made positive data more normal. But there's another and perhaps more fundamental reason: It often leads to differences between samples that we can interpret as a *multiplicative shift*. Some statisticians will go as far as to recommend log transforming positive data by default, though by the end of Cleveland's chapter 2, we'll see an example where that backfires.

We'll use the stereogram fusion time data in `fusion.time`, which contains the group variable `nv.vv` (where "NV" means no visual information and "VV" means visual information) and the quantitative variable `time` (a positive number.)

We first draw a two-sample QQ plot of the data.

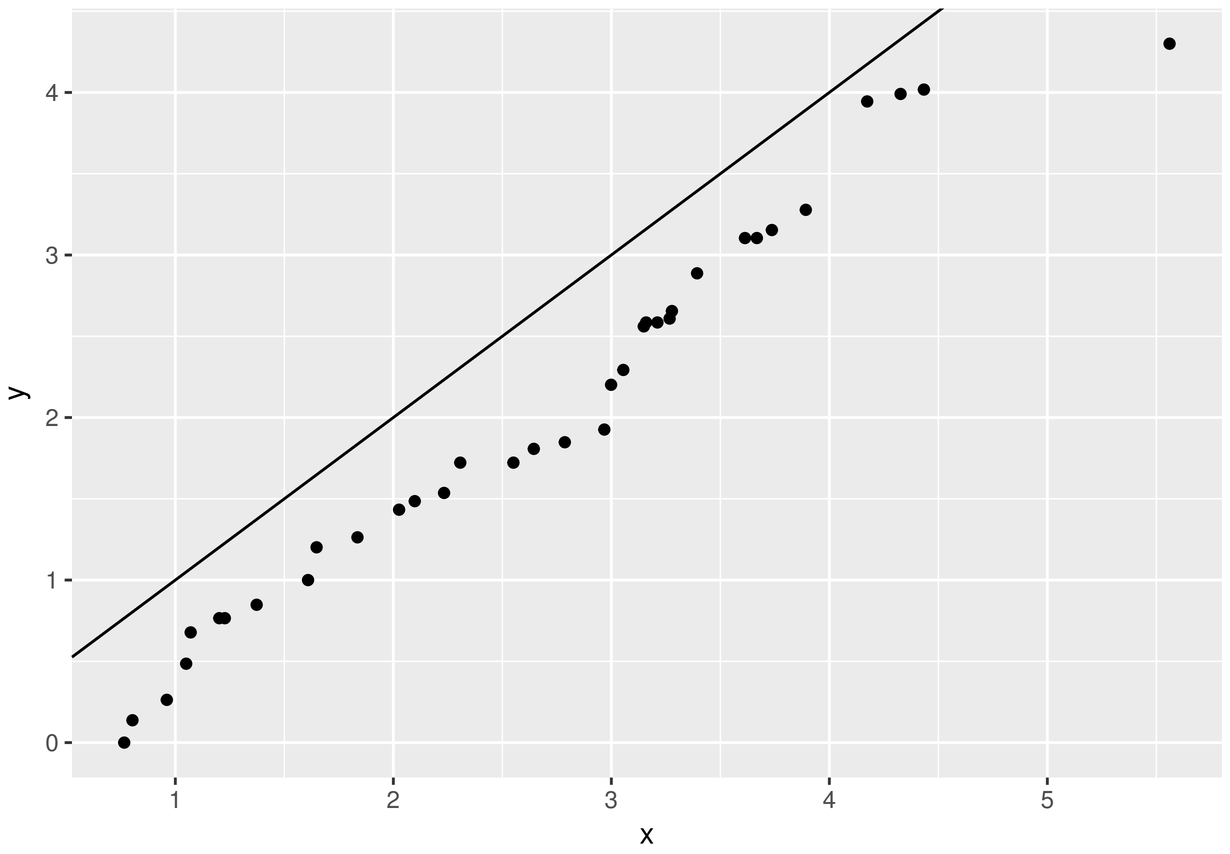
```
time = fusion.time$time
nv.vv = fusion.time$nv.vv
NV.times = sort(time[nv.vv == "NV"])
VV.times = sort(time[nv.vv == "VV"])
NV.VV.qq = as.data.frame(qqplot(NV.times, VV.times, plot.it = FALSE))
ggplot(NV.VV.qq, aes(x, y)) + geom_point() + geom_abline()
```



The line $y = x$ doesn't fit the plot. It seems like a line through the origin with a smaller slope – which would be equivalent to a multiplicative effect – would be a good fit, but it's hard to visually work out where the line should go.

Let's try a log transformation. In contrast to confirmatory data analysis, where natural logs are standard, in EDA it's usually much easier to understand what's going on if we use base 2 or base 10 logs.

```
NV.times.log = sort(log2(time[nv.vv == "NV"]))
VV.times.log = sort(log2(time[nv.vv == "VV"]))
NV.VV.qq.log = as.data.frame(qqplot(NV.times.log, VV.times.log, plot.it = FALSE))
ggplot(NV.VV.qq.log, aes(x, y)) + geom_point() + geom_abline()
```

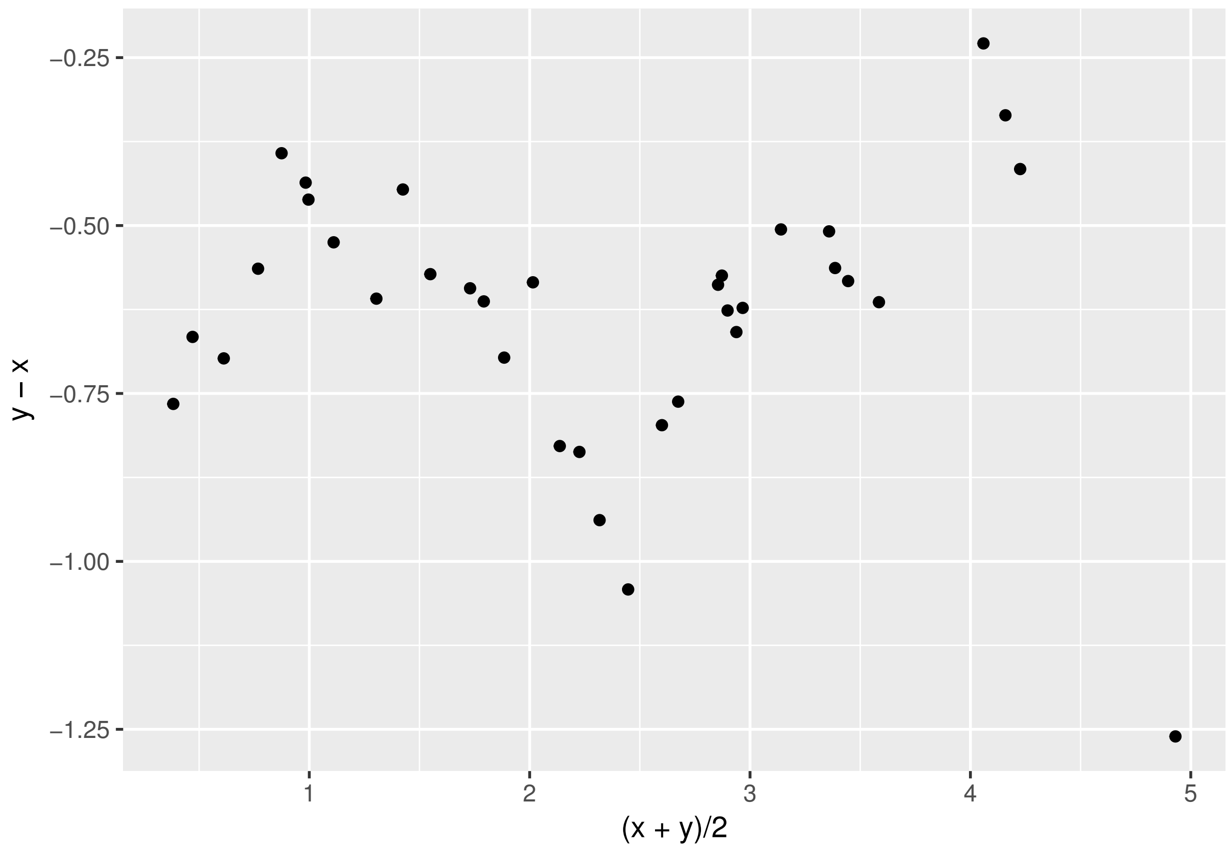


We can see the data is well-described by a straight line below $y = x$, indicating a multiplicative shift – roughly speaking, the distribution of the VV times is the NV distributions multiplied by some constant (less than 1.)

2.3.2 Tukey mean-difference plot

While the shift is pretty clear here, in messier cases it may be difficult to compare to the diagonal line. A *Tukey mean-difference* plot has the advantage that we can compare to the x-axis instead. It simply plots the difference $y - x$ against the mean $(x + y)/2$:

```r
ggplot(NV.VV.qq.log, aes((x + y)/2, y - x)) + geom_point()
```



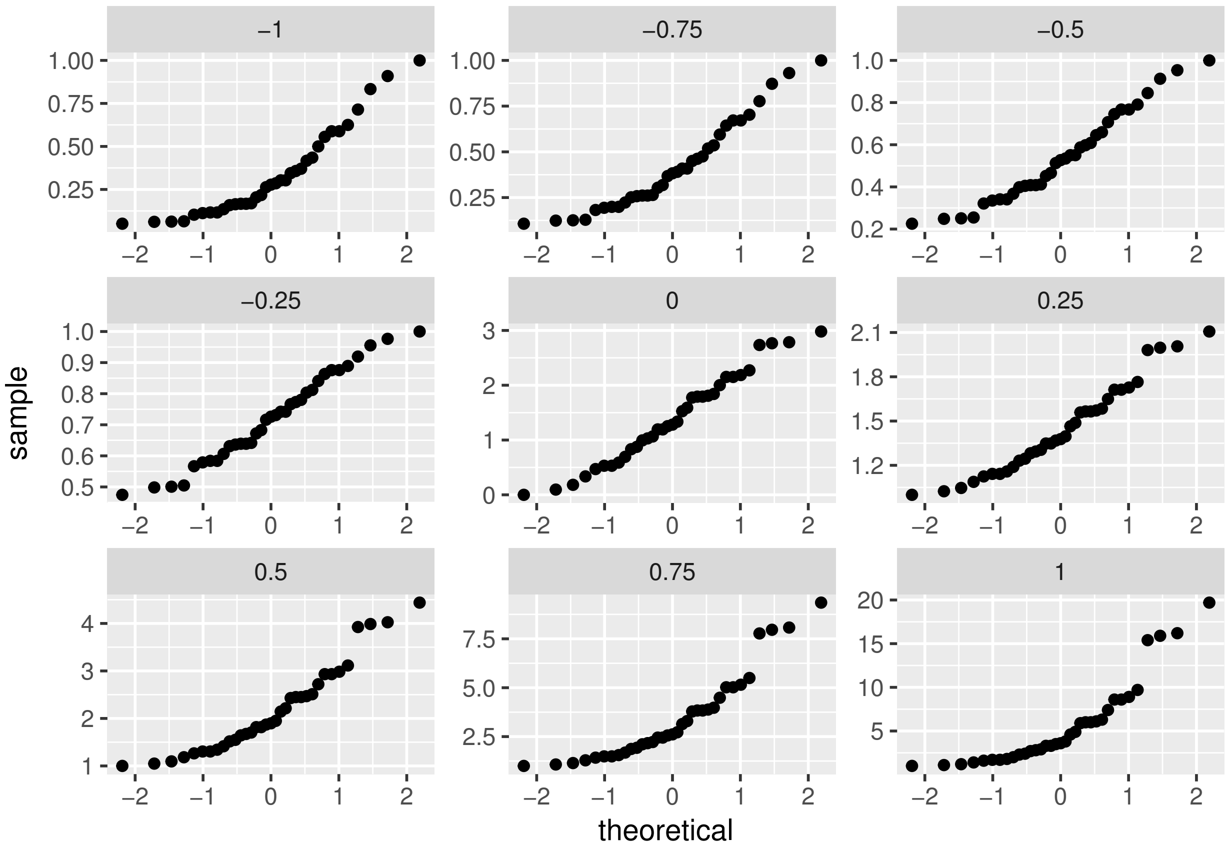
The points are scattered somewher around -0.75 to -0.5 on the y-axis. That means the multiplicative constant is around $2^{-0.75}$ to $2^{-0.5}$, or 0.6 to 0.7. Our best guess is that the visual information decreases fusion time by 30–40%.

2.3.3 Power transformation

The log transformation doesn't always work – for a start, you can't log zero or a negative number. **Power transformations** allow a wider range of options. Define a power transformation with parameter τ to be x^τ, and let the special case where $\tau = 0$ be the log transformation. Where practical, we'll hugely prefer $\tau = 1$ (no transformation), $\tau = 0$, or possibly $\tau = -1$ (inverse transformation) because they're much easier to interpret.

We can try out a variety of values of τ on the fusion time data, and see what gives us a distribution close to normal.

```r
n.VV = length(VV.times)
power = rep(seq(-1, 1, 0.25), each = n.VV)
VV.time = c(VV.times^-1, VV.times^-0.75, VV.times^-0.5, VV.times^-0.25, log(VV.times),
    VV.times^0.25, VV.times^0.5, VV.times^0.75, VV.times)
ggplot(data.frame(power, VV.time), aes(sample = VV.time)) + stat_qq() + facet_wrap(~power,
    scales = "free")
```

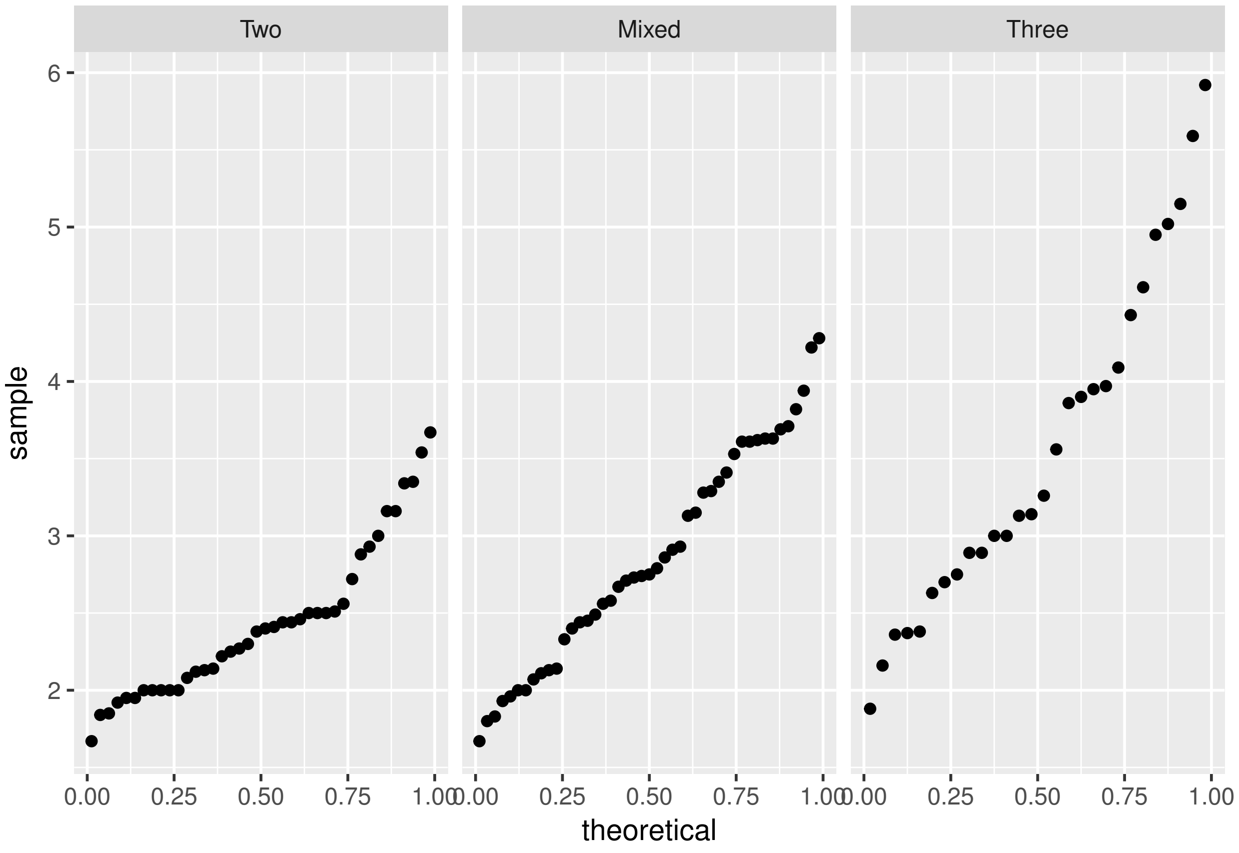


Here τ-values of 0 (the log transformation) and -0.25 give the straightest normal QQ plots. Since it's much, much easier to interpret $\log(\tau)$ than $\tau^{-0.25}$, we strongly prefer the log transformation.

2.3.4 Example: Food webs

The data set `food.web` contains the quantitative variable `mean.length` (the average number of links in the food chains in an ecosystem) and the categorical variable `dimension` (whether the ecosystem is two-dimensional like a plain, three-dimensional like a forest, or mixed.) We want to compare the average food chain length across these three type of ecosystem. Start with uniform QQ plots:

```
ggplot(food.web, aes(sample = mean.length)) + stat_qq(distribution = qunif) +
    facet_grid(~dimension)
```

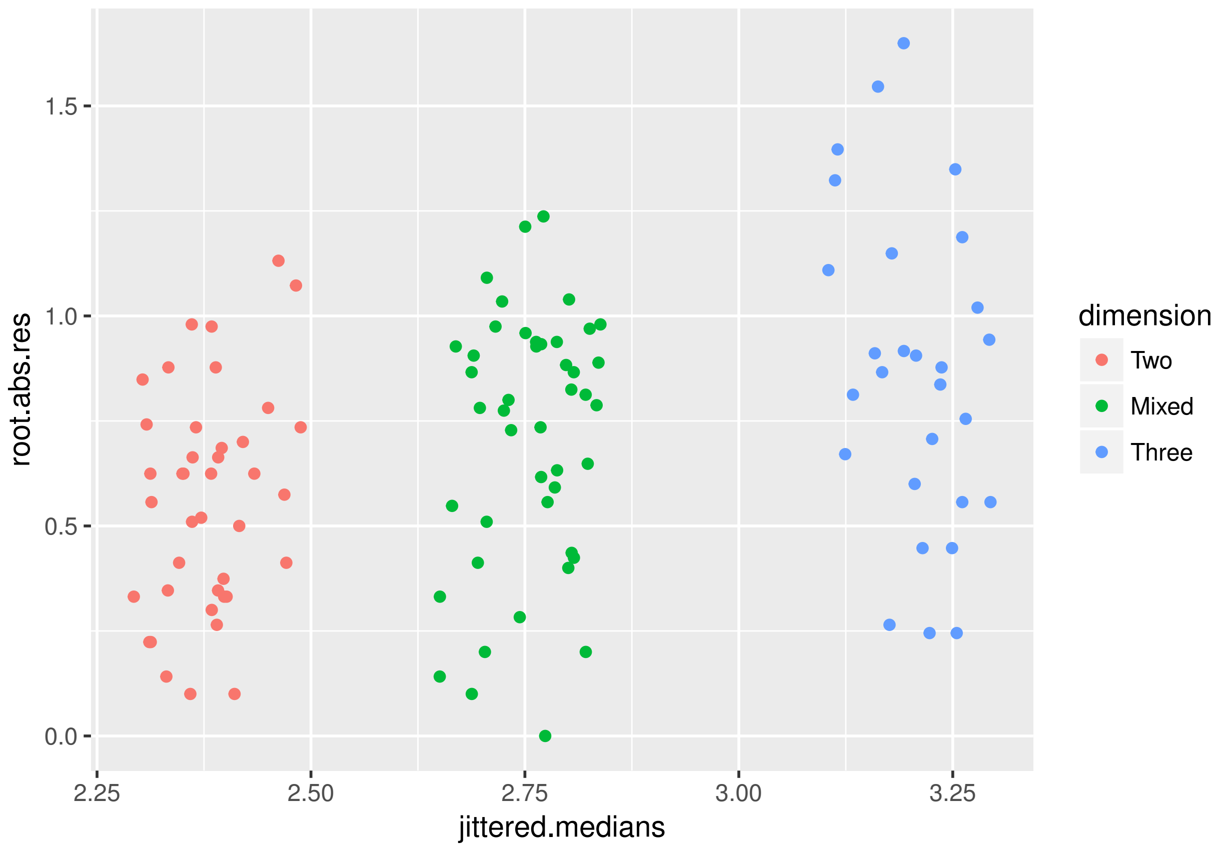


There's a clear difference in scale: Some of the three-dimensional webs approach a mean length of 6, while none of the two-dimensional webs even makes it to 4.

Is this a shift, or is there a difference in spread as well? This will be easier to see from the residuals. We'll display these in a **spread-location plot**:

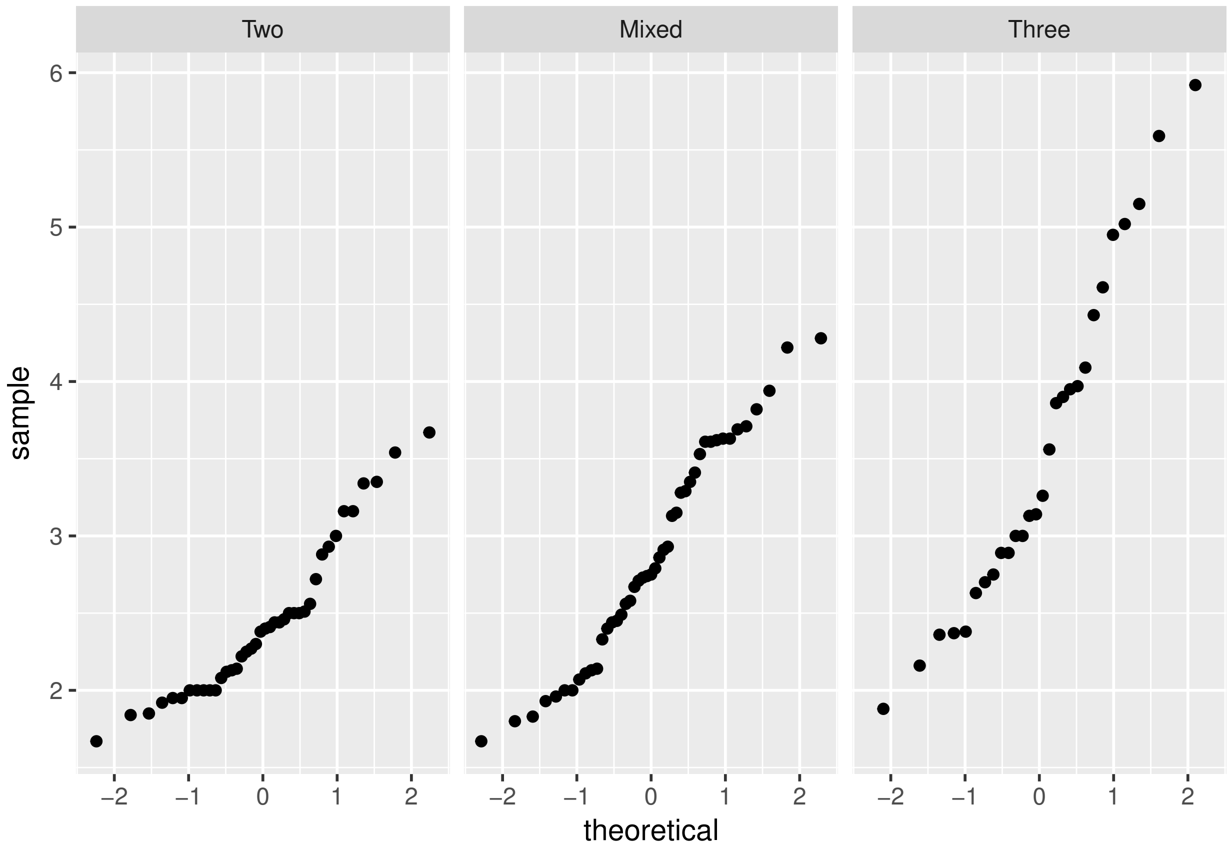
- The x-axis is the model estimate. Here I'll use the median since it's more resistant to outliers than the mean. Also, since the model is categorical, I'll add some random uniform noise (the size of the noise term can be determined by trial and error.)
- The y-axis has some transformation of the residuals. Cleveland uses the square root of the absolute residuals. I'm not sure there's any theoretical justification for the square root absolute transformation, but it tends to work well: the absolute value transforms the equal spread problem into an equal location problem, while the square root transformation intends to reduce skewness.

```
web.length = food.web$mean.length
dimension = food.web$dimension
n = nrow(food.web)
median.3 = median(web.length[dimension == "Three"])
median.2 = median(web.length[dimension == "Two"])
median.mixed = median(web.length[dimension == "Mixed"])
group.median = rep(NA, n)
group.median[dimension == "Three"] = median.3
group.median[dimension == "Two"] = median.2
group.median[dimension == "Mixed"] = median.mixed
jittered.medians = group.median + runif(n, -0.1, 0.1)
root.abs.res = sqrt(abs(web.length - group.median))
food.web.sl = data.frame(jittered.medians, root.abs.res, dimension)
ggplot(food.web.sl, aes(jittered.medians, root.abs.res, col = dimension)) +
    geom_point()
```



Pretty clearly the three sets of residuals are not all at the same average height. We also check for normality:

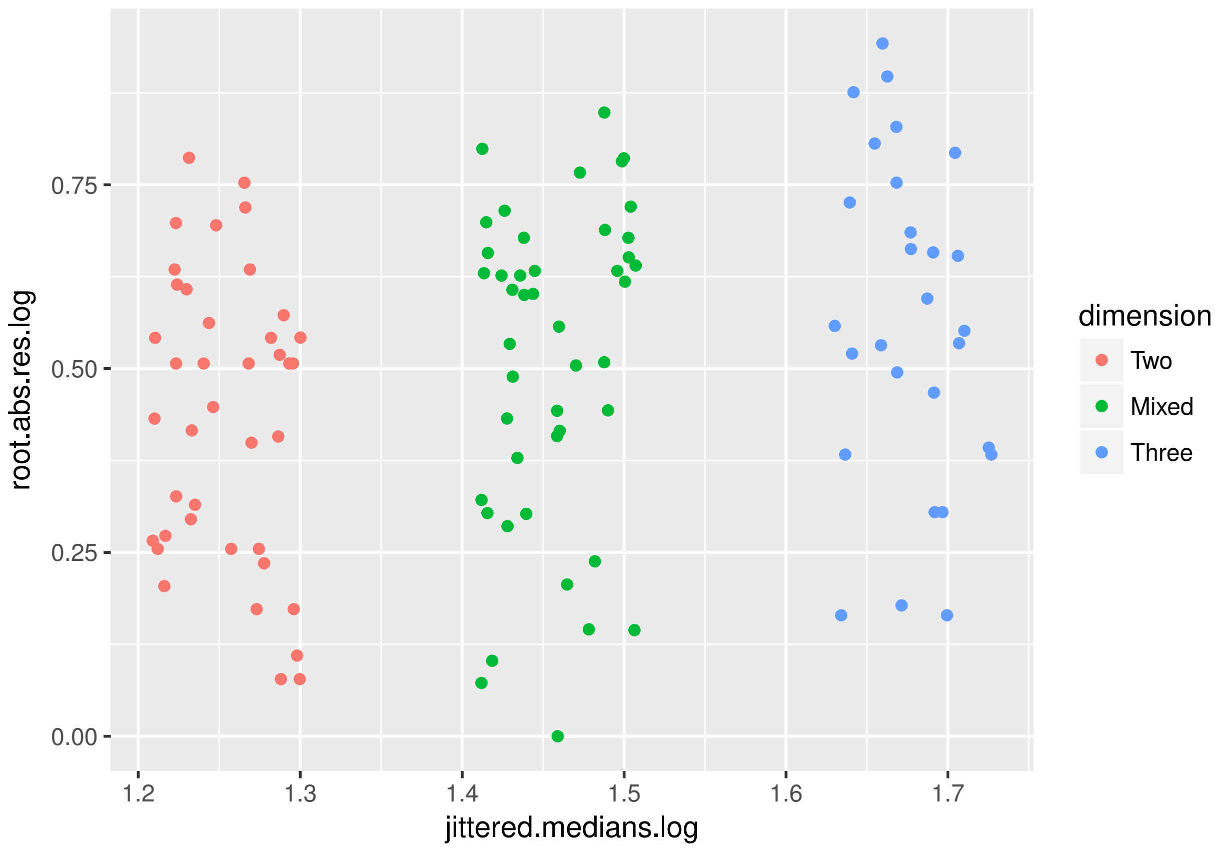
```
ggplot(food.web, aes(sample = mean.length)) + stat_qq() + facet_wrap(~dimension)
```



All the normal QQ plots look like they curve at least somewhat upward, indicating right skew. It looks like a transformation will help.

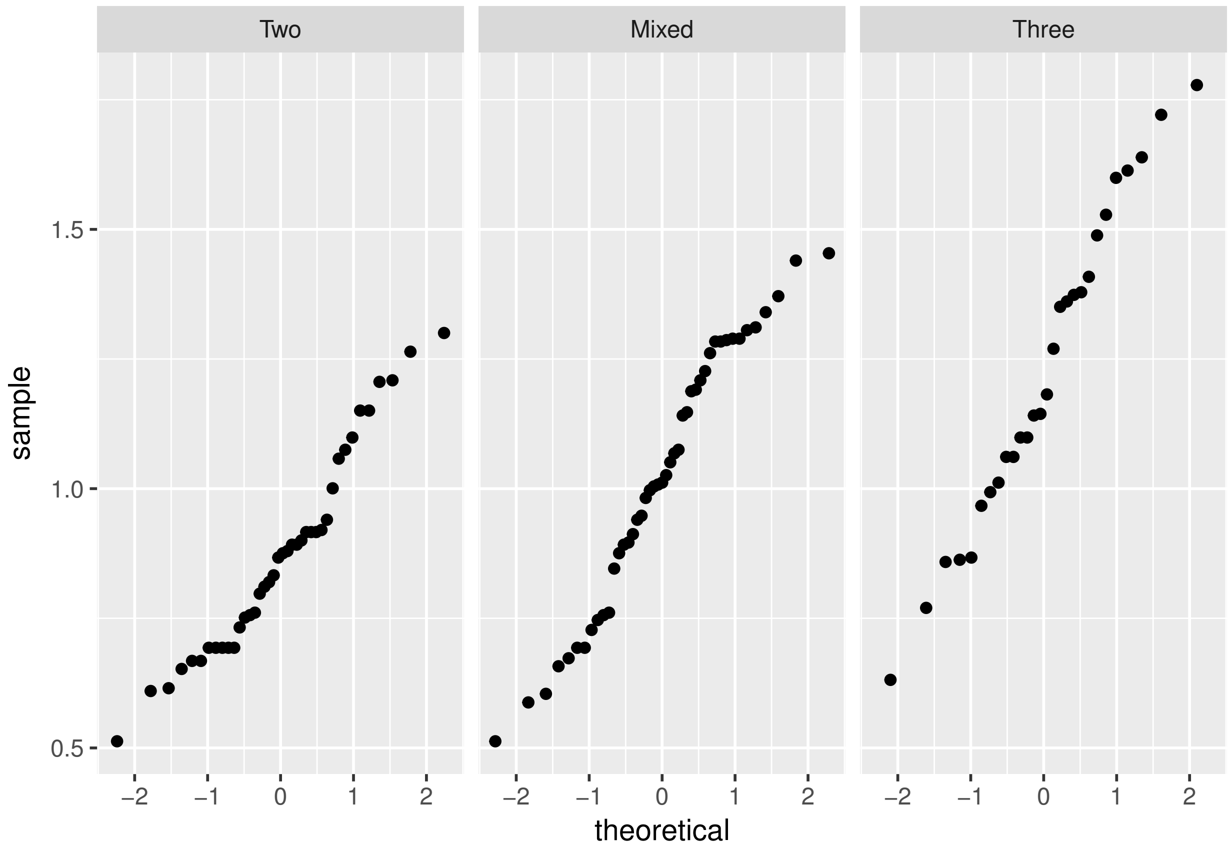
The food web mean lengths are positive, so we'll first try a log transformation. We repeat the spread-location plot on the transformed data:

```
log.web.length = log2(food.web$mean.length)
median.3.log = median(log.web.length[dimension == "Three"])
median.2.log = median(log.web.length[dimension == "Two"])
median.mixed.log = median(log.web.length[dimension == "Mixed"])
group.median.log = rep(NA, n)
group.median.log[dimension == "Three"] = median.3.log
group.median.log[dimension == "Two"] = median.2.log
group.median.log[dimension == "Mixed"] = median.mixed.log
jittered.medians.log = group.median.log + runif(n, -0.05, 0.05)
root.abs.res.log = sqrt(abs(log.web.length - group.median.log))
food.web.log.sl = data.frame(jittered.medians.log, root.abs.res.log, dimension)
ggplot(food.web.log.sl, aes(jittered.medians.log, root.abs.res.log, col = dimension)) +
    geom_point()
```



It's better but it still seems like there are differences between the three groups of residuals. We also check normality:

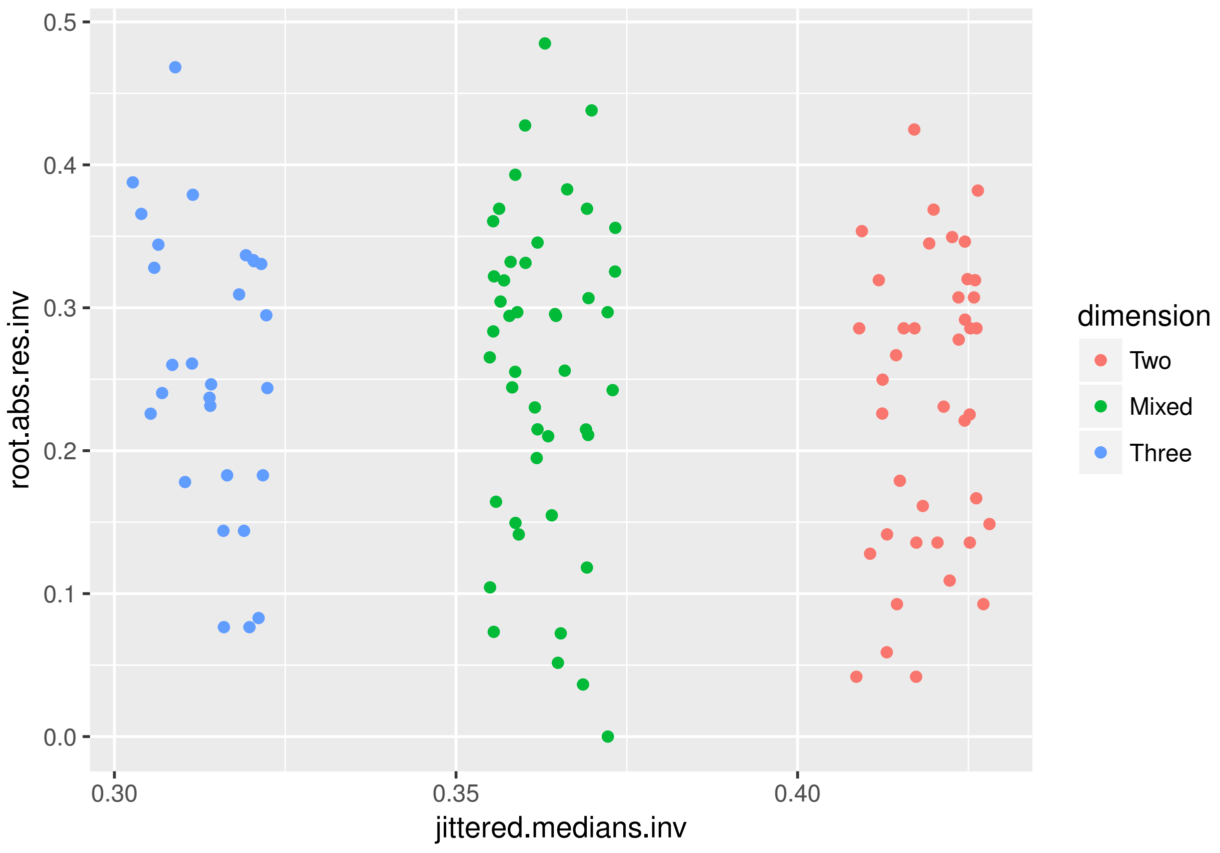
```r
ggplot(food.web, aes(sample = log(mean.length))) + stat_qq() + facet_wrap(~dimension)
```



The lines still don't look as straight as they could be – the log doesn't do enough to "normalize" the data.

We could try out a number of transformations, but if $\tau = 0$ didn't quite do enough, then $\tau = -1$, the inverse transformation, seems like the next candidate. Try the spread-location plot again:

```
inv.web.length = 1/food.web$mean.length
median.3.inv = median(inv.web.length[dimension == "Three"])
median.2.inv = median(inv.web.length[dimension == "Two"])
median.mixed.inv = median(inv.web.length[dimension == "Mixed"])
group.median.inv = rep(NA, n)
group.median.inv[dimension == "Three"] = median.3.inv
group.median.inv[dimension == "Two"] = median.2.inv
group.median.inv[dimension == "Mixed"] = median.mixed.inv
jittered.medians.inv = group.median.inv + runif(n, -0.01, 0.01)
root.abs.res.inv = sqrt(abs(inv.web.length - group.median.inv))
food.web.inv.sl = data.frame(jittered.medians.inv, root.abs.res.inv, dimension)
ggplot(food.web.inv.sl, aes(jittered.medians.inv, root.abs.res.inv, col = dimension)) +
    geom_point()
```



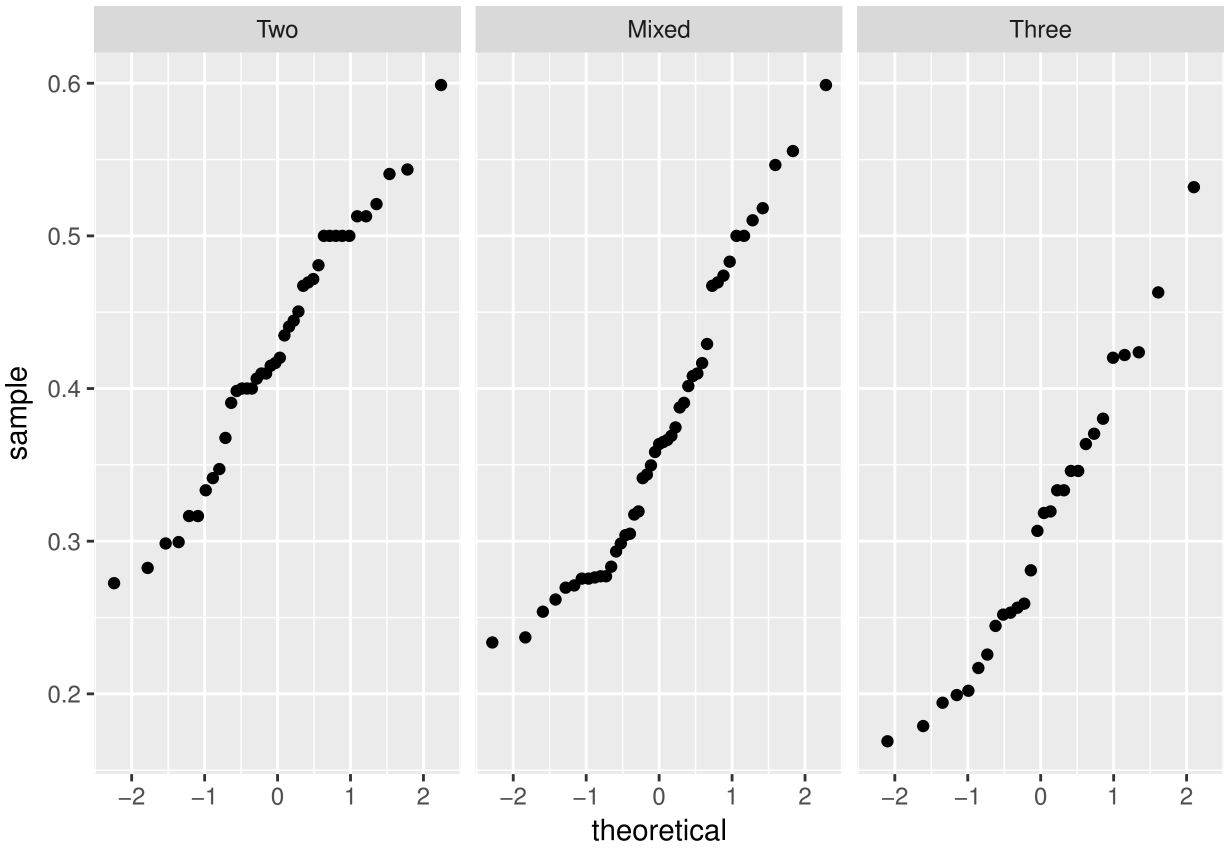
This looks the best of those we've seen. We can calculate the three group means:

```
aggregate(root.abs.res.inv ~ dimension, FUN = mean)
```

```
##   dimension root.abs.res.inv
## 1       Two        0.2332382
## 2     Mixed        0.2563367
## 3     Three        0.2579944
```

They're close enough that the differences can be explained as noise. What about normality?

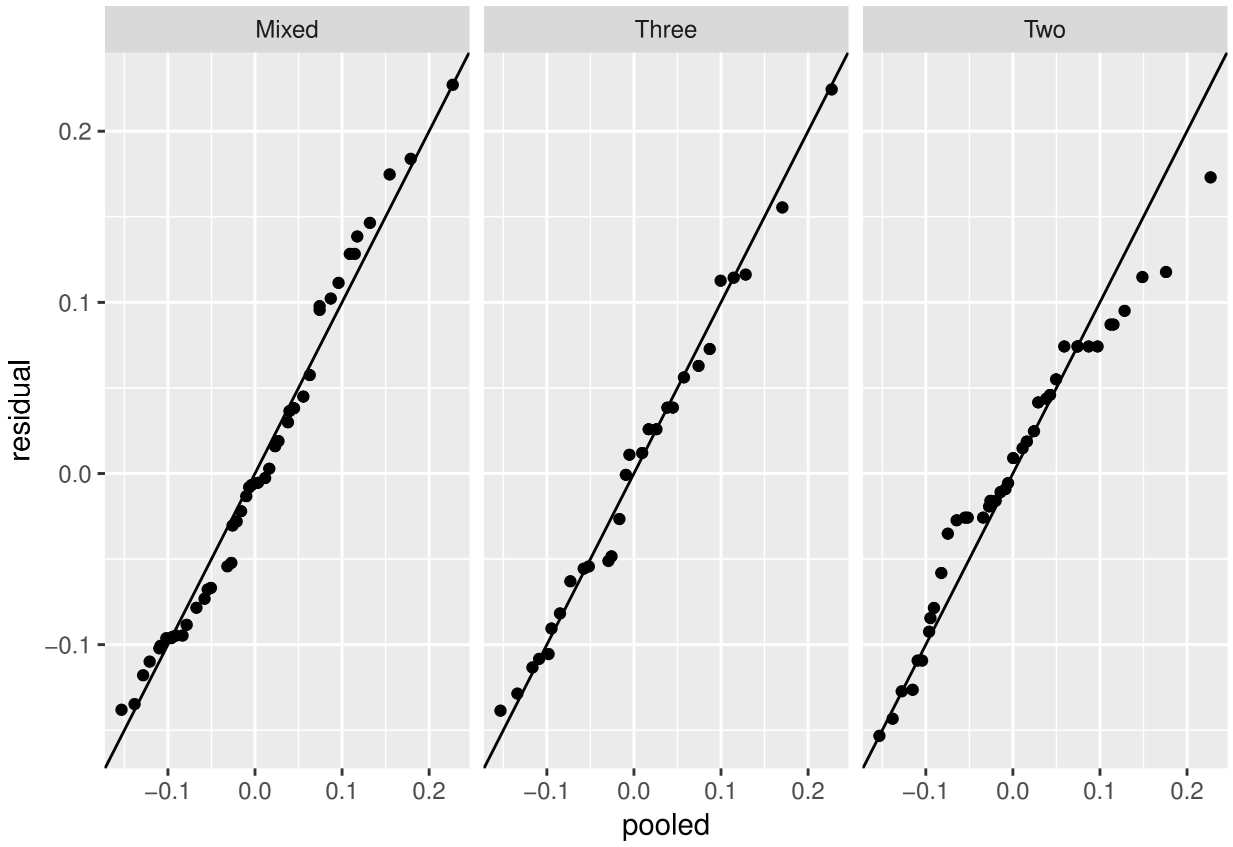
```
ggplot(food.web, aes(sample = 1/mean.length)) + stat_qq() + facet_wrap(~dimension)
```



These look straight. We're pretty happy with the inverse transformation. Furthermore, it's interpretable – we only have to think in terms of chains per link rather than links per change (though note we're no longer strictly dealing with a "mean.")

The spread is similar for all three QQ plots, suggesting that we might be able to pool the residuals. To check, this we create a vector of pooled residuals (`food.web.res` in the code below,) then draw a two-sample QQ plot for each of the three sets of residuals against the pooled set. We cheat by using the `qqplot()` function to calculate the quantiles for us. (Of course it would be much easier and only slightly uglier to just use `qqplot()` to draw the plots for us.)

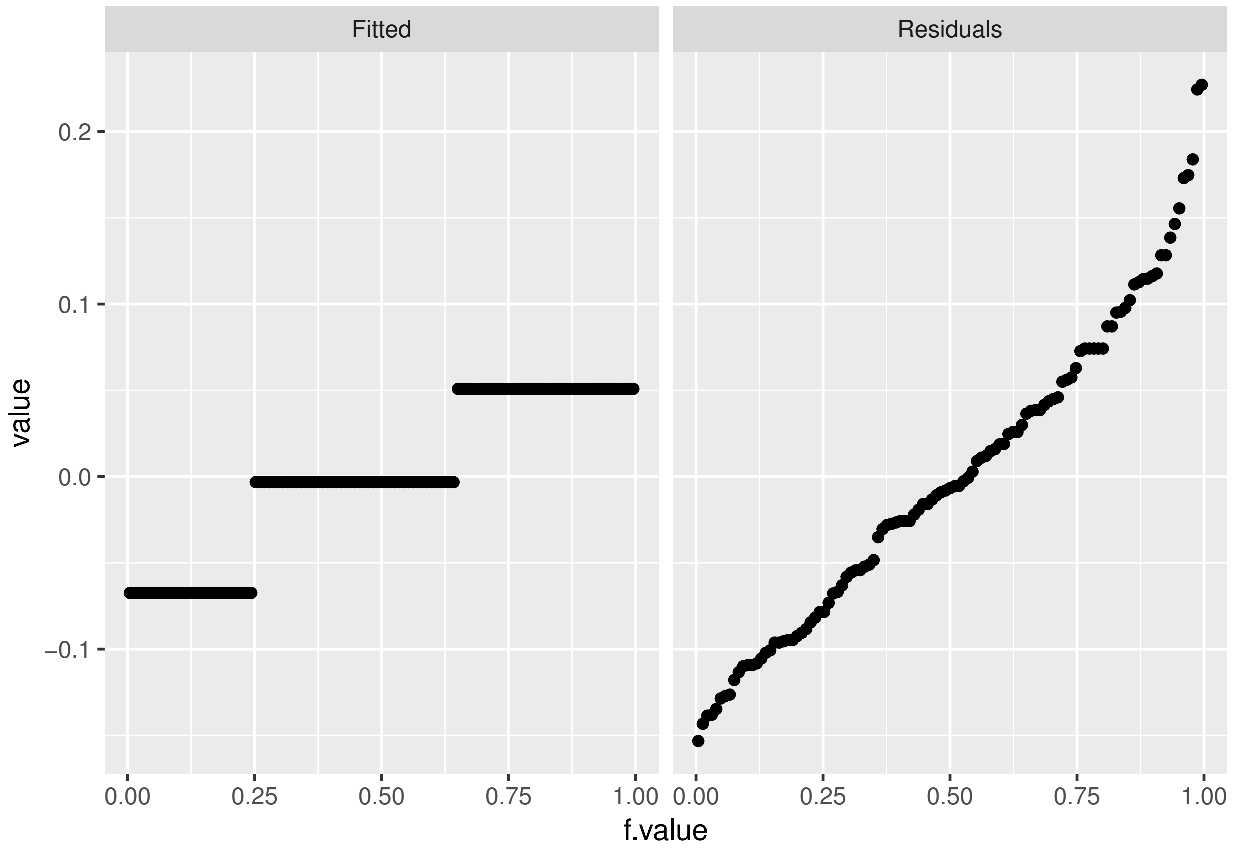
```r
food.web.lm = lm(inv.web.length ~ dimension)
food.web.res = residuals(food.web.lm)
res.qq.3 = qqplot(food.web.res, food.web.res[dimension == "Three"], plot.it = FALSE)
res.qq.2 = qqplot(food.web.res, food.web.res[dimension == "Two"], plot.it = FALSE)
res.qq.mixed = qqplot(food.web.res, food.web.res[dimension == "Mixed"], plot.it = FALSE)
food.web.res.qq = data.frame(pooled = c(res.qq.3$x, res.qq.2$x, res.qq.mixed$x),
    residual = c(res.qq.3$y, res.qq.2$y, res.qq.mixed$y), dimension = c(rep("Three",
        length(res.qq.3$x)), rep("Two", length(res.qq.2$x)), rep("Mixed", length(res.qq.mixed$x))))
ggplot(food.web.res.qq, aes(pooled, residual)) + geom_point() + geom_abline() +
    facet_wrap(~dimension)
```



It's not perfect, but the three graphs look reasonably similar, so pooling is justifiable.

We'll draw a residual-fit plot to see how much variation our model capture. Again, we'll use `gather()` to get our data in the right shape.

```
food.web.fitted = sort(fitted.values(food.web.lm)) - mean(fitted.values(food.web.lm))
n = length(inv.web.length)
f.value = (0.5:(n - 0.5))/n
food.web.fit = data.frame(f.value, Fitted = food.web.fitted, Residuals = sort(food.web.res))
library(tidyr)
food.web.fit.long = food.web.fit %>% gather(type, value, Fitted:Residuals)
ggplot(food.web.fit.long, aes(x = f.value, y = value)) + geom_point() + facet_wrap(~type)
```



The fitted values are close together compared to the residuals. While the model may be useful, we should remember it only accounts for a fraction of the variation in the data.

2.4 Robust fits

READ: Cleveland pp. 68–79.

Load the usual stuff:

```
library(ggplot2)
load("lattice.RData")
```

2.4.1 Bin packing

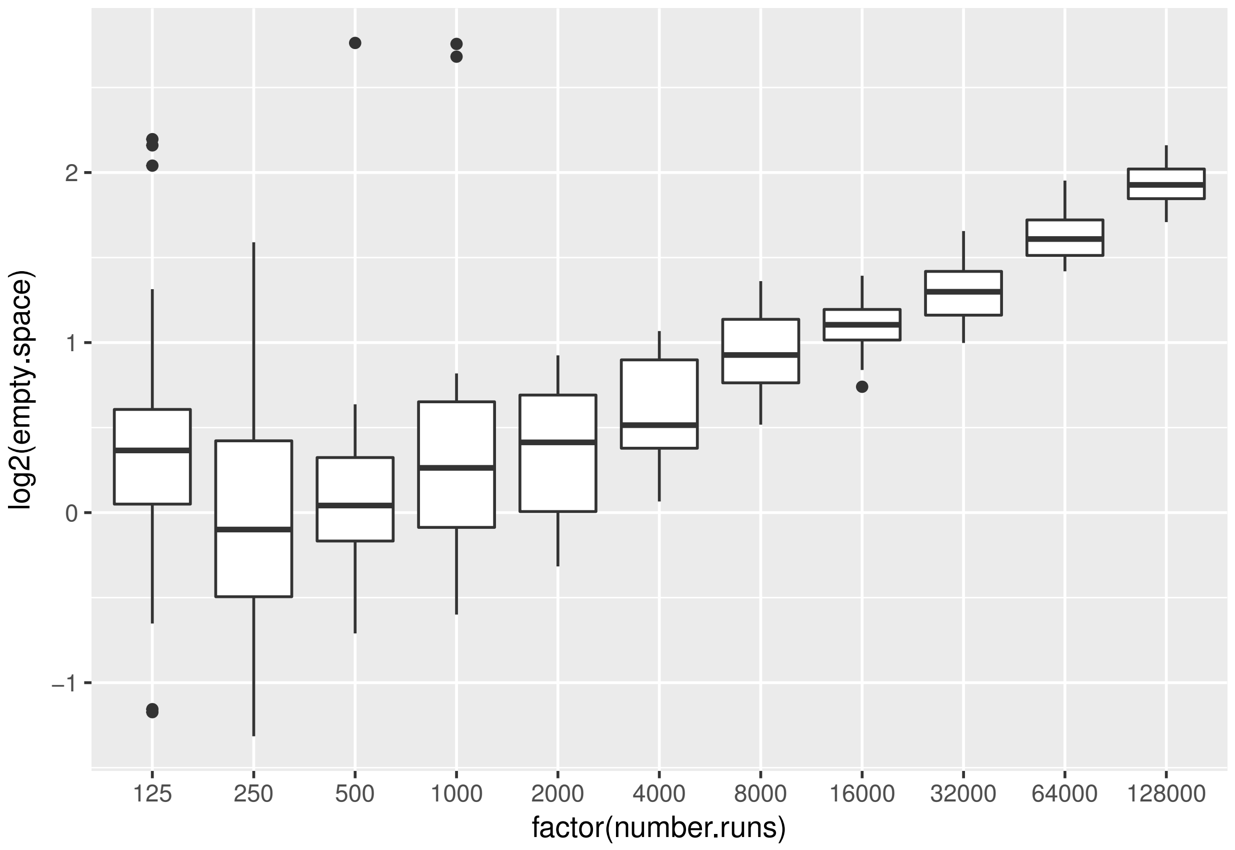
Suppose that we have a number of files, with sizes IID uniform on $[0, 0.8]$. We want to put these files on to disks of capacity 1 using the fewest number of disks possible, i.e.~wasting the smallest amount of empty space. We want to explore the performance of a "bin packing" algorithm that provides an approximate solution.

Cleveland's data frame `bin.packing` contains two variables:

- `empty.space`: The amount of empty space wasted.
- `number.runs`: The number of randomly generated files.

For now we'll treat `number.runs` as categorical. We suspect there could be multiplicative effects, so for now we'll study the base 2 log of `empty.space`. Let's draw boxplots, split by `number.runs`:

```r
ggplot(bin.packing, aes(factor(number.runs), log2(empty.space))) + geom_boxplot()
```

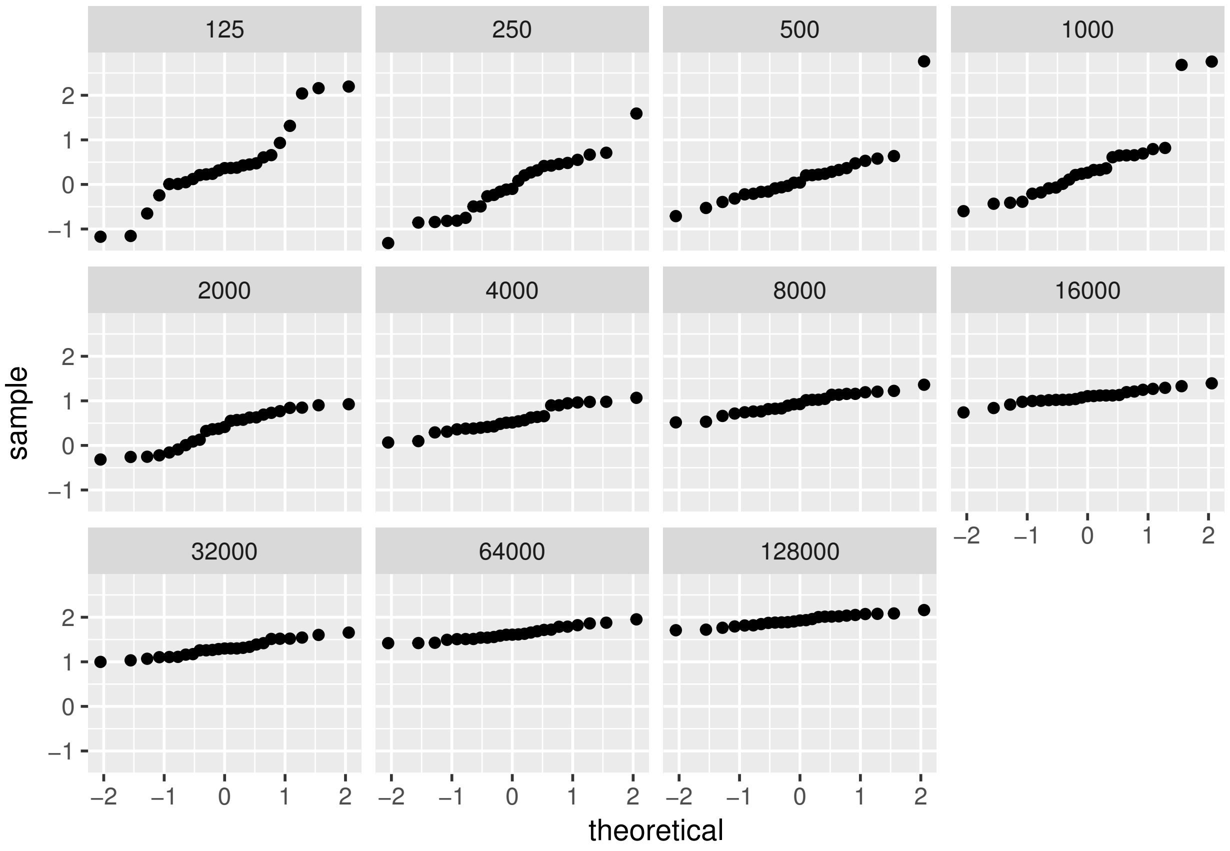


A couple of things are clear:

- Log empty space generally goes up with the number of runs.
- The data isn't homoscedastic: The spread decreases as number of runs increases.

Next we check the normal QQ plots, faceted by the number of runs.

```r
ggplot(bin.packing, aes(sample = log2(empty.space))) + stat_qq() + facet_wrap(~number.runs)
```



For large numbers of runs, the QQ plots are well-fitted by straight lines. However for smallest numbers of runs there are difficulties – especially for less than 1000 runs, where there are major outliers.

Because of the heteroskedasticity and outliers, we might prefer to both build our model and explore our residuals in a more robust way. The median is more outlier-resistant than the mean, so we'll use those as our fitted values.

In Cleveland's notation: Let b_{in} be the ith log empty space measurement for the bin packing run with n weights. Let l_n be the medians. The fitted values are

$$\hat{b}_{in} = l_n$$

and the residuals are

$$\hat{\epsilon}_{in} = b_{in} - \hat{b}_{in}$$

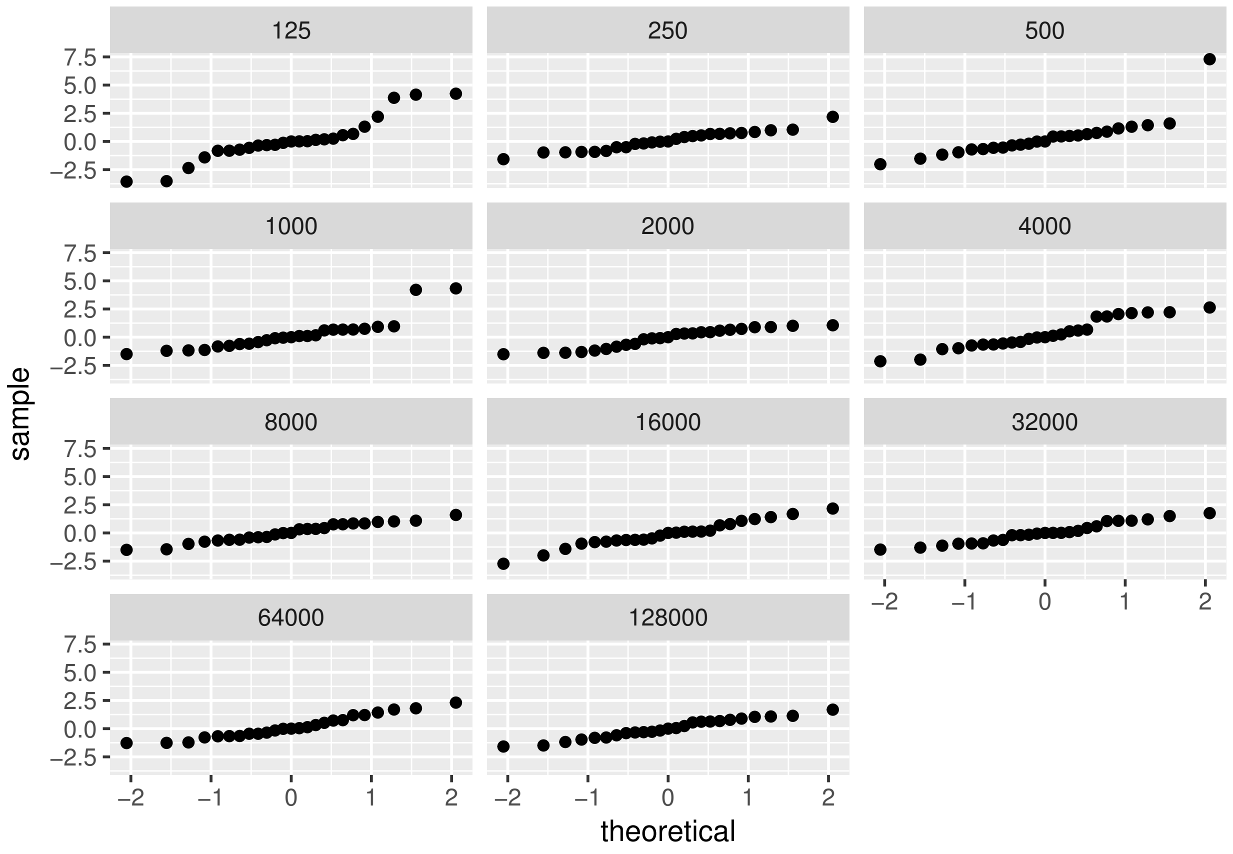
Now we adjust the residuals for heteroscedasticity. Let s_n be the **median absolute deviations** or **mads**: that is, for each n, the median of the absolute value of the residuals. The `mad()` function in R gives the median absolute deviations (multiplied by a constant `1/qnorm(3/4)` to put the estimate on the same scale as the standard deviation.)

The **spread-standardized residuals** are

$$\frac{\hat{\epsilon}}{s_n}$$

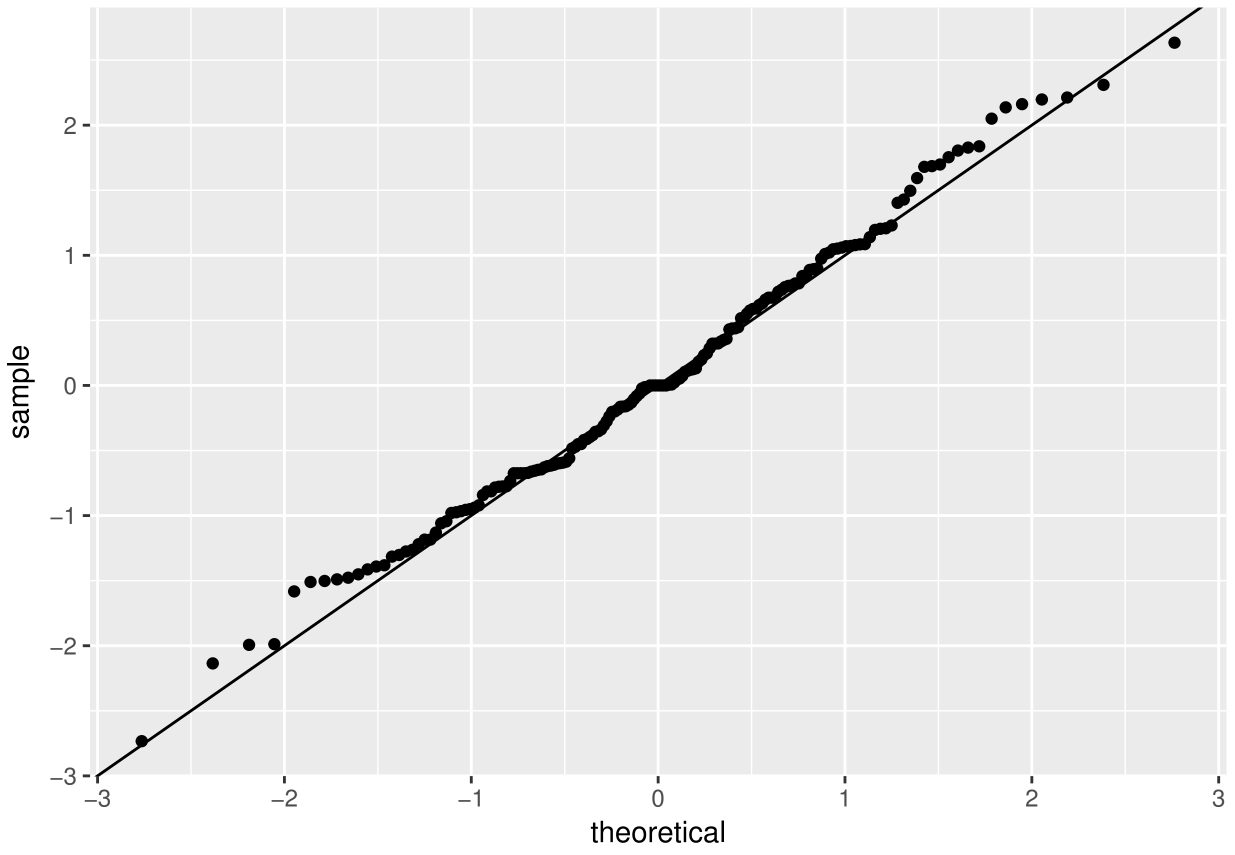
```
number.runs = bin.packing$number.runs
log2.space = log2(bin.packing$empty.space)
```

```
log2.packing = data.frame(log2.space, number.runs)
log2.space.medians = aggregate(log2.space ~ number.runs, median, data = log2.packing)
log2.space.mad = aggregate(log2.space ~ number.runs, mad, data = log2.packing)
n = nrow(log2.packing)
log2.space.fitted = rep(NA, n)
log2.space.madlist = rep(NA, n)
for (J in 1:n) {
    which.runs = which(log2.space.medians$number.runs == number.runs[J])
    log2.space.fitted[J] = log2.space.medians$log2.space[which.runs]
    log2.space.madlist[J] = log2.space.mad$log2.space[which.runs]
}
log2.space.residuals = log2.space - log2.space.fitted
log2.space.standardized = log2.space.residuals/log2.space.madlist
log2.model = data.frame(number.runs, residuals = log2.space.residuals, std.residuals = log2.space.standa
ggplot(log2.model, aes(sample = std.residuals)) + stat_qq() + facet_wrap(~number.runs,
    ncol = 3)
```



It’s still a bit hard to see what’s going on because of the outliers, particularly when the number of runs is 1000 or more. Let’s pool the residuals for those cases and draw a normal QQ plot.

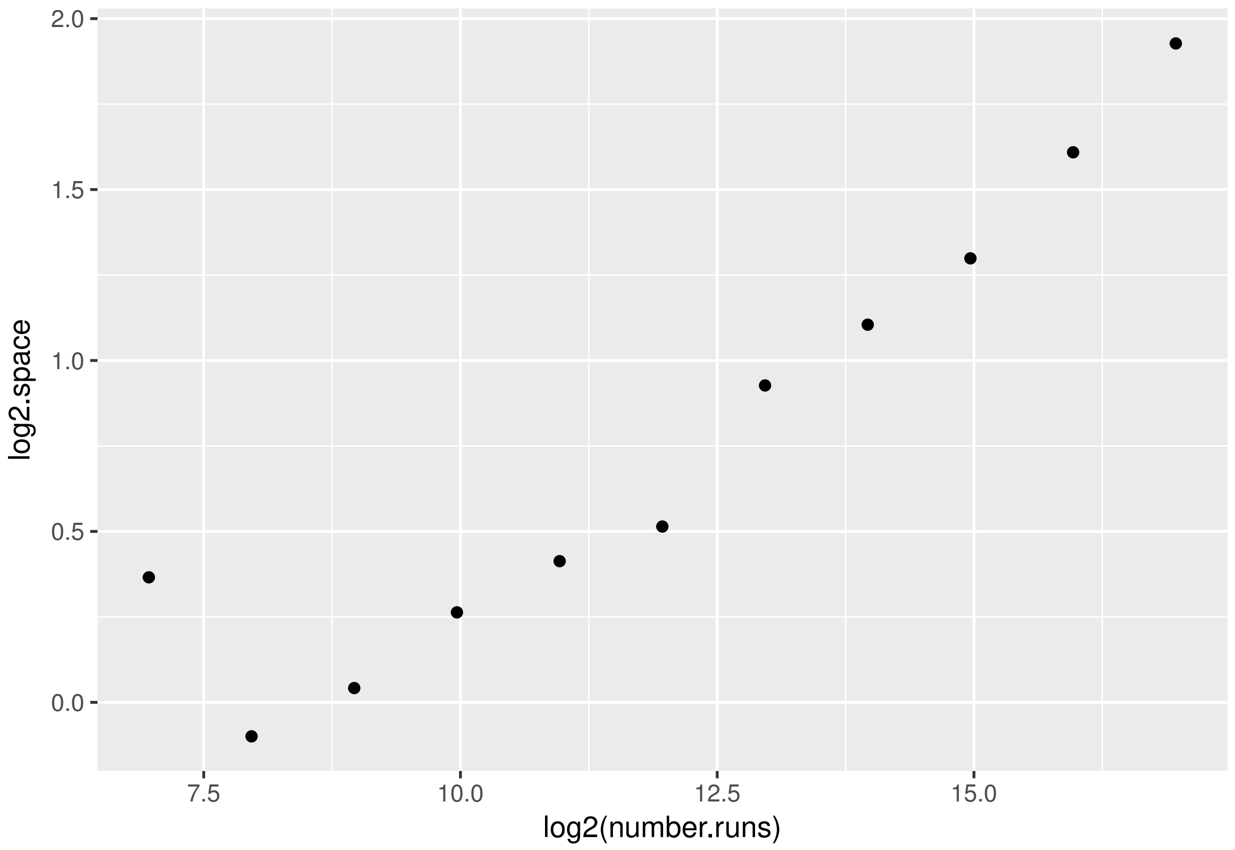
```
log2.model.big.n = log2.model[number.runs > 1000, ]
ggplot(log2.model.big.n, aes(sample = std.residuals)) + stat_qq() + geom_abline()
```



There's an S-shape to the plot. This can happen for many different reasons, but one possibility is over-transformation: maybe our log transformation is hurting rather than helping.

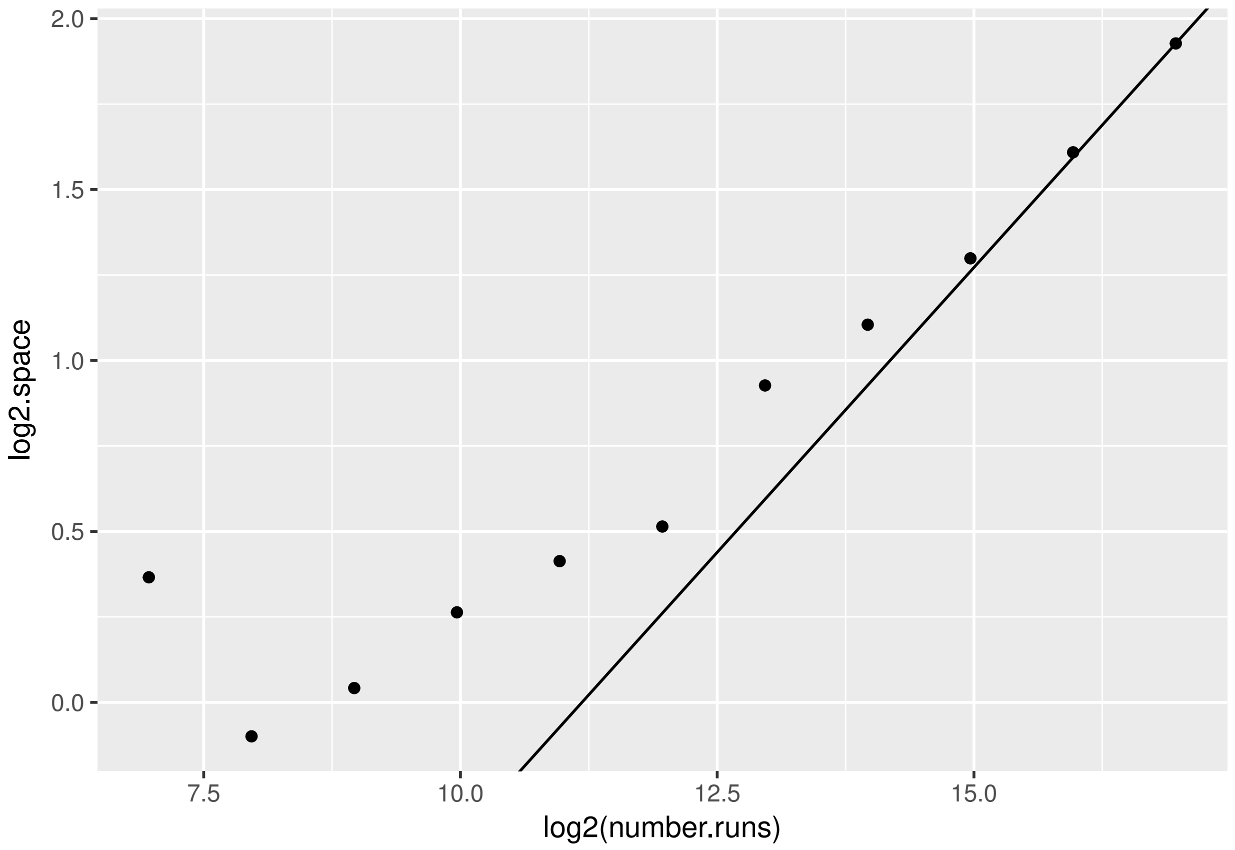
Theory suggests that on a log-log scale, then as the number of runs gets large, empty space approaches a linear function of number of runs with slope 1/3. We plot the median log empty space for each number of runs:

```r
k = nrow(log2.space.medians)
plot248 = ggplot(log2.space.medians, aes(x = log2(number.runs), y = log2.space)) +
    geom_point()
plot248
```



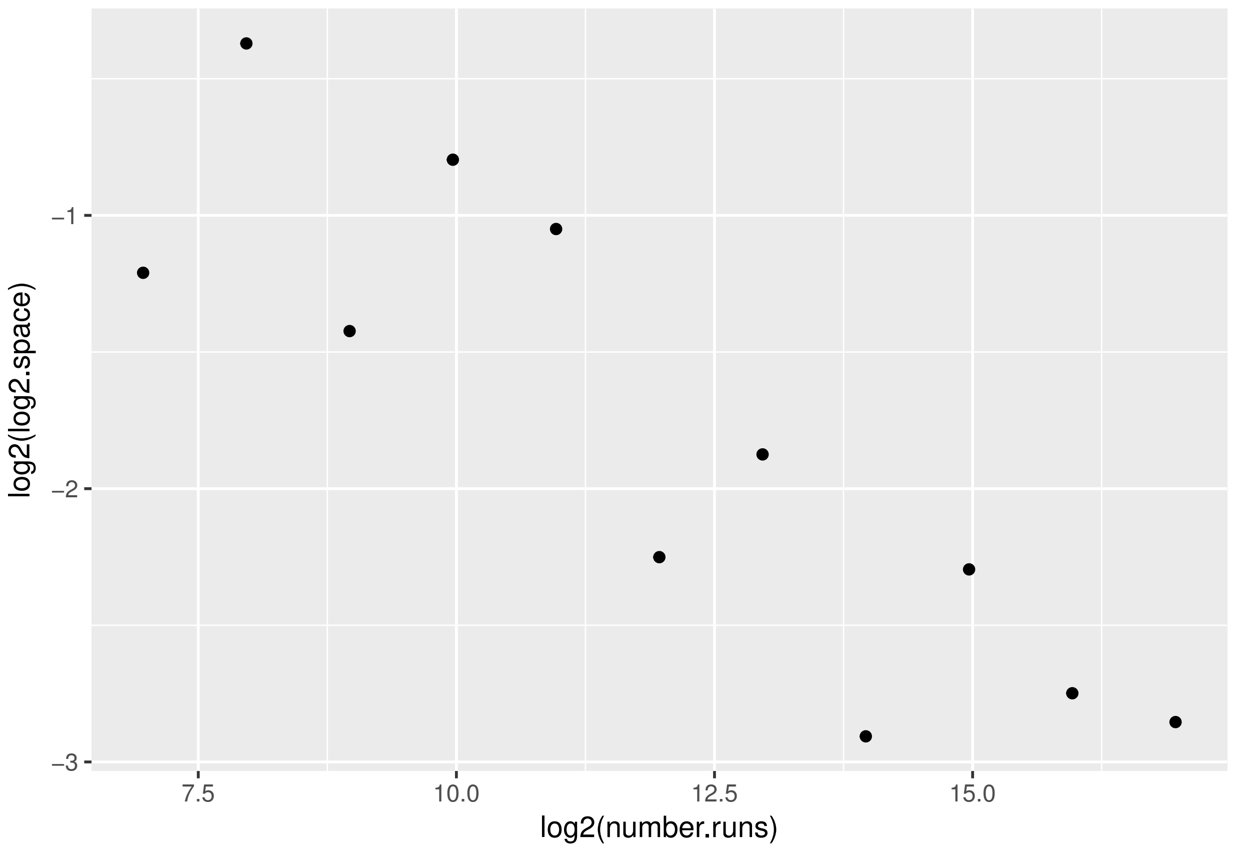
Then we add a straight line with slope 1/3 through the last point:

```
plot248 + geom_abline(slope = 1/3, intercept = log2.space.medians[k, 2] - 1/3 *
    log2(log2.space.medians[k, 1]))
```



The line does eventually provide a good fit, though we would've liked a few more points on the right-hand side to be sure. What about the spreads? Plot the log mads against the log number of runs:

```r
ggplot(log2.space.mad, aes(x = log2(number.runs), y = log2(log2.space))) + geom_point()
```



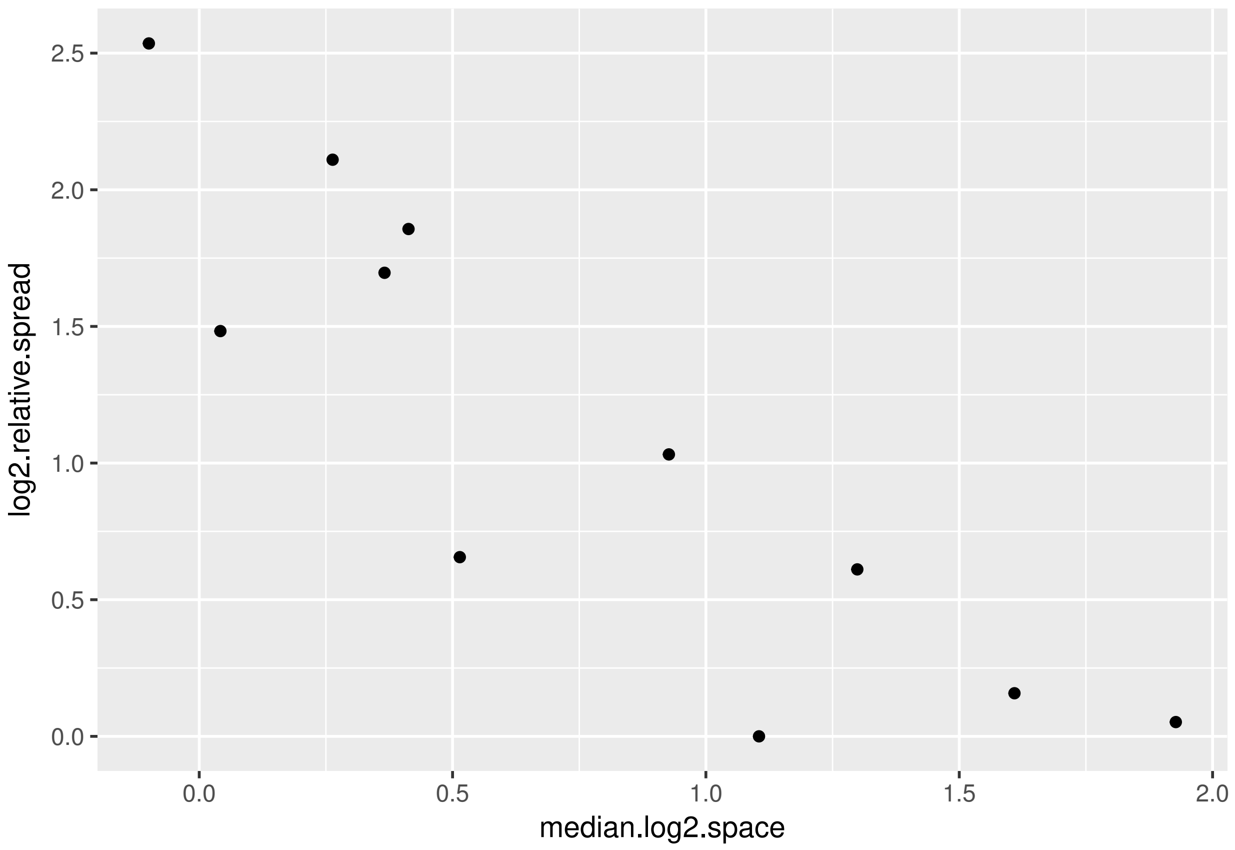
The mads decrease as the number of runs increases. This again is consistent with overtransformation.

To check more thoroughly for heteroscedasticty, we draw spread-location plots.

- For location, we'll use the median log empty space.
- For spread, we'll divide the mads by the smallest mad, then take logs.

If the data is homoscedastic after transformation, this plot should look like random noise.

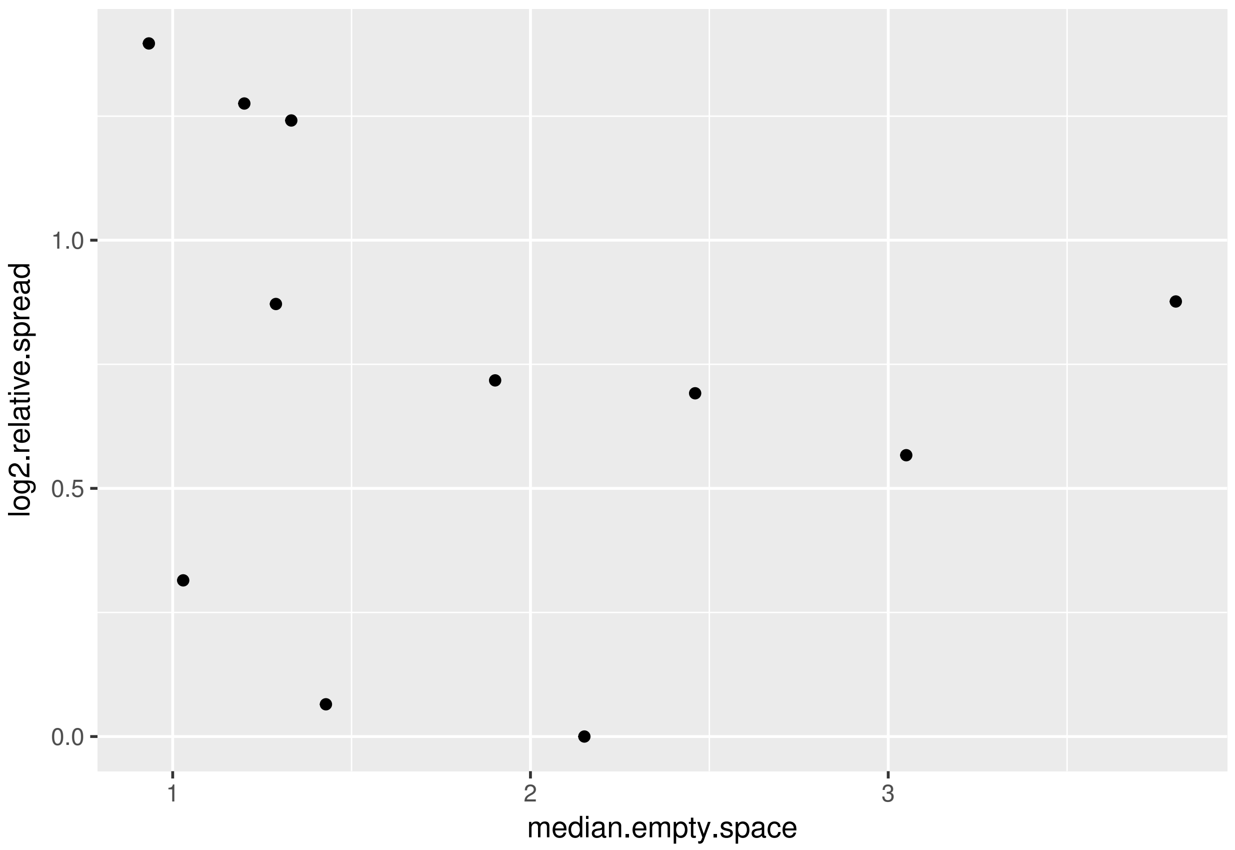
```
median.log2.space = log2.space.medians$log2.space
log2.relative.spread = log2(log2.space.mad$log2.space/min(log2.space.mad$log2.space))
log2.sl = data.frame(median.log2.space, log2.relative.spread)
ggplot(log2.sl, aes(x = median.log2.space, y = log2.relative.spread)) + geom_point()
```



It’s not random noise – it’s a downward sloping line.

We can now be confident that our log transformation has led to heteroscdesticity. In that case, let’s start over again *without* transformation. The spread-location plot is:

```
empty.space.medians = aggregate(empty.space ~ number.runs, median, data = bin.packing)
empty.space.mad = aggregate(empty.space ~ number.runs, mad, data = bin.packing)
empty.space.sl = data.frame(median.empty.space = empty.space.medians$empty.space,
    log2.relative.spread = log2(empty.space.mad$empty.space/min(empty.space.mad$empty.space)))
ggplot(empty.space.sl, aes(x = median.empty.space, y = log2.relative.spread)) +
    geom_point()
```



This time it looks pretty random. We're much closer to heteroscedasticity without the transformation.

The major moral: In EDA, we're allowed to do dumb things because we're explorers. We do need to check our models so that we'll find out if we're doing dumb things. If we are, there's no shame in starting all over again.