Workshop: Basic R and More, Part 3 of 4

Wake Forest School of Medicine
Winston-Salem, NC

Philip Turk, PhD[1]

Director, Biostatistics and Data Science
Center for Outcomes Research and Evaluation (CORE)
Atrium Health; Charlotte, NC

07/09/2021

Atrium Health

---

[1]<Philip.Turk@atriumhealth.org>

Graphs Using `ggplot2`

- The basic plotting commands in R are effective but the graphics are rudimentary and not easily extensible.
- `ggplot2` is a state-of-the-art data visualization package in R (a member of the `tidyverse`).
- On balance, `ggplot2` is every bit the equal of Tableau, Tableau Public, Data Studio, Power BI, etc., for data visualization. All have their pros and cons.
- The package does not fully anticipate what the user wants to do, but rather provides the mechanisms for combining different graphical components which the user chooses.

Graphs Using `ggplot2`

- *The Grammar of Graphics* (Wilkinson, 1999) provided a theoretical framework for the construction of graphics.
- `ggplot2` is based upon a layered grammar of graphics, an extension of Wilkinson's work. This "grammar" gives a framework which enables one to easily update a graphic by modifying a single feature at a time, and suggests aspects of a graph that can be built up component by component.
  - <https://ggplot2.tidyverse.org/reference>
- Along with `dplyr`, `ggplot2` has grown in use to become one of the most popular R packages. Companies now view the ability to work with these packages as a valuable technological skill.

Graphs Using `ggplot2`

- Let's create a *bar chart*. Here we consider a data set that gives the fuel efficiency of different classes of vehicles in two different years 1999 and 2008. This is a subset of data that the EPA makes publicly available and is included in the `ggplot2` package as `mpg`.

```
> data(mpg) ## Loads data set
> ?mpg ## Read about the data set
```

- There are three ways to determine what the columns are. First, we could submit `str(mpg)`, which describes the 'structure' of an R object; in this case, our data frame. Second, we could submit `glimpse(mpg)`, which is like `str()`, but tries to show you as much data as possible. Be aware that `glimpse()` requires the use of the `tibble` package. Third, we could use the Environment tab 👁 👁
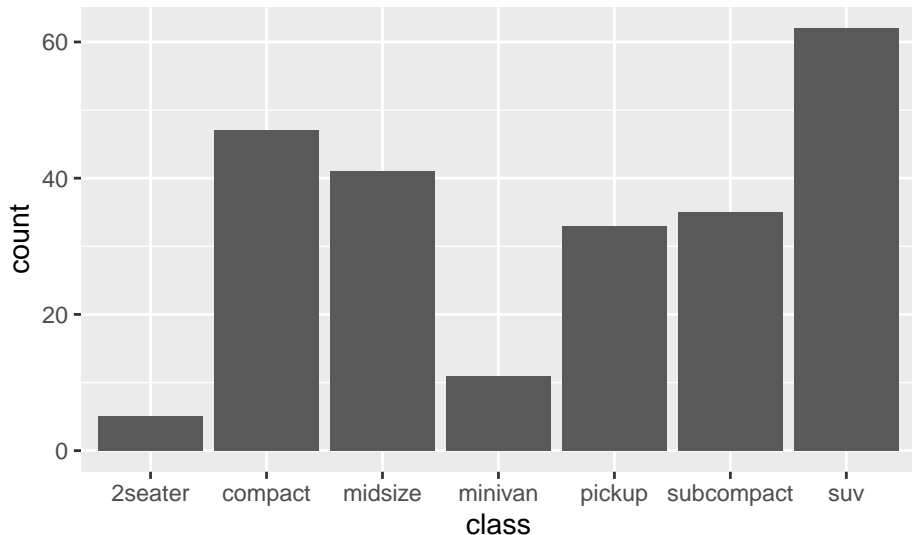
Graphs Using `ggplot2`

```
> plot01 <- ggplot(mpg, aes(x = class))
> plot01 <- plot01 + geom_bar()
```

- aes stands for *aesthetic*, a visual property of the graph. It says the x-axis will be the car's class, which is indicated by the column named `class`.
- geom_bar() creates a bar chart. geom stands for *geometric object*. They represent different ways to display variables and the relationship between them.
- Please note that the plus sign + serves as a pipe operator for `ggplot2`.
- When I create graphs using `ggplot2`, I prefer to code sequentially and save the graph.

Graphs Using `ggplot2`

```
> plot01
```

Graphs Using `ggplot2`

- Suppose we were interested in the distribution of highway miles per gallon (`hwy`).

```
> plot02 <- ggplot(mpg, aes(x = hwy))
> plot02 <- plot02 + geom_histogram()
```
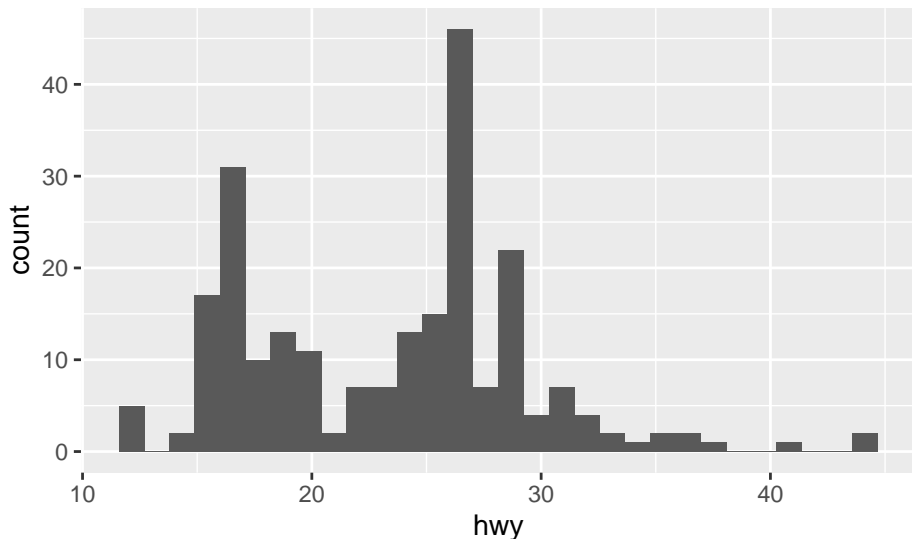
- `geom_histogram()` creates a frequency histogram.
- You will get an ominous message about the default of 30 bins, which gives us too fine a partition of the data. To get a sensible number of classes for a histogram, use the Freedman-Diaconis rule:
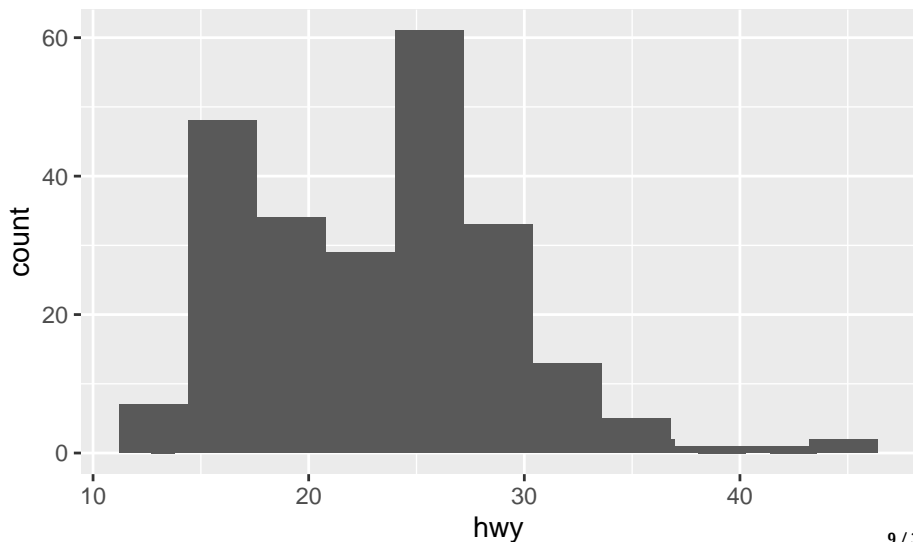
```
> nclass.FD(mpg$hwy)
```

```
[1] 11
```

Graphs Using `ggplot2`

```
> plot02
```

Graphs Using `ggplot2`

```
> plot02 <- plot02 + geom_histogram(bins = 11)
> plot02
```

Graphs Using `ggplot2`
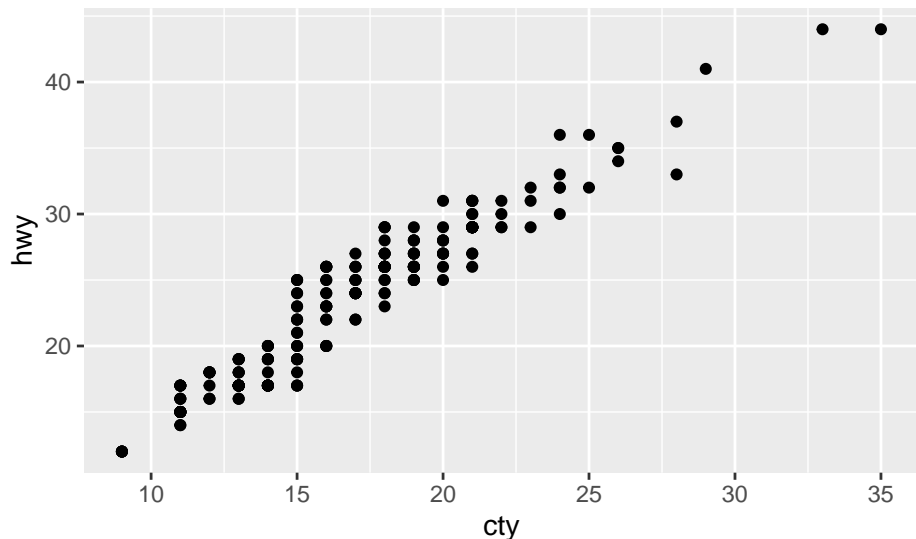
- Suppose we were also interested in the relationship between city miles per gallon (`cty`) and `hwy`.

```
> plot03 <- ggplot(mpg, aes(x = cty, y = hwy))
> plot03 <- plot03 + geom_point()
```

- `geom_point()` draws points for a scatter plot.
- There will be a minor *overplotting* problem. Counterintuitively, adding random noise to a plot can sometimes make it easier to read. *Jittering* is particularly useful for small data sets that are discretized. We will demonstrate jittering momentarily.
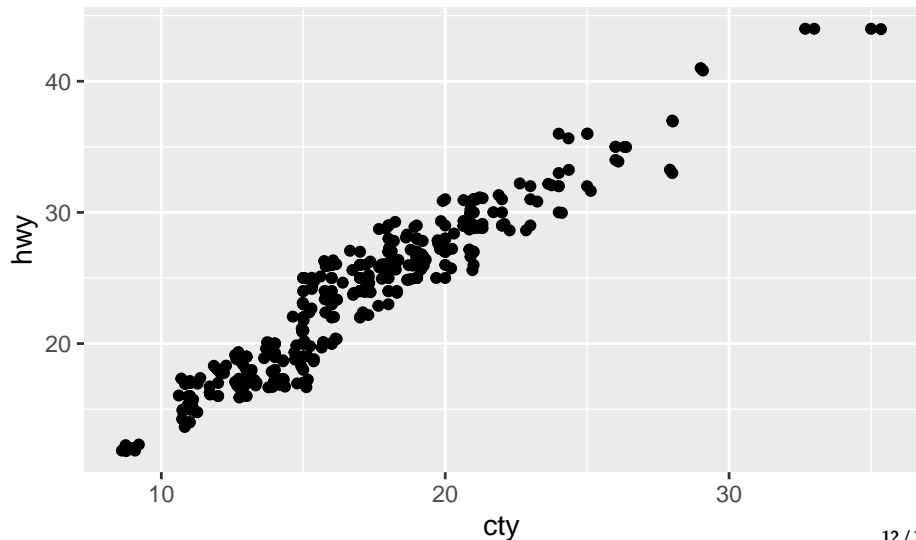
Graphs Using `ggplot2`

```
> plot03
```

Graphs Using `ggplot2`

```
> plot03 <- plot03 + geom_jitter()
> plot03
```

Graphs Using `ggplot2`

- We can also make a coded scatter plot. For example, suppose we were interested in the relationship between `cty` and `hwy` by `class`.

```
> plot03 <- ggplot(mpg, aes(x = cty, y = hwy,
+                           color = class))
> plot03 <- plot03 + geom_point()
> plot03 <- plot03 + geom_jitter()
> plot03 <- plot03 + labs(x = "City MPG", ## Add bling :)
+           y = "Highway MPG",
+           title = "Coded Scatterplot of MPG",
+           color = "Car Type")
```

Graphs Using `ggplot2`

Coded Scatterplot of MPG
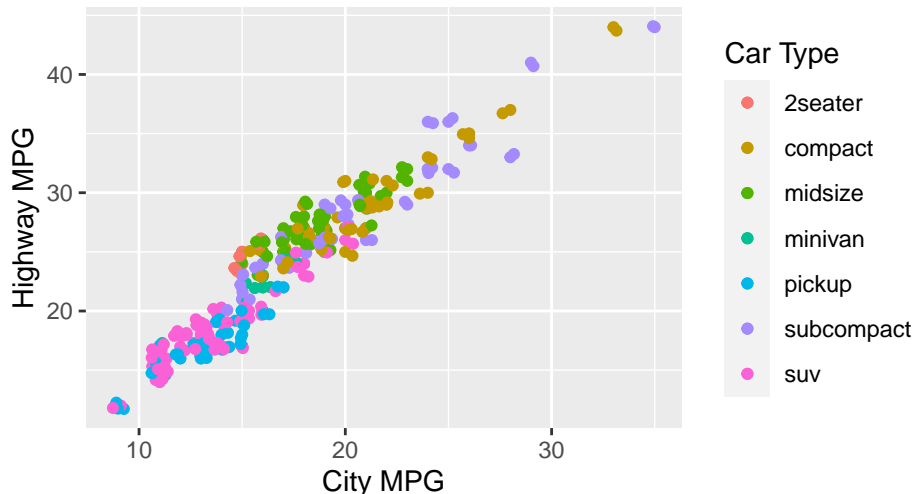
Graphs Using `ggplot2`

- Lastly, we can also make side-by-side box plots. For example, suppose we were interested in the relationship between `cty` by `year`, where `year` is the year of manufacture ({1998, 2008}).

```
> plot04 <- ggplot(mpg, aes(x = factor(year),
+                           y = cty))
> plot04 <- plot04 + geom_boxplot()
```

- `factor()` tells R to treat `year` as a category, not an integer.
- `geom_boxplot()` creates box plots.

Graphs Using `ggplot2`

- `ggplot2` is a powerful, extensible way to obtain even the most complicated (but beautiful!) graphs in order to visualize your data.
- For example, let's suppose you work for the EPA and are interested in the mean city miles per gallon for each year. You then want to superimpose these values directly on each box plot. This is straightforward to do in `ggplot2`.

```
> plot04 <- plot04 + stat_summary(fun = mean,
+             geom = "point", shape = 17,
+             size = 2, color = "red")
```

Graphs Using `ggplot2`

```
> plot04
```
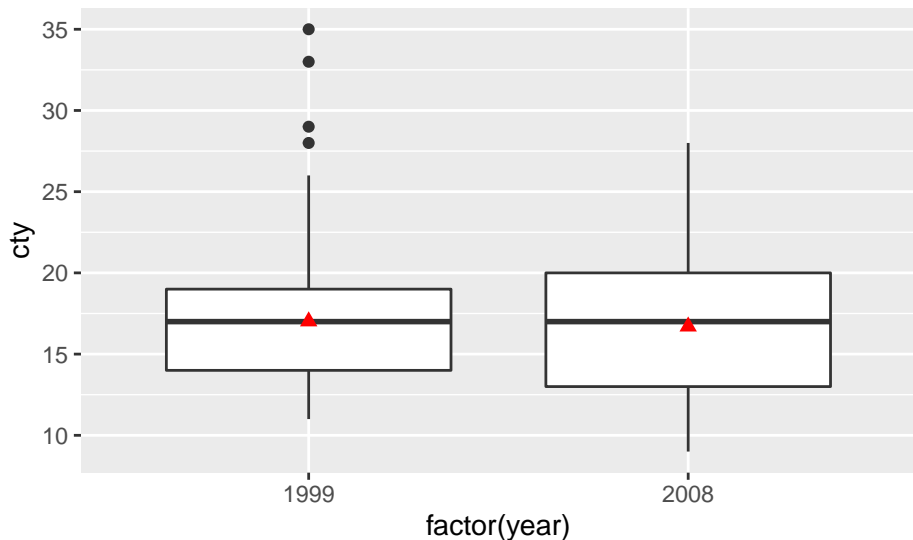
Useful Basic Programming

- It is often necessary to write R code that can perform different actions depending on the data or to automate a task that must be repeated many times. We now cover just a few of these tools.
- Here is an 'if' statement. (The 'else' part is sometimes optional.)

```
> flip <- rbinom(n = 1, size = 1, prob = 0.5)
> flip ## Flip a coin, and get a 0 or 1

[1] 0
```

```
> if(flip == 0){ ## Convert the 0/1 to Tail/Head
+     flip <- "Tail"
+ }else{ ## Keep else on same line as closing } for if
+     flip <- "Head"
+ }
> flip

[1] "Tail"
```

Useful Basic Programming

- In my experience, the `ifelse()` function is very useful because it operates on vectors.

```
> x <- 1:5
> x

[1] 1 2 3 4 5
> ifelse(x <= 2, "Small", "Large")

[1] "Small" "Small" "Large" "Large" "Large"
```

- Be aware that there is also an `if_else()` function in `dplyr` that is faster (but a bit more temperamental).

Useful Basic Programming

- A `while` loop will repeat a section of R code over and over until a stopping condition is met.

```
> x <- pi ## Initialize x
> while(x < 50){ ## Multiple lines of R code
+     x <- 2*x ## Important to update x
+     print(x)
+ }

[1] 6.283185
[1] 12.56637
[1] 25.13274
[1] 50.26548
```

- Aside: When "programming" at the command line for this workshop, do *not* close with a } brace until you are done typing. (We won't program this way in practice.)

Useful Basic Programming

- When we know in advance how many times we should go through a loop of R code, a 'for' loop can be quite useful.

```
> stuff <- NULL ## Create an empty object
> for(i in 1:4){ ## Append elements to stuff
+    stuff[i] <- i*i
+    print(stuff)
+ }

[1] 1
[1] 1 4
[1] 1 4 9
[1]  1  4  9 16
```

- In the for loop, 'i' can be called anything. It's a variable that you choose that is a placeholder.

Useful Basic Programming

- Sometimes it can be useful to define your own functions. A rigorous treatment of R programming would require its own short course :)
- Let's start by defining a function `F_to_C` that converts temperatures from Fahrenheit to Celsius.

```
> F_to_C <- function(temp_F){ ## temp_F is placeholder
+   temp_C <- ((temp_F - 32) * (5 / 9)) ## Do conversion
+   return(temp_C) ## Show result temp_C
+ }
```

Useful Basic Programming

- Calling our own function is no different from calling any other function already in R. We can do it for one temperature (°F) or a vector of temperatures (°F).

```
> F_to_C(32) ## Check  it!

[1] 0
```

```
> lots_temps <- c(0, 32, 68)
> F_to_C(lots_temps)

[1] -17.77778    0.00000   20.00000
```

```
> my_Cs <- F_to_C(lots_temps)
> my_Cs ## Can assign function output to object

[1] -17.77778    0.00000   20.00000
```

Working With Strings Using `stringr`

- Character strings can be an important class of data. Data being read into R often come in the form of character strings where different parts might mean different things.
  - For example a sample ID of "OBGYN06150902" might be a so-called "Patient Code" where OBGYN represents a department acronym, 061509 is the date in MMDDYY format, and 02 represents some designation for a location, doctor, etc.
- We need to have a set of utilities that allow us to split, combine, and examine character strings in a easy and consistent fashion. These will come in handy when managing and wrangling data and in more sophisticated use (e.g., text analytics).
- Here we will introduce a few of the more commonly used functions from the `stringr` package from the `tidyverse`.

Working With Strings Using `stringr`

- The `str_c()` allows us to concatenate two strings or two vectors of strings. This is demonstrated through the following example.

```
> first_names <- c("Michael", "Sarah", "Mitch", "Jennifer")
> last_names <- c("Long", "Pearson", "Jones", "Smith")
> full_names <- str_c(first_names, last_names)
> cbind(first_names, last_names, full_names)

     first_names last_names full_names
[1,] "Michael"   "Long"     "MichaelLong"
[2,] "Sarah"     "Pearson"  "SarahPearson"
[3,] "Mitch"     "Jones"    "MitchJones"
[4,] "Jennifer"  "Smith"    "JenniferSmith"
```

Working With Strings Using `stringr`

- The `str_length()` function calculates the length of each string in the vector of strings passed to it. This could be important in text analytics.

```
> str_length(full_names)
```

```
[1] 11 12 10 13
```

- A powerful string operation is to take a string and match some pattern within it. For example, we might be interested in mining through `full.names` to detect if a pattern occurs.

```
> str_detect(full_names, pattern = fixed("Mi"))
```

```
[1]  TRUE FALSE  TRUE FALSE
```

Working With Strings Using `stringr`

- To figure out positions where the first `Mi` patterns are in each string, we can use the `str_locate()` function.

```
> str_locate(full_names, pattern = fixed("Mi"))

     start end
[1,]     1   2
[2,]    NA  NA
[3,]     1   2
[4,]    NA  NA
```

- One of my favorite `stringr` functions is `str_count()`. It counts the number of designated matches in a string (e.g., the number of vowels/full name). Note that '|' is a logical operator that stands for 'or'.

```
> str_count(full_names, "a|e|i|o|u|A|E|I|O|U")

[1] 4 5 3 4
```

Using Dates and Times in `lubridate`

- Using dates in any software package can be surprisingly difficult.
- They require some special organization because there are several conventions for how to write them and because the sort order should be in the order that the dates occur in time.
- We are going to use the package `lubridate` (which belongs to the tidyverse) to work with dates and times. `lubridate` gets installed as part of the tidyverse installation. However, because it's so specialized, it does not belong to the *core* tidyverse, so you have to load it explicitly with `library(lubridate)`.

Using Dates and Times in `lubridate`

- To create a `Date` object, we need to take a string or number that represents a date and give R a *consistent order* for the date so that it can uniquely identify which piece is the year, the month, and the day.
- We demonstrate the use of the `mdy()` function. Many other functions exist depending on how the date is specified.

```
> library(lubridate)
> mdy("April 16, 2019", "4-16-19", "4-16-2019", "4/16/19")

[1] "2019-04-16" "2019-04-16" "2019-04-16" "2019-04-16"
```

Using Dates and Times in `lubridate`

- Suppose we also had the hours and minutes. Then we can add this time to the date if we wanted to using the `mdy_hm()` function.

```
> mdy_hm("April 16, 2019 1:00 PM", "4-16-2019 13:00")
```

```
[1] "2019-04-16 13:00:00 UTC" "2019-04-16 13:00:00 UTC"
```

- UTC stands for Coordinated Universal Time. However, there are options for specifying a different time zone (e.g., EST, EDT, etc.). See `OlsonNames()` for your personal favorite. Notice the `tz` option.

```
> mdy_hm("4-16-2019 13:00", tz = "US/Eastern")
```

```
[1] "2019-04-16 13:00:00 EDT"
```

Using Dates and Times in `lubridate`

- The `lubridate` package provides many functions for extracting information from the date and time. We demonstrate just a few of them.

```
> x <- mdy_hm("4-16-2019 13:00", tz = "US/Eastern")
> year(x) ## Return year

[1] 2019

> hour(x) ## Return hour of the day

[1] 13

> wday(x) ## Return day of the week (Sunday = 1)

[1] 3

> yday(x) ## Return day of the year

[1] 106
```

Using Dates and Times in `lubridate`

- Once we have two or more `Date` objects defined, we can perform appropriate mathematical operations. For example, we might want to the know the number of days there are between two dates, aka 'tenure time'.
- For example, for some patient, suppose s/he was admitted into the hospital on 2019-6-23 and discharged on 2019-9-15.

```
> Admission <- ymd("2019-6-23") ## "time when"
> Discharge <- ymd("2019-9-15") ## "time when"
> LOS <- Discharge - Admission
> LOS ## "time duration"

Time difference of 84 days
```

- Let's pause for a moment. This is a great demonstration of the power of R as a tool in data science. Why?
- We have demonstrated proof of concept for one patient. But what if a hospital had such data for thousands of people? Scaling up to get *everyone's* time difference would present no challenge at all to R. The creation of a 'time difference' variable is sometimes called *feature extraction* among machine learning modelers.
- This feature could then be used in a machine learning model to help us understand and predict some behavior that might be of interest to the hospital. Of course, the modeling would be done in R in a seamless workflow.