

Data Visualization with R

Principles, Tools and Applications

Jekaterina Zukovska & Sigbert Klinke

<https://hu.berlin/skdv>

2026-06-15

Workshop plan

Prerequisite: Basic R knowledge

16:00	Welcome
16:05	Introduction and Principles: Visualisation, Pitfalls, Chartjunk, Plotting Systems, Exercise: Misleading graphics
16:25	Base R Plots: High-level Plotting Functions, Layering, Low-level Plotting Functions, Multipanel Plots, Exercise: Analysing iris data
17:25	lattice and grid
17:50	Discussion

18:00	Break
18:15	ggplot2
19:15	Interactive graphics: Techniques, plotly, echarts4r, Shiny, Exercise: Explore the Iris Dataset
19:45	Discussion

20:00	End
-------	------------

Packages

- crosstalk – Linked selections across interactive widgets
- DT – Interactive data tables (DataTables)
- echarts4r – Interactive charts via Apache ECharts
- ggiraph – Interactive ggplot2 graphics (hover/click)
- ggplot2 – Grammar of Graphics plotting system
- htmlwidgets – Framework for JavaScript R visualizations
- HKRbook – Datasets and tools for HKR teaching material
- lattice – Trellis conditioned graphics system
- plotly – Interactive web-based plotting library
- shiny – Web app framework for R
- shinydashboard – Dashboard layout system for Shiny apps
- TeachingDemos – Teaching-focused stats and plotting utilities
- tourr – High-dimensional data visualization tours

```
install.packages(c("crosstalk", "DT", "echarts4r", "ggiraph",  
  "ggplot2", "htmlwidgets", "HKRbook", "lattice", "plotly",  
  "shiny", "shinydashboard", "TeachingDemos", "tourr"))
```

Introduction and Principles

Why Visualization?

A plot is not decoration — it is a statistical instrument

Why does this matter? You don't just “see” data — you discover it.

- A table of 100 numbers → hard to interpret.
- A scatterplot → instantly shows: “Hey, there are two groups!”
- Visualization = Discovery: “Why are there two clusters?” → That's where research begins.

Tip

- Always plot your data before running models. Even simple plots can reveal outliers, patterns, or data issues ([Anscombe's Quartet](#)).

Visualization is not just about seeing patterns — it's about modeling the data-generating process (Tukey, 1977)

- It's a cognitive tool that reduces cognitive load by offloading mental processing onto the visual system.
- It supports exploratory data analysis (EDA) as a form of hypothesis generation, not just confirmation.
- As Tukey said: “The greatest value of a picture is when it forces us to notice what we never expected to see.”

Critical note

- Visual perception is not objective — it's influenced by expectations, context, and bias ([the invisible gorilla](#)).
- That's why well-designed visuals are essential — to minimize misinterpretation.

Tukey, J. W. (1977). Exploratory Data Analysis. Addison-Wesley.

Visualization in the Analytical Pipeline

Visualization is relevant throughout the analysis pipeline

- Data cleaning: Spot missing values or weird numbers (e.g., age = 200).
- Transformation: See if a log-scale helps reveal a pattern.
- Model building: Check if your model fits the data (e.g., residuals plot).
- Diagnostics: Find outliers or violations of assumptions.
- Communication: Show your findings clearly to others.

Tip

- Use a simple plot before every step — it saves hours of debugging later.

Choosing the Right Visualization

- What to visualize: distribution, parameters, summary statistics, relationships, etc.
- Variable type(s)
- Number of observations
- Number of variables

Common Pitfalls

- Interpretation depends on experience and bias
- Incorrect representation leads to incorrect conclusions

Exploratory vs. Communicative Visualization

The purpose of a plot determines its design

- Exploratory = Your lab notebook
 - ▶ You're testing ideas, trying things, making messes.
 - ▶ You want everything visible — even if it's cluttered.
- Communicative = Your research paper or presentation
 - ▶ You're telling a story.
 - ▶ You remove distractions and focus on one key message.

Tip

- Use exploratory plots when you're learning your data.
- Use communicative plots when you're presenting to others.

- Exploratory Visualization
 - ▶ Hypothesis generation
 - ▶ Structure discovery
 - ▶ Anomaly detection
 - ▶ Model diagnostics
 - ▶ High information density
 - ▶ Iterative process
- Core Purpose
 - ▶ Discover patterns, test assumptions, refine models
 - ▶ Driven by curiosity and data-driven insight
- Objective
 - ▶ Visualization as a tool for model criticism
- Communicative Visualization
 - ▶ Hypothesis presentation
 - ▶ Argument construction
 - ▶ Audience-centered design
 - ▶ Reduced information density
 - ▶ Focus on key messages
- Core Purpose
 - ▶ Convey insights clearly and persuasively
 - ▶ Guide the audience to a specific conclusion
- Shift in Objective
 - ▶ From *discovering* patterns to *communicating* findings

Avoiding Systematic Errors

Visibility determines interpretability

- Problems

- ▶ Overplotting = Like trying to read a book where every word is written on top of the others → You can't see individual letters → you can't read.
- ▶ Hidden structure = The pattern exists, but it's buried under noise → Like a whisper in a noisy room.
- ▶ Artificial clustering = You think you see groups, but they're just random → Like seeing faces in clouds.

- Ask yourself

- ▶ "Can I actually see what's going on?"
- ▶ "Is this just a trick of the eye?"

Tip

- Use `alpha = 0.5` (transparency) for overlapping points
- Try `jitter()` (random noise) for categorical/discrete data
- Use binning for 2D density.
- If you have many points, subsample or aggregate.

- Statistical problem

- ▶ Overplotting → leads to underestimation of density and misjudgment of distribution shape. It can mask outliers or clusters, leading to incorrect model assumptions.
- ▶ Hidden structure → occurs when visual encoding fails to reveal underlying patterns (e.g., non-linear relationships, heteroscedasticity). This undermines exploratory analysis and model diagnostics.
- ▶ Artificial clustering → arises from perceptual grouping (Gestalt principles), where the human brain interprets random patterns as meaningful groups - a well-documented cognitive bias.

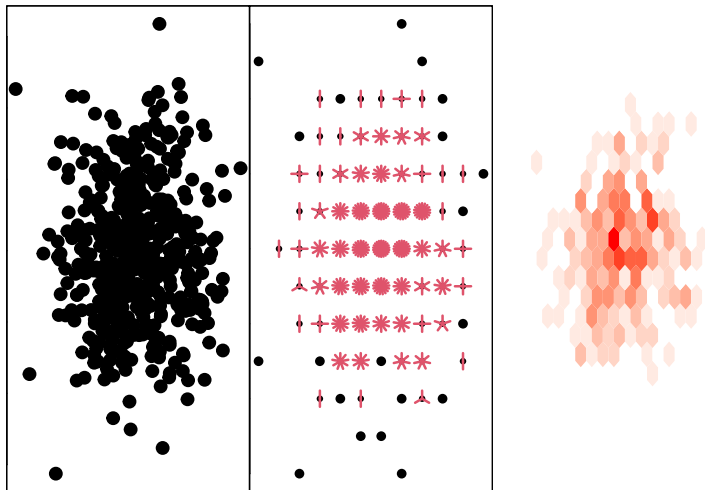
- References

- ▶ Cleveland, W. S. (1993). Visualizing Data. Hobart Press.
- ▶ Tufte, E. R. (1983). The Visual Display of Quantitative Information. Graphics Press.

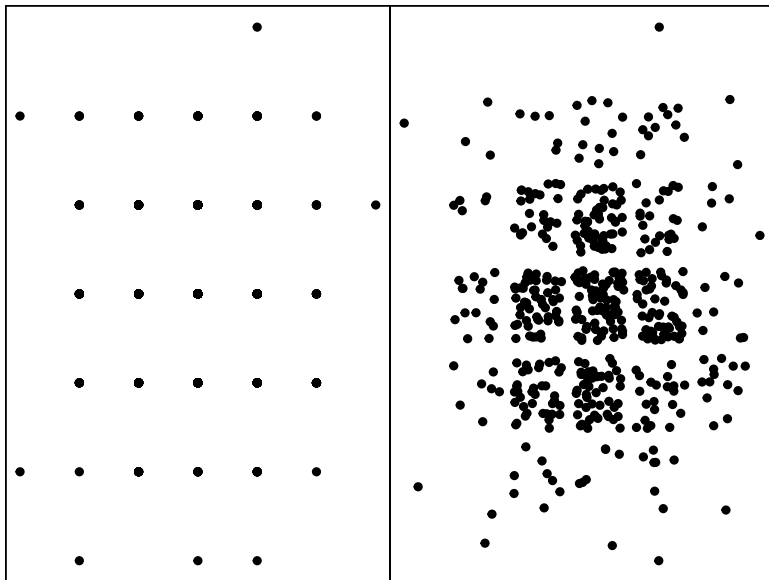
Tip

- Always validate your visual findings with quantitative checks (e.g., correlation, clustering algorithms, residual analysis)
- Use visual + statistical validation — never rely solely on what “looks right”

Overplotting vs. Sunflower & Hexagonal bins



Overplotting vs. Jittering



Misleading Scales in Data Visualization

- **Undisclosed transformations**

- ▶ Logarithmic or other transformations alter perceived slope and variance — without warning, they mislead.

- **Unequal bin widths**

- ▶ In histograms, can create artificial peaks and distort the true shape of the distribution.

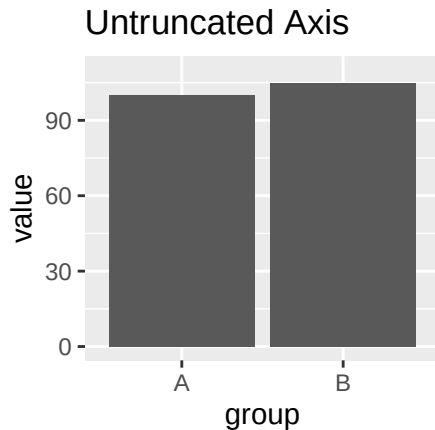
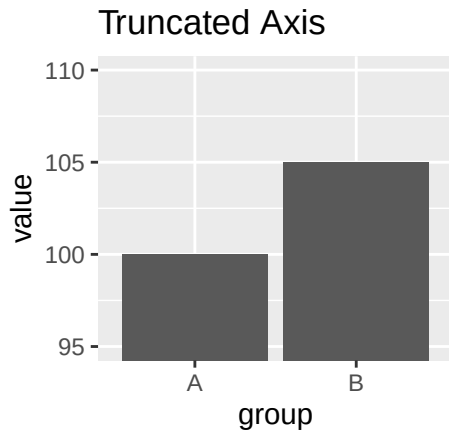
- **Dual axes**

- ▶ Risk suggesting spurious correlations due to arbitrary or incompatible scaling.

- **Aspect ratio**

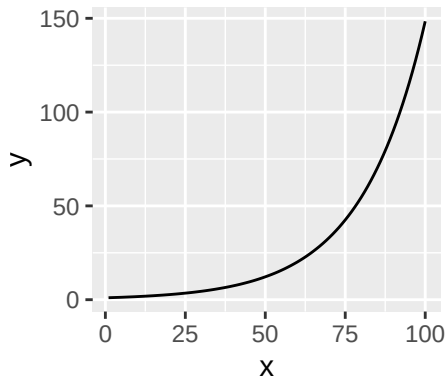
- ▶ Influences perceived slope; a steeper line may appear more dramatic than it is.

Truncated Baseline

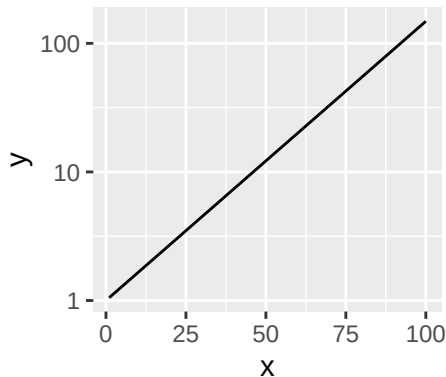


Log Scale Without Clear Framing

Linear Scale

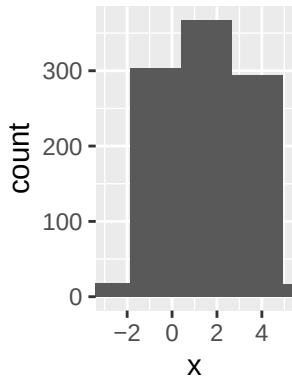


Log Scale

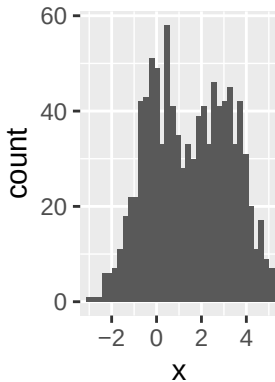


(Unequal) Binning

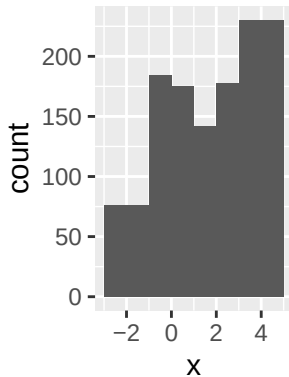
Coarse



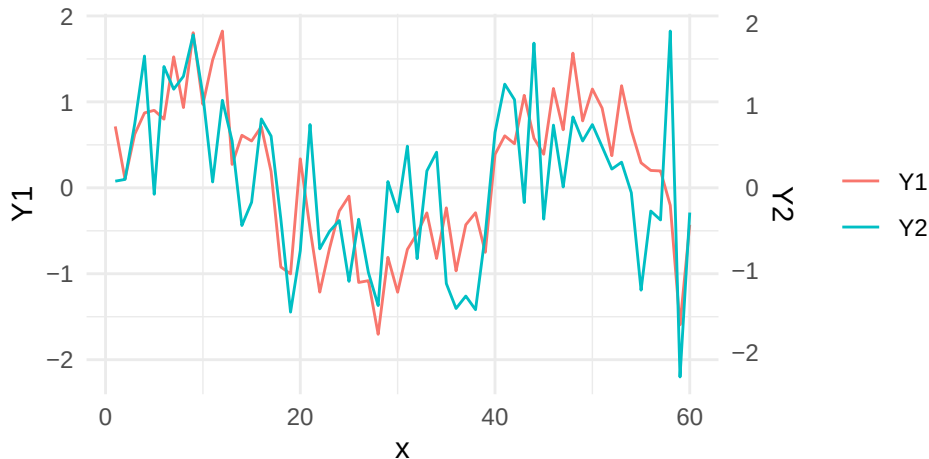
Finer



Unequal



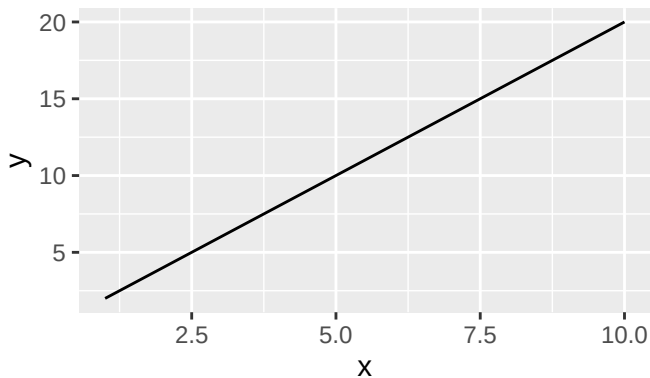
Dual Axes



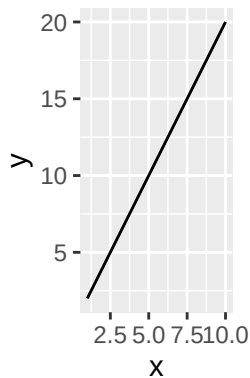
- Dual axes can visually imply relationships that are purely artifacts of scaling

Aspect Ratio Distortion

Default



Distorted



It's anything in a chart that doesn't help you understand the data

- Examples

- ▶ Background images (e.g., a coffee cup behind a bar chart)
- ▶ Fancy borders or 3D effects
- ▶ Too many gridlines
- ▶ Ornamental icons (e.g., little stars or flags)
- ▶ Overlapping lines that create Moiré patterns

Tip

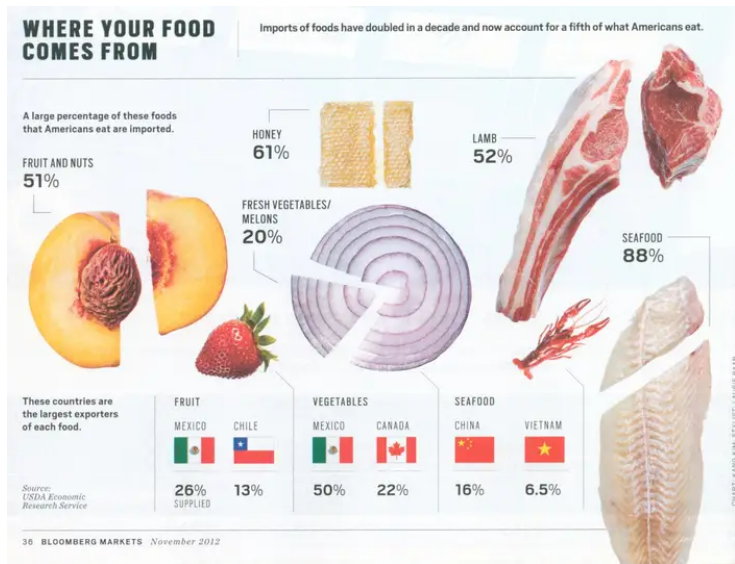
- *If I remove this, will the message still be clear?* If yes → remove it.

The best graphs are those that convey the most information with the least ink (Tufte, 1983)

- Cognitive load: Limited mental resources for processing visual information
- Signal-to-noise ratio: Minimizing irrelevant visual elements
- Pre-attentive interference: Irrelevant features drawing attention away from data
- Practice:

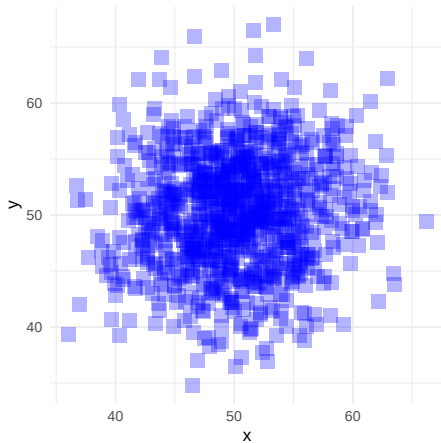
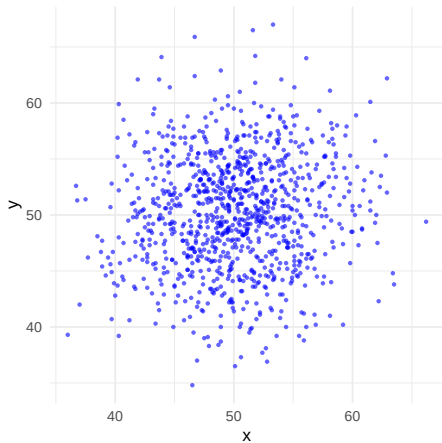
$$\frac{\text{Ink/Pixel for data}}{\text{Ink/Pixel}} \longrightarrow \text{maximize}$$

Background Images

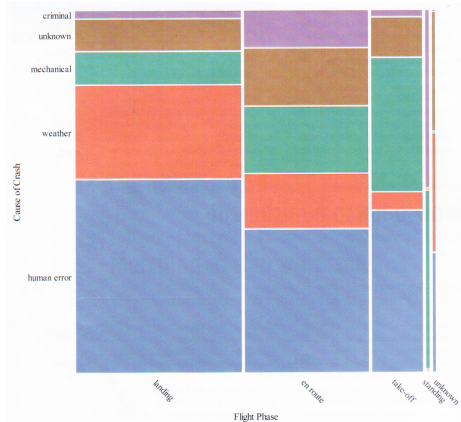
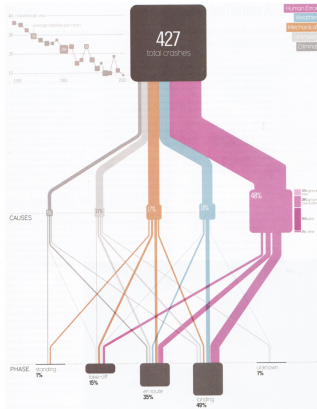


36 BLOOMBERG MARKETS November 2012

Scatterplot: Clean vs. Moiré



Air Crash Statistics (Cause vs. Phase)



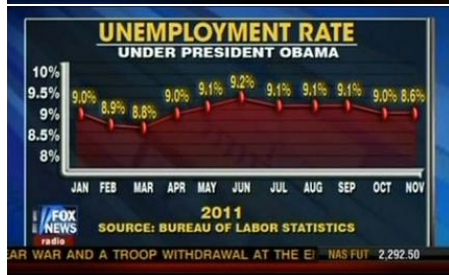
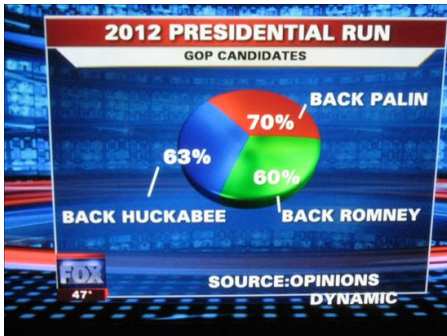
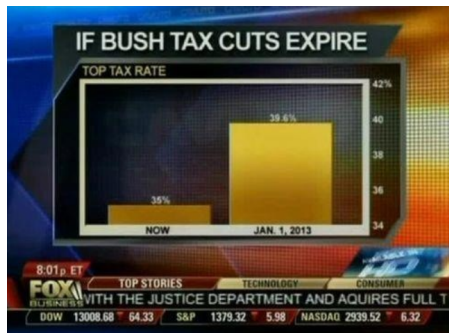
KISS = Keep It Simple, Stupid!

Base R vs lattice vs ggplot2

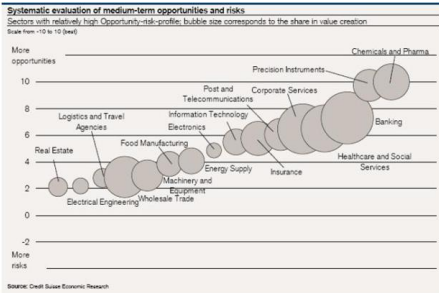
Aspect	Base R	lattice	ggplot2
Paradigma	Command-based	Formula-based	Grammar-based
Layout	Manual	Multi-panel	Facets / patchwork
Overplotting	Manual	Limited	Built-in
Flexibility	High (low-level control)	Medium (structured)	High (layered design)
Composability	Low (stateful)	Moderate (function-based)	High (layered, declarative)
Interactivity	None	None	Optional
Learning curve	Low	Moderate	Steep
Reverse import	?	400+	1000+

- Base R: Becker, Chambers, & Wilks (1988) – The New S Language.
- Lattice: Sarkar (2008) – Lattice: Multivariate Data Visualization with R.
- ggplot2: Wickham (2009) – ggplot2: Elegant Graphics for Data Analysis.

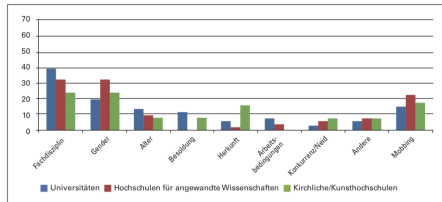
Missleading Graphics I



Missleading Graphics II



Die häufigsten Diskriminierungsgründe in Abhängigkeit vom Hochschultyp



Base R Plots

mtcars Dataset

- **Dataset:** mtcars (Motor Trend Car Road Tests, 1974)
- **Observations:** 32 cars
- **Variables:** 11 numeric variables
- **Purpose:** Explore relationships between car features and fuel efficiency (mpg).

Variable	Description (English)	Beschreibung (Deutsch)
mpg	Miles/(US) gallon	Meilen pro Gallone
cyl	Number of cylinders	Anzahl der Zylinder
disp	Displacement (cuIn.)	Hubraum
hp	Gross horsepower	Brutto-PS
drat	Rear axle ratio	Hinterachsübersetzung
wt	Weight (1000 lbs)	Gewicht
qsec	1/4 mile time	Zeit auf 1/4 Meile
vs	Engine (0=V, 1=straight)	Motor
am	Transmission (0=auto, 1=man.)	Getriebe
gear	# forward gears	# Vorwärtsgänge
carb	# carburetors	# Vergaser

plot |

- Command: `plot(x, y, ...)` – creates a plot dependent on `class(x)`
- Generic function: adapts to numeric, factor, or time series data
 - ▶ Scatterplot: `plot(x, y)` → numeric x & y → default scatterplot.
 - ▶ Factor + numeric: `plot(factor, numeric)` → creates boxplot-style plot automatically.
 - ▶ Time series object: `plot(ts_object)` → R automatically plots a line chart over time.

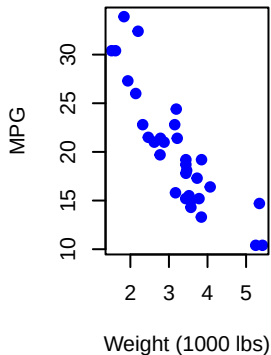
```
plot(mtcars$wt, mtcars$mpg, main="wt vs mpg", xlab="Weight (1000 lbs)",  
     ylab="MPG", pch=19, col="blue")  
plot(factor(mtcars$cyl), mtcars$mpg, main="mpg by cylinders",  
     xlab="Cylinders", ylab="MPG", col="lightgreen")  
plot(AirPassengers, main="AirPassengers", xlab="Year", ylab="Passengers",  
     type="o", col="red")
```

```
class(AirPassengers)  
[1] "ts"
```

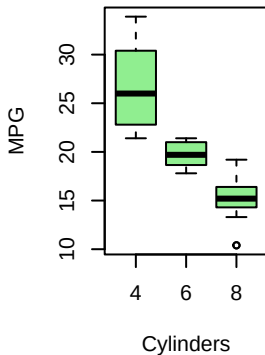
`plot` → `plot.ts` or if `plot.ts` not exists `plot.default`

plot II

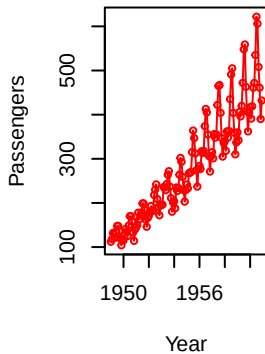
wt vs mpg



mpg by cylinders

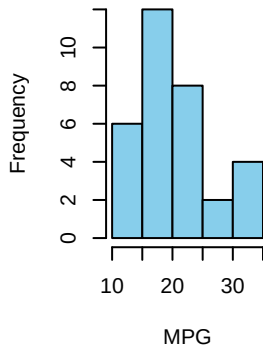


AirPassengers

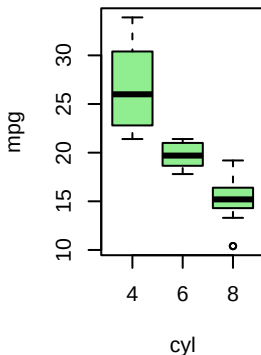


Distribution / Summary Plots

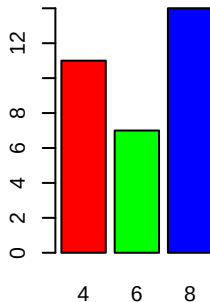
hist()



boxplot()



barplot()



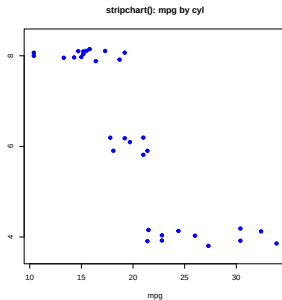
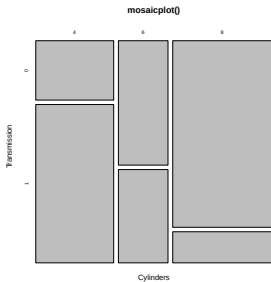
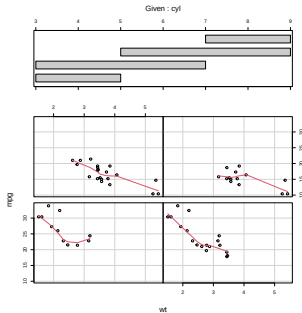
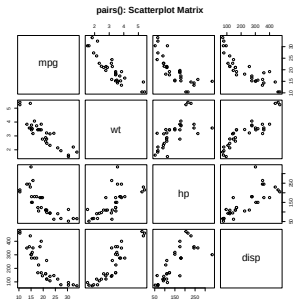
```
hist(mtcars$mpg, main="hist()", col="skyblue", xlab="MPG")
boxplot(mpg ~ cyl, data=mtcars, main="boxplot()", col="lightgreen")
barplot(table(mtcars$cyl), main="barplot()", col=c("red","green","blue"))
?funcname # help for funcname
```

Relationship Plots I

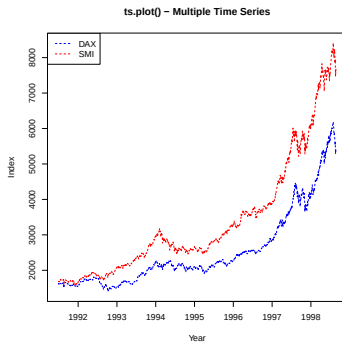
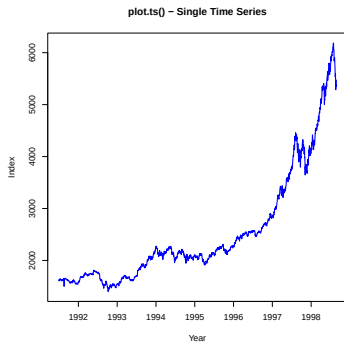
- Relationship plots help us understand associations between variables.
- They are pre-model diagnostics: before running regressions, ANOVA, or other analyses, you want to know:
 - ▶ Are variables linearly related?
 - ▶ Are there clusters or outliers?
 - ▶ Are there interaction effects?
 - ▶ Are assumptions (like homoscedasticity) likely met?

```
pairs(mtcars[, c("mpg", "wt", "hp", "disp")],  
      main="pairs(): Scatterplot Matrix")  
coplot(mpg ~ wt | cyl, data=mtcars, panel=panel.smooth,  
       main="coplot(): mpg ~ wt | cyl")  
mosaicplot(table(mtcars$cyl, mtcars$am), xlab="Cylinders",  
            ylab="Transmission", main="mosaicplot()")  
stripchart(mpg ~ cyl, data=mtcars, method="jitter", pch=19,  
           col="blue", main="stripchart(): mpg by cyl")
```

Relationship Plots II

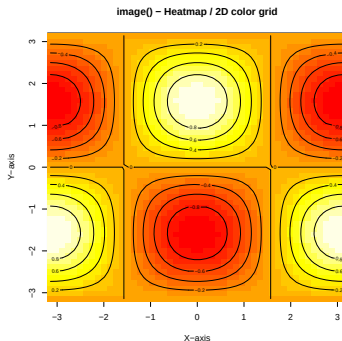
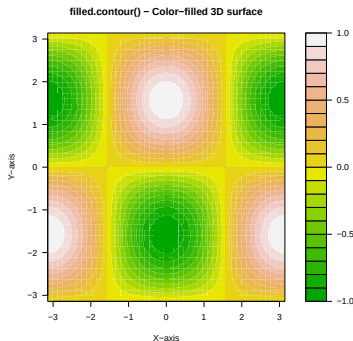


Time Series / Line Plots



```
plot.ts(EuStockMarkets[,1], col="blue", lwd=2, main="plot.ts()",
        ylab="DAX", xlab="Year")
ts.plot(EuStockMarkets[,1:2], col=c("blue","red"), lty=2,
        gpars=list(xlab="Year", ylab="Index", main="ts.plot()"))
legend("topleft", legend=c("DAX", "SMI"), col=c("blue", "red"), lty=2)
```

Specialized Plots



```
y <- x <- seq(-pi, pi, length=50)
z <- outer(x, y, function(x,y) cos(x)*sin(y)) # 2D function
filled.contour(x, y, z, color.palette=terrain.colors,
               xlab="X-axis", ylab="Y-axis",
               main="filled.contour() - Color-filled 3D surface")
image(x, y, z, col=heat.colors(20), xlab="X-axis",
      ylab="Y-axis", main="image() - Heatmap / 2D color grid")
contour(x, y, z, add=TRUE)
```

The Formula Interface in R Plots

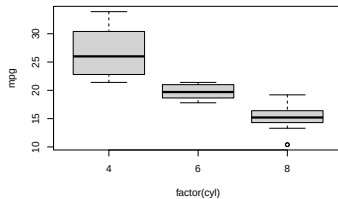
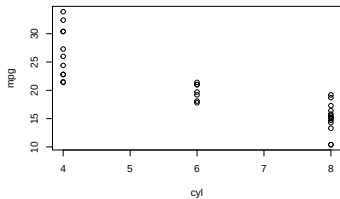
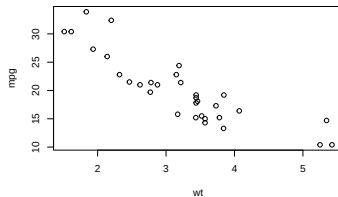
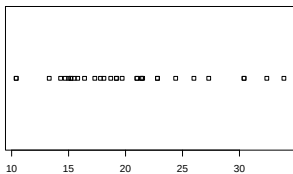
**The formula interface is a powerful way to tell R:
"Show me the relationship between these variables."**

```
plot(~ mpg, data = mtcars)
plot(mpg ~ wt, data = mtcars)
plot(mpg ~ cyl, data = mtcars)
plot(mpg ~ factor(cyl), data = mtcars)
```

Tip

● Rules:

- ▶ Use ~ (tilde) — not =
- ▶ y is the variable you want to explain or show
- ▶ x is the variable you think influences it
- ▶ data tells R where to find the variables



- The formula interface is not just syntax — it's a unified modeling language in R
 - ▶ Visualization
 - ★ `lattice::xyplot(mpg ~ wt | cyl, data = mtcars)`
 - ▶ Modeling:
 - ★ `lm(mpg ~ cyl, data = df)`
 - ★ `glm(am ~ wt, data = mtcars, family = binomial)`
 - ▶ Data manipulation:
 - ★ `aggregate(mpg ~ cyl, data = mtcars, FUN = mean)`
- Examples
 - ▶ $y \sim x \rightarrow$ response variable (y) modeled as a function of predictor (x)
 - ▶ $y \sim x_1 + x_2 \rightarrow$ multiple predictors
 - ▶ $y \sim x_1 * x_2 \rightarrow$ includes interaction terms
 - ▶ $y \sim \text{factor}(x) \rightarrow$ treats x as categorical
 - ▶ $y \sim x \mid \text{group} \rightarrow$ conditioning
- Delays evaluation until the data is provided (via `data =`)
- Enables automatic handling of grouping, faceting, and conditioning

Univariate Plots

Function	Input Type	Primary Purpose
<code>boxplot()</code>	Formula/List	Summarize distribution
<code>density()</code>	Vector	Estimate density
<code>hist()</code>	Vector	Show distribution
<code>stem()</code>	Vector	Text-based stem plot
<code>stripchart()</code>	Vector	1D scatter for small samples
<code>barplot()</code>	Vector/Matrix	Compare category magnitudes
<code>dotchart()</code>	Vector	Readable alternative to barplot
<code>pie()</code> 🤖	Vector	Show proportions

Bivariate Plots

Function	Input Type	Primary Purpose
<code>sunflowerplot()</code>	Two Vectors	Scatterplot for overlapping points
<code>smoothScatter()</code>	Two Vectors	Density-based scatterplot
<code>boxplot()</code>	Formula	Summarize distribution by group
<code>cdplot()</code>	Formula/Factor	Conditional density plot
<code>spineplot()</code>	Formula/Factor	Stacked bar by group size
<code>assocplot()</code>	Matrix	Deviations from independence
<code>mosaicplot()</code>	Table/Matrix	N-way contingency visualization
<code>fourfoldplot()</code>	2x2xK Table	2x2 association across strata

Multivariate Plots

Function	Input Type	Primary Purpose
<code>pairs()</code>	Data Frame	Scatterplot matrix
<code>coplot()</code>	Formula	Conditional relationships
<code>stars()</code>	Matrix/DF	Star/spider chart
<code>symbols()</code>	Vectors	Shapes with mapped sizes
<code>contour()</code>	Matrix	3D surfaces via contour lines
<code>filled.contour()</code>	Matrix	3D surfaces with color
<code>image()</code>	Matrix	Heatmap / colored grid
<code>persp()</code>	Matrix	3D perspective plot

Other Plots

Function	Input Type	Primary Purpose
Specialized & Mathematical Plots		
<code>curve()</code>	Expression	Plot mathematical functions
<code>matplot()</code>	Matrix	Columns vs columns plot
Generic Plot(s)		
<code>plot()</code>	Universal	Generic plotting

Tip

- What kind of data do you have? (Measurement Level)
- What are you trying to show? (Research Question → Formula)
 - ▶ Compare groups (`mpg ~ cyl`): Boxplot, Barplot
 - ▶ Show distribution (`~ mpg`): Histogram, Density plot
 - ▶ Show relationship (`mpg ~ wt`): Scatterplot
 - ▶ Show change over time (`value ~ time`): Line plot
- Match to the Right Plot

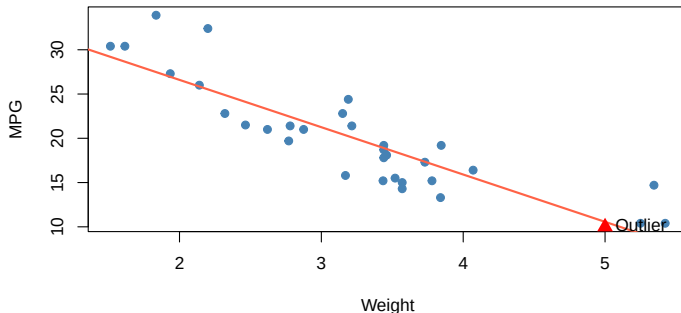
Measurement Level	Goal	Best Plot
Nominal	Compare categories	Barplot, Pie chart
Ordinal	Show ordered groups	Barplot, Dotplot, Stacked bar
Interval/Ratio	Show distribution	Histogram, Density plot, Boxplot
Interval/Ratio	Show relationship between two variables	Scatterplot, Line plot
Interval/Ratio	Show change over time	Line plot, Time series plot
Two nominal variables	Show association	Mosaic plot, Stacked bar
One nominal + one ratio	Compare distributions across groups	Boxplot, Faceted histogram, Stripchart
Multiple ratio variables	Explore relationships	Scatterplot matrix

Layers I

- Base R graphics uses a “pen-and-paper” model:
 - ▶ Start with a plot frame (`plot()`, `plot.ts()`, `hist()`, etc.)
 - ▶ Add additional elements on top using functions like:
 - ★ `points()` → add extra points
 - ★ `lines()` → add lines or fitted models
 - ★ `abline()` → add straight lines (regression, `hline`, `vline`)
 - ★ `text()` / `mtext()` → add labels or annotations
 - ★ `axis()` → add axes
 - ★ `legend()` → add legends
 - ★ `box()` → add box around a plot
- **Order matters:** later layers overlay earlier ones.
- Layers are powerful
 - ▶ Build plots incrementally for clarity
 - ▶ Combine raw data, model fits, and annotations in one graphic
 - ▶ Makes exploratory and diagnostic plots more informative

Layers II

Base R layering example



```
plot(mtcars$wt, mtcars$mpg, pch=19, col="steelblue",  
     main="Base R layering example", xlab="Weight", ylab="MPG")  
abline(lm(mpg ~ wt, data=mtcars), col="tomato", lwd=2)  
points(5, 10, col="red", pch=17, cex=1.5)  
text(5, 10, labels="Outlier", pos=4)
```

Base R Plotting Workflow

① Create the plot

- ▶ `plot()` initializes the plotting region and axes

② Add graphical layers

- ▶ `points()` – add data points
- ▶ `lines()` – add lines or curves
- ▶ `abline()` – reference lines
- ▶ `text()` – annotations

③ Add explanatory elements

- ▶ `legend()` – explain symbols or colors
- ▶ `title()` – add titles
- ▶ `axis()` – add axes

④ Optional: add further elements if needed

- ▶ `grid()` – grid lines
- ▶ `polygon()` – shaded areas
- ▶ `arrows()` – arrows for annotations
- ▶ `segments()` – line segments

Functions and Common Parameters

- **High-Level vs Low-Level Functions (API)**

- ▶ **High-level:** create plot frame/axes, e.g. `plot()`, `hist()`, `boxplot()`, `plot.ts()`, ...
- ▶ **Low-level:** add layers / annotations, e.g. `points()`, `lines()`, `abline()`, `text()`, `legend()`, ...

- **Shared Parameter** groups across functions

- 1 Layout, Grid & Margins (Structure), `par()`
- 2 Titles, Labels & Content (Text Input), `plot()`
- 3 Colors & Fills (Aesthetics), `plot()` (local) or `par()` (global)
- 4 Sizing & Typography (Fonts), `plot()` or `par()`
- 5 Points & Lines (Data Styling), `plot()` or `par()`
- 6 Axis Fine-Tuning & Geometry (Scaling), `plot()` or `par()`
- 7 Text Alignment & Rotation (Fine-Tuning), `par()`

- **Number of Parameters:** 85+

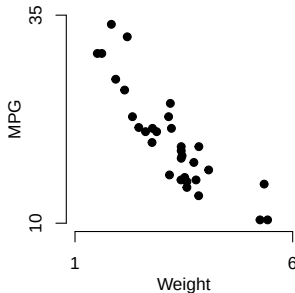
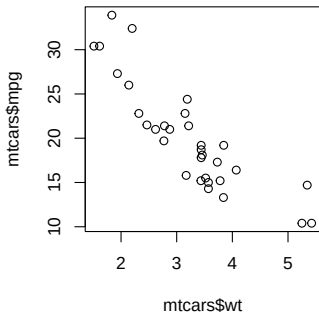
- ▶ 72 parameters in `par`, 13 only readable (device)
- ▶ specific parameters for a plot type

Shared Parameters Across Functions

	Parameter Name(s)	Short Description
1	<code>btty</code>	Box border type.
1	<code>mar</code>	Margin sizes.
1	<code>mfrow, mfcoll</code>	Layout for multiple plots <code>c(rows, cols)</code> .
2	<code>main, sub</code>	Main title (top) and subtitle (bottom).
2	<code>xlab, ylab</code>	X-axis and y-axis text labels.
3	<code>col, col.main, col.sub, col.lab, col.axis</code>	Colors for data, titles, subtitles, labels, and ticks.
3	<code>bg</code>	Background / Point fill color.
4	<code>cex, cex.main, cex.sub, cex.lab, cex.axis</code>	Text and symbol size scaling factors (default is 1).
4	<code>font</code>	Font style (1=plain, 2=bold, ...)
5	<code>pch</code>	Point symbol type (integers 0:25).
5	<code>lty, lwd</code>	Line type and thickness.
5	<code>type</code>	Plot style ("p"=points, "l"=lines, ...).
5	<code>add</code>	Overlays data onto existing plot if TRUE.
6	<code>xlim, ylim</code>	Min/max data limits for x and y axes.
6	<code>las</code>	Axis tick label orientation.
6	<code>axes, xaxt, yaxt</code>	Suppress all/specific axes.

Custom Axes with axis()

```
plot(mtcars$wt, mtcars$mpg)
plot(mtcars$wt, mtcars$mpg, axes=FALSE, xlab="", ylab="",
      xlim=c(1, 6), ylim=c(10,35), pch=19)
axis(1, at=c(1, 6))
axis(2, at=c(10, 35))
mtext("Weight", side=1, line=2)
mtext("MPG", side=2, line=2)
```



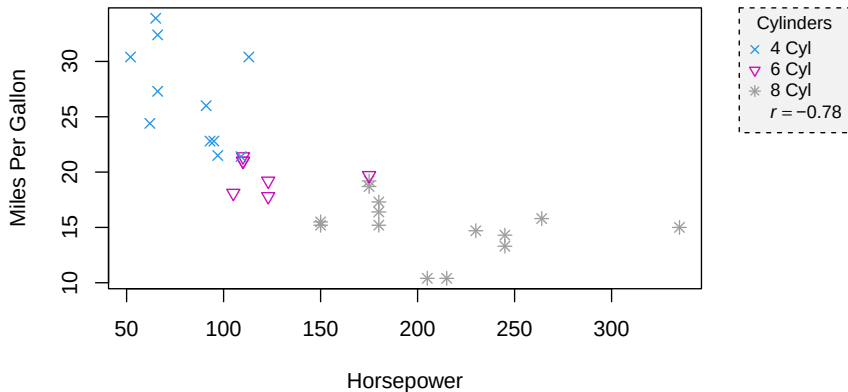
legend()

- legend() adds legend(s) to plots
- includes legends for points, lines, colors

```
par(mar=c(5, 4, 4, 8) + 0.1)
plot(mtcars$hp, mtcars$mpg, pch=mtcars$cyl,
     col=mtcars$cyl, main="Engine Performance",
     xlab="Horsepower", ylab="Miles Per Gallon")
r_val <- round(cor(mtcars$hp, mtcars$mpg), 2)
legend("topright", inset=c(-0.25, 0), xpd=TRUE,
     title="Cylinders", col=c(4, 6, 8, NA),
     pch=c(4, 6, 8, NA), text.font=c(1, 1, 1, 2),
     bg="gray95", box.lty=2, cex=0.8,
     legend=c("4 Cyl", "6 Cyl", "8 Cyl",
              as.expression(substitute(italic(r)==R, list(R=r_val))))))
```

Result of legend()

Engine Performance



Graphical Parameters with `par()`

Purpose

- `par()` controls **graphical parameters** in base R
- Parameters affect **subsequent plots**
- Used to control appearance of plots globally

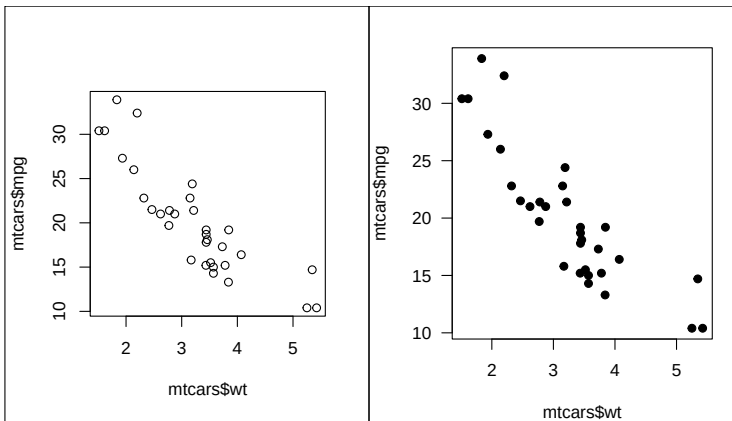
```
oldpar <- par(cex=1.2)
par(col="steelblue")
par(lwd=2)
par(oldpar)  # restore par
```

Margins and Axis Control

- `mar` plot margins (bottom, left, top, right)
- `las` orientation of axis labels
- `mgp` position of axis titles and labels
- `tck` tick mark length

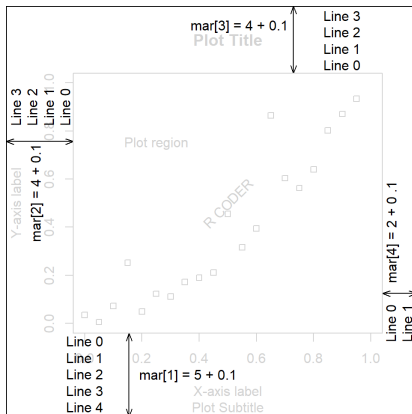
Margins and Axis Control

```
# default: c(5.1, 4.1, 4.1, 2.1)
plot(mtcars$wt, mtcars$mpg)
par(mar=c(4,4,2,1))
par(las=1)
plot(mtcars$wt, mtcars$mpg, pch=19)
```

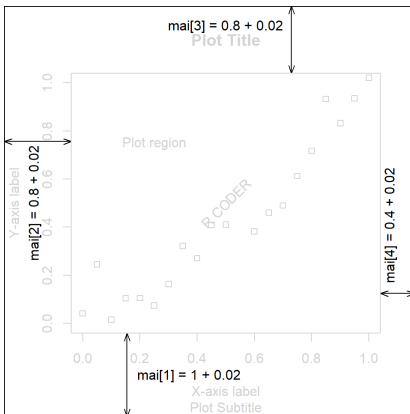


mar vs. mai

mar: margins in lines



mai: margins in inches



Source: [Margins in R](#)

Multipanel Plots

Multi-panel plots are essential for **Exploratory Data Analysis (EDA)** and storytelling:

- **Comparison:** View different categories side-by-side without overlapping data.
- **Context:** Place summary statistics (histograms) next to raw data (scatter plots).
- **Efficiency:** Show multiple dimensions of a dataset in a single graphic export.
- **Clarity:** Reduce “visual noise” by giving each variable its own space.
- **Best for:** Highly customized or non-linear arrangements.
- **Example:** Scatterplot matrix

Three Ways to Multipanel

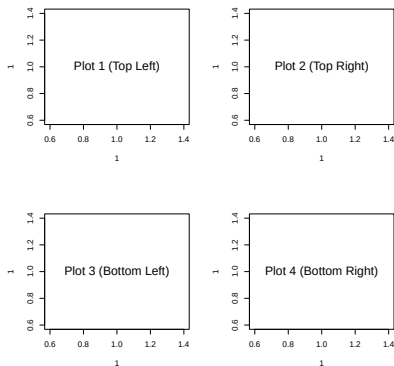
- `par(mfrow=c(rows, cols))`
 - ▶ Divides the device into equal-sized cells.
 - ▶ **Best for:** Ideal for simple, regular layouts with equal-sized panels.
- `layout(matrix)`
 - ▶ Uses a matrix to define plot positions.
 - ▶ **Best for:** Complex layouts where some plots are larger than others.
- `split.screen()`¹
 - ▶ Manually defines regions (screens) that can be further subdivided.

Feature	<code>par(mfrow)</code>	<code>layout()</code>	<code>split.screen()</code>
Ease of Use	High	Medium	Low
Symmetry	Equal cells only	Variable sizes	Fully manual
Best use case	Regular grids	Asymmetric layouts	Nested layouts

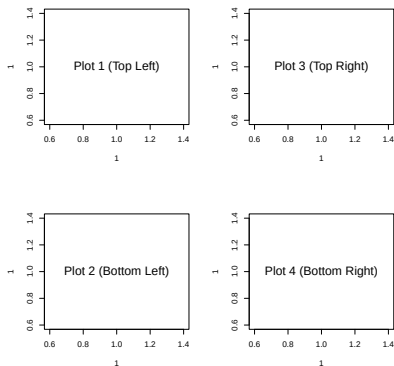
¹ only rarely used

par() with mfrow and mfc col

par(mfrow=c(2,2))



par(mfcol=c(2,2))



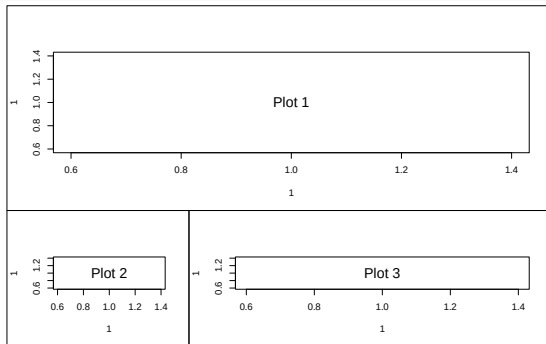
- `plot.new()` makes an empty plot
- `par(mfg=c(row, col))` jump to plot at position (row, col)

layout()

```
m <- matrix(c(1, 1, 2, 3), nrow=2, byrow=TRUE)
m
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    3
```

```
layout(m, heights=c(6,4), widths=c(1,2))
# plot commands ...
```



split.screen()

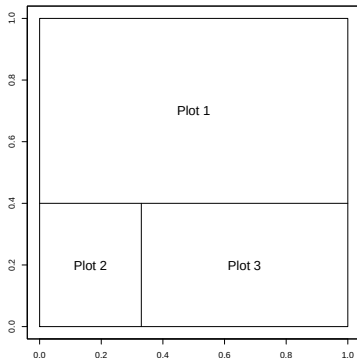
- works on a normalized coordinates $[0, 1] \times [0, 1]$
- a 4-column matrix (xmin, xmax, ymin, ymax) defines the plotting areas
- plotting areas may overlap (subplots !)
- `screen(i)` jumps to screen `i`

```
m <- rbind(c(0, 1, 0.4, 1),  
           c(0, 0.33, 0, 0.4),  
           c(0.33, 1, 0, 0.4))
```

```
m
```

```
##      [,1] [,2] [,3] [,4]  
## [1,] 0.00 1.00  0.4  1.0  
## [2,] 0.00 0.33  0.0  0.4  
## [3,] 0.33 1.00  0.0  0.4
```

```
res <- split.screen(figs=m)  
# plot commands  
close.screen(all=TRUE)
```

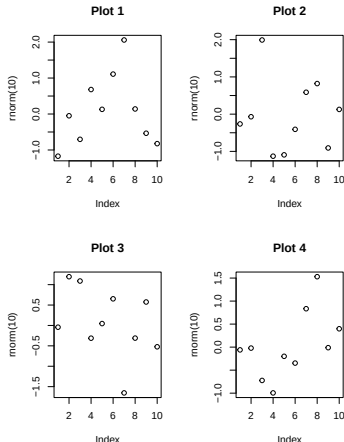


The Master Title

- oma and omi for outer margins
- default: c(0,0,0,0) - no outer margins
- mtext() for texts in the outer margins

```
par(oma=c(0, 0, 3, 0))
par(mfrow=c(2, 2))
main <- sprintf("Plot %i", 1:4)
for(i in 1:4)
  plot(rnorm(10), main=main[i])
  mtext("Random Normal Distributions",
        outer=TRUE, cex=1.5,
        line=1, font=2)
```

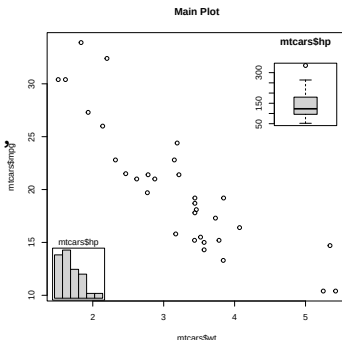
Random Normal Distributions



TeachingDemos::subplot()

- simplifies the creation of subplot(s) in a plot
- subplot() embed a new plot within an existing plot
- uses the user coordinates

```
library("TeachingDemos")
plot(mtcars$wt, mtcars$mpg,
     main="Main Plot")
subplot(
  boxplot(mtcars$hp, main="mtcars$hp"),
  x=5, y=29, size=c(1.2, 1.2))
subplot({
  hist(mtcars$hp, xlab="", ylab="",
       main= "", xaxt="n", yaxt="n")
  box()} ,
  x=1.8, y=12, size=c(1, 1))
text(1.8, 15, "mtcars$hp")
```



Iris Data and Exercises

The `iris` dataset contains measurements for 150 flowers from three different species.

Variable	Description (English)	Beschreibung (Deutsch)
<code>Sepal.Length</code>	Sepal length (cm)	Kelchblattlänge (cm)
<code>Sepal.Width</code>	Sepal width (cm)	Kelchblattbreite (cm)
<code>Petal.Length</code>	Petal length (cm)	Kronblattlänge (cm)
<code>Petal.Width</code>	Petal width (cm)	Kronblattbreite (cm)
<code>Species</code>	Species (Setosa, Versicolor, Virginica)	Spezies (Art)

- Create a scatter plot of Sepal Length vs Sepal Width. Use `plot()`, label both axes and add a title.
- Create a scatter plot of Petal Length vs Petal Width. Color the points by Species, use different plotting symbols (`pch`) and add a legend.

Exercises

- Create a boxplot of Sepal Length by Species, add colors and axis labels.
- Create a histogram of Petal Length with an appropriate number of bins. Add a color and a title.
- Create a pairs plot of all numeric variables. Color the points by Species.
- Compare the distribution with `hist` of Petal.Length across species using side-by-side plots. Make sure that the plots are comparable.
- Create a 2×2 matrix where the top row is all 1s, and the bottom row has 2 and 3.
 - ▶ Ratios: Use `layout(mat, heights=c(2, 1))` to make the top plot twice as tall as the bottom ones.
 - ▶ Plot 1 (Top): A scatter plot of Petal.Length vs Petal.Width.
 - ▶ Plot 2 (Bottom Left): A boxplot of Sepal.Length by Species.
 - ▶ Plot 3 (Bottom Right): A stripchart (jittered points) of Sepal.Width.

The Next Step: lattice

Tufte's Ideas

Edward Tufte (1983) The Visual Display of Quantitative Information :

- **Small Multiples**

- ▶ Concept: Consistent visual structure across multiple panels.
- ▶ Objective: Enables rapid comparison by reducing cognitive load.
- ▶ Goal: Shift focus from the axes to the changes in the data.

- **High Data-Ink Ratio**

- ▶ Concept: Maximize visual elements that convey data.
- ▶ Objective: Every mark on the page should represent a number.
- ▶ Goal: Minimize non-data ink (e.g., gridlines, borders, labels).

- **Visual Parallelism**

- ▶ Concept: Side-by-side comparison using identical scales and axes.
- ▶ Objective: Detecting differences between objects sitting side-by-side.
- ▶ Goal: Supports “at-a-glance” integrity and honest visual comparisons.

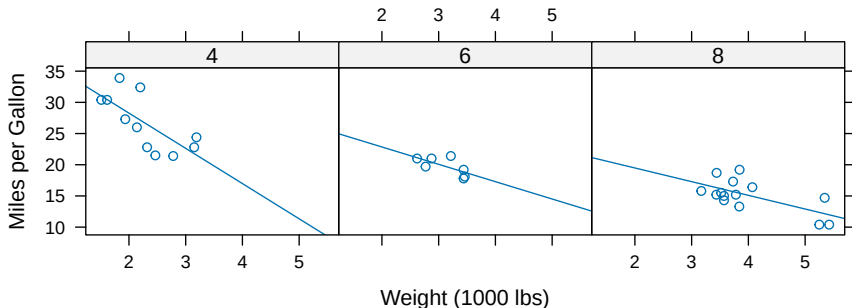
Implementation in `lattice`

- **Trellis Display:** a system of small multiples where data are split into subsets
- **The Conditioning Operator (`|`)**
 - ▶ **Method:** Uses the formula interface $y \sim x \mid g$.
 - ▶ **Realization:** Automatically creates a separate panel for each category. No manual loops or data splitting required.
- **Automated Aesthetic Mapping**
 - ▶ **Method:** Calculates global axis limits across the entire dataset *before* drawing.
 - ▶ **Realization:** Supports consistent comparison; can be overridden if needed.
- **Strips and Shared Axes**
 - ▶ **Method:** Moves labels to outer edges and uses a “Strip” to identify subsets.
 - ▶ **Realization:** Reduces redundant axis titles and improves data-ink ratio.

A Simple lattice example

```
xyplot(mpg ~ wt | factor(cyl), data=mtcars,  
       type=c("p", "r"), layout=c(3, 1),  
       main="Mtcars: MPG vs Weight by Cylinder",  
       xlab="Weight (1000 lbs)",  
       ylab="Miles per Gallon")
```

Mtcars: MPG vs Weight by Cylinder

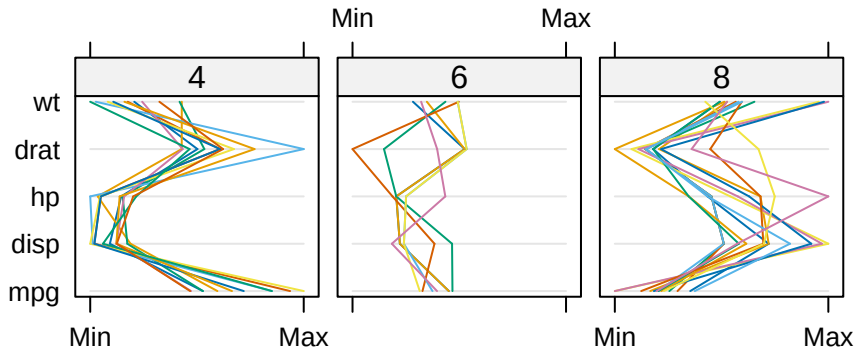


Plot Overview

Function	Input Type	Primary Purpose
<code>xyplot()</code>	<code>y ~ x g</code>	Scatter plot
<code>xyplot.ts()</code>	<code>y ~ x g</code>	Time series plot
<code>bwplot()</code>	<code>y ~ g</code> or <code>y ~ x g</code>	Boxplot
<code>histogram()</code>	<code>~ x g</code>	Histogram
<code>barchart()</code>	<code>~ x g</code>	Bar chart
<code>densityplot()</code>	<code>~ x g</code>	Smoothed histograms
<code>dotplot()</code>	<code>y ~ x g</code>	Dot plot
<code>stripplot()</code>	<code>y ~ x g</code>	Strip plot
<code>qq()</code>	<code>y ~ x g</code>	QQ plot
<code>cloud()</code>	<code>z ~ x * y g</code>	3D scatter plot
<code>wireframe()</code>	<code>z ~ x * y g</code>	3D surface plot
<code>levelplot()</code>	<code>z ~ x * y g</code>	Heatmaps
<code>contourplot()</code>	<code>z ~ x * y g</code>	Contour plots
<code>spiom()</code>	<code>z ~ x * y g</code>	Scatter plot matrix
<code>parallelplot()</code>	<code>z ~ x * y g</code>	Parallel coordinate plot

Parallel Coordinate Plot

```
parallelplot(~ mtcars[, c(1, 3:6)] | factor(cyl), data=mtcars)
```

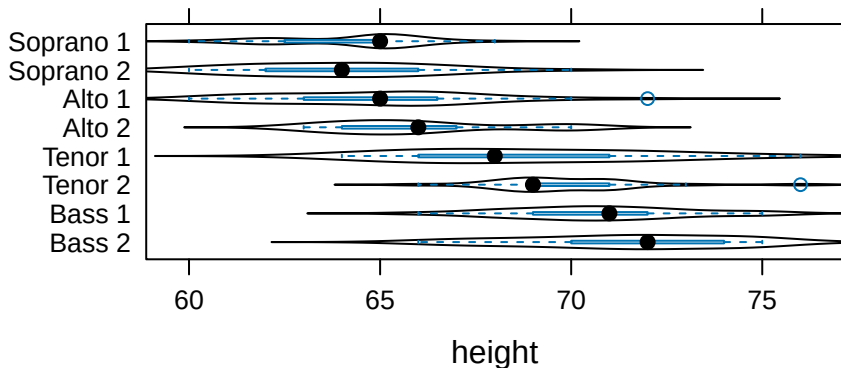


Panels (Low-Level-plotting)

- Modular structure: Each panel is generated by a panel function (e.g., `panel.xyplot`, `panel.bwplot`) that draws the data for a specific subset.
- Conditional display: Panels are created for each combination of conditioning variables (e.g., `x ~ y | g` produces one panel per level of group).
- Customization: The behavior of individual panels can be modified by defining a custom panel function or by adjusting the `par.settings` argument.
- Shared axes: Panels typically share axis scales unless explicitly overridden (e.g., via `scales=list(relation="free")`).
- Combining: Panels can be combined, e.g. `panel.loess()`, `panel.abline()`, `panel.smoothScatter()`, `panel.spline()`
- `latticeExtra`: Provides several new high-level functions and methods, additional utilities such as panel and axis annotation functions.

Stacked Panels

```
bwplot(voice.part ~ height, singer,
       panel=function(..., box.ratio) {
         panel.violin(..., col="transparent",
                     varwidth=FALSE, box.ratio=box.ratio)
         panel.bwplot(..., fill=NULL, box.ratio=.1)
       } )
```



The Foundation: grid

Transition – From lattice to ggplot2

- **The grid Foundation:**

- ▶ Both packages rely on the grid system for multi-panel and layered visuals, despite having different syntaxes.

- **What is grid?**

- ▶ A low-level framework (introduced in R 2.10.0) for building graphics independently of base R and designed for composability.

- **Core Architecture of grid**

- ▶ **Graphical objects (grobs):** Represent visual elements (e.g., text, lines, rectangles).
- ▶ **Viewport:** Define coordinate systems and spatial regions; support nesting and scaling.
- ▶ **Rendering pipeline:** Converts grobs into pixels or vector output (PDF, SVG, etc.).
- ▶ **Device independence:** Output is consistent across formats and devices.

Graphical Objects (Grobs)

- A **grob** is a structured, self-contained representation of a graphical primitive or composite element.
- **Key characteristics:**
 - ▶ **Atomic grobs:** Represent basic shapes (e.g., `textGrob`, `rectGrob`, `circleGrob`).
 - ▶ **Composite grobs:** Combine multiple grobs (e.g., `grobTree`, `gList`).
 - ▶ **Attributes:** Each grob carries properties such as position, size, color, and font.
 - ▶ **Inheritance:** Grobs can inherit or override attributes from parent objects.
 - ▶ **Example:** `textGrob("Label", x=0.5, y=0.5, gp=gpar(col="blue"))` creates a text grob with specified position and color.
- Grobs are not rendered directly; they are passed through the rendering pipeline.

Viewports

- A **viewport** defines a region of the graphics device with its own coordinate system. It enables nested, hierarchical layout and coordinate transformation.
- **Structure:**
 - ▶ **Viewport origin:** Defines the (x, y) reference point.
 - ▶ **Width and height:** Specify the size of the region.
 - ▶ **Coordinate system:** Can be user-defined (e.g., data coordinates) or device-based.
 - ▶ **Parent-child relationships:** Viewports can be nested, enabling complex layouts.

Example:

```
vp <- viewport(x=0.5, y=0.5, width=0.8, height=0.8)
pushViewport(vp)
```

creates a viewport centered in the device, scaled to 80% of its (parents) size.

The Rendering Pipeline and Device Independence in grid

- **Scene construction:**

- ▶ Grobs (e.g., text, lines, rectangles) are assembled into a hierarchical structure.
- ▶ Viewports define spatial regions and coordinate systems.

- **Coordinate transformation and clipping:**

- ▶ User coordinates are mapped to device coordinates.
- ▶ Clipping is applied based on viewport boundaries.

- **Rendering:**

- ▶ For vector formats (PDF, SVG): grobs are converted into scalable path commands.
- ▶ For raster formats (PNG, JPEG): the scene is rendered pixel-by-pixel.

- **Device independence:**

- ▶ Backend-specific drivers (`pdf()`, `png()`, etc.).
- ▶ Consistent coordinate mapping and device-level handling of fonts, colors, and resolution.

How lattice Uses grid

- `lattice` generates multi-panel displays using `grid` for layout and rendering.
- Each panel is rendered as a `grid grob`, and the overall figure is constructed via a viewport hierarchy.
- For example:
 - ▶ `xyplot(y ~ x | group, data)` creates a trellis display.
 - ▶ The conditioning variable (`group`) determines the number of panels.
 - ▶ `grid` manages spacing, alignment, and axis placement across panels.
 - ▶ The `print()` method for `lattice` objects invokes `grid` rendering, which handles the final output.

How ggplot2 Uses grid

- ggplot2 implements a layered grammar of graphics.
- Each layer (e.g., `geom_point()`, `scale_x_continuous()`) contributes a grid grob.
- The `ggplot()` object is a list of components:
 - ▶ Data
 - ▶ Aesthetics
 - ▶ Geoms
 - ▶ Scales
 - ▶ Facets
 - ▶ Themes
- When `print()` or `ggsave()` is called, ggplot2 converts this structure into a grid-based scene.
- The final rendering is performed by `grid`, which composes all grobs into a single output.

ggplot2

Introduction to ggplot2

- ggplot2 is the dominant visualization system in R.
- Built on the **Grammar of Graphics** by L. Wilkinson.
- Part of the **tidyverse** ecosystem (<https://tidyverse.org/>).
- Highlight advantages:
 - ▶ Layered approach makes plots easier to build and modify.
 - ▶ Highly customizable for publication-quality graphics.
 - ▶ Works with pipelines (`%>%`) and tidy data frames.
- Packages:
 - ▶ ggplot2 (for data visualization)
 - ▶ dplyr (for pipe-operator `%>%` and data manipulations)

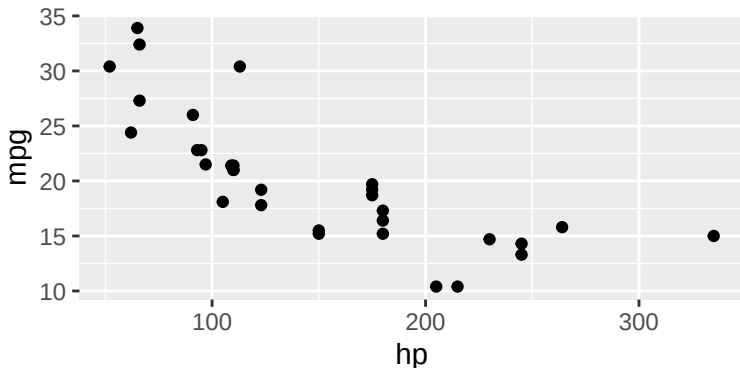
Grammar of Graphics Structure

- Plot structure: **data + aesthetics + geoms + stat + scales + coord + facet + theme**
- Components:
 - ▶ **data**: the dataset used.
 - ▶ **aesthetics**: mapping variables to visual properties.
 - ▶ **geoms**: geometric objects like points, bars, lines.
 - ▶ **stat**: add computed summaries like means or model fits.
 - ▶ **scales**: control how data is translated into color, size, axes.
 - ▶ **coord**: control axes, scaling, and plot geometry (e.g., flip).
 - ▶ **themes**: control non-data styling.
 - ▶ **facets**: split data into subplots.
- Each component is independent.
- Plots are built by adding layers with +

Basic ggplot() Syntax

- Basic ggplot syntax: `ggplot(data = ..., aes(...)) + geom_X()`

```
library(ggplot2)           # activates the plotting library
ggplot(data = mtcars,      # initializes the plot
  aes(x = hp, y = mpg)) +  # defines mappings
geom_point()               # adds a visual layer (scatterplot)
```



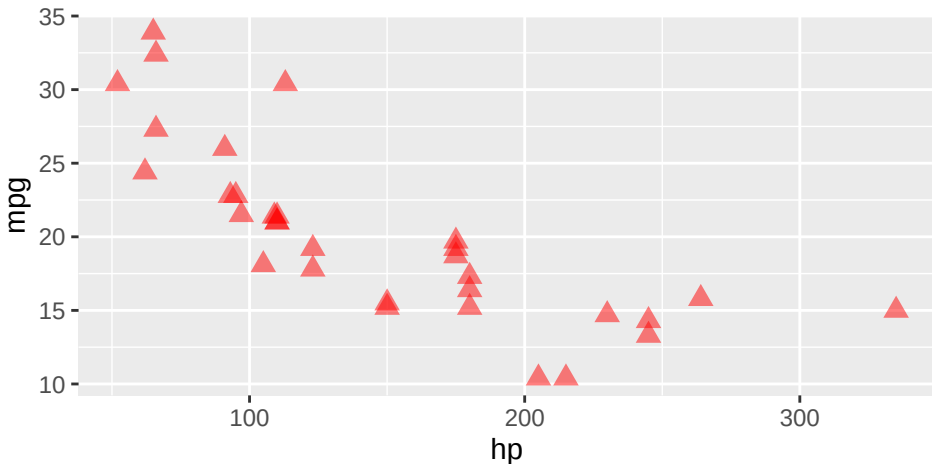
Core Geoms in ggplot2

- `geom_point()`: scatterplots for relationships between two numeric variables
- `geom_line()`: trends over continuous x-values
- `geom_bar()`: counts of categorical data
- `geom_col()`: bars for pre-summarised values
- `geom_histogram()`: distribution of a numeric variable
- `geom_boxplot()`: median, quartiles, and outliers
- `geom_violin()`: smoothed distribution shape
- `geom_smooth()`: trend lines with optional confidence bands
- `geom_density()`: smooth density estimate
- `geom_jitter()`: scattered points with noise to avoid overlap

Scatterplots: `geom_point()`

- Show relationships between two continuous variables.
- Fundamental for correlation and regression analysis.
- Useful for detecting patterns, outliers, clusters.
- Main parameters:
 - ▶ `shape` = point symbol: triangle, circle, etc. (numbers 0–25)
 - ▶ `size` = point size, eg. `size = 2`
 - ▶ `color` = point color, eg. `color = "red"`, `color = "#D55E00"`
 - ▶ `alpha` = transparency, eg. `alpha = 0.7`
 - ▶ `jitter` = adds noise to reduce overlap
 - ▶ `group` = defines separate groups for aesthetics or faceting

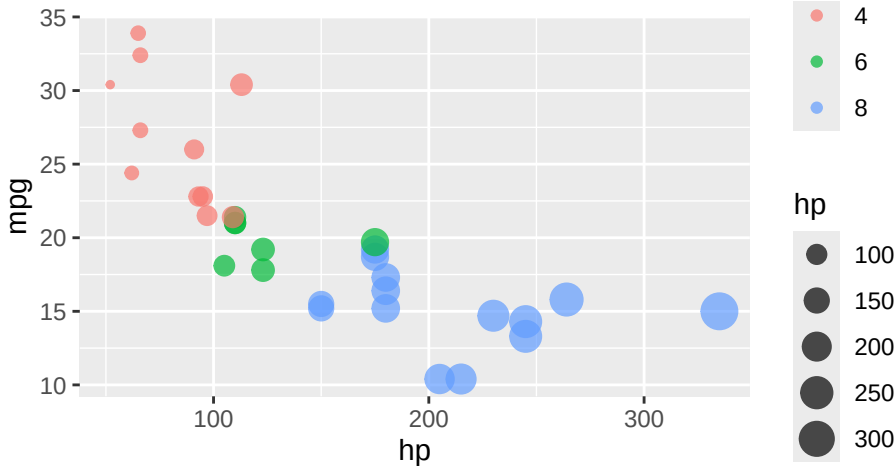
```
ggplot(mtcars, aes(hp, mpg)) + # compact form  
  geom_point(shape = 17,      # fixed triangle shape  
            size = 3,         # fixed point size  
            alpha = 0.5,      # fixed transparency  
            color = "red")    # fixed color
```



Aesthetics: color, shape, size, fill

- Aesthetics (`aes`) control how data is visually encoded.
- They map variables to visual properties such as color, fill, shape, size, etc.
- Aesthetics help reveal structure in the data: groups, patterns, and relationships.
 - ▶ `color` - distinguishes groups by color
 - ▶ `shape` - distinguishes groups by point symbol
 - ▶ `size` - emphasizes magnitude or importance
- Aesthetics inside `aes()` depend on data; outside `aes()` they are fixed.

```
ggplot(mtcars, aes(hp, mpg,  
  color = factor(cyl), # color mapped to cylinders  
  size = hp)) +        # size reflects horsepower  
geom_point(alpha = 0.7) # fixed transparency
```

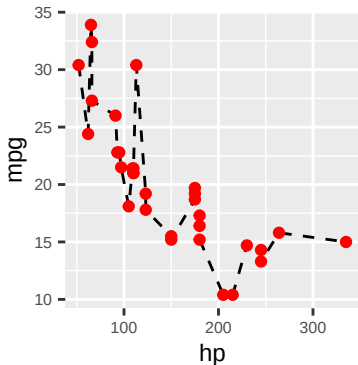


Line Charts: `geom_line()`

- Show trends across a continuous x-variable (time or ordered sequences).
- Support multiple series using color, linetype, or group.
- Can include trend smoothing (e.g., `geom_smooth()`).
- Main parameters:
 - ▶ `color` - line color
 - ▶ `linetype` - “solid”, “dashed”, “dotted” etc. (numbers 0-6)
 - ▶ `linewidth` - line thickness
 - ▶ `alpha` - transparency
 - ▶ `group` - defines separate groups

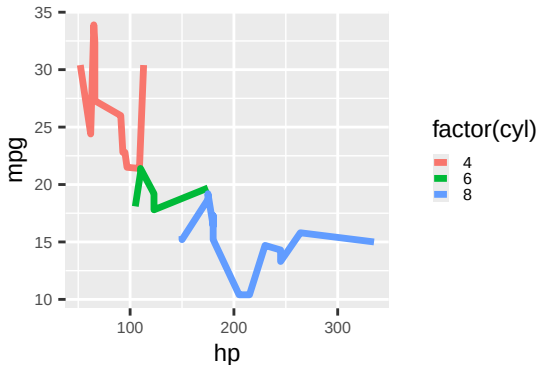
```
# Simple line
```

```
ggplot(mtcars,  
  aes(x = hp, y = mpg)) +  
  # 1st geom, dashed line  
  geom_line(linetype = 2) +  
  # 2nd geom, red color  
  geom_point(color = "red")
```



```
# Grouped lines
```

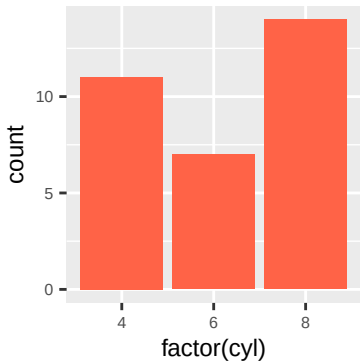
```
ggplot(mtcars,  
  aes(x = hp,  
    y = mpg,  
    # subgroup by cyl  
    color = factor(cyl))) +  
  geom_line(linewidth = 1.2)
```



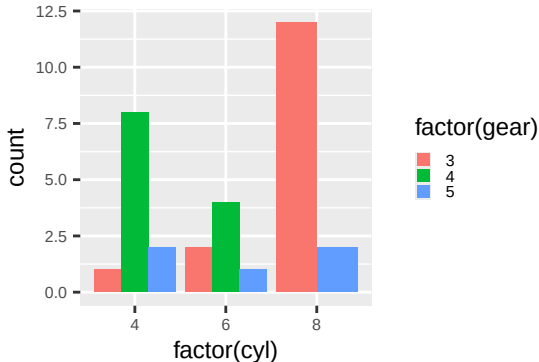
Bar Charts: `geom_bar()`

- Bar charts show counts of observations for each category.
- Require categorical x-values (factor-variable).
- With raw (unsummarised) data, `geom_bar()` counts rows automatically.
- Grouped bar charts use `fill` to show subgroups.
- Main parameters:
 - ▶ `fill` = bar color (inside)
 - ▶ `color` = outline color
 - ▶ `alpha` = transparency
 - ▶ `position` = "stack" (default), "dodge", "fill"
 - ▶ `stat` = "count" (default) counts rows; `stat` = "identity" uses pre-summarised data.

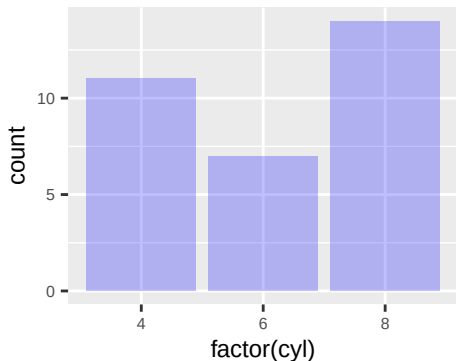
```
# Simple bar chart
ggplot(mtcars,
  # factor-variable
  aes(x = factor(cyl))) +
  geom_bar(fill = "tomato")
```



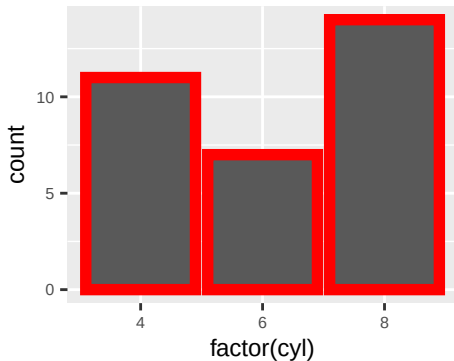
```
# Grouped bar chart by number of gears
ggplot(mtcars,
  aes(x = factor(cyl),
    fill = factor(gear))) + # subgroup by gear
  geom_bar(position = "dodge") # side-by-side
```



```
ggplot(mtcars,
      aes(x = factor(cyl))) +
  geom_bar(
    fill = "blue", # bar color
    alpha = 0.25) # transparency
```



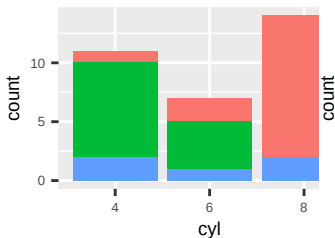
```
ggplot(mtcars) +
  aes(x = factor(cyl)) +
  geom_bar(
    color = "red", # outline color
    linewidth = 2) # outline width
```



• Position Options:

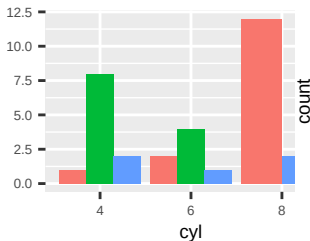
- ▶ `position = "stack"` (default) → stacked bars
- ▶ `position = "dodge"` → side-by-side bars
- ▶ `position = "fill"` → proportional bars (100%) (marginal distributions)

position stack



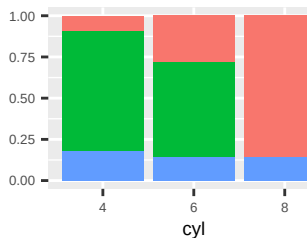
gear 3 4 5

position dodge



gear 3 4 5

position fill

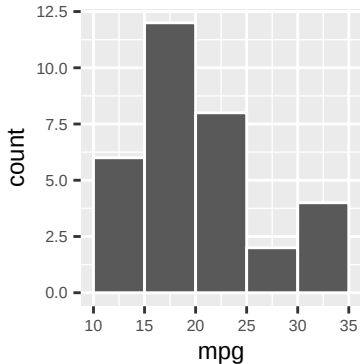


gear 3 4 5

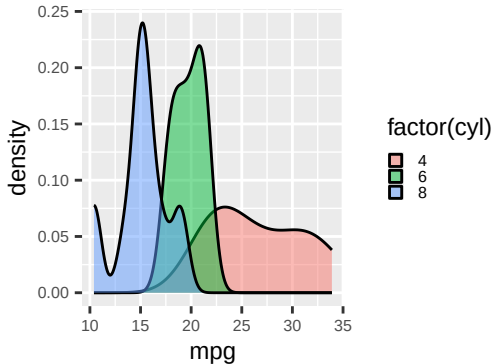
Histograms: `geom_histogram()` & Density Plots: `geom_density()`

- Both plots show distributions of a numeric variable.
- Histograms (**`geom_histogram`**) display raw counts in bins.
- Density plots (**`geom_density`**) show a smoothed estimate of the distribution and are especially useful for comparing groups.
- Key histogram parameters:
 - ▶ `bins` = number of bins (default = 30)
 - ▶ `binwidth` = width of each bin
 - ▶ `boundary` = where bins start
 - ▶ `center` = center of one bin
- Density plots depend on bandwidth (`bw`); too small → noisy, too large → oversmoothed.

```
# Histogram
ggplot(mtcars, aes(mpg)) +
  geom_histogram(
    color = "white", # outline col.
    binwidth = 5, # bin width
    boundary = 10) # bin start point
```



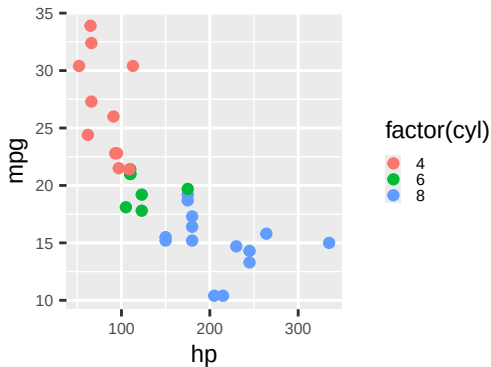
```
# Density plot
ggplot(mtcars,
  aes(mpg, fill = factor(cyl))) +
  geom_density(
    color = "black", # outline col.
    alpha = 0.5) # transparency
```



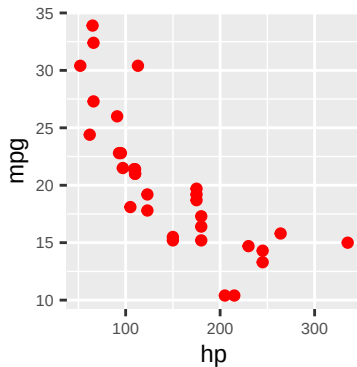
Mapping vs Setting

- Mapping (inside `aes()`) means the visual property depends on data.
- Example: `aes(color = cyl)` → color varies by cylinder.
- Setting (outside `aes()`) means the property is fixed.
- Example: `color = "red"` → all points are red.
- Advanced notes:
 - ▶ Mapping creates a legend; setting does not.
 - ▶ Mapping requires a variable; setting requires a value.
 - ▶ You can map and set different aesthetics in the same layer.

```
# Mapping
ggplot(mtcars,
       aes(hp, mpg,
           color = factor(cyl))) +
  geom_point()
```



```
# Setting
ggplot(mtcars,
       aes(hp, mpg)) +
  geom_point(
    color = "red")
```

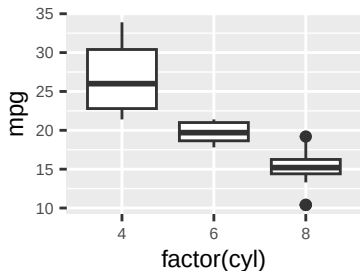


Boxplots: `geom_boxplot()` & Violin Plots: `geom_violin()`

- Both require a categorical x and numeric y.
- Boxplots show the distribution summary: median, quartiles, and potential outliers.
- Violin plots show the full distribution shape using kernel density.
- Both are useful for comparing groups.
- Boxplots emphasize summary statistics.
- Violin plots emphasize distribution shape.
- They can be combined for richer interpretation.

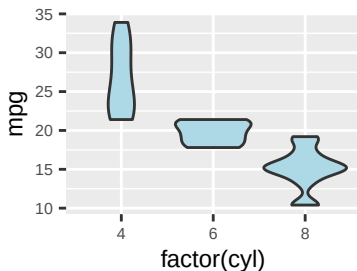
- Key boxplot parameters:
 - ▶ width = width of boxes
 - ▶ outlier.colour = color of outliers
 - ▶ outlier.size = size of outliers
 - ▶ coef = whisker length (default $1.5 \times \text{IQR}$)

```
# Boxplot
ggplot(mtcars,
       aes(factor(cyl), mpg)) +
  geom_boxplot()
```

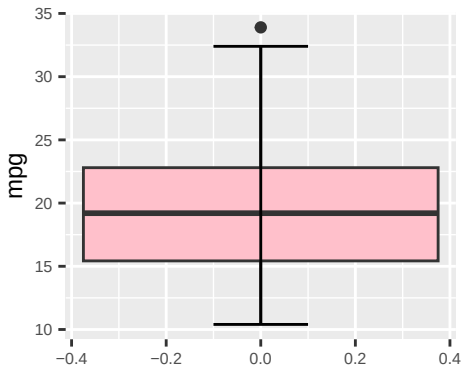


- Key violin plot parameters:
 - ▶ fill = interior color
 - ▶ color = outline color
 - ▶ alpha = transparency
 - ▶ trim = trim tails (default TRUE)
 - ▶ bw = bandwidth for density smoothing

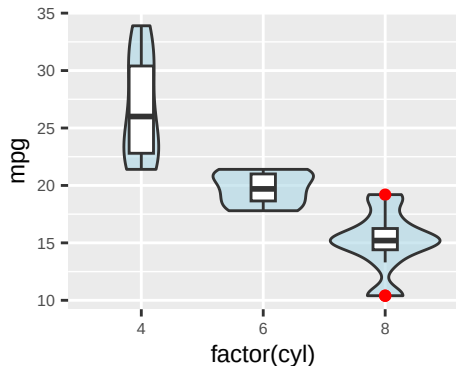
```
# Violin plot
ggplot(mtcars,
       aes(factor(cyl), mpg)) +
  geom_violin(fill = "lightblue")
```



```
# Boxplot for only one variable
ggplot(mtcars, aes(y = mpg)) +
  geom_boxplot(fill = "pink") +
  # add whiskers as error bars
  stat_boxplot(geom = "errorbar",
              width = 0.2)
```



```
# Combined plot
ggplot(mtcars, aes(factor(cyl), mpg)) +
  geom_violin(fill = "lightblue",
              alpha = 0.6) +
  geom_boxplot(width = 0.2,
              outlier.colour = "red")
```



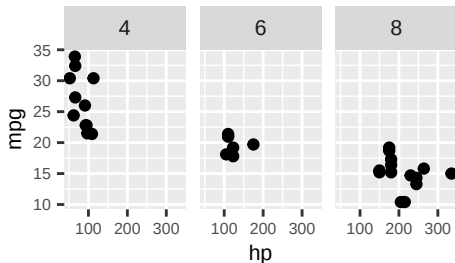
Faceting

- Splits data into multiple panels for easier comparison.
- Helps reveal patterns across groups.
- Two main approaches: `facet_wrap` and `facet_grid`.
- `facet_wrap` - small multiples.
 - ▶ Uses one (or several) factors arranged in a flexible 1D layout.
 - ▶ Panels wrap automatically into rows/columns.
 - ▶ Best for many levels of a single factor.
- `facet_grid` - two-factor comparison.
 - ▶ Uses two factors (rows $\times$ columns).
 - ▶ Produces a fixed grid (matrix) layout.
 - ▶ Best for structured comparisons across two categorical variables.
- Faceting formulas always rely on the tilde `~` to define the faceting structure.

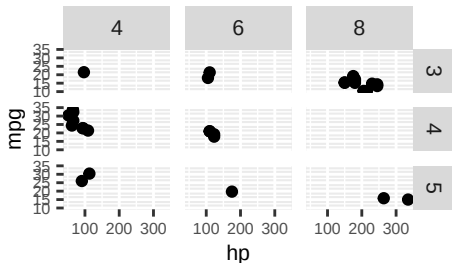
Faceting formulas

Use case	Formula	Meaning
facet_wrap	<code>~ cyl</code>	wrap panels by one factor
facet_grid	<code>gear ~ cyl</code>	rows = gear, columns = cyl
columns only	<code>. ~ cyl</code>	facet by columns only
rows only	<code>gear ~ .</code>	facet by rows only

```
ggplot(mtcars, aes(hp, mpg)) +  
  geom_point() +  
  facet_wrap(~ cyl)
```



```
ggplot(mtcars, aes(hp, mpg)) +  
  geom_point() +  
  facet_grid(gear ~ cyl)
```



Scales

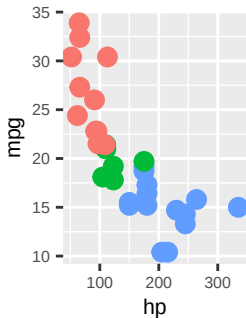
- Scales = the “translator” between data and visual encoding.
- Scales control how data values are translated into visual properties: axes, colors, sizes, shapes, etc.
- Why scales matter
 - ▶ adjust breaks, limits, labels
 - ▶ apply transformations (log, reverse)
 - ▶ choose color palettes
 - ▶ improve readability and interpretation
 - ▶ format labels (% , \$, scientific notation)
- Types of scales
 - ▶ Position scales — x/y axes.
 - ▶ Color & fill scales.
 - ▶ Size, shape, alpha scales.
 - ▶ Transformations.

Position Scales (Axes)

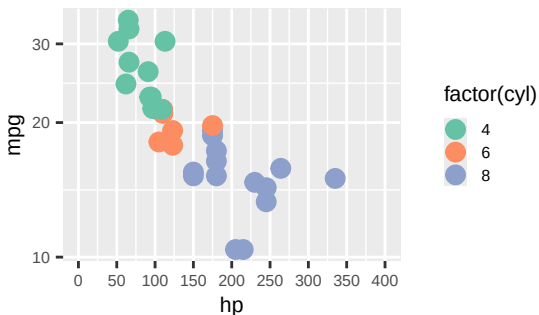
- Continuous scales
 - ▶ `scale_x_continuous` / `scale_y_continuous`
 - ▶ transformation: `scale_x_log10`, `scale_x_reverse`
- Discrete scales
 - ▶ `scale_x_discrete` / `scale_y_discrete`
- When to use
 - ▶ continuous numeric variables → continuous scales
 - ▶ factors/categories → discrete scales
 - ▶ skewed data → log transform
 - ▶ need to zoom without dropping data → use limits carefully

```
ggplot(mtcars, aes(hp, mpg, color = factor(cyl))) +
  geom_point(size = 3) +
  scale_y_log10() + # log transform
  scale_x_continuous(breaks = seq(0, 400, 50), # axis x breaks
                    limits = c(0,400)) + # axis x limits
  scale_color_brewer(palette = "Set2") # color scale
```

original



with scale transformations

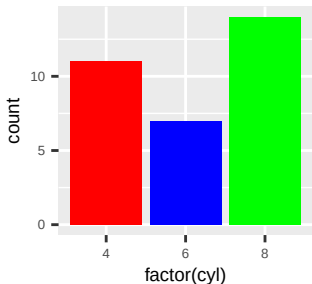


Color & Fill Scales

- Manual scales `[colors()]`
 - ▶ `scale_color_manual / scale_fill_manual`
- Brewer palettes (categorical)
`[RColorBrewer::display.brewer.all()]`
 - ▶ `scale_color_brewer / scale_fill_brewer`
- Viridis (colorblind-friendly)
 - ▶ `scale_color_viridis_d / scale_fill_viridis_c`
- Grey scale
 - ▶ `scale_fill_grey / scale_color_grey`
- When to use
 - ▶ categorical groups → Brewer, Viridis discrete
 - ▶ continuous gradients → Viridis continuous
 - ▶ publications → grey or Viridis
 - ▶ colorblind-safe → Viridis

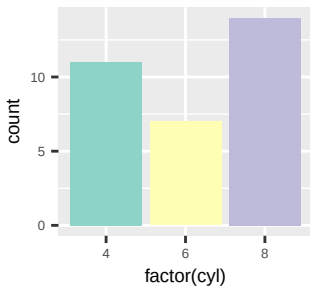
```
plot <- ggplot(mtcars, aes(factor(cyl), fill = factor(cyl))) +  
  geom_bar() # basic plot
```

```
plot +  
  scale_fill_manual(  
    values = c("red",  
              "blue", "green"))
```



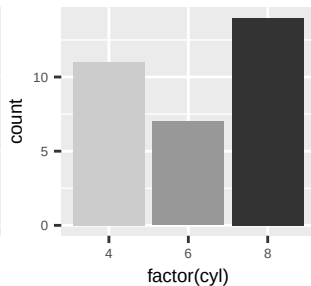
factor(cyl) ■ 4 ■ 6 ■ 8

```
plot +  
  scale_fill_brewer(  
    palette = "Set3"  
  )
```



factor(cyl) ■ 4 ■ 6 ■ 8

```
plot +  
  scale_fill_grey(  
    start = 0.8, end = 0.2  
  )
```

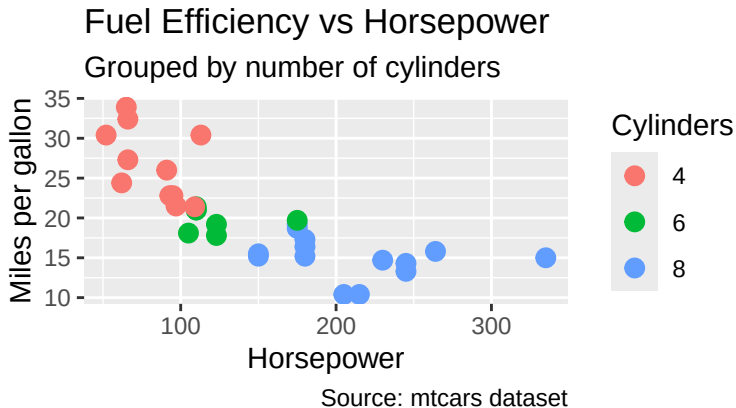


factor(cyl) ■ 4 ■ 6 ■ 8

Labels: titles, subtitles, captions

- Labels help explain what the plot shows, what variables mean, and how to interpret the graphic.
- Use `labs()` to set all textual elements in one place.
- Main label components:
 - ▶ `title` = main headline
 - ▶ `subtitle` = additional context
 - ▶ `caption` = data source, notes
 - ▶ `x` / `y` = axis labels
 - ▶ `color` / `fill` = legend titles

```
ggplot(mtcars, aes(hp, mpg, color = factor(cyl))) +  
  geom_point(size = 3) +  
  labs(title = "Fuel Efficiency vs Horsepower",  
        subtitle = "Grouped by number of cylinders",  
        x = "Horsepower",  
        y = "Miles per gallon",  
        color = "Cylinders",  
        caption = "Source: mtcars dataset")
```



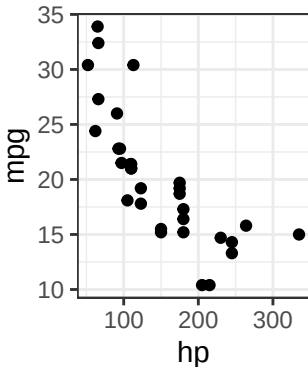
Themes: Quick Styles

- Themes control the overall appearance of a plot: background, gridlines, fonts, borders, legend placement.
- Quick themes provide instant styling without manual customization.
- Most common quick themes
 - ▶ `theme_gray` = default ggplot2 theme
 - ▶ `theme_bw` = white background, black gridlines
 - ▶ `theme_minimal` = very clean, minimal grid
 - ▶ `theme_classic` = no gridlines, classic axes
 - ▶ `theme_light` = light gridlines, soft background
 - ▶ `theme_dark` = dark background, high contrast
 - ▶ `theme_void` = no axes, no background (for maps, logos)
 - ▶ `theme_linedraw` = black-and-white technical style

```
plot1 <- ggplot(mtcars, aes(hp, mpg)) +  
  geom_point() # basic plot
```

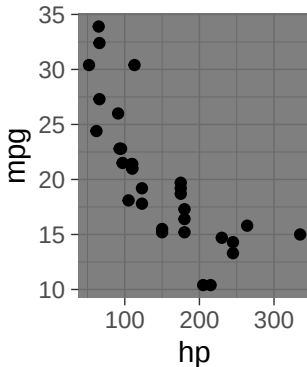
```
# base theme
```

```
plot1 +  
  theme_bw()
```



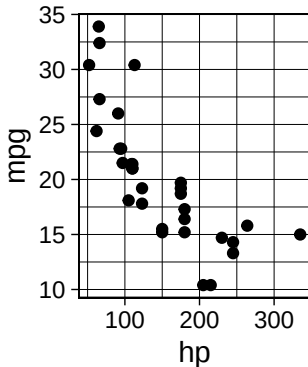
```
# dark theme
```

```
plot1 +  
  theme_dark()
```



```
# linedraw theme
```

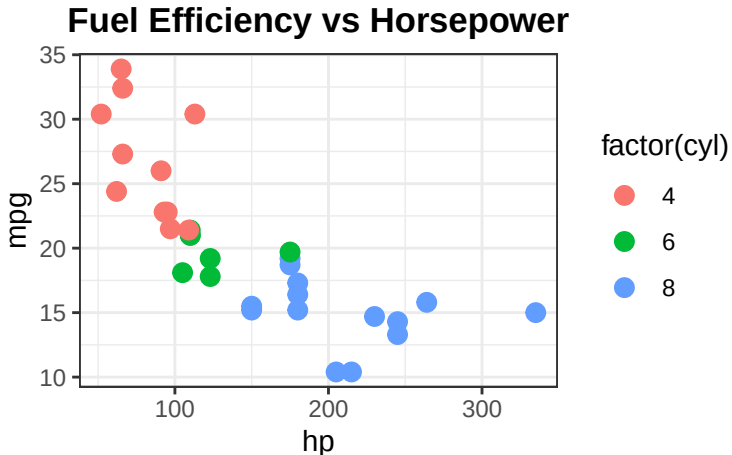
```
plot1 +  
  theme_linedraw()
```



Fine-Tuning Themes

- Quick themes set the overall style, but `theme()` allows precise control of plot appearance: legend placement, text formatting, alignment, backgrounds, gridlines, margins.
- Common elements to customize:
 - ▶ `legend.position` = move legend to “bottom”, “top”, “left”, “right”, or “none”
 - ▶ `plot.title` = font face, size, alignment
 - ▶ `axis.text` = axis tick labels
 - ▶ `axis.title` = axis titles
 - ▶ `panel.grid` = major/minor gridlines
 - ▶ `plot.margin` = outer spacing

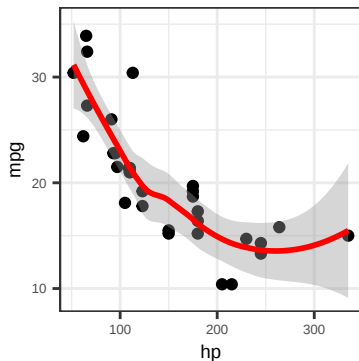
```
ggplot(mtcars, aes(hp, mpg, color = factor(cyl))) +
  geom_point(size = 3) +
  labs(title = "Fuel Efficiency vs Horsepower") +
  theme_bw() + # base theme
  theme(plot.title = element_text(
    face = "bold", # bold title
    hjust = 0.5)) # center alignment
```



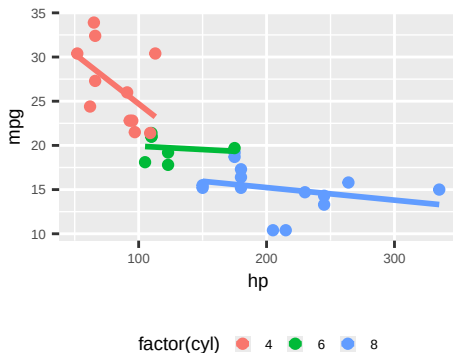
Trend Lines: `geom_smooth()`

- Adds a fitted trend line to a scatterplot.
- Useful for visualizing regression, direction, and strength of relationships.
- Supports linear models, LOESS, and more.
- Default settings: `method = "loess"`, `se = TRUE`

```
# default settings
#`method = "loess"`, `se = TRUE`
ggplot(mtcars,
       aes(hp, mpg)) +
  geom_point() +
  geom_smooth(color = "red") +
  theme_bw()
```



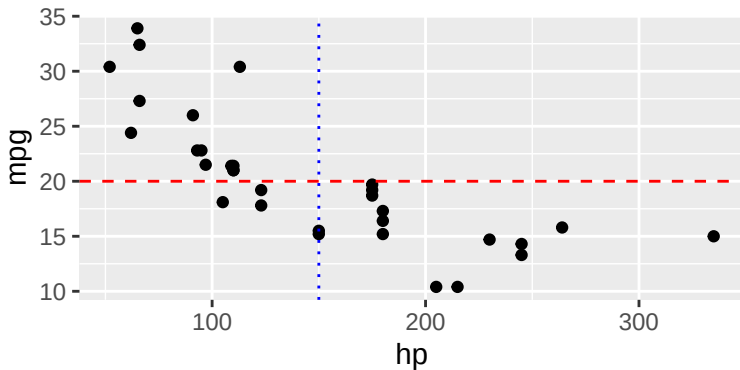
```
# linear regression
ggplot(mtcars, aes(hp, mpg,
                   color = factor(cyl))) +
  geom_point() +
  geom_smooth(method = "lm",
             se = FALSE) +
  theme(legend.position = "bottom")
```



Horizontal & Vertical Reference Lines

- Reference lines help highlight important values on the plot: means, medians, thresholds, acceptable ranges, target levels, cut-off points or domain boundaries.
- Use `geom_hline()` for horizontal lines and `geom_vline()` for vertical lines.
- Required parameters:
 - ▶ `yintercept` / `xintercept`

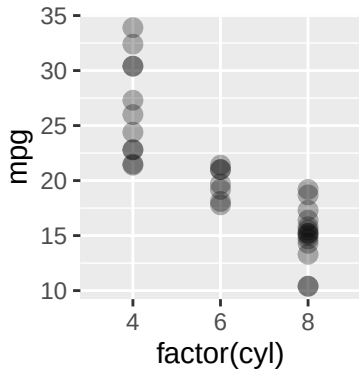
```
ggplot(mtcars, aes(hp, mpg)) +  
  geom_point() +  
  # Horizontal line  
  geom_hline(yintercept = 20,  
             color = "red", linetype = "dashed") +  
  # Vertical line  
  geom_vline(xintercept = 150,  
             color = "blue", linetype = "dotted")
```



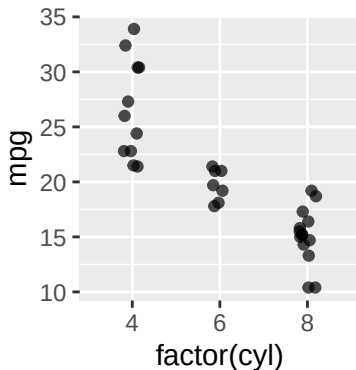
Positioning: `geom_jitter()`

- When many points overlap, it becomes hard to see how many observations each category has.
- `geom_jitter()` adds a small random displacement to points to reduce overplotting.
- **Why jitter is useful**
 - ▶ reveals hidden points
 - ▶ shows distribution within categories
 - ▶ avoids misleading “one point = one value” impression
 - ▶ works well with categorical x and numeric y
- **Key parameters**
 - ▶ `width` = horizontal jitter amount
 - ▶ `height` = vertical jitter amount
- **When to use jitter**
 - ▶ categorical x-axis (e.g., `cyl`, `gear`)
 - ▶ many identical or similar values
 - ▶ when boxplot/violin hides individual observations
 - ▶ when you want to show raw data + distribution

```
# Transparency
ggplot(mtcars,
aes(factor(cyl), mpg)) +
  geom_point(
    size = 3,
    alpha = 0.3 # transparency
  )
```



```
# Jittering
ggplot(mtcars,
aes(factor(cyl), mpg)) +
  geom_jitter(
    width = 0.1, # horizontal spread
    height = 0, # no vertical jitter
    alpha = 0.7) # transparency
```



Interactive graphics

Techniques

- Navigation
 - ▶ panning / zoomin: move visible area (+) / change scale (+)
 - ▶ rotating (3D) / touring (>3D): (+) / explore projections (-)
- Selection
 - ▶ selecting / highlighting / brushing: select data points or regions (+)
 - ▶ fisheye view: distortion-based focus (-)
- Linking & Coordination
 - ▶ linking various plots (via Dash)
- Data query & Details
 - ▶ querying data: hover / tooltips (+)
 - ▶ filtering / search: subset data (via Dash) / find data (-)
- Data transformation
 - ▶ sorting / reordering of variables (+)
 - ▶ aggregation / drill-down: change level of detail
- Presentation Enhancements
 - ▶ animation / faceting (small multiples): (+)
 - ▶ annotate: by user (+)
 - ▶ semantic zoom: (-)

Grand Tour & Projection Pursuit

- Grand Tour: inspect a lot of 1D/2D projections
- Projection Pursuit:
 - ▶ find interesting “projections” (non-normal)
 - ▶ for each projection assign an index value for interestingness
 - ▶ maximize the index

```
#install.packages("tourr")
library("tourr")
# Sepal.Length, Sepal.Width, Petal.Length, Petal.Width
x <- mtcars[, 1:7],
# copy to console!
animate(x, tour_path=grand_tour(), rescale=TRUE,
        display=display_xy())
animate(x, tour_path=grand_tour(), rescale=TRUE,
        display=display_xy(col=as.integer(iris$Species)))
```

Plotly

Why Plotly

- Plotly is an open-source library for interactive data visualization
 - ▶ enables interactive, web-based visualizations
- Supports multiple programming languages:
 - ▶ R, Python, JavaScript
- Built on: plotly.js
- Advantages:
 - ▶ Hover information
 - ▶ Zoom and pan
 - ▶ Toggle data visibility
- Web-ready: Runs directly in the browser
- Easy sharing: Embedding in websites and dashboards
- Typical use cases:
 - ▶ Data exploration
 - ▶ Presentations
 - ▶ Dashboards
 - ▶ Reporting

Foundations & Limitations

Technical foundations

- Uses web technologies:
 - ▶ HTML/CSS/JavaScript
 - ▶ Rendering happens in the browser
- Conceptual pipeline:
 - ▶ Data → JSON → plotly.js → Visualization

Supported Chart types

- Standard: Line charts, bar charts, scatter plots
- Advanced: Heatmaps, 3D plots, Surface plots
- Specialized: Geographic maps, Network graphs

Limitations

- More complex than static libraries
- Performance issues with very large datasets
- Sometimes less fine-grained control than low-level tools

Plotly Dash: Interactive Dashboards

- Dash is a Python/R framework for building interactive web dashboards
- Built on top of:
 - ▶ **Plotly.js** for visualization
 - ▶ **Flask** (Python) for web server
 - ▶ **React.js** for front-end components
- Key Features:
 - ▶ Fully interactive dashboards without writing HTML/JS
 - ▶ Components include: Graphs (dcc.Graph), sliders, dropdowns, input boxes, tabs, tables, buttons
 - ▶ Callback system: User actions → update charts/data dynamically
- Basic Workflow:
 - ▶ Import data and prepare Plotly charts
 - ▶ Define layout (components & arrangement)
 - ▶ Write callbacks to connect user inputs to outputs
 - ▶ Launch as a **web app** (SVM example)

Create a plot from R

```
library("plotly")
mtcars$cyl <- as.factor(mtcars$cyl) # Zylinder als Faktor behandeln
fig <- plot_ly(data=mtcars, x=~hp, y=~mpg,
  color=~cyl, size=~wt, colors="Dark2",
  text=~paste("Auto:", rownames(mtcars)), # Custom Hover-Text
  type='scatter', mode='markers') %>%
layout(title="Leistung vs. Verbrauch (mtcars)",
  xaxis=list(title="Pferdestärken (hp)"),
  yaxis=list(title="Meilen pro Gallone (mpg)"),
  legend=list(title=list(text="Zylinder")))
fig
# save as html: saveWidget(fig, "plotly_scatter.html")
browseURL("plotly_scatter.html")
```

plot_ly Parameters

- type=NULL: plot_ly tries to guess from the data given (like plot)
- Classical Graphics
 - ▶ type='bar', type='box', type='histogram', type='pie'
- Time series und Line plots
 - ▶ type='scatter' & mode='lines', type='scatter' & mode='lines+markers', fill='tozero' (Area Chart)
- Multivariate & Scientific plots
 - ▶ type='heatmap', type='scatter3d', type='surface'
- Special plots
 - ▶ type='choropleth', type='bubble', type='candlestick'
- R Example codes under <https://plotly.com/r/>

Example: Hover / Tooltips & 3D Scatter Plot

Hover / Tooltips, Zoom / Panning

```
library("plotly")
plot_ly(data=mtcars, x=~wt, y=~mpg, type="scatter",
        color=~factor(cyl), hoverinfo="text", mode="markers",
        text=~paste("Car:", rownames(mtcars),
                     "<br>HP:", hp,
                     "<br>Gears:", gear)
)
```

3D Scatter Plot

```
library("plotly")
plot_ly(mtcars, x=~wt, y=~mpg, z=~hp, color=~factor(cyl),
        type="scatter3d", mode="markers"
)
```

Example: Linked Views

```
library("plotly")
library("crosstalk")  # alternative to a data frame
library("DT")         # interactive table
library("htmlwidgets") # dashboard base
shared_mtcars <- SharedData$new(mtcars, key=rownames(mtcars))
# Create the Plotly Scatterplot using the shared data
p <- plot_ly(shared_mtcars, x=~hp, y=~mpg, color=~as.factor(cyl),
             type="scatter", mode="markers", marker=list(size=10),
             colors="Dark2") %>%
  layout(title="Select points here or in the table below",
         xaxis=list(title="Horsepower"),
         yaxis=list(title="Miles per Gallon"))
tab <- datatable(shared_mtcars, width="100%",
                 options=list(pageLength=5))
# Build Dashboard
bscols(p, tab)
# save as html: saveWidget(., "plotly_dashboard.html")
browseURL("plotly_dashboard.html")
```

Dashboard

A dashboard is a type of interface or visualization designed to monitor, explore, and analyze data quickly. In R, it usually refers to an interactive web page or application that combines multiple visualizations, tables, and controls in one place.

Feature	Single Plot	Dashboard
Scope	One visualization	Multiple linked visualizations (charts, tables, maps)
Interaction	Optional (hover/zoom)	Full interactivity (filters, selections, linked views, dropdowns, sliders, buttons)
Purpose	Display data	Explore, monitor, and analyze data

ggplotly

- `ggplotly()` turns static `ggplot2` visualizations into interactive Plotly charts with minimal effort
- What works well
 - ▶ Scatter plots, Line plots, Bar charts, Facets
- What works less well
 - ▶ Very complex ggplots (e.g. custom geoms), Some themes/layouts don't transfer perfectly, Not all ggplot features are supported

```
library("ggplot2")
p <- ggplot(mtcars, aes(x=wt, y=mpg, color=factor(cyl))) +
  geom_point()
#
library("plotly")
ggplotly(p)
```

Example: Facets

- Split the data into multiple views using faceting, and Plotly automatically makes each panel interactive

```
library("ggplot2")  
# Separate panels for each gear (3, 4, 5)  
p <- ggplot(mtcars, aes(x = wt, y = mpg, color = factor(cyl)))  
  geom_point() +  
  facet_wrap(~gear)  
# Convert to interactive plot  
library("plotly")  
ggplotly(p)
```

ggiraph

ggiraph

- An R package for interactive ggplot graphics
- Works by creating SVG-based plots with embedded interactivity
- Designed to stay very close to ggplot2 syntax

Key idea: *Make ggplot interactive while keeping full control over appearance*

Feature	ggiraph	ggplotly()
Approach	Native ggplot extension	Conversion of ggplot
Interactivity	Hover, click, highlight	Hover, zoom, pan, legend
Zoom / Pan	No	Yes
Styling control	Very precise	Limited
Complex ggplots	Works well	May have issues
Output	SVG (HTML)	HTML + JavaScript
Use case	Reports, annotations	Exploration, dashboards

Example: ggiraph

- Hover shows tooltips
- Points highlight on hover

```
library("ggplot2")
library("ggiraph")
p <- ggplot(mtcars, aes(
  x = wt,
  y = mpg,
  tooltip = rownames(mtcars)
)) +
  geom_point_interactive()
girafe(ggobj = p)
```

ggiraph Characteristics

- Interaction Model (event-based)
 - ▶ Interactivity is based on SVG + JavaScript events
 - ▶ Supports: hover, click, selection
- Fine-grained element-level Control via Aesthetics:
`aes(tooltip = ..., data_id = ...)`
- Much more control than Plotly via CSS: hover color, opacity, transitions
`girafe_options(opts_hover(css="fill:red;stroke:black;"))`
- Integration with RMarkdown / HTML: Lightweight compared to Plotly
- Selection & Highlighting Logic
 - ▶ Supports persistent selection (click), hover highlighting
 - ▶ Can define selection groups
- Performance: Efficient for small to medium datasets

echarts4r

echarts4r

- R wrapper for [Apache ECharts](#)
- Provides interactive charts based on a powerful JS engine
- Part of the htmlwidgets ecosystem (like Plotly)

Key idea: *High-performance, highly customizable interactive charts with a dashboard feel*

```
library("echarts4r")  
mtcars |>  
  e_charts(wt) |>  
  e_scatter(mpg, symbol_size = 10) |>  
  e_tooltip()
```

Features & Examples

Key Features

- Built-in interactivity: hover, zoom, tooltips
- Very smooth animations
- Rich chart types: bar, line, scatter, radar, heatmap, network, ...
- Strong support for dashboards

Example: Zoom slider

```
mtcars |>  
  e_charts(wt) |>  
  e_scatter(mpg, symbol_size = 10) |>  
  e_x_axis(name = "Car Weight (1000 lbs)") |>  
  e_y_axis(name = "Miles per Gallon") |>  
  e_datazoom(type = "inside")
```

Shiny

Shiny is an R framework for building interactive web applications directly from R code

- Create a UI (buttons, slider, tables) and a server (R logic: computation, models, data)
- Use of reactive programming → outputs update automatically when inputs change.
- No need for deep knowledge of HTML/CSS/JavaScript.

Packages

- shiny central framework for web apps in R
- shinydashboard dashboards mit sidebar, header etc.
- shinythemes ready-to-use CSS theme for fast styling
- shinyWidgets provides enhanced UI components (switches, sliders, picker inputs, etc.)
- htmltools tools for generating HTML and building UI components
- shinyjs adds JavaScript functionality directly into Shiny apps
- reactlog debugging reactive dependencies in Shiny
- ...

Example: Histogram

```
library("shiny")
# UI
ui <- fluidPage(
  titlePanel("Histogram Example"),
  sliderInput("n", "Number of observations:", 10, 1000, 100),
  plotOutput("histPlot")
)
# Server
server <- function(input, output) {
  output$histPlot <- renderPlot({
    data <- rnorm(input$n)
    hist(data)
    rug(data)
  })
}
# Run app
shinyApp(ui, server)
```

How does Shiny work?

- User Interface: `ui <- ...` defines the layout
- Server: `server <- function(input, output) { ... }`
- Run App: `shinyApp(ui, server)`
- Reactivity (the core)
 - ▶ `input$...` → user inputs
 - ▶ `output$...` → outputs
 - ▶ `render*()` → generates output
 - ▶ `reactive()` → reusable logic

Example: When the slider moves, the histogram is automatically recalculated.

Example: Dashboard

```
library("shiny")
library("shinydashboard")
# UI
ui <- dashboardPage(
  dashboardHeader(title = "Histogram Dashboard"),
  dashboardSidebar(sliderInput("n", "# obs:", 10, 1000, 100)),
  dashboardBody(plotOutput("histPlot"))
)
# Server
server <- function(input, output) {
  output$histPlot <- renderPlot({
    data <- rnorm(input$n)
    hist(data)
    rug(data)
  })
}
# Run the app
shinyApp(ui, server)
```

plotly / echarts4r / Shiny

- What does ...
 - ▶ plotly → adds interactive plots (easy)
 - ▶ echarts4r → adds more advanced / animated visualizations
 - ▶ Shiny → builds a whole app
- Typical setup
 - ▶ Shiny + plotly → fast & simple dashboards
 - ▶ Shiny + echarts4r → more advanced / polished visualizations
- Rule of thumb
 - ▶ Use plotly → if you want quick interactivity
 - ▶ Use echarts4r → if you want high-end visuals & animations
 - ▶ Use Shiny → if you want a full web application

Drawbacks

- Plotly
 - ▶ Can be slow with very large datasets
 - ▶ Limited animation/advanced chart options compared to echarts4r
 - ▶ Custom styling sometimes tricky / heavy dependency on JavaScript
- echarts4r
 - ▶ Steeper learning curve than Plotly
 - ▶ Less seamless integration with some R functions
 - ▶ Documentation slightly less beginner-friendly
 - ▶ Requires more setup for very simple plots
- Shiny
 - ▶ Can become slow for complex apps or large datasets / performance issues
 - ▶ Not a visualization library by itself (needs plotly/echarts4r/ggplot2 for charts)
 - ▶ Scalability requires more planning (e.g., server resources, modules)

```
library("HKRbook")  
men_corr()
```

Exercise: Explore the Iris Dataset

- Create a basic Shiny app
 - ▶ Define a UI with a title for your app.
 - ▶ Reserve space in the UI for two plots:
 - ▶ One Plotly histogram of Sepal.Length
 - ▶ One ECharts4R scatter plot of Sepal.Length vs Petal.Length
- Write the server logic
 - ▶ For the Plotly histogram, plot Sepal.Length with a color of your choice.
 - ▶ For the ECharts4R scatter plot, map Sepal.Length to the x-axis and Petal.Length to the y-axis, and enable tooltips.
- Run your Shiny app
 - ▶ Use `shinyApp(ui, server)` to launch the app.
 - ▶ Check that the Plotly histogram and ECharts4R scatter plot display correctly.
- Optional Challenge
 - ▶ Try changing the colors of the histogram bars or scatter points.
 - ▶ Explore plotting other iris variables, like Petal.Width or Sepal.Width.