Syllabus :: Module -4 (Statistics with R- Apply Level)

R Graphics- Overview, Customizing Charts, Graphical parameters, Basic Graphics functions, Lattice Graphics - Lattice functions, Customizing Lattice Graphics, Ggplot.

(CO 4: Visualize different types of data (Cognitive Knowledge level: Apply)]

Overview of Textbook Contents Aligned with the Syllabus [Module 4]

Based on the "AIT 362 Programming in R Syllabus", Module 4 focuses on **Data Visualization**. Here's an overview of each topic within this module, drawing upon both the syllabus and "R in a Nutshell 2nd edition":

- **R Graphics- Overview**: This introductory topic likely covers the fundamental concepts of how graphics are generated in R. "R in a Nutshell" mentions an **overview of R graphics** in Chapter 13, noting that it will discuss various types of plots such as scatter plots, time series plots, bar charts, and pie charts. It also mentions that R comes with a number of different packages, including `graphics` and `grDevices`, which implement basic features of the R language or R environment for creating visualizations. The syllabus itself indicates that this section will provide a general introduction to R's graphical capabilities.

- **Customizing Charts**: This topic will likely delve into how to modify the appearance of R graphs beyond the defaults. "R in a Nutshell" has a section on **customizing charts** , which discusses common arguments to chart functions like `main`, `sub`, `xlab`, and `ylab` for titles and labels. It also covers **graphical parameters** that control various aspects of the plots, such as margins (`mar`, `mai`), plotting region (`pty`), and axis labels (`las`). The syllabus highlights that customizing charts is a key aspect of this module.

- **Graphical parameters**: This topic specifically focuses on the settings that control the look and feel of R plots. Chapter 13 of "R in a Nutshell" details **graphical parameters** , explaining how to set them using the `par` function. It provides a table of common graphical parameters, covering aspects like color (`col`), point character (`pch`), line type (`lty`), line width (`lwd`), font size (`cex`), and margins (`mar`). The syllabus lists "Graphical parameters" as a distinct topic within Module 4.

- **Basic Graphics functions**: This section will likely cover the core functions in R's base graphics system used to create different types of plots. "R in a Nutshell" describes several **basic graphics functions**, such as `plot` for scatter plots, `lines` for adding lines, `points` for adding points, `barplot` for bar charts, `hist` for histograms, `boxplot` for box plots, `abline` for adding lines based on intercept and slope, `curve` for plotting mathematical functions, `text` for adding text, `title` for adding titles, `axis` for customizing axes, `legend` for adding legends, `polygon` for drawing polygons, and `segments` for drawing line segments . The syllabus explicitly mentions "Basic Graphics functions".

- **Lattice Graphics - Lattice functions**: This topic introduces the `lattice` package, which provides a powerful system for creating trellis graphs. "R in a Nutshell" dedicates Chapter 14 to **Lattice Graphics** , explaining that it is built on the `grid` graphics system. It discusses **high-level lattice plotting functions** like `xyplot` for scatter plots, `barchart` for bar charts, `histogram` for histograms, `densityplot` for density plots, `dotplot` for Cleveland dot plots, `stripplot` for strip charts, `qqmath` for quantile-quantile plots, and `levelplot` and `contourplot` for visualizing three-dimensional data . The syllabus lists "Lattice Graphics - Lattice functions" as a topic.

- **Customizing Lattice Graphics**: This section focuses on how to modify the appearance of graphs created using the `lattice` package. "R in a Nutshell" explains how to customize lattice plots using arguments within the high-level functions, as well as through **panel functions** and **trellis options** set with `trellis.par.set` and retrieved with `trellis.par.get` . It discusses customizing aspects like axes, titles, legends (keys), and the appearance of panels . The syllabus includes "Customizing Lattice Graphics" as a specific topic.

- **Ggplot**: This topic introduces the `ggplot2` package, which implements the grammar of graphics, providing a flexible and powerful approach to data visualization. "R in a Nutshell" has a chapter on **ggplot2** , explaining its fundamental components such as **data**, **aesthetics**, **geometries**, **statistical transformations**, **scales**, **coordinate systems**, **faceting**, and **positional adjustments**. It highlights the use of the `qplot` function for quick plots and demonstrates how to build more complex plots layer by layer using the `ggplot()` function . The syllabus includes "Ggplot" as the final topic in Module 4.

R Graphics- Overview

- **Variety of Chart Types:** R includes tools for drawing most common types of charts, such as **bar charts**, **pie charts**, **line charts**, and **scatter plots**. Additionally, R can create less common charts like **quantile-quantile (Q-Q) plots**, **mosaic plots**, and **contour plots**. The `graphics` package contains a wide variety of functions for plotting data.

- **Key Packages:** The "R in a Nutshell" book highlights three popular packages for plotting graphics: **graphics**, **lattice**, and **ggplot2**. The `graphics` package is easy to use for customizing and modifying charts or interacting with plots on the screen. The `lattice` package provides an alternative set of functions well-suited for splitting data by a conditioning variable. `ggplot2` employs a different approach based on the grammar of graphics for creating visually appealing charts quickly. The syllabus mentions "Basic Graphics functions" separately, suggesting a focus on the functions within the base `graphics` package.

- **Flexibility and Customization:** R offers a significant amount of control over graphics, allowing users to control almost every aspect of a chart. "Customizing Charts" is a dedicated topic in Module 4, and Chapter 13 in "R in a Nutshell" also discusses this. This includes adjusting titles, labels (`xlab`, `ylab`, `main`, `sub`), controlling axes, and using various **graphical parameters**.

- **Graphical Parameters:** The `par` function can be used to set and query **graphical parameters**. These parameters control various aspects of the plots, such as background color (`bg`), margin size (`mai`, `mar`), plotting region (`pty`), and text properties (`ps`, `cex`, `cex.axis`, `cex.lab`, `cex.main`, `cex.sub`). A list of common graphical parameters is available.

- **Basic Graphics Functions:** Module 4 includes "Basic Graphics functions" as a specific topic. Chapter 13 of "R in a Nutshell" details these, mentioning functions like `plot` for scatter plots, `lines` for adding lines , `points` for adding points , `barplot` for bar charts, `hist` for histograms, `boxplot` for box plots, `abline` for adding lines, `curve` for plotting functions, `text` for adding text , `title` for adding titles , `axis` for customizing axes, `legend` for adding legends, `polygon` for drawing polygons , and `segments` for drawing line segments. These functions can be used to either modify existing charts or create new ones from scratch.

- **Output Devices:** R graphics can be displayed on the screen or saved in various formats. The "Graphics Devices" section (page 246 in "R in a Nutshell") explains how to choose output methods.

- **Annotation:** Titles and axis labels are referred to as chart annotation, and their control is managed by the `ann` parameter. Setting `ann=FALSE` suppresses the printing of titles and axis labels.

In essence, the "R Graphics- Overview" in Module 4 lays the groundwork for understanding the capabilities of R's base graphics system, the various types of plots it can generate, and the fundamental tools available for customizing their appearance. It sets the stage for subsequent topics in the module that delve into more advanced graphics systems like Lattice and Ggplot.

Customizing Charts

Overview of Customization Methods

"R in a Nutshell" highlights several ways to customize R plots:

- **Arguments to Charting Functions**: Many R plotting functions have specific arguments that allow you to directly control aspects of the chart, such as titles, labels, colors, and more.
- **Setting Session Parameters**: You can modify global graphical settings that will affect all subsequent plots within your R session. This is primarily done using the `par()` function to set **graphical parameters**.
- **Functions that Modify Charts**: After creating a basic plot, you can use additional functions to add elements like titles (`title`), legends (`legend`), trend lines (`abline`), or more points (`points`).
- **Writing Custom Charting Functions**: For highly specific or repetitive tasks, you have the flexibility to create your own functions to generate plots.

Common Arguments to Chart Functions

Many charting functions in R share some common arguments, as outlined in "R in a Nutshell":

- `add`: A logical value indicating whether the plot should be added to an existing plot on the device (`TRUE`) or if the device should be cleared first (`FALSE`).
- `axes`: A logical value controlling whether axes should be plotted on the chart (`TRUE` by default).
- `log`: Controls whether points are plotted on a logarithmic scale; you can specify `"x"`, `"y"`, or `"xy"`.
- `main`: The main title for the plot.
- `sub`: The subtitle for the plot.
- `xlab`: The label for the x-axis.
- `ylab`: The label for the y-axis.
- `ann`: A logical value; if `TRUE` (default), axis titles and overall titles are included. If `FALSE`, these annotations are not included.
- `axes`: A logical value specifying whether axes should be drawn (`TRUE` by default).
- `frame.plot`: A logical value specifying whether a box should be drawn around the plot (defaults to the value of `axes`).
- `panel.first`: An expression that is evaluated after the axes are drawn but before points are plotted.

Graphical Parameters

Graphical parameters control various aspects of the appearance of plots. These can be set and queried using the `par()` function. "R in a Nutshell" provides a detailed list of these parameters, and they cover a wide range of features:

- **Annotation**:
  - `ann`: Controls whether titles and axis labels are printed.
  - `main`, `sub`, `xlab`, `ylab`: Specify the text for titles and labels.
  - `cex.main`, `cex.sub`, `cex.lab`, `cex.axis`: Control the size of the text for different annotation elements.
  - `font.main`, `font.sub`, `font.lab`, `font.axis`: Specify the font style for different annotation elements.
- **Margins**:
  - `mar`: A numeric vector `c(bottom, left, top, right)` specifying the margin sizes in lines of text.
  - `mai`: A numeric vector `c(bottom, left, top, right)` specifying the margin sizes in inches.
  - `mex`: A value controlling the size of a line of text in the margins.
  - `omi`: A numeric vector `c(bottom, left, top, right)` specifying the outer margin in lines.
  - `omd`: A numeric vector `c(bottom, left, top, right)` specifying the outer margin as a fraction of the device size.
- **Text Properties**:
  - `ps`: The default point size of text.
  - `cex`: A default scaling factor for text.
  - `adj`: Controls the alignment of text.
  - `lheight`: Controls the spacing between lines of text.
  - `crt`: Rotates individual characters.
  - `srt`: Rotates whole strings.
  - `family`: Specifies the font family.
  - `font`: Specifies the text style (plain, bold, italic, bold-italic).
- **Line Properties**:
  - `lend`: Controls the line end style (round, butt, square).
  - `ljoin`: Controls the line join style.
  - `lmiter`: Controls the line miter limit.
  - `lty`: Specifies the line type (solid, dashed, dotted, etc.).
  - `lwd`: Specifies the line width.
  - `bty`: Specifies the type of box to draw around the plot.
- **Axes**:
  - `lab`: Controls axis annotation.
  - `las`: Specifies the style of axis labels (parallel, horizontal, perpendicular, vertical).

    - `mgp`: Controls the margin for the axis title, labels, and lines.
    - `tcl`, `tck`: Specify the size of tick marks.
    - `xaxp`, `yaxp`: Control the minimum and maximum tick mark locations and the number of tick marks.
    - `xaxs`, `yaxs`: Control the calculation method for axis intervals.
    - `xaxt`, `yaxt`: Specify the axis type; use `"n"` for no axis.
    - `col.axis`, `col.lab`, `col.ticks`: Control the colors of axis elements.
- **Background and Clipping**:
    - `bg`: The background color for the device region.
    - `xpd`: Controls clipping of drawing to the plot region, figure region, or device region.
- **Multiple Plots**:
    - `mfg`: Specifies the location of the next plot in a matrix of subplots.
    - `mfcol`, `mfrow`: Set up a matrix of subplots.
    - `new`: Indicates whether the plotting routine should pretend the device has been freshly initialized, allowing plotting on top of another plot.
- **Points**:
    - `pch`: Specifies the plotting character to use for points.
    - `cex`: Controls the size of points.
    - `col`: Specifies the color of points.
- **Logarithmic Scales**:
    - `log`: For chart functions, controls logarithmic scales on axes.
    - `xlog`, `ylog`: Graphical parameters to specify logarithmic scales.

Basic Graphics Functions and Customization

The "Basic Graphics functions" mentioned in the syllabus (and detailed in "R in a Nutshell") are often used in conjunction with customization techniques. These low-level functions can be called directly and their appearance can be controlled through graphical parameters and function-specific arguments. For example:

- `plot()`: Used for creating scatter plots and other types of plots. Arguments like `type`, `xlim`, `ylim`, `col`, `pch`, etc., allow for direct customization .
- `lines()` and `points()`: Add lines or points to an existing plot. Their appearance is controlled by arguments like `col`, `lty`, `lwd` (for `lines`), and `col`, `pch`, `cex` (for `points`) .
- `barplot()`: Creates bar charts with arguments to control colors (`col`), labels (`names.arg`), and orientation (`horiz`) .
- `hist()`: Generates histograms with arguments for breaks, colors, and titles .
- `boxplot()`: Creates box plots with options to control notch, color, and outliers .
- `title()`, `xlab()`, `ylab()`: Functions to add or modify plot titles and axis labels .
- `axis()`: Provides fine-grained control over drawing and customizing plot axes .
- `legend()`: Adds a legend to the plot with many customization options .

By understanding and utilizing these arguments and graphical parameters, you can effectively customize the appearance of your charts in R to best represent your data and communicate your findings.

Extra : **Use of par( ) function** **[A revision]**

The `par()` function in R is used to **set or query graphical parameters** for the current graphics device. It allows you to control many aspects of the plots you create using the base `graphics` package.

Here's a breakdown of how `par()` is used, based on the sources:

- **Setting Graphical Parameters:** You can set graphical parameters by using the parameter name as an argument name within the `par()` function. For example, to change the background color of subsequent plots to white, you would use `par(bg="white")`. These settings will become the **defaults for any new plot** until you close the device.

- **Querying Graphical Parameters:** To check the current value of a specific graphical parameter, you pass the name of the parameter as a character string to `par()`. For instance, to see the current background color, you would use `par("bg")`.

- **Getting All Graphical Parameters:** Calling `par()` with no arguments returns a **list of all current graphical parameters** and their values.

- **Scope of Parameters:** The `par()` function sets the graphical functions for a **specific graphics device**. The new settings become the defaults for that device. If there is no active device when `par()` is called, it will open the default device. Changes made with `par()` in the current environment will alter the attributes but will not affect the attributes in an enclosing environment.

- **Usefulness:** `par()` is useful when you want to **set a consistent set of graphical parameters** across multiple plots. You could even write a function to set a desired set of parameters and call that function before plotting.

- **Caution:** While many plotting functions have arguments with the same names as graphical parameters set by `par()`, they might have **different meanings or apply only to that specific function call**. Therefore, it's always a good idea to check the help file for each function to understand the exact behavior of its arguments.

Examples of Graphical Parameters

The sources mention several graphical parameters that can be controlled by `par()`:

- `bg`: Specifies the **background color** for plots.
- `cex`: A default **scaling factor for text**.
- `cex.axis`: Text **magnification for axis annotation text**.
- `col`: Default **plotting color**.
- `col.axis`: **Color for axis annotation**.
- `col.lab`: **Color for axis labels**.
- `las`: Specifies the **style of the axis labels**.

The "Graphical Parameters" section in Chapter 13 of "R in a Nutshell" (page 247 onwards) provides a more comprehensive list and details about each graphical parameter. You can also get a list of all parameters by calling `par()` with no arguments.

Graphical parameters **[Already Discussed]**

Graphical parameters in R control various aspects of the appearance of plots created using the base `graphics` package. You can customize almost every detail of a chart by manipulating these parameters.

There are primarily two ways to work with graphical parameters:

- **As arguments to charting functions**: Many high-level plotting functions in R, such as `plot()`, `barplot()`, `hist()`, etc., accept graphical parameters directly as arguments. For example, you can set the main title using the `main` argument, axis labels using `xlab` and `ylab`, and axis limits using `xlim` and `ylim`. The type of plot elements (points, lines, etc.) can be controlled by the `type` argument. Colors are often controlled by the `col` argument.

- **Using the `par()` function**: The `par()` function is specifically designed to **set or query graphical parameters** for the current graphics device.

    - **Setting parameters**: You can set graphical parameters by providing their names as arguments to `par()` with the desired values. These settings become the **defaults for all subsequent plots** created on that device until the device is closed. For instance, `par(bg="white")` sets the background color to white.
    - **Querying parameters**: To check the current value of a specific graphical parameter, you pass its name as a character string to `par()`, like `par("bg")`.
    - **Getting all parameters**: Calling `par()` without any arguments returns a list of all current graphical parameters and their values.

The sources provide a comprehensive list of graphical parameters, categorized by the aspects of the plot they control:

- **Annotation**: Parameters like `ann` (control overall titles and labels), `main` (main title), `sub` (subtitle), `xlab` (x-axis label), `ylab` (y-axis label), `cex.main`, `cex.sub`, `cex.lab`, `cex.axis` (control text sizes), and `font.main`, `font.sub`, `font.lab`, `font.axis` (control font styles).

- **Margins**: Parameters such as `mar` (margin size in lines of text), `mai` (margin size in inches), `mex` (expansion factor for `mar`), `omi` (outer margin in lines), `omd` (outer margin as fraction of device), and `mgp` (margin line for axis title, labels, and lines). Figure 13-24 in "R in a Nutshell 2nd edition.pdf" illustrates how margins work.

- **Text Properties**: Parameters like `ps` (default point size), `cex` (default scaling factor), `adj` (text alignment), `lheight` (line spacing), `crt` (character rotation), `srt` (string rotation), `family` (font family), and `font` (text style).

- **Line Properties**: Parameters including `lend` (line end style), `ljoin` (line join style), `lmiter` (miter limit), `lty` (line type), `lwd` (line width), and `bty` (box around the plot).

- **Axes**: Parameters such as `lab` (axis annotation), `las` (axis label style), `mgp` (axis margin), `tcl`, `tck` (tick mark sizes), `xaxp`, `yaxp` (tick mark locations and number), `xaxs`, `yaxs` (axis interval calculation), and `xaxt`, `yaxt` (axis type, e.g., `"n"` for no axis).

- **Points**: The `pch` parameter controls the plotting character used for points, and `cex` and `col` also affect the appearance of points.

- **Background and Clipping**: Parameters like `bg` (background color) and `xpd` (controls clipping).

- **Multiple Plots**: Parameters for managing multiple plots within a single device include `mfcol`, `mfrow` (set up a matrix of subplots), and `mfg` (specify the location of the next plot).

It's important to note that while some plotting functions have arguments with the same names as graphical parameters set by `par()`, they might have **different meanings or apply only to that specific function call**. Therefore, always refer to the help documentation for each function to understand the precise behavior of its arguments.

The `par()` function is useful for setting a consistent look for multiple plots without repeatedly specifying the same arguments in each plotting function call. You can even create custom functions to set a specific theme of graphical parameters.

In contrast to the base `graphics` package, the `lattice` package has its own system for controlling plot aesthetics using `trellis.par.get()` and `trellis.par.set()`, with a hierarchical structure of parameters. While some underlying concepts might be similar (e.g., controlling colors, lines, text), the mechanisms and parameter names differ. Similarly, `ggplot2` utilizes a "grammar of graphics" with aesthetic properties (`aes`) defined within the `ggplot()` and subsequent layer functions to control the visual aspects of plots.

Basic Graphics functions in R

Basic Graphics functions in R, found within the `graphics` package, are the foundational tools for creating a wide variety of statistical charts and customizing their appearance. These functions can be used at an application level to both generate standard plot types and to build highly customized visualizations from the ground up.

Here's a more detailed look at their application:

- **Generating Common Chart Types:** Basic Graphics functions allow you to create essential plot types directly. The sources list several such functions and their corresponding chart types:
  - **Scatter Plots:** The `plot()` function is a generic function used for plotting R objects, and when given two numeric vectors, it generates a scatter plot. You can specify the type of plot using the `type` argument (e.g., `"p"` for points, `"l"` for lines). `points()` can add points to an existing plot, and `lines()` can add line segments. `matplot()` is useful for plotting columns of one matrix against columns of another. `smoothScatter()` is preferred for a very large number of points to visualize density.
  - **Bar and Column Charts:** `barplot()` creates bar plots with vertical or horizontal bars. The data for barplot is specified with the `height` argument.
  - **Histograms:** `hist()` computes and plots a histogram of the given data values.
  - **Pie Charts:** `pie()` draws a pie chart. Arguments like `clockwise` and `init.angle` control the appearance.
  - **Box Plots:** `boxplot()` produces box-and-whisker plots of given (grouped) values. `bxp()` draws box plots based on given summaries.
  - **Dot Plots:** `dotchart()` draws a Cleveland dot plot.
  - **Quantile-Quantile (Q-Q) Plots:** `qqnorm()` creates a Q-Q plot against a normal distribution, and `qqplot()` plots the quantiles of two data sets against each other.
  - **Contour Plots:** `contour()` creates contour plots or adds contour lines to an existing plot. `filled.contour()` creates contour plots with filled areas.
  - **Perspective Plots:** `persp()` draws perspective plots of surfaces over the x-y plane. The `trans3d()` function can be used to add lines or points to a perspective plot.
  - **Image Plots:** `image()` creates a grid of colored or grayscale rectangles.

- **Customizing Charts:** Basic Graphics functions, often in conjunction with `par()`, provide a high degree of control over the appearance of plots.

    - **Adding Titles and Labels:** `title()` adds labels to a plot, including the main title, subtitle, and axis labels. The `xlab` and `ylab` arguments are common to many charting functions for setting axis labels. The `ann` parameter can control whether these annotations are included.
    - **Modifying Axes:** The `axis()` function adds a suitable axis to the current plot, allowing for customization of the axis position, labels, and tick marks. Parameters like `las` control the style of axis labels, and `cex.axis` controls the size of axis annotation text. `log` controls logarithmic scales.
    - **Adding Lines and Shapes:** `abline()` adds one or more straight lines, `segments()` draws line segments between pairs of points, `polygon()` draws polygons, and `rect()` draws rectangles. `curve()` plots a curve corresponding to a given function or expression.
    - **Adding Text and Legends:** `text()` draws strings at specified coordinates, and `mtext()` writes text in one of the four margins. `legend()` adds legends to plots.
    - **Controlling Appearance:** Graphical parameters set by `par()` (as discussed previously) can influence colors (`col`, `bg`, `col.axis`, `col.lab`, `col.main`, `col.sub`), line types (`lty`), line widths (`lwd`), point characters (`pch`), text size (`cex`, `cex.axis`, `cex.lab`, `cex.main`, `cex.sub`), and margins (`mar`, `mai`, `oma`, `omi`).
- **Relationship with Higher-Level Functions:** Higher-level plotting functions like `plot()`, `barplot()`, `hist()`, etc., internally call these basic graphics functions to do the actual drawing. This means that many of the arguments in high-level functions correspond to parameters used by the lower-level functions, allowing for convenient customization without directly using the basic functions in all cases. For instance, you can often pass `xlab`, `ylab`, `main`, `col`, `lty`, `lwd`, `pch`, and `cex` directly to `plot()` because `plot()` will pass these arguments down to functions like `points()` and `lines()`.

- **Building Plots from Scratch:** For more intricate or less standard visualizations, you can use the basic graphics functions directly. This involves setting up a plot area with `plot.new()` and `plot.window()`, and then sequentially adding graphical elements using functions like `points()`, `lines()`, `polygon()`, `text()`, `axis()`, and `title()`.

In summary, Basic Graphics functions in R provide a powerful and flexible toolkit for creating and customizing a wide range of plots. They are used at an application level both directly for building custom graphics and indirectly through higher-level plotting functions that simplify the creation of standard chart types while still allowing for extensive customization. The control offered by these functions and the associated graphical parameters makes the base `graphics` package a fundamental part of data visualization in R.

If you are interested here are some Basic Graphics functions in R from the `graphics` package:

Function Name	Description	Typical Use Cases	Related High-Level Functions (if applicable)
`plot()`	A **generic function for plotting R objects**. It dispatches to specific plotting functions based on the object's class.	Creating **scatter plots**, line plots, or other basic visualizations depending on the input data.	Serves as the foundation for many basic plots.
`points()`	**Adds points to an existing plot**.	Highlighting specific data points, adding multiple sets of points to a single plot.	Called internally by `plot()` for scatter plots and can be used to enhance existing plots.
`lines()`	**Adds line segments connecting points to an existing plot**.	Showing trends, connecting data points in time series or other sequential data.	Called internally by `plot()` for line plots and can be used to enhance existing plots.
`abline()`	**Adds one or more straight lines across the plot area** with specified intercept and slope, or horizontal/vertical lines.	Indicating thresholds, showing linear trends or regression lines.	Can be used to add reference lines to any type of plot.
`segments()`	**Draws line segments between pairs of specified points**.	Connecting specific data points that are not necessarily sequential, visualizing movement or relationships between pairs.	Can be used to add customized line connections to plots.

`polygon()`	**Draws polygons defined by a sequence of connected vertices**.	Shading areas, creating custom shapes on a plot.	Used in functions like `pie()` and can be used for custom area fills.
`rect()`	**Draws rectangles**.	Highlighting regions in a plot, creating bar-like elements in custom visualizations.	Can be used to build custom bar charts or highlight areas.
`curve()`	**Draws a curve corresponding to a given function or expression** over a specified interval.	Plotting mathematical functions.	Can be used to overlay theoretical curves on data plots.
`text()`	**Draws strings (text) at specified coordinates**.	Labeling specific points, adding annotations to a plot.	Can be used to add informative labels to any plot type.
`mtext()`	**Writes text in one of the four margins of the current figure region or the outer margins of the device region**.	Adding information outside the main plotting area, such as additional titles or comments.	Useful for adding marginal notes to any plot.
`title()`	**Adds labels to a plot**, including the main title, subtitle, and axis labels.	Providing context and identifying the plot.	Most high-level plotting functions call this internally or have arguments for titles and labels.

`axis()`	**Adds a suitable axis to the current plot**, allowing for customization of the side, position, labels, and other options.	Customizing axis appearance, adding additional axes.	High-level functions usually draw axes automatically, but `axis()` allows for fine-grained control.
`box()`	**Draws a box around the current plot** with a given color and line type. The `bty` parameter controls the type of box drawn.	Defining the boundaries of the plot area.	Most high-level functions draw a box by default.
`legend()`	**Adds legends to plots** to identify different elements like colors or symbols.	Distinguishing multiple data series or groups on a single plot.	Essential for interpreting plots with multiple components.
`grid()`	**Adds an nx-by-ny rectangular grid to an existing plot**.	Providing visual guides for reading values on the axes.	Can be overlaid on various plot types.
`arrows()`	**Draws arrows between pairs of points**.	Indicating direction or flow in a visualization.	Useful for showing movement or relationships with directionality.
`rug()`	**Adds a rug representation (1D plot) of the data to the plot**, showing the location of individual data points along the axes.	Displaying the distribution of data points, especially useful in conjunction with other plots like scatter plots or density plots.	Can be added to existing plots to show marginal distributions.

`locator()`	**Reads the position of the graphics cursor when the (first) mouse button is pressed**.	Allowing interactive selection of points on a plot, for example, to label them.	Useful for user interaction with plots.
`plot.new()` / `frame()`	**Starts a new plot**. This function causes the completion of plotting in the current plot (if there is one) and an advance to a new graphics frame.	Initializing a new, blank plotting area.	Called internally by `plot()` and other high-level functions at the beginning of a new plot.
`plot.window()`	**Sets up the world coordinate system for a graphics window**. It is called by higher-level functions such as `plot.default()` after `plot.new()`.	Defining the range and scaling of the plot axes.	Called internally to prepare the plotting area.

This table provides a comparison of some fundamental Basic Graphics functions available in R's `graphics` package, highlighting their primary purpose and typical applications based on the information in the sources. These functions can be used individually to build custom graphics or are called internally by higher-level plotting functions.

Extra : Pie Chart **[A detailed explanation]**

What is a Pie Chart?

A pie chart is described as one of the most popular ways to plot data. It is considered an effective way to compare different parts of a quantity. However, the source also notes that there are many reasons not to use pie charts. Specifically, it mentions that pie charts are good for showing how much bigger one number is than another and for taking up a lot of space on a page, but they are not good at showing subtle differences between the sizes of different slices. The help file for the `pie` function itself is quoted as saying, "**Pie charts are a very bad way of displaying information. The eye is good at judging linear measures and bad at judging relative areas. A bar chart or dot chart is a preferable way of displaying this type of data.**".

The `pie()` Function in R

You can draw pie charts in R using the `pie()` function. The syntax for the function is as follows:

```
pie(x, labels = names(x), edges = 200, radius = 0.8,
    clockwise = FALSE, init.angle = if(clockwise) 90 else 0,
    density = NULL, angle = 45, col = NULL, border = NULL,
    lty = NULL, main = NULL, ...)
```

Arguments of the `pie()` Function

The following table describes the arguments to the `pie()` function and their default values:

Argument	Description	Default
`x`	**A vector of nonnegative numeric values that will be plotted.**	
`labels`	An expression to generate labels, a vector of character strings, or another object that can be coerced to a graphicsAnnot object and used as labels.	`names(x)`
`edges`	A numeric value indicating the number of segments used to draw the outside of the pie.	`200`
`radius`	A numeric value that specifies how big the pie should be. (Parts of the pie are cut off for values over 1.)	`0.8`
`clockwise`	**A logical value indicating whether slices are drawn clockwise or counterclockwise.**	`FALSE`

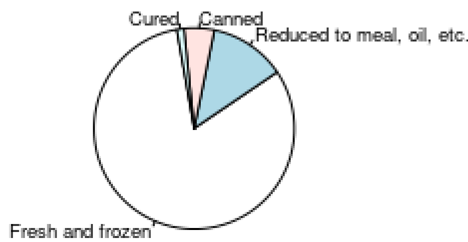
`init.angle`	A numeric value specifying the starting angle for the slices (in degrees).	`if (clockwise) 90 else 0`
`density`	A numeric value that specifies the density of shading lines in lines per inch. `density=NULL` means that no lines are drawn.	`NULL`
`angle`	A numeric value that specifies the slope of the shading lines (in degrees).	`45`
`col`	A numeric vector that specifies the colors to be used for slices. If `col=NULL`, then a set of six pastel colors is used.	`NULL`
`border`	Arguments passed to the `polygon` function to draw each slice.	`NULL`
`lty`	The line type used to draw each slice.	`NULL`
`main`	A character string that represents the title.	`NULL`
`...`	Additional arguments passed to other graphical routines.	

Executing Code Example

The source provides the following example to create a pie chart showing what happened to fish caught in the United States in 2006:

```
# 2006 fishery data from

#   http://www.census.gov/compendia/statab/tables/09s0852.xls

# units are millions of pounds of live fish
```

```
domestic.catch.2006 <- c(7752, 1166, 463, 108)

names(domestic.catch.2006) <- c("Fresh and frozen",

                     "Reduced to meal, oil, etc.", "Canned", "Cured")

# note: cex.6 setting shrinks text size by 40% so you can see the labels

pie(domestic.catch.2006, init.angle=100, cex=.6)
```



Output Pie chart (for the above code)

This code first creates a numeric vector `domestic.catch.2006` with the amounts of fish caught in different categories and then assigns names to these categories. Finally, it uses the `pie()` function to generate a pie chart. The `init.angle` argument sets the starting angle of the first slice, and `cex=.6` reduces the text size of the labels by 40% to prevent them from overlapping. The resulting chart (Figure 13-10 in the source) visually represents the proportion of fish caught in each category, showing that most of the fish (by weight) was sold fresh or frozen.

Extra : Scatter Plot **[A detailed explanation]**

What is a Scatter Plot?

A scatter plot is a way to visually represent the relationship between two numeric variables. Each point on the plot corresponds to a pair of values for the two variables, with one variable plotted on the horizontal x-axis and the other on the vertical y-axis. Scatter plots are useful for identifying patterns, trends, and correlations between variables.

Basic Scatter Plots with the `plot()` Function

The `plot()` function in the `graphics` package (which is part of base R) is the most common way to create simple scatter plots.

The basic syntax for `plot()` when creating a scatter plot is:

plot(x, y, ...)

where `x` and `y` are vectors containing the data for the two variables you want to plot. The `...` represents additional graphical parameters that you can use to customize the appearance of the plot.

Common Arguments to the `plot()` Function for Scatter Plots

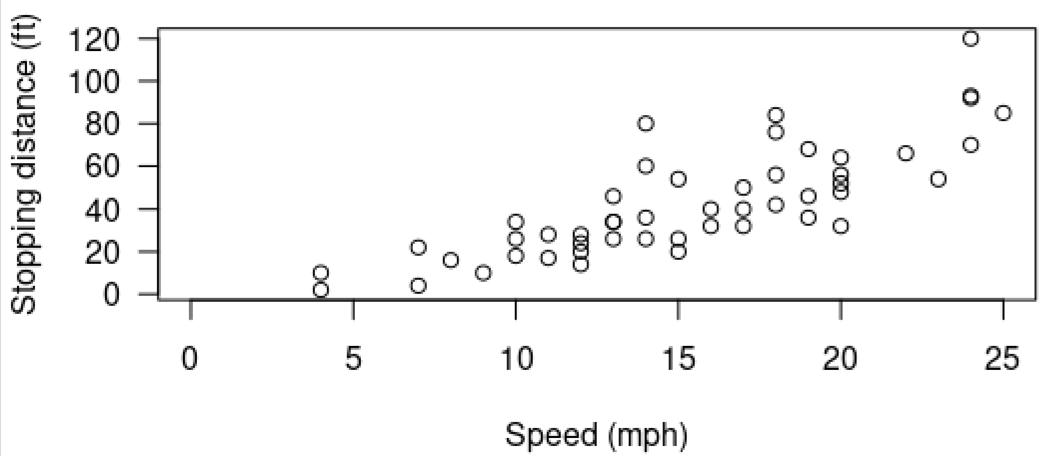
Argument	Description	Default
`x`	The data to be plotted on the x-axis.	
`y`	The data to be plotted on the y-axis. If `y` is `NULL`, `plot(x)` can behave differently depending on the type of `x`.	`NULL`
`type`	Specifies the type of plot. For a scatter plot, use `"p"` for points.	`"p"`
`xlim`	A numeric vector of length 2 specifying the minimum and maximum x-axis limits.	`NULL`
`ylim`	A numeric vector of length 2 specifying the minimum and maximum y-axis limits.	`NULL`
`xlab`	A character string for the label of the x-axis.	`NULL`

`ylab`	A character string for the label of the y-axis.	`NULL`
`main`	A character string for the main title of the plot.	`NULL`
`sub`	A character string for the subtitle of the plot.	`NULL`
`col`	The color to be used for the points.	Determined by graphical parameters.
`pch`	The plotting character to be used for the points.	Determined by graphical parameters.
`cex`	A numeric value giving the factor by which plotting text and symbols should be scaled relative to the default.	`NULL`
`axes`	A logical value indicating whether axes should be drawn.	`TRUE`
`frame.plot`	A logical value indicating whether a box should be drawn around the plot.	`axes`
`asp`	The desired aspect ratio (y/x) of the plot.	`NA`
`...`	Additional graphical parameters.	

Executing Code Example using `plot()`

The source provides an example comparing vehicle speed and stopping distance using the built-in `cars` dataset:

```
plot(cars, xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
    las = 1, xlim = c(0, 25))
```



Output Scatter Plot

This code generates a scatter plot with "Speed (mph)" on the x-axis and "Stopping distance (ft)" on the y-axis. `las = 1` makes the axis labels horizontal, and `xlim = c(0, 25)` sets the limits of the x-axis. The resulting plot (Figure 3-4) shows a roughly proportional relationship between speed and stopping distance.

Scatter Plots with the `xyplot()` Function (Lattice Graphics)

The `lattice` package provides the `xyplot()` function for creating scatter plots, particularly when you want to examine relationships conditioned on or grouped by other variables.

The basic syntax is:

```
xyplot(y ~ x | conditioning_variable, groups = grouping_variable, data = data_frame, ...)
```

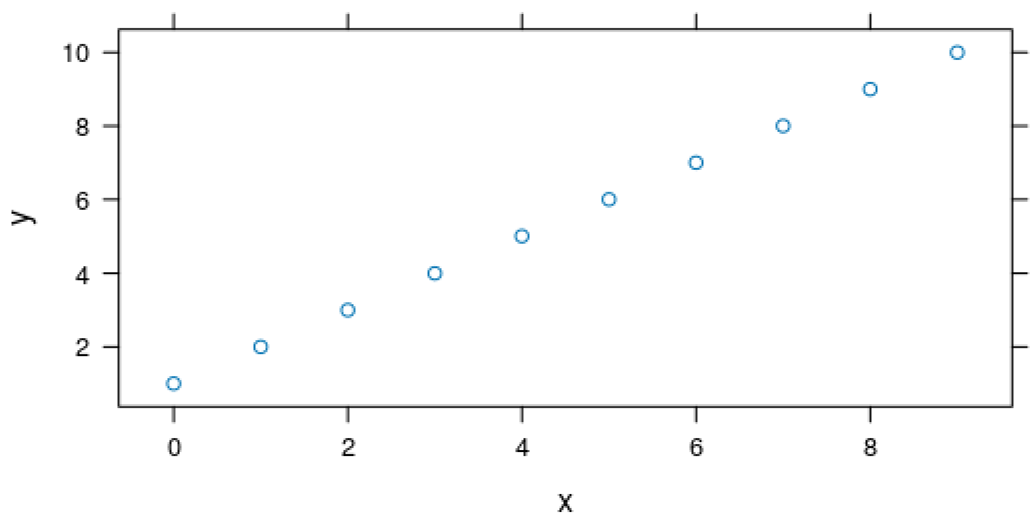
- `y ~ x`: This is a formula specifying the variable `y` to be plotted on the y-axis as a function of the variable `x` on the x-axis.
- `| conditioning_variable`: This is optional and specifies a variable to condition on, creating separate panels in the plot for each level of the conditioning variable.

- `groups = grouping_variable`: This is also optional and specifies a variable to group the data points by, which are typically displayed with different colors or symbols within the same panel.
- `data = data_frame`: Specifies the data frame containing the variables.
- `...`: Additional arguments for customization.

Executing Code Example using `xyplot()`

The source demonstrates `xyplot()` with a created data frame `d`:

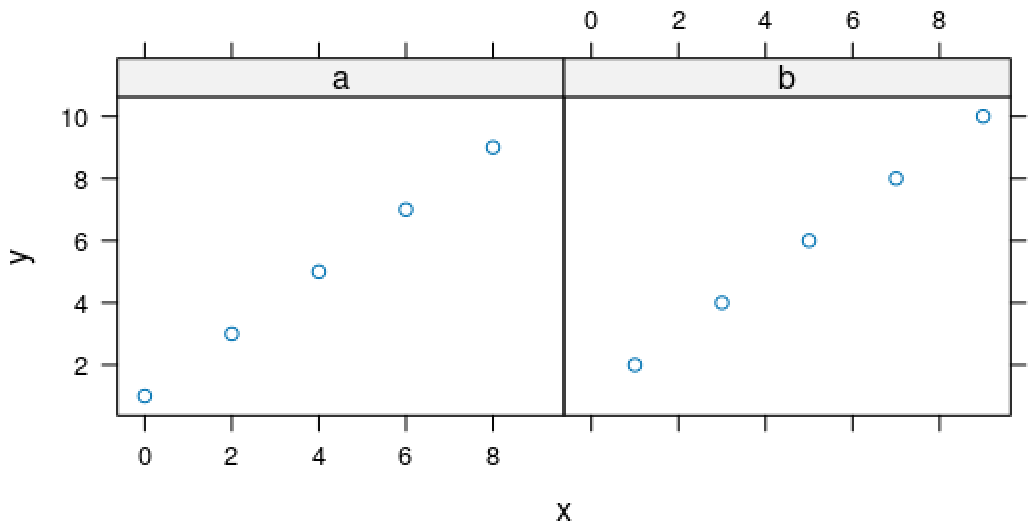
```
library(lattice)

d <- data.frame(x=c(0:9), y=c(1:10), z=c(rep(c("a", "b"), times=5)))

xyplot(y ~ x, data=d)
```



This creates a simple scatter plot of `y` against `x`.

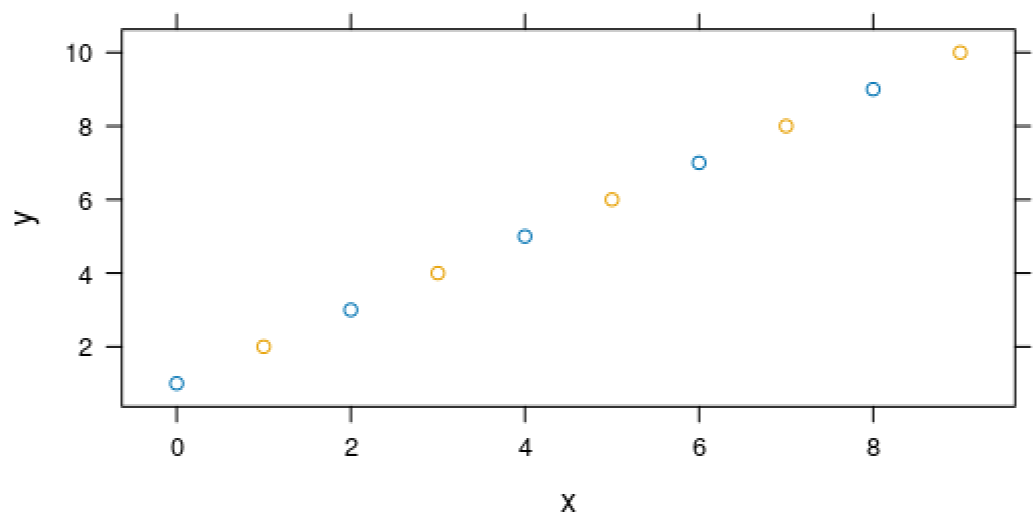
The following example adds a conditioning variable `z` to create separate panels:

```
xyplot(y ~ x | z, data=d)
```



And this example uses a grouping variable `z` to show different symbols for each group in a single panel:

xyplot(y ~ x, groups=z, data=d)



Scatter Plots with the `qplot()` Function (ggplot2)

The `ggplot2` package offers a powerful and flexible system for creating graphics based on the grammar of graphics . The `qplot()` function provides a quick way to generate plots, including scatter plots.

The basic syntax for a scatter plot using `qplot()` is:

library(ggplot2)

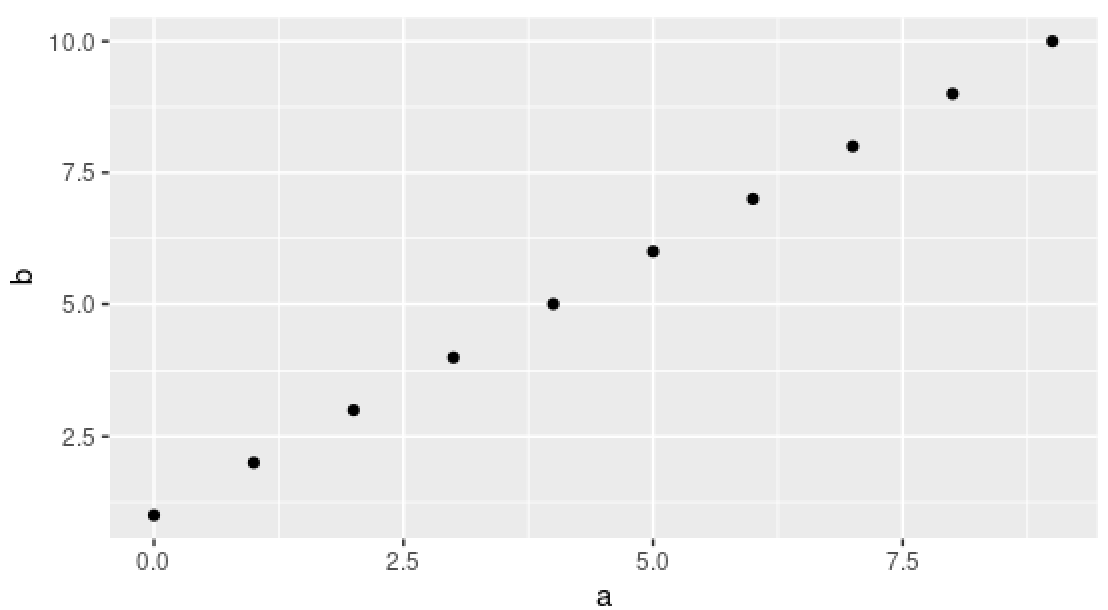
qplot(x = variable1, y = variable2, data = data_frame, ...)

- `x = variable1`: Specifies the variable for the x-axis.
- `y = variable2`: Specifies the variable for the y-axis.
- `data = data_frame`: Specifies the data frame.
- `...`: Additional arguments to customize the plot.

Executing Code Example using `qplot()`

The source provides a simple example using a created data frame `d` (with slightly different names in this context):

```
library(ggplot2)

d_ggplot <- data.frame(a=c(0:9), b=c(1:10), c=c(rep(c("a", "b"), times=5)))

qplot(x=a, y=b, data=d_ggplot)
```



This will create a basic scatter plot of `b` against `a`.

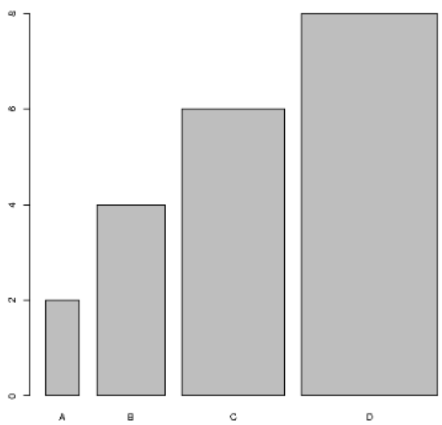
Other Useful Functions for Scatter Plot Related Visualizations

- **`matplot()`**: To plot multiple sets of columns of matrices against each other.
- **`smoothScatter()`**: For visualizing the density of a large number of points in a scatter plot using color shading.

- `pairs()`: To generate a matrix of scatter plots for all pairs of variables in a data frame.

These functions offer different ways to visualize the relationship between two or more numeric variables using scatter plot concepts. Remember that the choice of function depends on the specific requirements of your data exploration and visualization goals.

Extra : Bar and Column Chart [A detailed explanation]



Overview

Bar and column charts are fundamental ways to visualize categorical data or the magnitude of quantitative data across different categories. R provides several packages for creating these charts, including `graphics` (base R), `lattice`, and `ggplot2`.

1. Basic Bar and Column Charts with the `barplot()` Function (graphics package)

The `barplot()` function in the `graphics` package is used to create bar plots (vertical or horizontal columns).

Basic Syntax:

barplot(height, ...)

where `height` is a vector or matrix of values to be plotted.

Common Arguments:

Argument	Description	Default
`height`	A numeric vector or matrix representing the bar heights. If a matrix and `beside = FALSE`, bars are stacked. If `beside = TRUE`, bars are juxtaposed.	
`width`	A numeric vector specifying the widths of the bars.	`1`
`space`	Space between bars. If `beside = FALSE`, a fraction of the average column width. If `beside = TRUE`, a two-element vector for space within and between groups.	If matrix & `beside=TRUE`: `c(0, 1)` else `0.2`
`names.arg`	A character vector of names to be plotted below or beside each bar (or group of bars).	If matrix: `colnames(height)` else `names(height)`
`legend.text`	A character vector for legend labels or a logical value to use row names of `height` for the legend (useful when `height` is a matrix).	`NULL`
`beside`	A logical value indicating if bars should be stacked (`FALSE`) or drawn beside each other (`TRUE`) when `height` is a matrix.	`FALSE`
`horiz`	A logical value to draw bars horizontally (`TRUE`) or vertically (`FALSE`).	`FALSE`

`col`	A vector of colors for the bars.	Gray if `height` is a vector; gamma-corrected gray palette if a matrix
`border`	The color for the border of the bars.	`par("fg")`
`main`	The overall title for the plot.	`NULL`
`sub`	The subtitle for the plot.	`NULL`
`xlab`	The label for the x-axis.	`NULL`
`ylab`	The label for the y-axis.	`NULL`
`xlim`	Limits for the x-axis.	`NULL`
`ylim`	Limits for the y-axis.	`NULL`
`axes`	A logical value indicating whether axes should be drawn.	`TRUE`
`axisnames`	A logical value to draw and label the second axis if `names.arg` is not `NULL`.	`TRUE`
`cex.axis`	Numeric value for the size of numeric axis labels relative to other text.	`par("cex.axis")`

`cex.names`	Numeric value for the size of axis names (from `names.arg`) relative to other text.	`par("cex.axis")`
`add`	A logical value to add bars to an existing plot (`TRUE`) or create a new one (`FALSE`).	`FALSE`
`args.legend`	A list of arguments to pass to the `legend()` function.	`NULL`
`...`	Additional graphical parameters to pass to `par()`.	

Executing Code Example using `barplot()`:

The source provides an example showing doctorates by field and gender using `barplot()`:

Assuming 'doctorates.m' is a matrix with years as columns and genders as rows

barplot(doctorates.m, beside=TRUE, horiz=TRUE, legend=TRUE, cex.names=.75)

This code would create a horizontal, juxtaposed bar plot with a legend, where bar names are scaled down.

2. Bar Charts with the `barchart()` Function (lattice package)

The `lattice` package provides the `barchart()` function for creating bar and column charts, often with conditioning or grouping.

Basic Syntax:

barchart(x, data, ...)

where `x` can be a formula or a table, and `data` is the data frame (if using a formula).

Common Arguments:

Argument	Description	Default
`x`	A formula (e.g., `y ~ category`) or an object of class `table`, `array`, `matrix`, or a numeric vector.	
`data`	A data frame in which the formula `x` is evaluated.	
`panel`	The panel function used to draw the bars (`panel.barchart` by default).	`lattice.getOption("panel.barchart")`
`box.ratio`	Numeric value specifying the ratio of the width of the rectangles to the inner rectangle space.	`2`
`horizontal`	A logical value to draw bars horizontally (`TRUE`) or vertically (`FALSE`).	`TRUE` for table method, otherwise depends
`groups`	A variable or expression describing groups of data to be displayed with different colors or symbols. Setting `groups=FALSE` can lead to separate panels.	`TRUE` for table method
`stack`	A logical value indicating whether bars should be stacked (`TRUE`) or	`TRUE` for table method

	not (`FALSE`) when there are groups (for the table method).	
`origin`	The base value from which bars originate (for the table method).	`0`
`auto.key`	A logical value or a list to automatically generate a legend for groups.	`FALSE`
`xlab`	A character value for the label of the x-axis.	
`ylab`	A character value for the label of the y-axis.	
`scales`	A list that specifies how the x- and y-axes should be drawn. Allows customization of tick marks, labels, etc.	Default scales are used
`layout`	A numeric vector of length 2 specifying the number of columns and rows in the layout of multiple panels (if conditioning is used).	Panels are arranged to fill available space
`subscripts`	A logical value indicating whether a vector named `subscripts` should be passed to the panel function.	

`subset`	Specifies a subset of values from `data` to plot using a logical vector or an expression.	All values are used
`xlim`	Specifies the minimum and maximum values for the x-axis.	
`ylim`	Specifies the minimum and maximum values for the y-axis.	
`drop.unused.levels`	A logical value to drop unused levels of factors.	Depends on the function
`main`	A character value or expression for the main title.	
`sub`	A character value or expression for the subtitle.	
`par.strip.text`	A list of parameters to control the strip text (for conditioned plots).	Default parameters are used
`...`	Additional arguments passed to the panel function or other lattice functions.	

Executing Code Example using `barchart()`:

The source provides several examples using the `births2006.smpl` dataset:

library(lattice)

```
data(births2006.smpl) # Assuming this dataset is loaded

births.dow <- table(births2006.smpl$DOB_WK)

barchart(births.dow)
```

This creates a horizontal bar chart of the count of births for each day of the week.

Another example shows births by day of week and delivery method:

```
dob.dm.tbl <- table(births2006.smpl$DOB_WK, births2006.smpl$DELIVERY)

barchart(dob.dm.tbl, stack=FALSE, auto.key=TRUE) # Unstacked bars with a legend


barchart(dob.dm.tbl, horizontal=FALSE, groups=FALSE) # Separate panels for each delivery method
```

3. Bar Charts with `geom_bar()` (ggplot2 package)

The `ggplot2` package uses the grammar of graphics to create plots. Bar charts are created using the `geom_bar()` geometric object.

Basic Syntax using `qplot()` (quick plot):

```
library(ggplot2)

qplot(x = category_variable, data = data_frame, geom = "bar", ...)
```

Common Arguments in `qplot()` for Bar Charts:

Argument	Description
`x`	The categorical variable to be displayed on the x-axis.

`data`	The data frame containing the data.
`geom`	Specifies the geometric object to use, which is `"bar"` for bar charts.
`weight`	Specifies a variable whose values determine the height of the bars (useful when data is already aggregated).
`fill`	A variable whose levels determine the fill color of the bars (for grouped or stacked bars).
`facets`	Specifies how to split the data into multiple panels (e.g., `~ grouping_variable` for rows, `grouping_variable ~ .` for columns).
`position`	Specifies the position adjustment for multiple bars within the same x category (e.g., `"dodge"` for side-by-side).
`xlab`	Label for the x-axis.
`ylab`	Label for the y-axis.
`xlim`	Limits for the x-axis.
`ylim`	Limits for the y-axis.
`fill`	Color to fill the bars.

`...`	Additional arguments to customize the plot.

Executing Code Example using `qplot()`:

The source provides an example using Medicare data:

```
library(ggplot2)

data(outcome.of.care.measures.national) # Assuming this dataset is loaded

bar.chart.example <- qplot(x=Condition,

                  data=outcome.of.care.measures.national,

                  geom="bar", weight=Rate, facets=Measure~., fill=Measure)

print(bar.chart.example)


qplot(x=Condition, data=outcome.of.care.measures.national,

    geom="bar", weight=Rate, fill=Measure, position="dodge")
```

The first `qplot` creates a faceted bar chart showing rates by condition for different measures. The second `qplot` creates a dodged bar chart comparing rates for different measures within each condition.

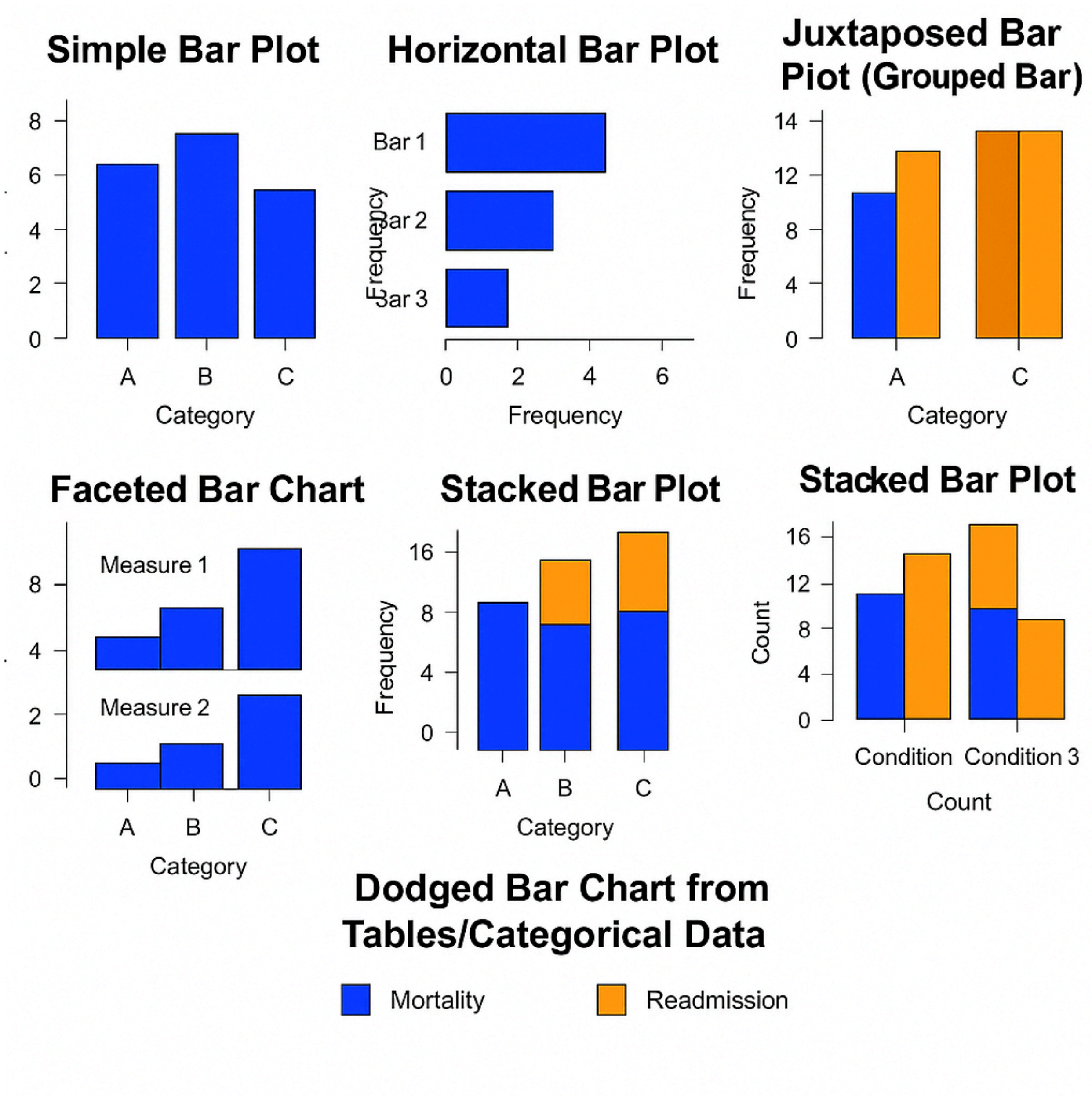
Different types of bar charts or bar plots serve various purposes in data visualization.

Type of Bar Chart/Plot	Description	Context/Use Case

Simple Bar Plot	Displays the magnitude of one or more categorical variables using the height (or length) of rectangular bars. By default in the `barplot` function, it shows the y-axis with the size of each bar and uses column names for the x-axis labels.	To show the absolute values or frequencies of different categories.
Horizontal Bar Plot	Bars are oriented horizontally instead of vertically. The `barplot` function can create these using the `horiz=TRUE` argument. The `barchart` function in the lattice package also defaults to horizontal bars.	Useful when category labels are long, as they are more readable horizontally. Also can be a matter of preference or convention depending on the data and audience.
Juxtaposed Bar Plot (Grouped Bar Chart)	For a matrix input in the `barplot` function with `beside=TRUE`, bars for different groups within each category are placed next to each other. This allows for direct comparison of subgroups within each main category. The lattice `barchart` can also achieve this by setting `stack=FALSE` and potentially using `groups`.	To compare the values of different subgroups for each category side-by-side.
Stacked Bar Plot	For a matrix input in the `barplot` function with the default `beside=FALSE` (or not specified), bars representing different groups within each category are stacked on top of each other. The total height of the bar represents the sum of the values for that category. The lattice	To show the contribution of different subgroups to the total value of each category.

	`barchart` function defaults to stacked bars when given a table.	
Faceted Bar Chart	Using `ggplot2`, bar charts can be split into multiple panels (facets) based on the levels of one or more additional categorical variables. This is achieved using the `facets` argument in the `qplot` or `ggplot` functions.	To examine the relationship between variables across different conditions or subgroups displayed in separate panels, making it easier to see patterns within each condition.
Dodged Bar Chart	In `ggplot2`, when plotting bars for different groups within each category, setting `position="dodge"` in the `qplot` or `geom_bar` functions will place these bars adjacent to each other instead of stacking them.	To compare the values of different groups directly within each category in a single panel, similar to a juxtaposed bar chart in base R.
Bar Chart from Tables/Categorical Data	The `barplot` function in base R works with numeric vectors or matrices. For categorical data, you would typically use the `table` function to get frequencies first and then use `barplot`. The lattice `barchart` function has a method that directly accepts objects of class `table` to visualize counts of categorical variables.	To visualize the frequency distribution of categorical data.

It's important to choose the type of bar chart that best highlights the patterns and comparisons relevant to your data and the message you want to convey. For instance, stacked bars are good for showing part-to-whole relationships, while juxtaposed or dodged bars are better for comparing the magnitudes of different groups within each category. Faceting allows for the examination of these relationships across various conditions.



The above image is Chat Gpt generated

Note : Run your own code (similar to textbook) since many of the data sets mentioned in this book need to be separately loaded.

Lattice Graphics - Lattice functions

Lattice graphics is an R package that provides a system for data visualization, inspired by **Trellis graphics** developed at Bell Labs. The **lattice package** is an implementation of Trellis graphics in R. It is built upon the **grid graphics engine** and is not readily compatible with traditional base R graphics. However, it offers powerful tools for creating informative and visually appealing charts, particularly for comparing different groups of data.

Key Characteristics of Lattice Graphics:

- **Focus on Conditioning and Grouping:** A primary strength of lattice graphics is its ability to split a chart into different **panels** arranged in a grid based on the values of one or more **conditioning variables** specified in a formula. It also allows for the display of different groups within the same panel using different colors or symbols based on a **grouping variable**.
- **Lattice Objects:** When you call a high-level lattice plotting function, it doesn't directly produce a plot. Instead, it returns a **lattice object** of class "trellis". To actually display the graphic, you need to explicitly use a **print** or **plot** command on this lattice object. On the R console, this often happens automatically.
- **Panel Functions:** Lattice graphics work by calling one or more **panel functions** that are responsible for plotting the actual data within each panel. You can customize the appearance of plots by specifying arguments to the high-level plotting function or by using or creating substitute **custom panel functions** to add extra graphical elements.
- **Formula Interface:** Most lattice functions utilize a **formula interface** to specify the variables to be plotted and any conditioning or grouping variables. The formula typically takes the form `y ~ x | z` where `y` is the dependent variable, `x` is the independent variable, and `z` is the conditioning variable. A grouping variable can be added using the `groups` argument, for example, `y ~ x | z`, `groups = g`. The tilde symbol `~` is used to show the relationship between the response variable (on the left) and the stimulus variables (on the right). The vertical bar `|` is used to specify conditioning variables. The plus sign `+` can be used to express a linear relationship between variables in the formula.
- **Consistency in Arguments:** Arguments within the **lattice package** are generally more consistent across different plotting functions compared to base R graphics. Many common arguments have similar effects in multiple lattice functions.

Common Lattice Functions and Plot Types:

The **lattice package** includes many high-level plotting functions that are often equivalent to similar functions in the base `graphics` package. Here are some key lattice functions and the types of plots they create:

- **`barchart()`**: For drawing **bar charts** and column charts. It can accept formulas and data frames, as well as objects of the class `table`. The default orientation is horizontal bars. You can create juxtaposed (unstacked) or stacked bar charts using the `groups` and `stack` arguments.

- `dotplot()`: For creating **Cleveland dot plots**, useful for showing data where there is a single point for each category. Similar to `barchart()`, it accepts formulas and data frames, as well as `table` objects.
- `histogram()`: For plotting **histograms** to visualize the distribution of a single numeric variable.
- `densityplot()`: For displaying **kernel density plots** to estimate the probability density function of a continuous variable.
- `stripplot()`: For producing **one-dimensional scatter plots** or dot plots, often used as an alternative to box plots, especially with small sample sizes.
- `qqmath()`: For generating **quantile-quantile (Q-Q) plots** to compare the distribution of a sample to a theoretical distribution (by default, a normal distribution).
- `xyplot()`: For creating **conditional scatter plots** to examine the relationship between two numeric variables, potentially conditioned on other variables and with grouping.
- `qq()`: For generating **quantile-quantile plots** to compare two distributions directly.
- `splom()`: For producing **scatter plot matrices** to visualize the relationships between multiple pairs of variables in a matrix or data frame.
- `levelplot()`: For plotting **three-dimensional data** on a flat grid using colors to represent the third dimension (similar to filled contour plots or heatmaps).
- `contourplot()`: For displaying **contour plots** to visualize three-dimensional data as lines representing constant values (like topographic maps).
- `cloud()` and `wireframe()`: For creating **perspective charts of three-dimensional data** (3D scatter plots and surfaces).
- `bwplot()`: For producing **box plots** (box-and-whisker plots) to summarize the distribution of a numeric variable across different groups.

Customizing Lattice Graphics:

The **lattice package** offers extensive options for customization. You can customize various aspects of the plots using arguments to the high-level functions, through **graphical parameters** specific to lattice, and by defining **custom panel functions**.

- **Common Arguments:** Lattice functions share many common arguments to control aspects like the formula, data, conditioning, grouping, axes labels (`xlab`, `ylab`), axis limits (`xlim`, `ylim`), scales, strips (panel labels), and more.
- **Scales Argument:** The `scales` argument (a list) allows fine control over how the x- and y-axes are drawn, including setting limits, the number of tick marks, whether to draw the axis, and even applying log transformations.
- **Lattice Options and Parameters:** You can use `lattice.getOption()` to check the current values of lattice settings and `lattice.par.set()` to modify these parameters, which control various visual elements like text size, colors, line styles, and more. The `show.settings()` function can graphically display all the current settings.

- **Strips:** The `strip` argument controls whether panel labels (strips) are drawn. You can further customize strips using functions like `strip.default` and `strip.custom` to control the appearance of variable names and levels.
- **Keys and Legends:** The `auto.key` argument can automatically draw a key (legend) for grouped data. You can also use `simpleKey()` and `draw.key()` for more control over the legend appearance.
- **Panel Functions:** As mentioned before, you can define your own **panel functions** to add specific graphical elements or modify the default plotting behavior within each panel. Lattice provides a set of **low-level graphics functions** (e.g., `llines`, `lpoints`, `ltext`) and **panel functions** (e.g., `panel.abline`, `panel.grid`) that can be used within custom panel functions.

In summary, **Lattice Graphics** in R, through its **lattice functions**, offers a powerful and structured approach to data visualization, especially when dealing with the need to explore relationships across different conditions or groups within your data. Its formula-based interface and consistent argument structure, combined with extensive customization options, make it a valuable tool for creating insightful graphics.

Extra : **Customizing lattice graphics** **[in detail]**

Customizing lattice graphics involves various methods to fine-tune the appearance of your Trellis plots. The `lattice` package provides several ways to achieve this, as detailed in the sources.

Common Arguments to Lattice Functions: Lattice functions share many common arguments that control different aspects of the plot. Some of these include:

- **`x`**: Specifies the object to be plotted, which can be a formula, array, numeric vector, or table. The tilde (~) notation in formulas is used to define the relationship between variables, with the response variable on the left and the stimulus variables on the right. A vertical bar (`|`) in the formula specifies conditioning variables for splitting the plot into panels.
- **`data`**: When `x` is a formula, this argument specifies the data frame to be used.
- **`panel`**: Specifies the panel function used to draw the plots in each panel.
- **`aspect`**: Controls the aspect ratio of the panels.
- **`groups`**: Specifies a variable for grouping data within panels, often displayed with different colors or symbols.
- **`xlab`**, **`ylab`**: Character values for the x-axis and y-axis labels.
- **`xlim`**, **`ylim`**: Specifies the minimum and maximum values for the x-axis and y-axis.
- **`scales`**: A list that controls how the axes are drawn.
- **`subscripts`**: A logical value indicating whether a subscripts vector should be passed to the panel function.

- **`subset`**: Specifies a subset of the data to plot. Note that `subset` does not remove unused levels from plotted factors.
- **`drop.unused.levels`**: A logical value to drop unused levels of factors.
- **`default.scales`**: A list giving the default value of scales.
- **`lattice.options`**: A list of plotting parameters similar to `par` in base R graphics.
- **`allow.multiple`**: Specifies how to interpret formulas with multiple response variables.

Controlling How Axes Are Drawn (`scales` argument): The `scales` argument, which is a list, provides detailed control over axis appearance. You can specify a single list for both axes or separate lists for `x` and `y`. Available arguments within the `scales` list include:

- **`relation`**: Determines how axis limits are calculated for each panel (`"same"`, `"free"`, or `"sliced"`).
- **`tick.number`**: Suggests the number of tick marks.
- **`draw`**: A logical value to draw the axis.
- **`alternating`**: Controls whether axis locations alternate between panels.
- **`limits`**: Specifies the limits for each axis.
- **`at`**: A numeric vector or list specifying where to plot tick marks.
- **`labels`**: Labels for the tick marks.
- **`cex`**, **`font`**, **`fontface`**, **`fontfamily`**: Control the appearance of axis labels.
- **`tck`**: Specifies the length of tick marks.
- **`col`**: Sets the color of tick marks and labels.
- **`rot`**: Specifies the angle to rotate axis labels.
- **`abbreviate`**: A logical value to abbreviate labels.
- **`log`**: Specifies whether to use a logarithmic scale for the axis.

Lattice Graphics Parameters (`trellis.par.get` and `trellis.par.set`): Lattice graphics have their own set of parameters, organized hierarchically as lists of lists, which control various visual elements.

- You can **check the value** of a parameter using `trellis.par.get("parameter.name")`. For example, `trellis.par.get("axis.text")` shows settings for axis text.
- You can **change a setting** using `trellis.par.set(list(parameter.group = list(parameter = value)))`. For instance, to make axis text smaller: `trellis.par.set(list(axis.text = list(cex = 0.5)))`.
- Calling `trellis.par.get()` with no arguments returns a **list of all settings**.
- The `show.settings()` function **displays all settings graphically**.

There are 34 high-level groups of parameters, including:

- `grid.pars`: Global parameters.

- `fontsize`: Base font size.
- `background`: Plot background color.
- `clip`: Clipping for panels and strips.
- `add.line`, `add.text`: Appearance of lines and text added by helper functions.
- `plot.polygon`, `plot.symbol`, `plot.line`: Default appearance of plotting elements.
- `superpose.line`, `superpose.symbol`, `superpose.polygon`: Appearance for grouped data.
- `strip.background`, `strip.shingle`, `strip.border`: Appearance of strips.
- `axis.line`, `axis.text`, `axis.components`: Appearance of axes.
- `layout.heights`, `layout.widths`: Control panel dimensions.
- `par.xlab.text`, `par.ylab.text`, `par.zlab.text`: Text labels.
- `par.main.text`, `par.sub.text`: Titles and subtitles.

Customizing Strips (`strip.default` and `strip.custom`): Strips are the labels that appear for each panel in a conditioned plot. You can customize them by:

- **Writing a custom strip function**: This usually involves creating a wrapper around `strip.default`. The arguments to `strip.default` provide the data for drawing the strip, such as variable names and factor levels.
- **Using the `strip.custom` function**: This is a simpler way to modify strips. It accepts the same arguments as `strip.default` and returns a new function that can be passed to the `strip` argument of a lattice function. You can modify aspects like `horizontal` (label orientation), `bg` (background color), `fg` (foreground color), and `par.strip.text` (text appearance parameters).

Customizing Keys (Legends) (`auto.key` and `simpleKey`): For plots with multiple groups, you can customize the legend (key):

- **The `auto.key` argument**: Setting `auto.key=TRUE` automatically draws a key using `simpleKey` with default arguments. You can also pass a list of arguments to `auto.key` that will be passed to `simpleKey`.
- **The `simpleKey` function**: This function generates a list suitable for drawing a key. You can specify arguments like `text` (legend labels), `points`, `rectangles`, `lines` (whether to show these elements), and graphical parameters like `col`, `cex`, `alpha`, `font`, etc..

Custom Panel Functions: For more advanced customization, you can define your own **panel functions**. Lattice functions use panel functions to do the actual plotting within each panel.

- You specify a built-in or custom panel function to the `panel` argument of a high-level lattice function.

- You can create your own panel function using **low-level graphics functions** provided by the `lattice` package, such as `llines`, `lpoints`, `ltext`, `lsegments`, `lpolygon`, `larrows`, and `lrect`. You can also use existing panel functions like `panel.abline` or `panel.grid` within your custom function.
- Custom panel functions typically receive the x and y data (and other relevant data) for the panel as arguments.
- For grouped data within panels, you might use `panel.superpose` to apply a panel function for each group.

`plot.trellis` Function: Lattice functions return a "trellis" object, and the actual plotting happens when you `print` or `plot` this object. `plot.trellis` (or `print.trellis`) is the function that sets up the panels and calls the panel functions. You can pass arguments to `plot.trellis` through the `plot.args` argument of high-level lattice functions. Arguments to `plot.trellis` include controlling the position and layout of the plot, specifying a `packet.panel` function, and setting `panel.height` and `panel.width`.

By combining these methods, you can achieve a high degree of control over the appearance of lattice graphics.

Ggplot

`ggplot2` is described as one of the **most popular R packages for creating graphics**. It is highlighted as a tool for producing **readable charts** quickly and easily creating **stunning charts**. Unlike the base `graphics` package, `ggplot2` utilizes a **different metaphor for graphics**, based on the **grammar of graphics**.

Here's a more detailed breakdown of `ggplot2` based on the information provided:

- **Underlying Principle: The Grammar of Graphics**
    - `ggplot2` is built upon the concept of the grammar of graphics, which involves describing the components of a chart rather than just selecting a chart type.
    - This approach allows for more flexibility and a deeper understanding of the charting process.
    - The key components of a chart in the grammar of graphics, as implemented in `ggplot2`, are:
        - **Data**: The dataset being visualized.
        - **Mappings (aes)**: How variables in the data are mapped to visual attributes (aesthetics) of the plot, such as x-position, y-position, color, size, and shape.

        - **Geometric Objects (geom)**: The visual elements used to represent the data, such as points (`geom_point`), bars (`geom_bar`), lines (`geom_line`), etc.. Different geoms are used for different types of plots.
        - **Aesthetic Properties (aes)**: These determine the look of the plot, including sizes, labels, and tick marks.
        - **Scales**: Control how the mappings from data values to aesthetic values are performed (e.g., the range of colors or the breaks on an axis).
        - **Facets**: Describe how the data is divided into subsets and displayed in multiple panels. The `facet_wrap` function can be used for this.
        - **Positional adjustments**: Provide fine-grained control over the placement of geometric objects, especially when they might overlap (e.g., dodging bars).
- **Quick Plot Function (`qplot`)**

    - `qplot` is a convenient function in `ggplot2` for quickly creating plots with a simple syntax.
    - It allows you to specify the x and y variables, the data frame, and other aesthetic attributes as arguments.
    - `qplot` can handle one-dimensional data (e.g., creating histograms or density plots by default) as well as two-dimensional data (e.g., scatter plots).
    - You can specify the type of geometric object to use with the `geom` argument (e.g., `geom="bar"`, `geom="density"`).
    - Other arguments in `qplot` include `data`, `facets`, `margins`, `stat` (statistical transformations), `position`, `xlim`, `ylim`, `log` (for logarithmic scales), `main` (title), `xlab` (x-axis label), `ylab` (y-axis label), and `asp` (aspect ratio).
- **Creating Graphics with `ggplot()` and Layers**

    - For more flexibility and control, you can create `ggplot2` objects using the `ggplot()` function, specifying the data and aesthetic mappings.
    - You then add layers to the `ggplot` object using the `+` operator.
    - Layers typically consist of **geometric objects (`geom_`)** and **statistical transformations (`stat_`)**.
    - Examples of geometric functions include `geom_point`, `geom_line`, `geom_bar`, `geom_histogram`, `geom_density`, `geom_boxplot`, and many others.
    - Statistical transformation functions (`stat_`) perform calculations on the data before it is plotted (e.g., `stat_density` calculates density estimates for plotting). Some convenience functions combine a statistical transformation with a geom (e.g., `geom_smooth` adds a smoothed conditional mean).
    - You can also use the `layer()` function to specify geometric objects by their short names (e.g., `"point"`).
- **Customization**

- `ggplot2` allows for extensive customization of plots by modifying aesthetic properties, scales, and other components.
- Scales can control the appearance of axes, legends, colors, and other visual elements.
- Aesthetic properties within `aes()` can be set to specific values (using `I()`) or mapped to variables in the data.
- Faceting allows you to create multiple small plots based on the levels of one or more categorical variables, making it easier to visualize patterns within subgroups.

- **Examples**
  - The sources provide examples of using `qplot` to create scatter plots, bar charts, and density plots.
  - A more complex example using Medicare data demonstrates how to create a scatter plot of average payment versus the number of cases for different diagnoses, and how to improve its legibility by using a log scale, semi-transparent points (`alpha`), and adding a smoothing line (`geom_smooth`).

In summary, `ggplot2` offers a powerful and flexible system for creating graphics in R based on the grammar of graphics. It provides both a quick plotting interface (`qplot`) and a more structured approach using `ggplot()` and layered components, allowing for detailed customization and the creation of a wide variety of visually appealing and informative charts.

Extra : **Some Other Basic Plots/Charts in R** **[Images and codes are chatgpt generated]**

---

1. Histogram

Purpose: Show distribution of a single numeric variable by bins.

hist(rnorm(100), main="Histogram")

---

2. Box Plot

Purpose: Summarize distribution using median, quartiles, and outliers.

boxplot(rnorm(100), main="Box Plot")

---

3. Dot Plot

Purpose: Show individual data points; best for small datasets.

```
stripchart(rnorm(30), method="stack", main="Dot Plot")
```

---

4. Q-Q Plot (Quantile-Quantile Plot)

Purpose: Compare sample distribution to a theoretical distribution (e.g., normal).

```
qqnorm(rnorm(100)); qqline(rnorm(100), col = "red")
```

---

5. Contour Plot

Purpose: Show 3D data in 2D using contour lines.

```
x <- y <- seq(-pi, pi, length=50)

z <- outer(x, y, function(x, y) cos(x)*sin(y))

contour(x, y, z, main="Contour Plot")
```

---

6. Perspective Plot (3D Surface)

Purpose: 3D representation of z values over x and y.

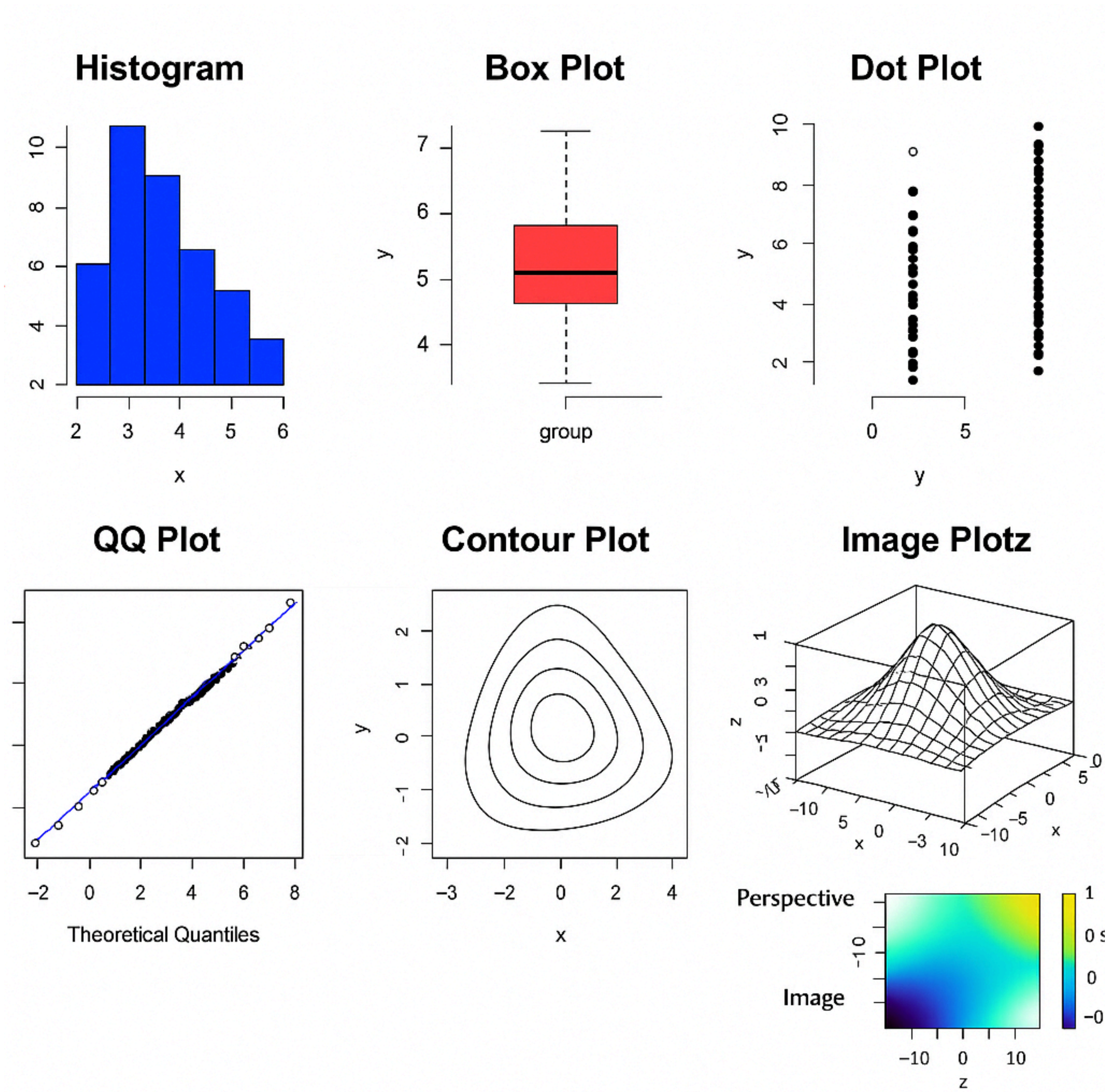
```
persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue", main="Perspective Plot")
```

---

7. Image Plot

Purpose: Heatmap-like plot showing pixel-level values.

```
image(x, y, z, col=terrain.colors(20), main="Image Plot")
```



Summary of Visuals

Plot Type	Description
Histogram	Distribution of numeric data across intervals.

Box Plot	Median, quartiles, and outliers.
Dot Plot	Dots representing individual values.
QQ Plot	Tests normality by comparing quantiles.
Contour Plot	3D surface shown as 2D lines.
Perspective Plot	3D mesh surface for Z ~ X,Y.
Image Plot	Heatmap-style color mapping of values.

Note : histograms are about understanding the distribution of a continuous variable, showing how many data points fall within certain ranges. Bar charts are about comparing values across different categories, where each bar represents a separate group and its associated magnitude.
