STAT 7650 - Computational Statistics Lecture Slides

Introduction

Elvan Ceyhan

Updated: January, 2025

AU

Acknowledgement:

Some of the content in this course is inspired by the lecture notes of Prof Ryan Martin of NCSU.

Outline

Review of Statistical Inference

Introduction to R

- Basics
- R Session
- R Graphics
- Programming in R
- Data Entry

Math & Stat Tools

- Probability Stuff
- Statistical Methods
- Linear Algebra - Matrix Operations in R

Introduction

- Based on parts of: Dalgaard's ISwR book, Chapter 1 in Givens & Hoeting (Computational Statistics), and Chapter 7 of Lange (Numerical Analysis for Statisticians).
- For more information: See the Course website under Canvas

What to compute?

- STAT 7650 is a course about computational statistics.
- It is important first to discuss what we want to compute in statistical problems.
- Concerned primarily with two problems:
  1. Maximizing the likelihood function.
  2. Integrating a "posterior distribution" (this includes sampling from the posterior too).
- The former should be familiar from Stat 7600/7610 (maximum likelihood or MLE).
- The latter, related to Bayesian statistics, may be new (unless you've taken STAT 7630).
- **Next:** A brief introduction to these concepts with an illustration.

Maximum Likelihood

- Consider n independent observations $y_1, \ldots, y_n$, from a density/mass function $p_{\boldsymbol{\theta}}$ depending on an unknown parameter (vector) $\boldsymbol{\theta}$.
- The likelihood and log-likelihood functions are defined as:

$$L(\boldsymbol{\theta}) = \prod_{i=1}^{n} p_{\boldsymbol{\theta}}(y_i)$$

$$\ell(\boldsymbol{\theta}) = \sum_{i=1}^{n} \log p_{\boldsymbol{\theta}}(y_i)$$

- The Maximum Likelihood Estimator (MLE) $\widehat{\boldsymbol{\theta}}$ of $\boldsymbol{\theta}$, which maximizes the likelihood, is given by:

$$\widehat{\boldsymbol{\theta}} = \operatorname*{argmax}_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) \quad \text{or} \quad \ell'(\widehat{\boldsymbol{\theta}}) = 0$$

- This requires optimization and/or finding roots of functions.

Maximum Likelihood (cont'd)

- Besides estimating the unknown parameter, it's important to assess its uncertainty.
- Under certain conditions, for large sample sizes (n), the distribution of $\widehat{\boldsymbol{\theta}}$ is approximately normal with mean $\boldsymbol{\theta}$ and (co)variance $I(\boldsymbol{\theta})^{-1}$, where $I(\boldsymbol{\theta})$ is the *Fisher information matrix*:

$$I(\boldsymbol{\theta}) = \mathbf{E}_{\boldsymbol{\theta}}\left[\ell'(\boldsymbol{\theta})\ell'(\boldsymbol{\theta})^\top\right] = -\mathbf{E}_{\boldsymbol{\theta}}\left[\ell''(\boldsymbol{\theta})\right]$$

- An approximate 95% confidence interval for θ_j is:

$$\widehat{\theta}_j \pm 1.96 \times \sqrt{[I(\widehat{\boldsymbol{\theta}})^{-1}]_{jj}}, \quad j = 1, \ldots, d$$

- This highlights the importance of computing derivatives and inverting matrices.

Bayesian Approach

- The Bayesian approach utilizes the rules of probability for inference.
- It begins with a prior distribution for $\boldsymbol{\theta}$, denoted as $\pi(\boldsymbol{\theta})$ - essentially a weight function.
- This leads to a conditional distribution for $\boldsymbol{\theta}$, given $\mathbf{Y} = \mathbf{y}$, called the 'posterior' distribution, expressed as:

$$\pi(\boldsymbol{\theta}|\mathbf{y}) = \frac{\pi(\boldsymbol{\theta})L(\boldsymbol{\theta})}{\int \pi(\boldsymbol{\theta})L(\boldsymbol{\theta})\, d\theta} \propto \pi(\boldsymbol{\theta})L(\boldsymbol{\theta})$$

- The focus is on $\pi(\boldsymbol{\theta}|\mathbf{y})$, with the aim to derive or estimate summaries like mean, variance, quantiles, probabilities, etc.
- Therefore, integrating functions or sampling from distributions is a crucial aspect of this approach.

Example: Probit Regression

- Observations $Y_1, \ldots, Y_n$ are independent (not iid) binary observations.
- Specifically, $Y_i \sim \text{Ber}(\Phi(\mathbf{x}_i^\top \boldsymbol{\theta}))$, for $i = 1, \ldots, n$, where:
  - "Ber" denotes a Bernoulli distribution.
  - $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are fixed d-dimensional covariates.
  - $\boldsymbol{\theta}$ is a d-dimensional parameter vector.
  - Φ is the standard normal distribution function[1].
- **Exercise:**
  1. Write out the log-likelihood function.
  2. Find MLE of $\boldsymbol{\theta}$.
  3. Calculate the Fisher information matrix, $I(\boldsymbol{\theta})$.

[1] **Note:** Other cdfs can be used, but then the model isn't referred to as "probit".

Remarks

- The course focuses on solving optimization and integration problems in statistical applications.
- A background in general numerical methods will be necessary.
- Software plays a crucial role; `R` will be used extensively.
- The course content ranges from simple to more challenging topics.
- The primary goal is to equip students with the knowledge to read current papers using computational statistics and implement their methods.

Outline

Review of Statistical Inference

Introduction to R

- Basics
- R Session
- R Graphics
- Programming in R
- Data Entry

Math & Stat Tools

- Probability Stuff
- Statistical Methods
- Linear Algebra - Matrix Operations in R

Outline

Review of Statistical Inference

Introduction to R

- Basics
- R Session
- R Graphics
- Programming in R
- Data Entry

Math & Stat Tools

- Probability Stuff
- Statistical Methods
- Linear Algebra - Matrix Operations in R

Main Features of R

- R is a free and open-source software platform, originally derived from S, with S-PLUS (a commercial version) now discontinued.
- Available for download across major operating systems (Windows, macOS, Linux) from `https://cran.r-project.org`.
- Provides an interactive environment for computation, while supporting script-based workflows for batch execution and reproducibility.
- Seamlessly integrates with lower-level programming languages such as C, C++, and FORTRAN to optimize computational performance.
- Renowned for its *versatility*, R enables users to define custom functions, extend existing packages, and adapt functionality to specific requirements.

Arithmetic in R

- R can perform arithmetic operations similar to a calculator.
- Basic binary (arithmetic) operations include:
  - $+$: Addition
  - $-$: Subtraction
  - $*$: Multiplication
  - $/$: Division
  - $\hat{}$ or $**$: Exponentiation
  - %/% : Integer division
  - %% : Modulus (remainder)

Variables and Assignments in R

- R enables storage of intermediate values by assigning them to variables.
- Assignment syntax:
  - `x <- 7`: Assigns the value 7 to the variable `x`.
  - The assignment operator <- is recommended and should be viewed as a single "left arrow" symbol.
  - The equal sign (=) can also be used for assignment but is less commonly recommended for clarity.
- Rules for variable names:
  - Variable names may include letters, numbers, underscores (_), or periods (.).
  - Examples: `pred.value`, `pred_value`.
  - Variable names cannot:
    - Start with a number or special character (other than a dot).
    - Contain spaces.

Expressions and Objects in R

- In R, user-entered expressions are evaluated by the system to produce output.
- These expressions are versatile, capable of generating formulas, graphs, datasets, and more.
- Expressions operate on *objects*, which are entities that can be assigned to variables.
- Syntax varies depending on the type of expression or object being used.
- This lecture will explore key types of expressions and objects in R.
- To examine the internal structure of an object X, use the command:

```
str(X)
```

Functions and Arguments in R

- Functions in R serve diverse purposes, including:
  - Mathematical operations, e.g., `log(x)`.
  - Non-mathematical tasks, e.g., `plot(x, y, pch=2)`.
- Functions are called with parentheses enclosing *arguments*.
- Types of arguments:
  - **Positional:** Recognized by their order in the argument list.
  - **Named:** Recognized by their name, making the order irrelevant.
- Functions may:
  - Have no arguments.
  - Include default arguments.
  - Accept arbitrary numbers of arguments.
- R includes a comprehensive set of *built-in functions* for various applications.
- Users can easily define custom functions using R's intuitive function syntax, which the same as ordinary R syntax.

Vectors

- Numeric vectors are fundamental in R and widely used.
- In addition to numeric vectors, two other key types of vectors are commonly utilized[2]:
  - **Character Vectors**: Composed of character strings.
    Example: `names <- c('Small', 'Medium', 'Large')`
  - **Logical Vectors**: Contain elements `TRUE` or `FALSE`, very useful for indexing data sets.
    Example:

```
gpa <- c(3.0, 2.8, 3.4, 3.7, 3.9, 3.3)
gpa > 3.5
[1] FALSE FALSE FALSE TRUE TRUE FALSE
```

[2] Complex vectors are also supported in R.

Vectors (Continued)

- Key functions to create vectors in `R`:
  - `c()`: Concatenates elements into a vector.
  - `seq()`: Generates a patterned sequence.
  - `rep()`: Repeats elements a specified number of times.
- Vectors in `R` must contain elements of the same data type.
- When combining variables of different types:
  - Elements are coerced into a common type, following a hierarchy: **Logical** $\rightarrow$ **Integer** $\rightarrow$ **Numeric (Double)** $\rightarrow$ **Complex** $\rightarrow$ **Character** $\rightarrow$ **Factor** $\rightarrow$ **List**[3].
  - Examples of coercion:
    - `c(FALSE, 7)` produces `[1] 0 7` (logical coerced to numeric).
    - `c(11.7, "abc")` produces `[1] "11.7" "abc"` (numeric coerced to character).

[3] The 'raw' type, representing raw bytes, is rarely used in standard data analysis.

Vectors (Continued)

- An interesting feature of `R` is that it supports **vectorized arithmetic**, applying operations element-wise across vectors:
  - **Example:**

```
x <- c(7, 10, 11)
y <- seq(5, 3, by = -1)
x + y  # Results in: [1] 12 14 14
```

- Handling vectors of different lengths:
  - The shorter vector is **recycled** to match the length of the longer vector.
  - An warning occurs if the length of the longer vector is not a multiple of the shorter vector.
- When defining custom functions, **be mindful of vectorization** and ensure assumptions align with your data structure.

Matrices and Arrays in R

- A matrix in R extends a vector by adding a double index (row and column).
  - **Example:** `M <- matrix(1:6, nrow = 3, ncol = 2)`
- Matrices are generally treated as vectors in R[4].
- Appending matrices:
  - Use `rbind()` to append rows.
  - Use `cbind()` to append columns.
- Assign row and column names using `rownames()` and `colnames()`.
- R also supports **arrays** (generalizations of vectors with n dimensions), though they are less common and harder to interpret visually.

[4] Linear algebra operations, such as matrix multiplication, are exceptions where matrices are treated differently.

Data Frames in R

- A **data frame** is R's representation of a data matrix or dataset:
  - Columns correspond to variables, and rows correspond to cases (observations).
- Unlike matrices, data frames can hold mixed data types, including numeric, character, and logical variables.
- Creating a data frame:
  - Syntax: `D <- data.frame(list_of_variables)`
- Reading external files into a data frame will be covered in later discussions.
- Data frames are central to R, with many statistical routines (e.g., linear regression) designed to operate directly on them.

Lists in R

- A **list** in R is a collection of objects, potentially of different types.
- Lists can include various object types such as vectors, matrices, functions, and even other lists.
- Syntax for creating a list:
  - `mylist <- list(list_of_objects)`
- **Example:**

```
M <- matrix(c(2, 5, 7, 7), nrow = 2)
f <- function(x) log(x) + x^2
mylist <- list(mymat = M, myfun = f)
mylist$myfun(mylist$mymat)
# Applies the function myfun to the matrix mymat.
```

Indexing in R

- **Indexing** is essential for accessing specific elements in vectors, matrices, arrays, data frames, and lists.
- Key indexing rules:
  - For a matrix M, `M[i, j]` retrieves the element at the i-th row and j-th column.
  - `M[, j]` returns the j-th column of M as a vector.
- Data frames are indexed similarly to matrices.
- Vectors are treated as one-dimensional matrices for indexing.
- List elements are accessed using the `$` operator, e.g., `mylist$mymat`.
- Examples:
  - `mylist$mymat[2, 2]`: Accesses the element in the second row and second column of `mymat`.
  - `mymat[-1, ]`: Excludes the first row of `mymat`, returning all remaining rows.

Subsetting in R

- Subsetting extends indexing by using logical variables to select specific parts of an object.
- **Example:** Using logical vectors for matrix subsetting:

```
row.log <- (1:nrow(M)) == 1
col.log <- (1:ncol(M)) == 2
M[row.log, col.log]  # Equivalent to M[1, 2]
```

- Logical variables allow more complex subsetting:
  - Example with a data frame:
    - Suppose data frame *D* has a variable age.
    - To select rows for adults (`age > 19`): `D[D$age > 19, ]`
- Generalizing row/column indexing:

```
x <- seq(5, 25, by = 5)
x[c(2, 3)]
[1] 10 15
```

Explicit Loops in R

- **Explicit loops** are used in R to perform repetitive tasks manually over elements of vectors, lists, or matrices.
- Common looping constructs in R:
  - `for()`: Iterates over a sequence or elements.
  - `while()`: Executes as long as a condition is `TRUE`.
  - `repeat`: Executes indefinitely until stopped with `break`.
- Syntax examples:
  - `for` loop: `for (i in 1:5) { print(i^2) }`: Prints squares of numbers 1 through 5.
  - `while` loop (Prints numbers 1 through 5):
    - `i <- 1`
    - `while (i <= 5) { print(i); i <- i + 1 }`.
  - `repeat` loop (Prints numbers 1 through 5):
    - `i <- 1`
    - `repeat { print(i); i <- i + 1; if (i > 5) break }`.
- Explicit loops are often less efficient than implicit loops and vectorized operations in R.

Implicit Loops in R

- **Implicit loops** simplify operations over rows/columns of matrices, data frames, or elements of lists.
- Common R functions for implicit loops:
  - `lapply()` and `sapply()` operate on lists.
  - `apply()` works on matrices or data frames.
- Syntax examples:
  - Suppose `x` and `y` are numeric vectors:
    - `mylist <- list(var1 = x, var2 = y)`
    - `lapply(mylist, mean)`: Returns a list of means.
    - `sapply(mylist, mean)`: Returns a vector of means.
  - Suppose `mymat` is a matrix:
    - `mymat <- cbind(var1 = x, var2 = y)`
    - `apply(mymat, 2, mean, na.rm = TRUE)`: Computes column means while ignoring `NA` values.

Sorting in R

- Sorting a single vector is straightforward using `sort()`:
  - **Example:** `sorted_x <- sort(x)`
- Sorting rows of a matrix or data frame by a specific column:
  - To sort the rows of a data frame D by the first column:
    - `D_sorted <- D[order(D[, 1]), ]`
  - To sort by the first column and then by the second column:
    - `D_sorted <- D[order(D[, 1], D[, 2]), ]`
- Use `decreasing = TRUE` within `order()` or `sort()` to sort in descending order:
  - **Example:** `D_sorted <- D[order(D[, 1], decreasing = TRUE), ]`
- Sorting can handle complex criteria, ensuring flexibility for data manipulation tasks.

Outline

Review of Statistical Inference

Introduction to R

- Basics
- R Session
- R Graphics
- Programming in R
- Data Entry

Math & Stat Tools

- Probability Stuff
- Statistical Methods
- Linear Algebra - Matrix Operations in R

Workspace and Directories in R

- **Working Directory:**
  - View the current directory with `getwd()`.
  - Change the working directory using `setwd("mydir")`.
- **Workspace Management:**
  - View all objects in the workspace with `ls()`.
  - Remove specific objects from the workspace using `rm(object_name)`.
- **Best Practices:**
  - Use `save.image("myworkspace.RData")` to save the current workspace.
  - Reload a saved workspace with `load("myworkspace.RData")`.

Workspace and Directories (Continued)

- **Saving the Workspace:**
  - Use `save.image()` to save all objects in the current workspace.
  - The default saved file is `.RData`, but a custom filename can be specified.
  - **Example:** Save specific objects (x, y, z) using:

```
save(x, y, z, file = "myfile.RData")
```

- **Loading a Saved Workspace:**
  - Reload a saved workspace using:

```
load(file = "myfile.RData")
```

- **Key Note:** Saving the workspace preserves objects but does not save output from the console.

Why is Saving the Workspace a Good Practice in R?

- Use `save.image()` to save the entire workspace to a `.RData` file.
- Reload the workspace with `load("myworkspace.RData")`.
- Benefits:
  - **Preservation:** Safeguards all objects for future use.
  - **Efficiency:** Avoids re-computation or re-importing of data.
  - **Organization:** Maintains the session state for multi-session projects.
  - **Reproducibility:** Ensures consistent analysis across sessions or with collaborators.
  - **Convenience:** Simplifies saving with a single command.
- Combine workspace saving with script-based workflows for transparency and reproducibility.

Saving Input and Output in R

- **Saving R Input (Commands):**
  - Store commands in a script file, e.g., `myscript.R`.
  - Execute the script using: `source("myscript.R")`
- **Redirecting Output to a File:**
  - Redirect output to a file, e.g., `myfile`, by using: `sink("myfile")`
  - All evaluated output will be stored in `myfile` instead of being displayed in the console.
  - Stop redirecting output with: `sink()`
- **Note:** The `save()` command preserves workspace objects but does not save input or output from the R console.

Getting Help in R

- **Accessing Function Documentation:**
  - Use `help(mean)` or `?mean` to get help for the `mean` function.
- **Comprehensive Help:**
  - Use `help("mean")` to open a detailed help file. In RStudio, this opens the documentation in a dedicated help window.
- **Online Resources:**
  - Google and/or ChatGPT searches often provide quick solutions and examples.
- **Extensive Documentation:**
  - Explore built-in guides like *Introduction to R* and *Writing R Extensions* for detailed learning.

Packages in R

- **Extensive Package Ecosystem:**
  - Thousands of packages provide specialized functions, tools, and datasets.
  - Many packages leverage compiled code (e.g., C or Fortran) for enhanced performance.
- **Discovering Packages:**
  - Browse the CRAN repository for a comprehensive list and package descriptions.
- **Installing a Package:**
  - Use `install.packages("pkg")` and follow the prompts.
- **Using a Package:**
  - Load an installed package with `library(pkg)` to access its functions and datasets.
- **Pro Tip:** Check the package documentation or vignettes for usage examples and detailed instructions.

Outline

- Review of Statistical Inference
- Introduction to R
  - Basics
  - R Session
  - R Graphics
  - Programming in R
  - Data Entry
- Math & Stat Tools
  - Probability Stuff
  - Statistical Methods
  - Linear Algebra - Matrix Operations in R

Introduction to Graphics in R

- R is widely recognized for its powerful and extensive graphical capabilities.
- Includes numerous built-in graphical functions while supporting user-defined customizations.
- This section explores various graphical tools and annotation techniques.
- R can directly generate high-quality PDF or Postscript graphics, ideal for LaTeX document integration.

Scatterplots in Base R

- Scatterplots are a fundamental graphical tool in R, offering extensive customization options.
- **Example Code:**

```
x <- runif(50, 0, 2); y <- runif(50, 0, 2)
plot(x, y, xlab = 'x-label', ylab = 'y-label',
     main = 'Main Title', sub = 'subtitle')
text(0.6, 0.6, 'text at (0.6, 0.6)')
abline(h = 0.6, v = 0.6, lty = 2)
for(s in 1:4)
  mtext(-1:4, side = s, at = 0.7, line = -1:4)
mtext(paste('side', 1:4), side = 1:4, line = -1,
        font = 2)
```

- The example demonstrates: Adding custom text (`text()`); Drawing reference lines (`abline()`); Annotating plot sides with margins (`mtext()`).

Scatterplots with `ggplot2`

- **ggplot2** is a versatile package for creating elegant and customizable plots in `R`.
- The below example demonstrates:
  - Adding custom text (`annotate()`);
  - Drawing reference lines (`geom_hline()`, `geom_vline()`);
  - Customizing titles and labels (`labs()`).

Scatterplots with ggplot2

Example Code:

```
library(ggplot2)
# Generate random data
set.seed(123)  # For reproducibility
x <- runif(50, 0, 2)
y <- runif(50, 0, 2)
data <- data.frame(x, y)

# Create scatterplot
ggplot(data, aes(x = x, y = y)) +
  geom_point(color = "blue") +
  geom_hline(yintercept = 0.6, linetype = "dashed", color = "red") +
  geom_vline(xintercept = 0.6, linetype = "dashed", color = "red") +
  annotate("text", x = 0.6, y = 0.6,
           label = "text at (0.6, 0.6)", hjust = -0.2) +
  labs( title = "Main Title", subtitle = "Subtitle",
    x = "x-label", y = "y-label"
  ) +
  theme_minimal()
```

Histograms in Base R

- Creating histograms in R is simple and highly customizable.
- Basic command: `hist(X)`, where X is the dataset or variable.
- Customization options include:
  - Adding density curves: `curve(dnorm(x, mean, sd), add = TRUE)`.
  - Overlaying legends: `legend("topright", legend = c("Mean", "Median"), lty = 1:2)`.
- **Example:** Using the `mean.med.hist` function:
  - Annotates the histogram with mean and median lines for better visualization.
  - Code available in the R script on Canvas.
- **Pro Tip:** Adjust parameters such as `breaks`, `col`, and `main` for enhanced visuals.

Histograms with `ggplot2`

- Histograms in **ggplot2** offer powerful customization and visual appeal.
- The below example demonstrates:
  - Adds a density curve (`stat_function()`) with mean and SD.
  - Annotates the plot with mean (solid line) and median (dotted line).
  - Enhances readability with customized `labs()` and `theme_minimal()`.

Histograms with ggplot2

Example Code:

```
library(ggplot2)
# Generate random data
set.seed(123)  # For reproducibility
data <- data.frame(X = rnorm(100, mean = 5, sd = 2))
# Create histogram
ggplot(data, aes(x = X)) +
  geom_histogram( aes(y = ..density..),
    bins = 20, fill = "lightblue", color = "black"
  ) +
  stat_function( fun = dnorm,
    args = list(mean = mean(data$X), sd = sd(data$X)),
    color = "red", linetype = "dashed"
  ) +
  geom_vline( aes(xintercept = mean(X)),
    color = "blue", linetype = "solid", linewidth = 1
  ) +
  labs( title = "Histogram with Density Curve",
    subtitle = "Mean Annotation",
    x = "Values", y = "Density"
  ) +
  theme_minimal() +  theme(legend.position = "none")
```

Boxplots in Base `R`

- Boxplots are a powerful tool for visualizing the location, spread, and potential outliers in distributions.
- Ideal for comparing multiple distributions side by side.
- Basic syntax: `boxplot(X)`, where X can be:
  - A numeric vector for a single boxplot.
  - A list or data frame of numeric vectors for grouped boxplots.
- Customization options include:
  - Adding titles: `main = "Title"`.
  - Customizing colors: `col = "lightblue"`.
  - Adding notches for median comparison: `notch = TRUE`.
- **Example:** The `mean_med_comp` function annotates boxplots with mean and median markers for enhanced comparison. See the `R` script on Canvas.

Boxplots with `ggplot2`

- Boxplots in **ggplot2** offer advanced customization and aesthetic appeal.
- The below example demonstrates:
  - Visualizes multiple distributions side by side with grouping by `group`.
  - Adds notches (`notch = TRUE`) to compare medians.
  - Includes mean annotations (`stat_summary()`) with a custom symbol.
  - Clean aesthetics with `theme_minimal()` and color-coded groups.

Boxplots with ggplot2

Example Code:

```
library(ggplot2)
# Generate random data
set.seed(123)  # For reproducibility
data <- data.frame(
  group = rep(c("A", "B", "C"), each = 50),
  values = c(rnorm(50, mean = 5, sd = 1),
             rnorm(50, mean = 6, sd = 1.5),
             rnorm(50, mean = 7, sd = 2)) )
# Create boxplot
ggplot(data, aes(x = group, y = values, fill = group)) +
  geom_boxplot(notch = TRUE, outlier.color = "red", outlier.shape = 19) +
  stat_summary(
    fun = mean, geom = "point", shape = 4, size = 3, color = "blue"
  ) +
  labs(
    title = "Boxplot of Distributions by Group",
    subtitle = "Notched Boxplot with Mean Annotation",
    x = "Group", y = "Values"
  ) +
  theme_minimal()
```

Outline

- Review of Statistical Inference
- Introduction to R
  - Basics
  - R Session
  - R Graphics
  - Programming in R
  - Data Entry
- Math & Stat Tools
  - Probability Stuff
  - Statistical Methods
  - Linear Algebra - Matrix Operations in R

Flow Control in R: if-else

- **Conditional Execution:** The `if-else` structure controls the flow of commands based on conditions.
- **Basic Syntax:**

```
if (condition1) {
  # Do something
} else if (condition2) {
  # Do something else
} else {
  # Do another thing
}
```

- Conditions evaluate to `TRUE` or `FALSE`.
- Logical operators for combining conditions:
  - `&`: AND, `|`: OR, `!`: NOT.
- Comparison operators:
  - Equal: `==`, Not equal: `!=`.

Flow Control in R: Loops

- **Major Looping Structures:** `for()`, `while()`, and `repeat()`.
- **Example:** Computing the square root of a non-negative number (`y`):
  - **Using** `while()`:

```
y <- 12345  # Initialize
x <- y / 2  # Initial guess
while (abs(x^2 - y) > 1e-10) {
  x <- (x + y / x) / 2
}
print(x)
```

  - **Using** `repeat()`:

```
x <- y / 2  # Initial guess
repeat {
  x <- (x + y / x) / 2
  if (abs(x^2 - y) < 1e-10) break
}
print(x)
```

Flow Control in R: Loops

- **Validation:**

```
sqrt(y)  # Built-in square root function
```

- **Highlights:**
  - `while()`: Continues as long as the condition is `TRUE`.
  - `repeat()`: Executes indefinitely until explicitly `break`.
  - Both loops ensure precision using `abs()` for error tolerance.

Flow Control in R: for() Loops (cont'd)

- **The for() loop is one of the most widely used looping structures in R.**
- **Basic Syntax:**
  - `for (i in sequence) { # Do something }`
- **Example: Generating Polynomial Curves**

```
x <- seq(0, 1, by = 0.05)
plot(x, x, type = "l", main = "Polynomial Curves")
for (j in 2:5) {
  lines(x, x^j, col = j)
}
```

- **Variations:**
  - Iterating over numeric sequences: `for(i in (1:10)^4)`
  - Iterating over specific values: `for (j in c(2, 5, 7))`
  - Iterating over object names: `for (var in names(data))`
  - Iterating over functions: `for (f in c(sin, cos, tan))`

Avoiding Loops in R

- Loops in R are easy to use but can be inefficient for large datasets.
- Vectorized functions, like apply, are often faster and cleaner.
- **Example: Finding the maximum value in each column of a matrix X:**
  - **Efficient Approach (with apply):**

    ```
    max.X <- apply(X, 2, max)
    ```
  - **Inefficient Approach (with a loop):**

    ```
    max.X <- rep(NA, ncol(X))  # Initialize
    for (j in 1:ncol(X)) { max.X[j] <- max(X[, j]) }
    ```
  - The apply function is:
    - Cleaner and more concise.
    - Optimized for performance, especially on large datasets.
- **Pro Tip:** Explore related functions like lapply, sapply, and vapply for different use cases.

Outline

Review of Statistical Inference

Introduction to R

- Basics
- R Session
- R Graphics
- Programming in R
- Data Entry

Math & Stat Tools

- Probability Stuff
- Statistical Methods
- Linear Algebra - Matrix Operations in R

Reading Data: `scan`

`scan` is a simple way to read numeric or character vectors.

- **Reading Data into a Vector:**

  ```
  X <- scan(file = "file.dat")
  #Reads file.dat into vector X
  ```

- **Reading Lists:**
  - Lists can be read, but the syntax is more complex.
  - Refer to: `help(scan)`.
- **Alternative for Tabular Data:**
  - Use `read.table()` or `read.csv()` for data frames.

    ```
    data <- read.table("file.dat", header = TRUE)
    data <- read.csv("file.csv")
    ```

Reading Data: `read.table`

`read.table` is ideal for importing tabular data as a data frame.

- **Default Behavior:**
  - Assumes columns separated by spaces; missing values represented as `NA`.
  - Customize with `sep`, `header`, and other arguments.
- **Example: Reading Comma-Separated Files:**

```
data <- read.table(file = "data.dat", header = TRUE,
                   sep = ",")
```

- **Recommended Alternatives:**
  - `read.csv()`: Simplified wrapper for CSV files.
  - `readr::read_csv()`: Faster and more robust import from the `tidyverse`.

```
library(readr)
data <- read_csv("data.csv")
```

The tidyverse Ecosystem

- The most comprehensive package for the **tidyverse** ecosystem in R is the `tidyverse` package itself. When installed, it loads a collection of core packages designed for data manipulation, visualization, and functional programming.
- **Core Packages in `tidyverse`:** `ggplot2` – Data visualization; `dplyr` – Data manipulation; `tidyr` – Data tidying; `readr` – Data import (CSV, TSV, etc.); `purrr` – Functional programming; `tibble` – Modern data frames; `stringr` – String manipulation; `forcats` – Factor handling
- **Other tidyverse-Compatible Packages:** `lubridate` – Works with date-time data; `sf` – Spatial data analysis; `haven` – Read SPSS, Stata, and SAS files; `httr`, `rvest`, `xml2` – Web scraping & APIs; `modelr`, `broom` – Modeling/tidying results
- **For more functionalities:** `tidymodels` - Machine learning, `dbplyr` - Database connections, and `reprex` - Reproducibility

Advanced Considerations for `read.table`

- **Key Arguments:**
  - `header`: Presence of a header row (`TRUE`/`FALSE`).
  - `sep`: Separator/delimiter (e.g., `","`, `"\t"`).
  - `na.strings`: Handling missing values.
  - `quote`: Quotes for character text.
  - `comment.char`: Ignoring comments in the file.
- **Shortcut Functions:**
  - `read.csv()`: For comma-separated files.
  - `read.delim()`: For tab-delimited files.
- **Pro Tip:** For large datasets:
  - Use `data.table::fread()` for optimal performance.
- For more details, see `help(read.table)`.

Writing Data to a File in R

- Exporting data from R is often required for use in other software or for sharing results.
- **Basic Command:** `write.table`
  - Writes a "rectangular" data object X (e.g., a data frame or matrix) to a text file.

    ```
    write.table(X, file = "output.txt", row.names = FALSE,
        col.names = TRUE)
    ```
  - Automatically converts X into a data frame format with optional headers.
- **Alternative Commands for Common Formats:**
  - `write.csv`: For exporting data as a CSV file.

    ```
    write.csv(X, file = "output.csv", row.names = FALSE)
    ```
  - `readr::write_csv`: Faster and more robust CSV writing from the `tidyverse`.

    ```
    library(readr)
    write_csv(X, "output.csv")
    ```
- **Pro Tip:** Use the `quote = FALSE` argument to exclude quotes around character fields for cleaner output.

Outline

Review of Statistical Inference

Introduction to R

- Basics
- R Session
- R Graphics
- Programming in R
- Data Entry

Math & Stat Tools

- Probability Stuff
- Statistical Methods
- Linear Algebra - Matrix Operations in R

Outline

Review of Statistical Inference

Introduction to R

- Basics
- R Session
- R Graphics
- Programming in R
- Data Entry

Math & Stat Tools

- Probability Stuff
- Statistical Methods
- Linear Algebra - Matrix Operations in R

Combinatorics in R

- **Key Functions for Counting Problems:**
  - `factorial(x)`: Computes $x!$ (factorial of x).
  - `choose(n, k)`: Calculates $\binom{n}{k}$ (combinations).
- **Related Functions:**
  - `gamma(x)`: Computes $\Gamma(x)$, the gamma function.
  - `lgamma(x)`: Returns $\log(\Gamma(x))$, useful for large values of x.
  - `digamma(x)`: Computes the derivative of $\log(\Gamma(x))$ (digamma function).
- **Examples:**

```
factorial(5)     # 5! = 120
choose(10, 3)    # 10 choose 3 = 120
gamma(5)         # Gamma(5) = 4! = 24
lgamma(10)       # Log of Gamma(10)
digamma(5)       # Digamma function at x = 5
```

- **Applications:** Useful in probability, combinatorics, and statistical modeling.

Random Sampling in R

- **The `sample()` Function:**
  - Used for random sampling from a finite set.
  - `sample(x)`: Generates a random permutation of elements in x.
  - For integer n: `sample(n)` is equivalent to `sample(1:n)`.
- **Options for Sampling:**
  - `size = k`: Specifies the number of elements to sample.
  - `replace = TRUE`: Allows sampling with replacement.
  - `prob`: Assigns probabilities to each element in X.
- **Example: Random Column Selection**

```
# A 10x10 matrix
X <- matrix(1:100, nrow = 10, ncol = 10)
# Matrix with 7 randomly selected columns
X[, sample(10, size = 7)]
```

- **Applications:**
  - Useful in bootstrapping, randomization tests, and simulations.
  - Probabilistic selection for modeling and data analysis.

Probability Distributions in R

- R **provides functions for probability calculations across many distributions.**
- **Supported Distributions:** Normal, binomial, Poisson, exponential, gamma, uniform, hypergeometric, and more.
- **Functions for a Distribution** `dist`**:**
  - `ddist`: Probability density (pdf) or mass function (pmf).
  - `pdist`: Cumulative distribution function (cdf).
  - `qdist`: Quantile function (inverse cdf).
  - `rdist`: Generate random variables.
- **Examples:**

```
dnorm(0, mean = 0, sd = 1) #pdf of standard normal at x=0
pbinom(3, size = 10, prob = 0.5) #cdf of Bin(10,0.5) at x=3
qpois(0.9, lambda = 5)  #90th percentile of Poisson(5)
rexp(10, rate = 2)      #10 random variables from Exp(2)
```

- **Pro Tip:** Refer to the help files (`?dnorm`, `?pbinom`, etc.) for distribution-specific parameterizations.

Probability Distribution Example: Binomial PDF and CDF

- Plotting a binomial probability mass function (pmf) and cumulative distribution function (cdf).
- **Example Code:**

```
# Parameters
n <- 25; p <- 0.4
# Set up plot
plot(0, 0, type = "n", xlim = c(0, n), ylim = c(0, 1),
     xlab = "x", ylab = "pdf and cdf")
# Add CDF (step function)
lines(0:n, pbinom(0:n, n, p), type = "s", lwd = 2, col = "gray")
# Add PMF (vertical lines)
lines(0:n, dbinom(0:n, n, p), type = "h", lwd = 2)
# Add legend
legend("right", inset = 0.05, lwd = 2, col = c("black", "gray"),
       legend = c("pdf", "cdf"))
```

- **Visualization:** Black lines represent the pmf (`dbinom`); Gray step function represents the cdf (`pbinom`).

Outline

Review of Statistical Inference

Introduction to R

- Basics
- R Session
- R Graphics
- Programming in R
- Data Entry

Math & Stat Tools

- Probability Stuff
- Statistical Methods
- Linear Algebra - Matrix Operations in R

Quick Summary

- `R` is extensively designed for statistical analysis with built-in functions for standard statistical methods.
- Examples of `R` functions:
  - `t.test`: Performs t-tests.
  - `lm`: Handles linear models (ANOVA, regression, etc.).
  - `glm`: Used for generalized linear models (like logistic regression).
- In STAT 7650, the focus is on learning computations, so built-in functions are primarily used for verification[5]

[5] **Note:** Outside of STAT 7650, using built-in functions for standard statistical methods is recommended.

Outline

Review of Statistical Inference

Introduction to R

- Basics
- R Session
- R Graphics
- Programming in R
- Data Entry

Math & Stat Tools

- Probability Stuff
- Statistical Methods
- Linear Algebra - Matrix Operations in R

Matrix Arithmetic

- Consider two matrices A and B of suitable dimensions.[6]
- Adding and subtracting matrices follows the usual element-wise operations.
- For multiplication, use the symbol %*%: `A %*% B`.
- Matrix division isn't as straightforward and typically involves matrix inversion or solving systems of linear equations.
- Matrix inversion and its implications will be discussed later.

[6] Care must be taken to ensure correct matrix dimensions to avoid unexpected results due to vectorization.

Matrix Operations in R

- **Determinants:** `det(M)`: Computes the determinant of matrix M.
- **Diagonal Entries:** `diag(M)`:
  - If M is a matrix, returns a vector of diagonal entries.
  - If M is a vector, creates a diagonal matrix with M on the diagonal.
- **Solving Linear Systems and Matrix Inversion:**
  - Solving $Ax = b$: `x <- solve(A, b)`
  - Inverting a matrix:
    ```
    solve(M)  # Returns the inverse of an invertible matrix
    ```
  - Generalized inverse (for non-invertible matrices):
    ```
    library(MASS)
    ginv(M)
    ```
- **Applications:** Solving systems of equations, optimization problems, and numerical methods.

The Spectral Theorem and Matrix Decompositions:

- **Spectral Theorem:**
  - For a symmetric $d \times d$ positive definite matrix M:

$$M = U \Lambda U^T$$

where:

- Λ: Diagonal matrix of eigenvalues of M.
- U: Orthonormal matrix of eigenvectors of M.

- **Eigenvalue Decomposition in R:**

```
eigen(M)  # Returns a list with $values (eigenvalues)
  # and $vectors (eigenvectors)
```

Matrix Decompositions in R

- **Other Common Decompositions:**
  - **Cholesky Decomposition:** For symmetric positive definite matrices.

    ```
    chol(M)  # Returns an upper triangular matrix R
             # such that M = R^T R
    ```

  - **Singular Value Decomposition (SVD):**

    ```
    svd(M)  # Returns U, D, and V such that M = U D V^T
    ```

- **Applications:** Useful in numerical linear algebra, optimization, PCA, and more.

Neat Example: `sweep` Operator

The `sweep` Operator on a Symmetric Positive Definite Matrix $M = (M_{ij})$:

- **Definition:** Sweeping on the k-th diagonal entry produces a new matrix $\widetilde{M} = (\widetilde{m}_{ij})$, where:

$$\widetilde{m}_{kk} = -\frac{1}{m_{kk}}, \quad \widetilde{m}_{ik} = \frac{m_{ik}}{m_{kk}}, \quad \widetilde{m}_{kj} = \frac{m_{kj}}{m_{kk}}, \quad \widetilde{m}_{ij} = m_{ij} - \frac{m_{ik} m_{kj}}{m_{kk}}$$

- **Properties:**
  - Sweeping has useful mathematical properties (see Chapter 7.5 in Lange).
  - Successively sweeping M along all diagonal entries (in any order) computes the inverse M^{-1}.
- **Implementation:**
  - `R` function `sweep` performs row/column operations but is not directly linked to matrix sweeping in this context.
  - For matrix sweeping, refer to custom `R` code provided on Canvas.