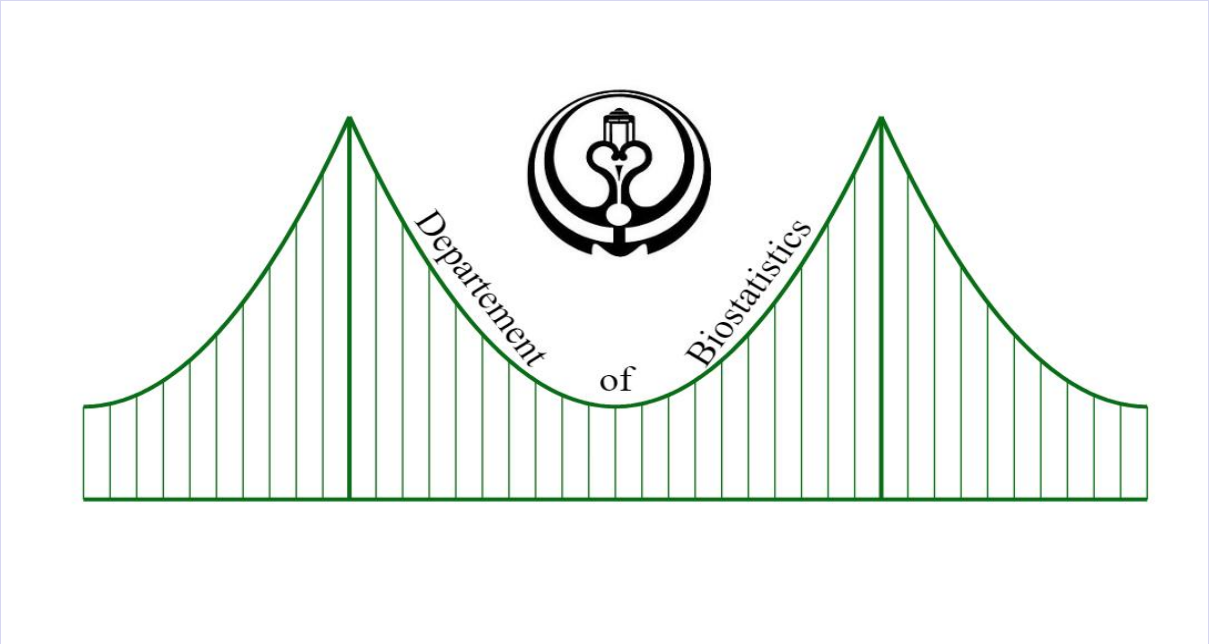


R Programming WorkShop

habib ezzat abadi

Shiraz University of Medical Scinece

1402-1403

Outline

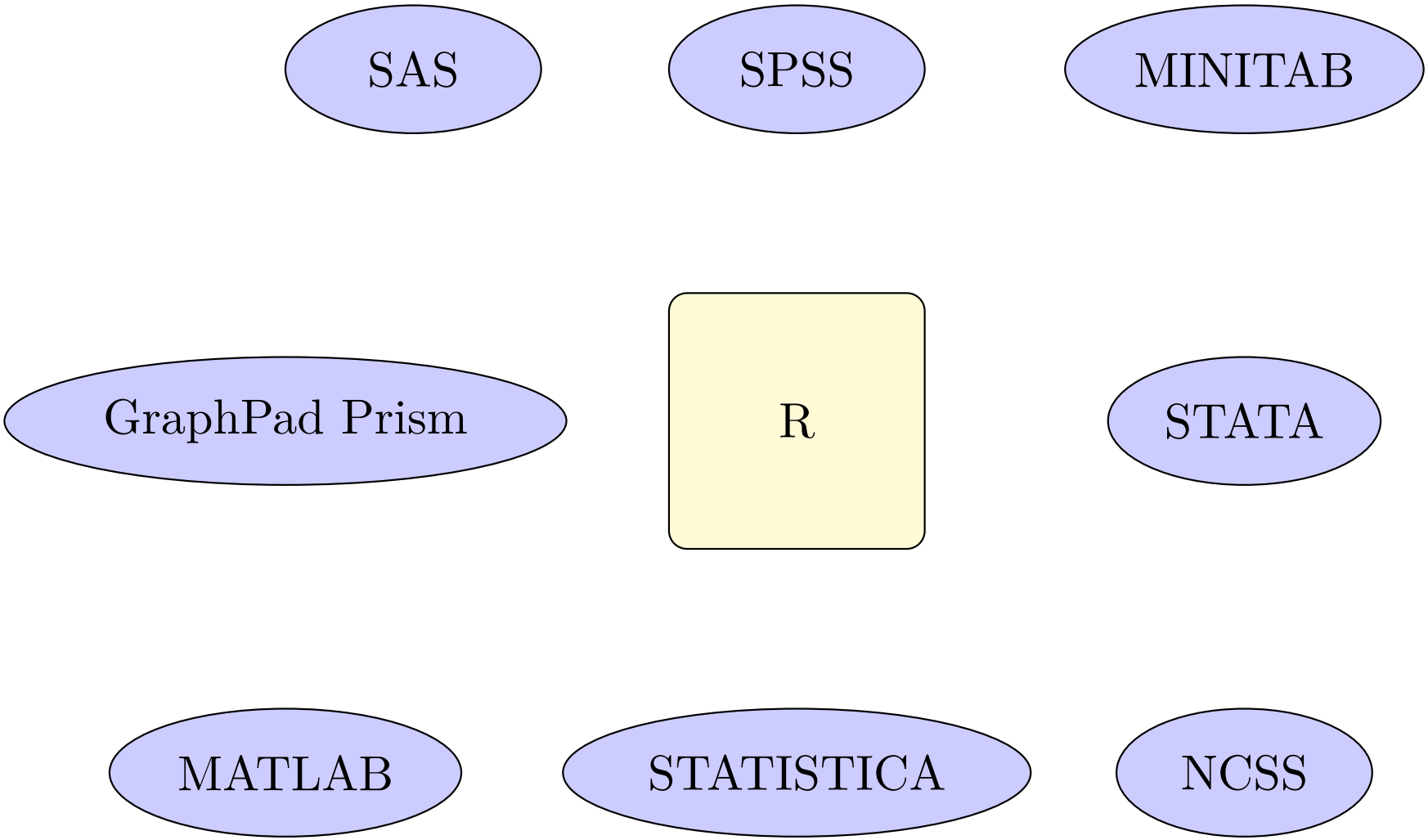
A Brief History of R

- 1976 S-Bell Labs: Fortran
  - John Chambers
  - Rick Becker
  - Allan Wilks
- 1988 S Version 3: C language
  - John Chambers
- 1991 R Created
  - Ross Ihaka
  - Robert Gentelman
- 1993 R Announced
- 2000 R Version 1 Released
- 2020 8th rank of programming languages
- 2023 16th rank of Programming languages





Why R?


SAS
SPSS
MINITAB
GraphPad Prism
R
STATA
MATLAB
STATISTICA
NCSS

Types of conditional structures in R

type of conditional structures in R

- if
- if-else
- ifelse

if structure

Why We Using if structure?

Sometimes we need a certain task to be done only when a condition is met, otherwise we want the normal flow of the program to be maintained if the condition is not met.

Example of if structure

Ex. (i)

Get a number from the user. If the remainder of this number is divided by 5 is 2 or 3, print "Great" in the output, otherwise, do nothing.

Labs

Go to the coding environment →

if-else

if-else

But there are times when we need to change the path of the program for any answer when we check the condition. And it is even possible to obtain different modes for different modes.

structure of if-else

CodeBlock

```
if (condition) {
command1
command2
...
} else{
if {
...
}
```

Example

Ex. (ii)

take a number from the user, with the condition that it is greater than 20, then if the remainder of this number is zero compared to five numbers, print the value of ”A” in the output, if was 1, print the value of ”B” in the output, if it was 2, print the value of ”CC” in the output and if it was 3, print the value of ”C” in the output, and finally, if it was 4 Print the value of ”D” in the output.

Labs

Go to the coding environment →

ifelse

ifelse

Sometimes we are faced with a binary situation, if the condition is met, one thing will happen, and if the condition If it is not established, one more thing will happen and our conditional structure will not extend further. R programming language has defined a very simple structure for this mode by the name of ifelse.

Example

Ex. (iii)

Get an input from the user. If the input was even, it returns the value ”even” and otherwise it returns the value ”odd”.

Labs

Go to the coding environment →

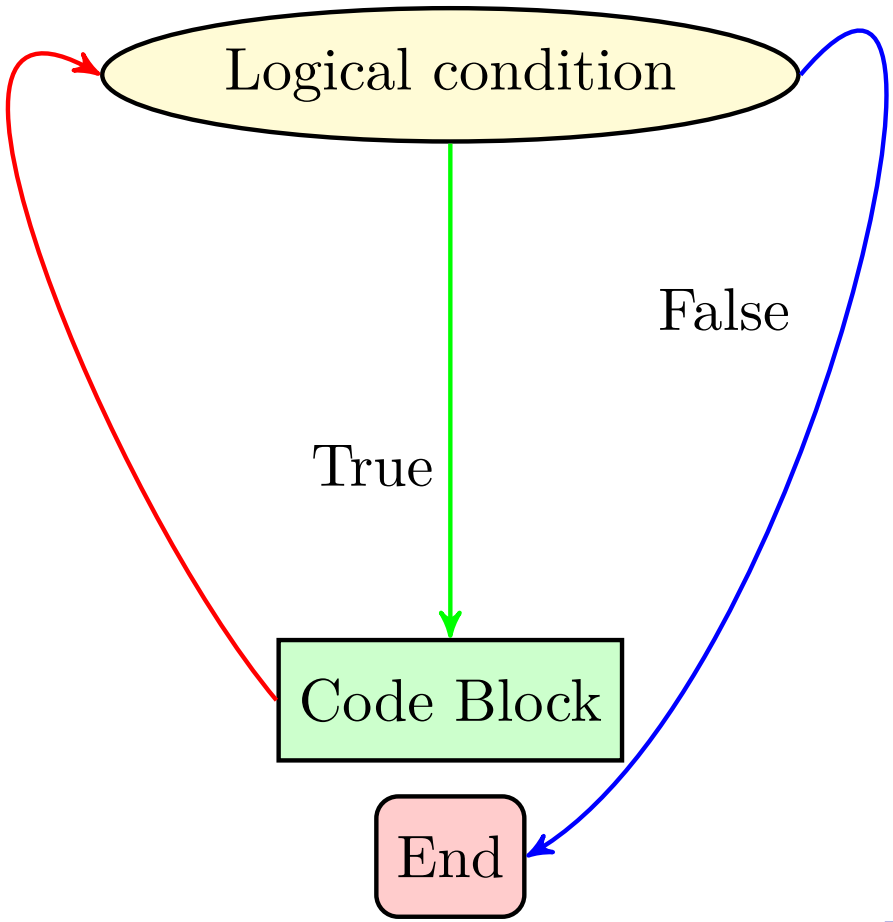
Why Using Loops in Programming?

loops are an essential tool in programming that allow you to execute a block of code repeatedly until a certain condition is met. They are useful for performing repetitive tasks without writing the same code multiple times.

Loops in R

- Using while loop
- Using repeat loop
- Using for loop

why do we use while loop?



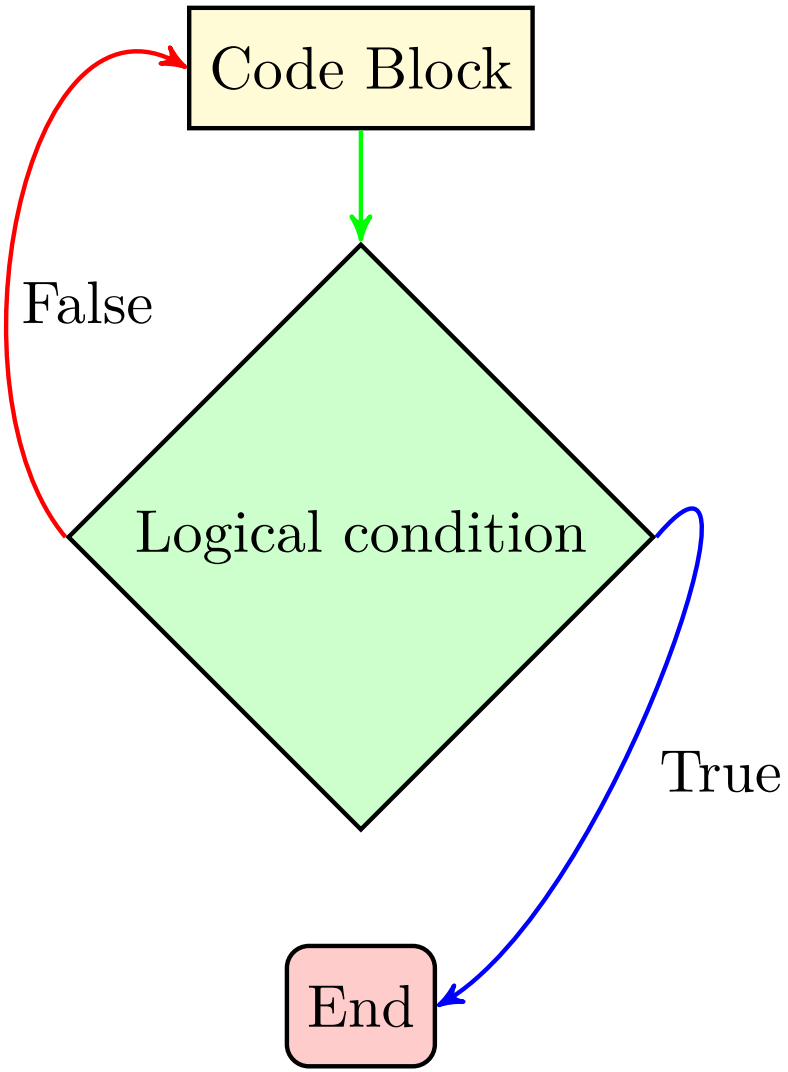
Ex (iv).

$$f(x) = \exp(x) - x^2,$$
$$\text{if} \quad f(x) = 0 \implies x = \ ?$$

Labs

Go to the coding environment →

why do we use repeat loop?



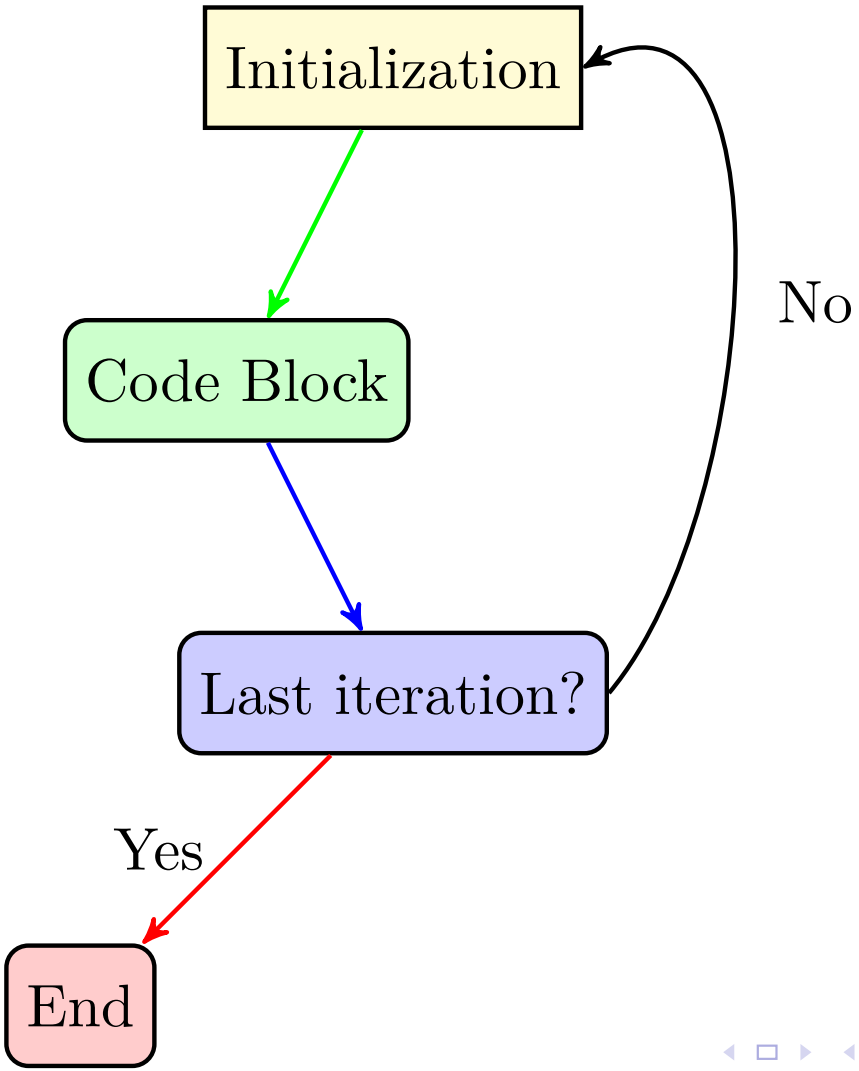
Ex (v).

By referring to the example in slide (20), get the numerical answer using loop repeat.

Labs

Go to the coding environment →

Why do we use For loop?



Example (vi):

Generate a matrix with 10 rows and 10 columns of integer values.

a) Calculate the row sum of this matrix using the for loop.

b) Using the "next" command, if the sum of a row is more than 500, do not print that row sum in the output.

c) If a row is calculated whose sum is greater than 600, the loop will stop. using the command (break)

Labs

Go to the coding environment →

Why do we need functions in Programming?

Functions in programming are essential for several reasons:

Functional Programming

1. Code Reusability: Functions allow us to write a piece of code once and reuse it multiple times. This can save a lot of time and effort, especially in large programs.
2. Modularity: Functions help to break down large programs into small, manageable parts. This makes the code easier to understand, debug, and maintain.
3. Abstraction: Functions hide the details of their operation, allowing us to use them without knowing exactly how they work. This is a key principle of software design known as abstraction.

Functional Programming

continue about Functional Programming

4. Namespace Separation: Variables defined inside a function are not visible outside the function. This helps to prevent naming conflicts in our code.
5. Testing and Debugging: It's easier to test and debug small functions than large monolithic code blocks. If a problem occurs, it's easier to pinpoint the issue in a small, isolated function.
6. Code Readability: Well-named functions can make the code more readable and self-explanatory, improving its understandability.

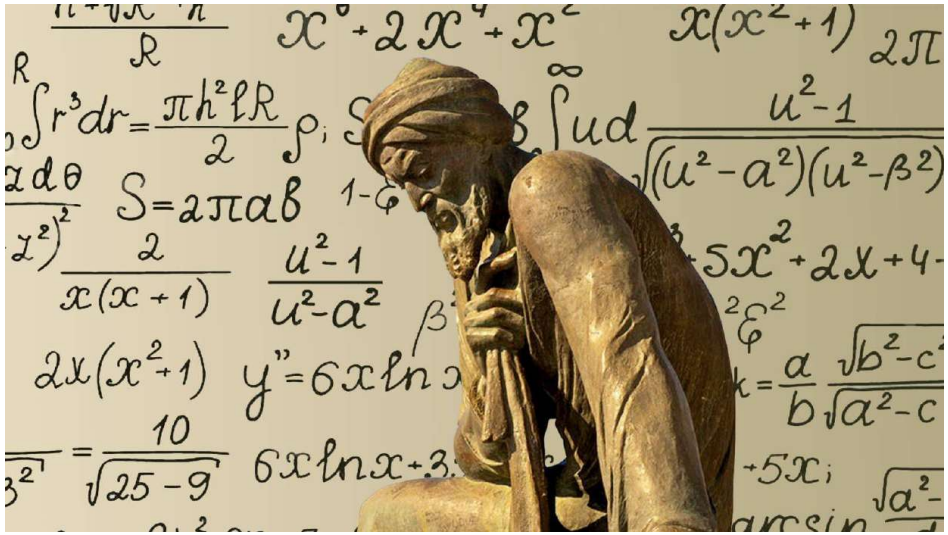
Functional Programming

summary

In summary, functions are a fundamental building block of programming that help us write better, more manageable, and efficient code. They are a key part of structured and object-oriented programming.

Creator

Abullah Mohammad bin Musa al Khwarizmi, Who is often referred to as "The Father of Algebra"



Definition

Definition

An algorithm is a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer. It's a sequence of instructions that a computer must perform to solve a well-defined problem. It essentially defines what the computer needs to do and how to do it. Algorithms can instruct a computer how to perform a calculation, process data, or make a decision.

Why do we need algorithms?

Algorithms are crucial in functional programming for several reasons:

1. Predictability: Algorithms provide a clear sequence of steps to solve a problem, which makes the program's behavior predictable.
2. Efficiency: Efficient algorithms can significantly reduce the time and space complexity of your program, making it run faster and consume less memory.
3. Problem Solving: Algorithms are essential tools for solving complex problems in programming. They provide a structured approach to problem solving, which is particularly important in functional programming where side effects are avoided.

Algorithms

continue about Algorithms

3. Code Reusability: Similar to functions, algorithms can be reused across different parts of a program or even different programs. This can save a lot of time and effort in development.
4. Understanding and Communication: Algorithms provide a way for programmers to communicate their ideas effectively. They are language-agnostic, meaning they can be implemented in any programming language.

Algorithms

Summary

In summary, algorithms are a fundamental part of functional programming and programming in general. They help us solve problems efficiently and effectively, and are a key tool in a programmer's toolkit.

Algorithms

Ex. (vii)

Write an algorithm that takes a number from the user and determines if this number is prime or not?

Algorithms

Result: True or False
Function isPrime(n):
if n is less than 2 then
    return False;
else
    for i from 2 to n do
        if n mod i equals 0 then
            return False;
        end
    end
    return True;
end

Algorithm 1: Prime Check

Objects and functions in R Programming

- Everything that exists is an object.
- Everything that happens is a function call.

John Chambers

function

Elements of a function

- formals
- body
- environment

formals of function

formals

These are the arguments of the function. They control how you can call the function. When a function is invoked, you can pass values to it through these arguments.

EX. (viii)

```
myfun <- function(x1, x2, x3) {
temp1 = x1 * x2
return(temp1)
}
```

Body of functions

Body

This is the code inside the function. It contains the sequence of statements that the function will execute when it is called.

Environment of functions

Environment

This is the "map" of the location of the function's variables. When a function is executed, it creates a new environment to hold its local variables.

lazy evaluation

lazy evaluation

Lazy evaluation, also known as call by need, is a technique used in R programming where the evaluation of an expression is delayed until its value is absolutely needed. This means that in R, an expression is not evaluated when it is not used. For instance, if a function argument is not used in the function, R will not evaluate it. This strategy increases the efficiency of the program, especially when used iteratively, as it avoids repeated evaluations.

Primitive functions

Primitive functions

In R programming, primitive functions are a special type of function that are implemented at a low level for performance reasons. They are only found in the base package and include language elements like ”if” and ”for”, operators like ”and” and ”$”, and mathematical functions like ”exp” and ”sin”.

Primitive functions

Primitive functions have several unique characteristics:

Primitive funcitons

- They operate at a low level, which can make them more efficient.
- They have different rules for argument matching.
- They do not allow any R-code in the function.
- They use a special technique for accessing C-code.

Generic functions

Generic functions

In R programming, generic functions play a crucial role in object-oriented programming. They are functions that behave differently based on the class of the input. This is known as polymorphism.
Generic functions use a system called ”S3”, which was introduced in ”S” version 3. You can identify an S3 generic function because its entire body is a call to the R function ”UseMethod”. For example, ”plot” and ”summary” are generic functions.

Introduction

basic concepts

Let's be a little more formal. Consider that we have a series of functions to run, f1, f2, etc. Serial processing means that f1 runs first, and until f1 completes, nothing else can run. Once f1 completes, f2 begins, and the process repeats. Parallel processing (in the extreme) means that all the f# processes start simultaneously and run to completion on their own.

Prallel Programming

The Serial-parallel scale

A problem can range from “inherently serial” to “perfectly parallel”. An “inherently serial” problem is one which cannot be parallelized at all, for example, if f2 depended on the output of f1 before it could begin, even if we used multiple computers, we would gain no speed-ups. On the other hand a “perfectly parallel” problem is one in which there is absolutely no dependency between iterations; most of the apply calls or simulations we’ve discussed in this WorkShop fall into this category. In this case, any f can start at any point and run to completion regardless of the status of any other f.

Parallel Programming

Terminology

- A core is a general term for either a single processor on your own computer (technically you only have one processor, but a modern processor like the i7 can have multiple cores - hence the term) or a single machine in a cluster network.
- A cluster is a collection of objecting capable of hosting cores, either a network or just the collection of cores on your personal computer.
- A process is a single running version of R (or more generally any program). Each core runs a single process.

The parallel package

The parallel package

There are a number of packages which can be used for parallel processing in R. Two of the earliest and strongest were multicore and snow. However, both were adopted in the base R installation and merged into the parallel package.

Initial Coammands

Initial Coammands

```
library(parallel)
detectCores()
```

Types of Parallel Programming

- SOCKET approach
- FORK approach

Types of Parallel Programming

Types of Prallel Programming

- The socket approach launches a new version of R on each core. Technically this connection is done via networking (e.g. the same as if you connected to a remote server), but the connection is happening all on your own computer mention this because you may get a warning from your computer asking whether to allow R to accept incoming connections, you should allow it.
- The forking approach copies the entire current version of R and moves it to a new core.

SOCKET

There are various pro's and con's to the two approaches:

PROS and CONS for Socket approach

- Works on any system (including Windows).
- Each process on each node is unique so it can't cross-contaminate.
- Each process is unique so it will be slower
- Things such as package loading need to be done in each process separately. Variables defined on your main version of R don't exist on each core unless explicitly placed there.
- More complicated to implement.

Forking

PROS and CONS for Forking approach

- Only works on POSIX systems (Mac, Linux, Unix, BSD) and not Windows.
- Because processes are duplicates, it can cause issues specifically with random number generation (which should usually be handled by parallel in the background) or when running in a GUI (such as RStudio). This doesn't come up often, but if you get odd behavior, this may be the case.
- Faster than sockets.
- Because it copies the existing version of R, your entire workspace exists in each process.
- Trivially easy to implement.

Labs

Labs Links

In order to be able to access the codes and examples, you can visit these two links;
First link
second link.