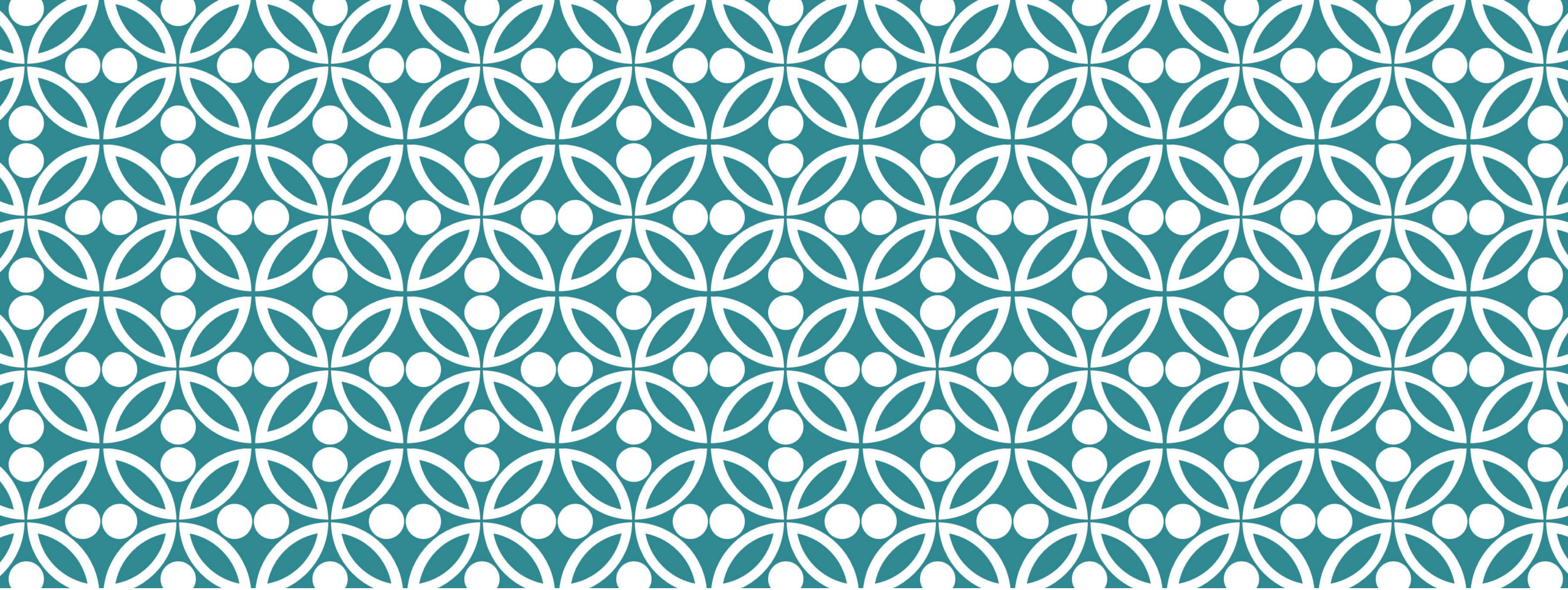
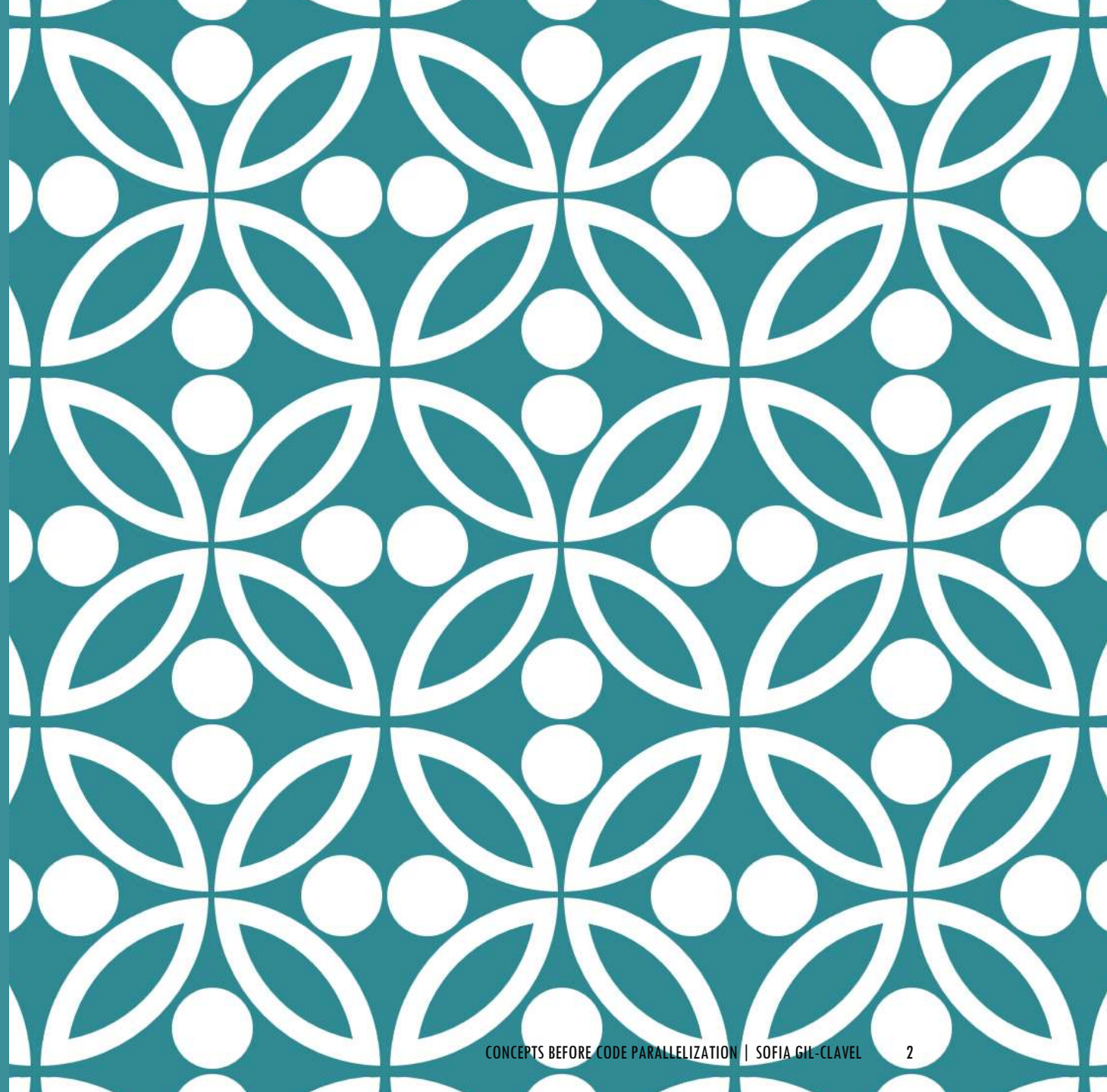


BASIC CONCEPTS BEFORE CODE PARALLELIZATION

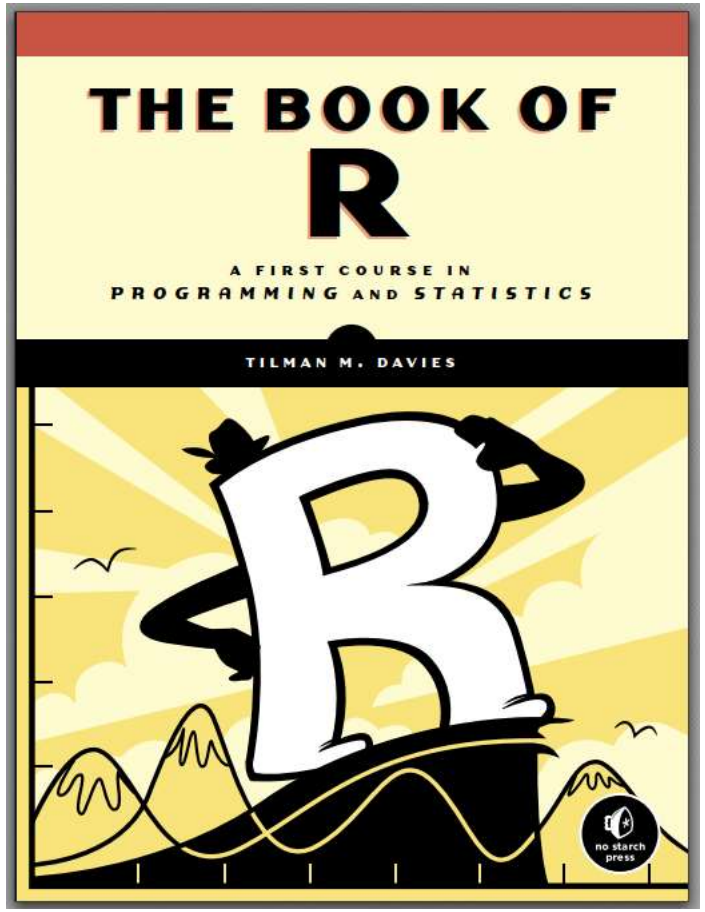
DR. SOFIA GIL-CLAVEL

1. Conditions and loops
2. From problems to pseudocode and to R-code
3. R properties to optimize loops
4. Speed-up code using lapply, vapply, etc.

1. CONDITIONS AND LOOPS



WE WILL FOLLOW:



Davies, Tilman M. ***The Book of R: A First Course in Programming and Statistics***. No Starch Press, 2016.

To write more sophisticated programs with R, you'll need to control the flow and order of execution in your code. One fundamental way to do this is to make the execution of certain sections of code dependent on a *condition*. Another basic control mechanism is the *loop*, which repeats a block of code a certain number of times. In this chapter, we'll explore these core programming techniques using `if-else` statements, `for` and `while` loops, and other control structures.

IF STATEMENTS

The if statement is the key to controlling exactly which operations are carried out in a given chunk of code. An if statement runs a block of code only if a certain condition is true. These constructs allow a program to respond differently depending on whether a condition is TRUE or FALSE.

```
if(condition){
    do any code here
}
```

- Exercise:

  Write the code where you test if a variable is bigger than 3. If it is bigger than 3, then you print the string "It is bigger than 3!".

IF-ELSE STATEMENTS

The if statement executes a chunk of code if and only if a defined condition is TRUE. If you want something different to happen when the condition is FALSE, you can add an else declaration.

```
if(condition){
    do any code in here if condition is TRUE
} else {
    do any code in here if condition is FALSE
}
```

- Exercise:

  To the previous code, now add the condition that if the value is smaller or equal to 3 then you print "It is not bigger than 3!".

INLINE IF-ELSE STATEMENTS

ifelse returns a value with the same shape as test which is filled with elements selected from either yes or no depending on whether the element of test is TRUE or FALSE.

ifelse(test, yes, no)

test: an object which can be coerced to logical mode.

yes: return values for true elements of test.

no: return values for false elements of test.

- Exercise:

Create a vector that goes from 0 to 10.

Now, use the inline if-else statement to test which ones are bigger than 5.

If they are bigger than 5, then it should return "It is bigger than 5!"

Otherwise, it should return "No, it is not!"

NESTING AND STACKING STATEMENTS

An if statement can itself be placed within the outcome of another if statement. By nesting or stacking several statements, you can weave intricate paths of decision-making by checking a number of conditions at various stages during execution.

```
if(mystring=="Homer"){
    foo <- 12
} else if(mystring=="Marge"){
    foo <- 34
} else if(mystring=="Bart"){
    foo <- 56
} else if(mystring=="Lisa"){
    foo <- 78
} else if(mystring=="Maggie"){
    foo <- 90
} else {
    foo <- NA
}
```

CODING LOOPS

Another core programming mechanism is the loop, which repeats a specified section of code, often while incrementing an index or counter. There are two styles of looping:

- The **for** loop repeats code as it works its way through a vector, element by element.
- The **while** loop simply repeats code until a specific condition evaluates to FALSE.

FOR

The R for loop always takes the following general form:

```
for(loopindex in loopvector){
    do any code in here
}
```

```
for(i in 1:5) {
  print(i)
}
```

Here, the loopindex is a placeholder that represents an element in the loopvector. it starts off as the first element in the vector and moves to the next element with each loop repetition. When the for loop begins, it runs the code in the braced area, replacing any occurrence of the loopindex with the first element of the loopvector. When the loop reaches the closing brace, the loopindex is incremented, taking on the second element of the loopvector, and the braced area is repeated. This continues until the loop reaches the final element of the loopvector, at which point the braced code is executed for the final time, and the loop exits.

EXERCISE 1.1

Write a code that runs on each letter of the sentence “This is a sentence.” while printing the concatenated string.

Hint: Use the function “paste” or “paste0”.

WHILE

To use for loops, you must know, or be able to easily calculate, the number of times the loop should repeat. In situations where you don't know how many times the desired operations need to be run, you can turn to the while loop. A while loop runs and repeats while a specified condition returns TRUE, and takes the following general form:

```
while(loopcondition){
    do any code in here
}
```

```
count <- 0
while(count < 10) {
  print(count)
  count <- count + 1
}
```

A while loop uses a single logical-valued loopcondition to control how many times it repeats. Upon execution, the loopcondition is evaluated. If the condition is found to be TRUE, the braced area code is executed line by line as usual until complete, at which point the loopcondition is checked again. The loop terminates only when the condition evaluates to FALSE, and it does so immediately—the braced code is not run one last time.

EXERCISE 1.2

Run the code from exercise 1.1, but now it stops when the code reaches the character ".". The code should not use the "for" loop at all.

OTHER CONTROL FLOW MECHANISMS

There are three more control flow mechanisms: break, next, and repeat. These mechanisms are often used in conjunction with the loops and if statements.

- Declaring break or next
- The repeat Statement

DECLARING BREAK OR NEXT

Break: Normally a for loop will exit only when the loopindex exhausts the loopvector, and a while loop will exit only when the loopcondition evaluates to FALSE. But you can also preemptively terminate a loop by declaring break.

```
count <- 0
repeat {
  print(count)
  count <- count + 1
  if(count == 10) break
}
```

Next: Instead of breaking and completely ending a loop, you can use next to simply advance to the next iteration and continue execution.

EXERCISE 1.3

To the code from exercise 1.2 add the following conditions:

- If the letter is a vocal, then the code should skip it.
- If the letter is a “c”, then it should break.

REPEAT

Another option for repeating a set of operations is the repeat statement.

```
repeat{
    do any code in here
}
```

```
count <- 0
repeat {
  print(count)
  count <- count + 1
  if(count == 10) break
}
```

Notice that a repeat statement doesn't include any kind of loopindex or loopcondition. To stop repeating the code inside the braces, you must use a break declaration inside the braced area (usually within an if statement); without it, the braced code will repeat without end, creating an infinite loop. To avoid this, you must make sure the operations will at some point cause the loop to reach a break.

EXERCISE 1.4

Write the code in exercise 1.3 replacing the “while” with “repeat” and “break”.

IN SUMMARY

Function/operator	Brief description
`if( ){ }`	Conditional check
`if( ){ } else { }`	Check and alternative
`ifelse`	Element-wise `if-else` check
`break`	Exit explicit loop
`next`	Skip to next loop iteration
`repeat{ }`	Repeat code until `break`

The control sequences are divided into:

- Bifurcation statements:
  - If/else - The if statement is the key to controlling exactly which operations are carried out in a given chunk of code.
  - Loops:
    - for: Runs a loop a finite number of times.
    - while: Runs a loop as long as a condition is true
  - Other control mechanisms:
    - break/next:
    - repeat (break): The repeat structure executes a loop infinitely. The only way to end the loop is to call the break function inside it.

Control structures describe the order in which statements or groups of statements are executed.

5 MINS BREAK

2. FROM PROBLEMS TO PSEUDOCODE AND TO R-CODE

ANALYSIS AND SPECIFICATION OF THE PROBLEM

- Analysis of a problem is required to ensure that the problem is well defined and clearly understood.
- Formulating a problem specification requires an accurate and as complete description as possible of the problem input (information available for problem resolution) and the required output.
- A good problem specification needs to answer the following questions:
  - ✓What should the program do?
  - ✓What output should it produce?
  - ✓What inputs are required to get the desired outputs?

DESIGN

- The solution design phase begins once the program specification is in place and consists of formulating the steps to solve the problem.
- Designing a solution requires the use of algorithms. An algorithm is a set of instructions or steps to solve a problem. The algorithms must process the data necessary for the resolution of the problem.
- Programs will be composed of algorithms that manipulate or process information.
- A good technique for designing an algorithm is to break down the problem into smaller, simpler subtasks or subtasks, until you get to subtasks that are easier to program.

ALGORITHMS & PSEUDOCODE

Algorithms are usually written in pseudocode, which consists of a set of words that represent tasks to be performed and a syntax of use like any language.

PROBLEM: CALCULATE THE ADDITION OF ALL THE NUMBERS FROM 1 TO N.

Solution:

We need to add up all the numbers between 1 and n.

For this, it is necessary to initialize a variable where the addition will be stored.

It is also necessary to add the values from 1 to n and store them in the previously defined variable.

Once this is done, it is possible to return the result.

Pseudocode:

Input: n

Output: The addition

Pseudocode:

- Start the variables i and sum
- i=1
- sum=0
- while(i<=n)
  - sum=sum + i
  - i=i+1
- return sum

PSEUDOCODE-TO-CODE TRANSLATION

Translation consists of translating our ideas written in pseudocode into the programming language that is being used.

EXERCISE 2.1

- Translate, write, and execute the pseudocode shown in the previous page.

WRITING FUNCTIONS

When the same algorithm must be applied to different values, it is better to encapsulate the code into a function.

function {base} R Documentation

Function Definition

Description

These functions provide the base mechanisms for defining new functions in the **R** language.

Usage

```
function( arglist ) expr
\( arglist ) expr
return(value)
```

Arguments

`arglist`	empty or one or more (comma-separated) '`name`' or '`name = expression`' terms and/or the special token `...`.
`expr`	an expression.
`value`	an expression.

WRITING FUNCTIONS

When the same algorithm must be applied to different values, it is better to encapsulate the code into a function.

function {base} R Documentation

Function Definition

Description

These functions provide the base mechanisms for defining new functions in the **R** language.

Usage

```
function( arglist ) expr
\( arglist ) expr
return(value)
```

Arguments

`arglist`	empty or one or more (comma-separated) '`name`' or '`name = expression`' terms and/or the special token `...`.
`expr`	an expression.
`value`	an expression.

This is an ellipsis!

EXERCISE 2.2

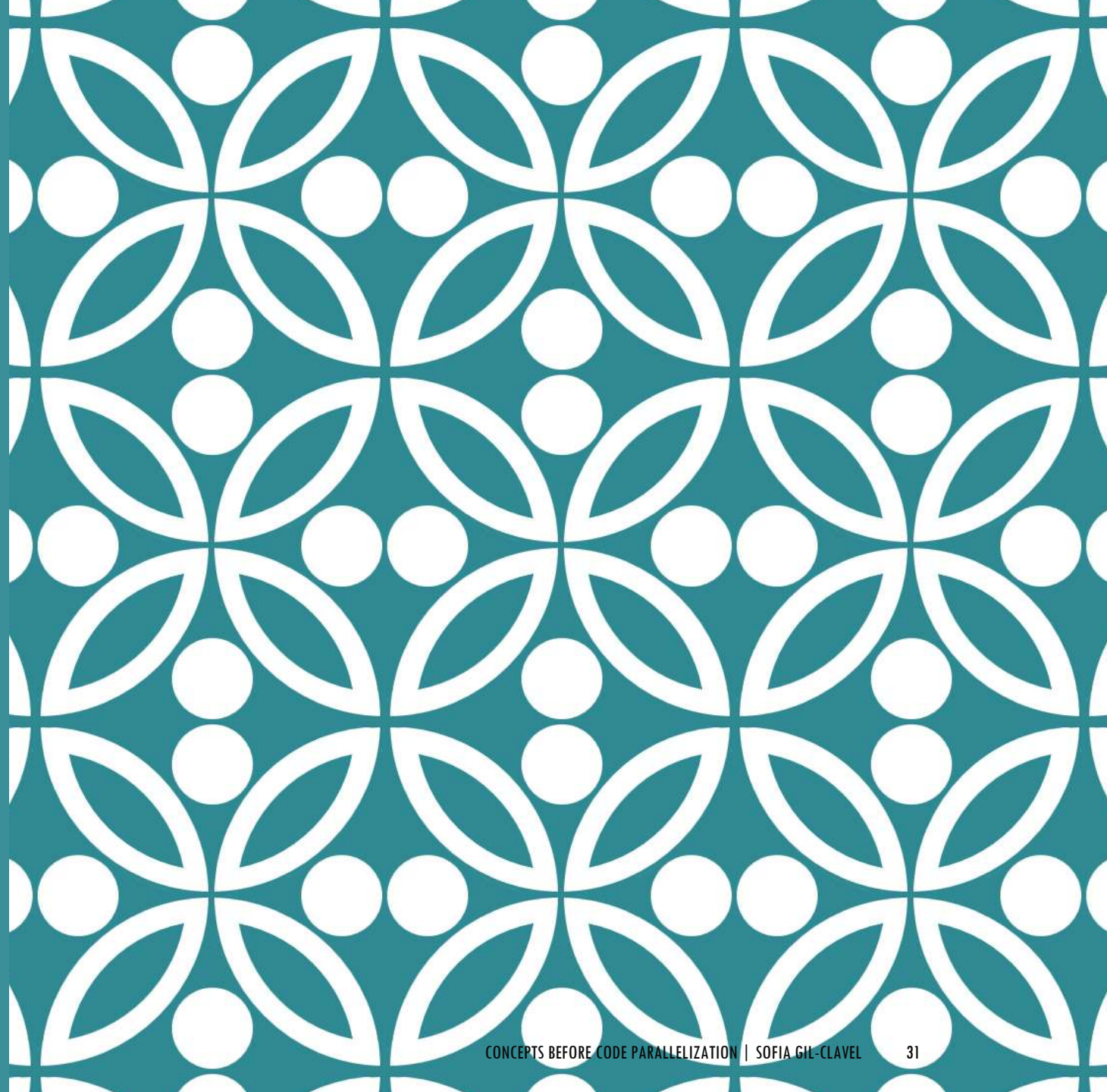
Turn the code from exercise 2.1 into a function and run it over different values of “n”.

EXERCISE 2.3

1. Write the pseudocode of a program that generates 30 databases and stores them in three different folders depending on the number of the database. The first 10 databases go in folder “first”, the databases between 11 and 20 go to “second”, and the lasts ones go into folder “third”. Hint: The databases can be a modification of an existing R-database.

2. Translate, write, and execute the previous pseudocode into R. Hint: Use the functions: function, data.frame, c(), paste or paste0, AND for, while, next, break, or repeat.

3. R PROPERTIES TO OPTIMIZE LOOPS



USING VECTORS TO OPTIMIZE LOOPS

Often, you'll want to perform the same calculations or comparisons upon multiple entities, for example if you're rescaling measurements in a data set. You could do this type of operation one entry at a time, though this is clearly not ideal, especially if you have a large number of items. R provides a far more efficient solution to this problem with vectors.

COLON OPERATOR

Colon Operator

Description

Generate regular sequences.

Usage

```
from:to
   a:b
```

Arguments

`from`	starting value of sequence.
`to`	(maximal) end value of the sequence.

Exercise:

- Create a vector from 0 to 100.

SEQUENCE GENERATION

Sequence Generation

Description

Generate regular sequences. `seq` is a standard generic with a default method. `seq.int` is a primitive which can be much faster but has a few restrictions. `seq_along` and `seq_len` are very fast primitives for two common cases.

Usage

```
seq(...)

## Default S3 method:
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)

seq.int(from, to, by, length.out, along.with, ...)

seq_along(along.with)
seq_len(length.out)
```

Exercise:

- Create a sequence between -1 and 1 with 100 numbers in between.
- Create the same sequence using increments.

REPETITION

Replicate Elements of Vectors and Lists

Description

`rep` replicates the values in `x`. It is a generic function, and the (internal) default method is described here.

`rep.int` and `rep_len` are faster simplified versions for two common cases. Internally, they are generic, so methods can be defined for them (see InternalMethods).

Usage

```
rep(x, ...)

rep.int(x, times)

rep_len(x, length.out)
```

Exercise:

- Create a vector that repeats the letters between a and d 4 times.
- Create a vector that repeats each letter between a and d 4 times.

VECTOR-ORIENTED BEHAVIOR

Vectors are so useful because they allow R to carry out operations on multiple elements simultaneously with speed and efficiency. This vector oriented, vectorized, or element-wise behavior is a key feature of the language, one that you will briefly examine here through some examples of rescaling measurements.

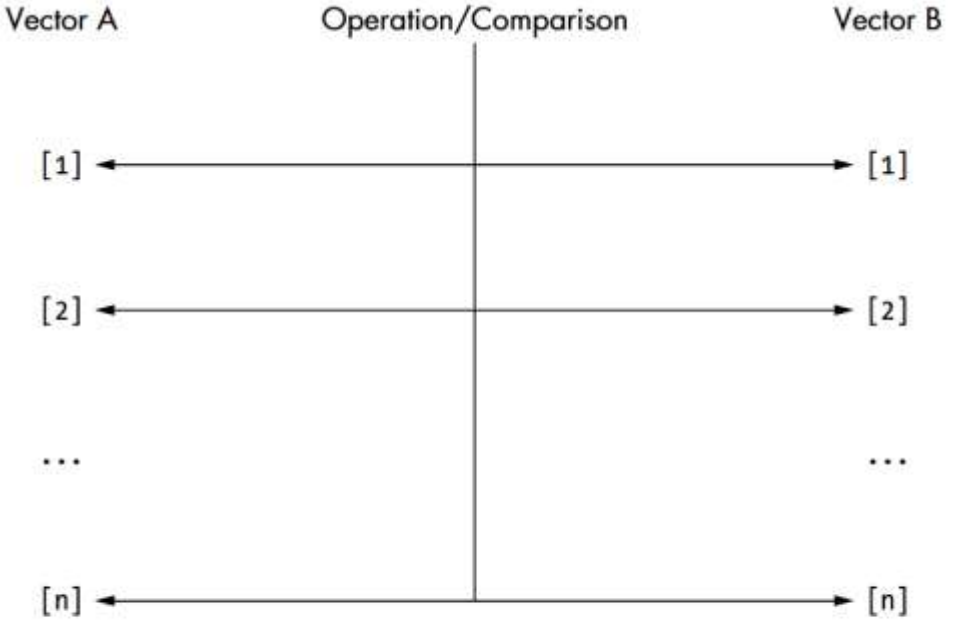


Figure 2-1: A conceptual diagram of the element-wise behavior of a comparison or operation carried out on two vectors of equal length in R. Note that the operation is performed by matching up the element positions.

EXERCISE 3.1

Choose two operators from each category and apply them to different pairs of vectors.

Operators					
Aritmetic		**Comparative**		**Logical**	
+	addition	<	less than	! x	logic NO
-	subtraction	>	more than	x & y	element-wise AND
*	multiplication	<=	less or equal than	x && y	single comparison AND
/	division	>=	more or equal than	x | y	element-wise OR
^	power	==	equal than	x || y	single comparison OR
%/%	integer division	!=	different than	xor(x,y)	exclusive OR

VECTOR-ORIENTED FUNCTIONS

`sum(x)`
`prod(x)`
`max(x)`
`min(x)`
`which.max(x)`
`which.min(x)`
`range(x)`
`length(x)`
`mean(x)`
`median(x)`
`var(x)` o `cov(x)`
`cor(x)`
`var(x, y)` o `cov(x, y)`
`cor(x, y)`

`round(x, n)`
`rev(x)`
`sort(x)`
`rank(x)`
`log(x, base)`
`scale(x)`
`pmin(x,y,...)`
`pmax(x,y,...)`
`cumsum(x)`
`cumprod(x)`
`cummin(x)`
`cummax(x)`

`match(x, y)`
`which(x == a)`
`choose(n, k)`
`na.omit(x)`
`na.fail(x)`
`unique(x)`
`table(x)`
`subset(x, ...)`
`sample(x, size)`

EXERCISE 3.2

Choose three functions from the previous page. What are they doing?

EXERCISE 3.3

Optimize the code you wrote in Exercise 2.3 using vector-oriented behavior.

5 MINS BREAK

4. SPEED-UP CODE USING APPLY, LAPPLY, ETC.

IMPLICIT LOOPING WITH APPLY

In some situations, especially for relatively routine for, you can avoid some of the details associated with explicit looping by using the apply function and other similar functions.

APPLY

The apply function is the most basic form of implicit looping—it takes a function and applies it to each margin of an array.

apply {base} R Documentation

Apply Functions Over Array Margins

Description

Returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.

Usage

```
apply(X, MARGIN, FUN, ..., simplify = TRUE)
```

Arguments

`X`	an array, including a matrix.
`MARGIN`	a vector giving the subscripts which the function will be applied over. E.g., for a matrix `1` indicates rows, `2` indicates columns, `c(1, 2)` indicates rows and columns. Where `X` has named dimnames, it can be a character vector selecting dimension names.
`FUN`	the function to be applied: see 'Details'. In the case of functions like `+`, `%*%`, etc., the function name must be backquoted or quoted.
`...`	optional arguments to `FUN`.
`simplify`	a logical indicating whether results should be simplified if possible.

EXERCISE 4.1

You want to get the mean value of each column of your data frame. What would you do?

For this exercise you need to write your own function to calculate the mean value.

1. Write the pseudocode.
2. Write the code using loops.
3. Use the function apply.

LAPPLY AND SAPPLY

Using lapply, you can apply the function to each element of a list. The returned value is itself a list. Another variant, sapply, returns the same results as lapply but in an array form.

You can think of a data frame as a list with values (columns) consisting of vectors of the same length.

EXERCISE 4.2

Solve exercise 4.1 using lapply and sapply.

EXTRA ARGUMENTS?

All of R's apply functions allow for additional arguments to be passed to FUN; most of them do this via an ellipsis, i.e. the "…" at the end of the function's argument.

EXERCISE 4.3

Introduce some “NA”s to your data frame values. Now, in your previous code, replace your self written “mean” function with the base R “mean” function.

What happened?

What extra argument do you need to pass to the apply functions?

EXERCISE 4.4

How would you optimize exercise 2.3 using the apply functions and the vector oriented behavior?

Exercise 2.3: Write a program that generates 30 databases and stores them in three different folders depending on the number of the database. The first 10 databases go in folder “first”, the databases between 11 and 20 go to “second”, and the lasts ones go into folder “third”. Hint: The databases can be a modification of an existing R-database.

Hint: You can pass the existing R-database as an extra argument of a function.

[hybrid]R-Workshop: Servers and Code-parallelization in R

This is the last workshop in the R-series. The final session will focus on using servers for parallel processing. During this session, we will use all the learnings from “Basic concepts before code-parallelization in R” to speed up codes using computing servers. These

https://societal-analytics.nl/events/20250620_workshop/

Join us!

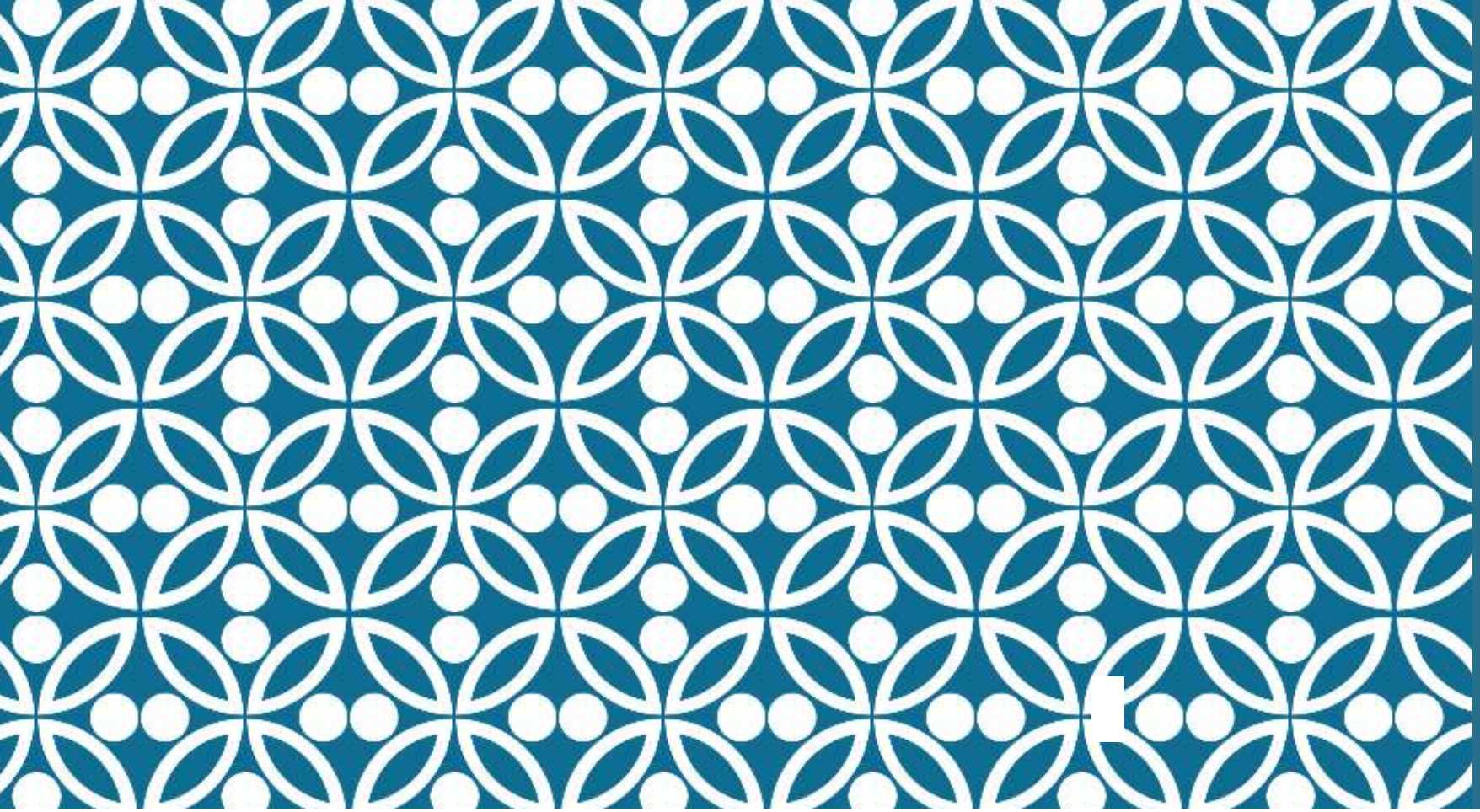
https://forms.office.com/e/8Bgd2YsasJ

All about the lab:

https://societal-analytics.nl/

Contact us at:

analytics-lab.fsw@vu.nl.



https://sofiag1l.github.io/

THANKS!

Dr. Sofia Gil-Clavel