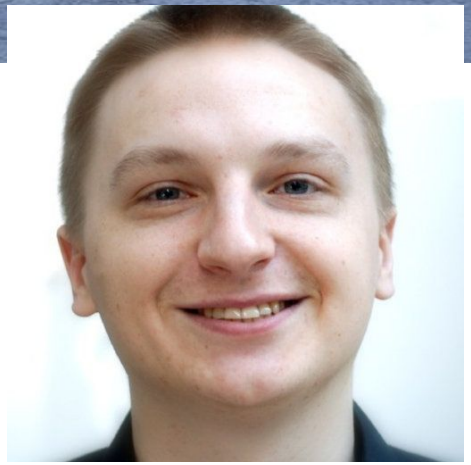
R Fundamentals and Best Practices for Mass Spectrometry Data Analysis



Saturday, November 7 (12:00 - 3:15pm Eastern)

Mateusz Staniak, University of Wrocław

Module #1: Intermediate R: functions and clean code

Module #2: Intermediate R: optimizing R code

About this workshop

About me

- second year PhD students in Mathematics @ University of Wrocław, Poland
- research in applications of statistics to MS-based protein quantification

- work experience: data analysis and software development (R, Python)
- currently: part of the MSstats team working on MSstats refactoring

- teaching experience: *Intro to R* and *Software development with R* courses at the University of Wrocław, short R/ML courses at R conferences

- contact: mtst@mstaniak.pl / staniak@math.uni.wroc.pl

Goals

ASMS Workshop focus:

- *coding fundamentals and best practices to support reproducible data analysis*
- *good practices for coding style, documentation (...)*

This section of the workshop will:

- present writing functions in R in depth,
- relate that to the principles of writing clean code,
- present methods of measuring performance of R code and optimizing it,
- put the those topics in the context of MS data processing and analysis.

What this section will not cover:

- data visualization (day 2 - visualization workshop)
- out-of-memory data processing (day 4 - Cardinal workshop)

Agenda

Part 1: functions in R

- quick R recap
- functions in R
- clean code principles
- object-oriented programming (methods in R)

Part 2: optimizing R code

- debugging
- profiling and benchmarking R code
- memory and time in R
- tabular data manipulation with data.table
- Rcpp basics
- parallel frameworks in R

Acknowledgments

This workshop is based on:

- May Institute 2020 Online - Kylie Bemis: Intermediate R crash course for MS practitioners (some slides are borrowed)
- My *Software development with R* course @ University of Wrocław
- Hadley Wickham's Advanced R
- Robert C. Martin's Clean Code: A Handbook of Agile Craftsmanship
- Life (:

Recap: R essentials

R essentials

- basic data types:
  - vectors,
  - matrices,
  - lists,
  - data.frames
    -> differences? Characteristics?
- subsetting:
  - types of subsetting,
  - type-preserving methods,
    -> how many ways to subset are there?
- control flow
  - loops
  - conditional statements
- vectorizations

Functions and functional programming

To understand computations in R, two slogans are helpful:

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

— John Chambers, creator of S

Why functions matter?

- good functions are a basis of clean code
- good functions ensure that the code is testable and extensible:
  - modifying code is easier when functions are small and have a single responsibility
  - small functions are easy to test
- functions help avoid duplication
  - fewer errors
- functions allow for sharing code
- function allow for re-using code

Functions: details

Functions in R

Functions are *first-class citizens* in R - they can be

- passed as an argument,
- returned from a function,
- modified,
- assigned to a variable.

Functions in R: structure

Functions consist of three parts:

- body – the code inside the function
- formals – a list of parameters used to call the function
- environment – where the function will look for objects associated with names that it uses

Exceptions: *Primitive* functions (base R functions that call C code directly)

Functions in R: parameters

- function may have multiple parameters

  (but functions with many parameters are harder to understand)
- some parameter can be optional: `missing()`
- parameter may be given default values: `parameter = “default”`
- a variable number of parameters can be passed via …
- parameters are evaluated in a **lazy** manner

Closures

- functions may return functions
- because of **lexical scoping**, a function created within another function has access to parameters of the parent function
- such functions are called **closures**

-> what are some scenarios in which this pattern can be useful?

Lexical scoping

Lexical scoping is a way in which R finds values associated with names of variables found in the body of a function:

1. R looks for variables in the **environment in which the function was created**.
2. If a variable cannot be found in that environment, parent environment is searched.
3. The search continues until the variable is found or empty environment is reached.

*apply family of packages

The *apply functions are often used to replace loops. The functions apply a function to each element of a vector / list:

- lapply function always returns a list,
- sapply function attempts to simplify results into a vector, matrix, or array,
- vapply function simplifies the result according to a user-specified template.

-> in what situation a loop cannot be replaced by an *apply function?

-> are *apply functions faster than loops?

Functions: functional programming

Functions: clean code principles

Clean code

Clean code is:

(benefits of clean code)

1. Easy to understand:
    a. readable
    b. functions and classes have clear responsibilities
    c. relationships between objects, functions, etc are comprehensible
2. Easy to change
    a. extendable
    b. unit tests

Cost of messy code:

1. Time wasted on
    a. reading and understanding
    b. refactoring
2. Slow (or no) development
    a. changes require big rewrites
    b. changes may break the existing code
3. The code is prone to bugs

General rules for writing clean functions

1. Functions must be **small**
2. Each function must have a **single responsibility**
3. **Single level of abstraction** for each function

Important:

- separate computations from the output
- use good names for functions and objects

```
public static String testableHtml(
  PageData pageData,
  boolean includeSuiteSetup
) throws Exception {
  WikiPage wikiPage = pageData.getWikiPage();
  StringBuffer buffer = new StringBuffer();
  if (pageData.hasAttribute("Test")) {
    if (includeSuiteSetup) {
      WikiPage suiteSetup =
        PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_SETUP_NAME, wikiPage
        );
      if (suiteSetup != null) {
        WikiPagePath pagePath =
          suiteSetup.getPageCrawler().getFullPath(suiteSetup);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -setup .")
              .append(pagePathName)
              .append("\n");
      }
    }
    WikiPage setup =
      PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
    if (setup != null) {
      WikiPagePath setupPath =
        wikiPage.getPageCrawler().getFullPath(setup);
      String setupPathName = PathParser.render(setupPath);
      buffer.append("!include -setup .")
            .append(setupPathName)
            .append("\n");
    }
  }
```

```
  buffer.append(pageData.getContent());
  if (pageData.hasAttribute("Test")) {
    WikiPage teardown =
      PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
    if (teardown != null) {
      WikiPagePath tearDownPath =
        wikiPage.getPageCrawler().getFullPath(teardown);
      String tearDownPathName = PathParser.render(tearDownPath);
      buffer.append("\n")
            .append("!include -teardown .")
            .append(tearDownPathName)
            .append("\n");
    }
    if (includeSuiteSetup) {
      WikiPage suiteTeardown =
        PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_TEARDOWN_NAME,
                wikiPage
        );
      if (suiteTeardown != null) {

        WikiPagePath pagePath =
          suiteTeardown.getPageCrawler().getFullPath (suiteTeardown);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -teardown .")
              .append(pagePathName)
              .append("\n");
      }
    }
  }
  pageData.setContent(buffer.toString());
  return pageData.getHtml();
}
```

Code excerpt from Clean Code

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite
) throws Exception {
  boolean isTestPage = pageData.hasAttribute("Test");
  if (isTestPage) {
    WikiPage testPage = pageData.getWikiPage();
    StringBuffer newPageContent = new StringBuffer();
    includeSetupPages(testPage, newPageContent, isSuite);
    newPageContent.append(pageData.getContent());
    includeTeardownPages(testPage, newPageContent, isSuite);
    pageData.setContent(newPageContent.toString());
  }
    return pageData.getHtml();
}
```

Code excerpt from Clean Code

Single level of abstraction: scrambled eggs example

```
add(butter)
while (butter is not melted) {
…
}
…
knife.addEnergy()
knife.hit(egg)
add(egg)
…
for (peppercorn in pepper) {
  add(peppercorn)
}
# stir
```

```
pan.add(butter)
pan.heat()
pan.add(cut(meat))
pan.add(cut(mushrooms))
pan.add(break(eggs))
pan.add(salt)
pan.add(pepper)
stir(pan)
```

Adapted from: http://pawelwlodarski.blogspot.com/2011/02/poziomy-abstrakcji.html

Naming: rules

1. Describe the intentions
2. One word = one concept
3. Easy pronunciation
4. Use domain-specific terms
5. Don’t try to be funny
6. Class names - nouns, function/method names - verbs
7. Avoid misinformation

Recommendations for clean functions

1. Code should be read top to bottom, each function on a lower level of abstraction
2. Functions should be small
3. Functions should do just one thing (and do it well)
4. Single level of abstraction
5. Functions should have 1-3 parameters (hard to do in R!)
6. Do not optimize code prematurely (correctness first!)
7. Functions need appropriate names

Comments

- general rule: **clean code should explain itself**
- comments will never fix a messy code

Exceptions:

- TODO/FIXME comments can be useful
- comments that explain specific implementation details

Test-driven development

Write tests before (production) code

Three rules of TDD:

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test

Functions: style

Coding styles

- Tidyverse coding style:

```
# Good
if (y < 0 && debug) {
  message("y is negative")
}

if (y == 0) {
  if (x > 0) {
    log(x)
  } else {
    message("x is negative or zero")
  }
} else {
  y^x
}

test_that("call1 returns an ordered factor", {
  expect_s3_class(call1(x, y), c("factor", "ordered"))
})
```

- Google coding style (variant of Tidyverse)

```
# Good
DoNothing <- function() {
  return(invisible(NULL))
}
```

The names of private functions should begin with a dot.

```
# Good
.DoNothingPrivately <- function() {
  return(invisible(NULL))
}
```

Clean code tools in R

Unit tests:

- testthat package
- tinytest package

Code style:

- styler
- lintr
- formatR

Clean code checks:

- cleanr package

Functions: methods

Object oriented programming in R

R provides several tools for object oriented programming. Two major frameworks are:

- S3 system - commonly used in base R and CRAN packages
  - no class definitions
  - very simple
  - is a base for function such as `summary()` or `print()`
  - single dispatch
- S4 system - preferred in Bioconductor packages
  - formal class definitions
  - allows for multiple dispatch

Object-oriented programming in R

In languages such as C++ or Python, methods belong to class.
Typical method calls look like this:

- `api.get_dataset(id = 1)[object.method(params)]`

R takes a functional programming approach: methods belong to *generic* functions.
Typical method call look like this:

- `get_dataset(api, id = 1)[method(object)]`

S3 and S4 systems in brief

	S3	S4
check class	`class(x)`	`is(x)`
create class instance	`class(x)<-/structure()`	`new("class_name", x)`
inheritance	`class(x) = c("class", "parent")`	`setClass("class", contains = "parent")`
method definition	`function_name.class_name`	`setGeneric` + `setMethod`
object validity	no / custom functions	`validObject()` method

Implications for code design

- class systems allow for hiding details of implementations
- classes allow for re-using code (inheritance)
- classes add flexibility (with new data types, it is enough to write a new method for an existing generic function)

Time for a break!

(10 minutes)

Debugging and profiling R code

Debugging

Debugging use cases

1. Finding a source of an error in foreign function calls
    a. sometimes, functions that are used in our code will throw an error (for example due to incorrect input)
    b. debugging tools can help pinpoint the source of an error
2. Finding a bug in our own code
    a. while testing our code, we may find that it produces incorrect results
    b. debugging tools help find the code that produces an

It is hard to write flawless code. Usually, at least at first the code will produce wrong results.

Tools for debugging

R provides several tools for debugging:

- `traceback()`: prints the stack at the time the error occurred
- `browser()`: creates a pseudo-breakpoint that allows you to jump into code at particular point
- `options(error=recover)`: allows you to enter the browser anywhere on the stack after an error occurs
- `debug()` and `undebug()`: enters browser at the beginning of a particular function call

Tools for debugging

The `traceback()` is extremely useful for the first use case.
Finding the exact location of a code that produced error is a first step in finding its cause.

The `browser()` is useful for the second use case. By allowing us to explore the environment within a function call, it can help finding the reason a bug exists.

Understanding `traceback()`

- finding the source of an error is a first debugging step
- `traceback()` provides a full list of functions calls starting with the outermost function call and ending with the function that throws an error
  (from bottom to the top)
- `traceback()` shows which function threw an error, but the cause of this error may be in an earlier call
  (for example because an earlier function changed column names in a data.frame)

Understanding browser()

Adding `browser()` to the code or using `debug()` on a function (or setting `options(error=recover)`) will start debugging mode. RStudio provides a special graphical interface for this process. Operations in the browser can be done via command line:

- n : Execute the next line
- s : Execute the next line and step into it if it's a function
- f : Finish execution of current loop or function
- c : Continue execution, exiting interactive debugging mode
- Q : Quit debugging mode without continuing execution

Note: `options(warn = 2)` converts warnings into errors

Note: debugging S4 methods requires adding the signature parameter to the debug function (otherwise, only the generic, not specific function will be debugged)

Defensive programming

1. “Fail fast” and “fail early”: it is better for a function to fail as soon as it receives bad input rather than propagate that bad input.

    a. Solution: check whether the arguments passed to your function are valid values for those arguments – immediately stop() with an appropriate message if your input isn’t valid

2. Write short, simple functions that do one thing (importance of clean code!)

    a. longer functions that do many things and handle multiple input types are more error-prone
    b. shorter, simpler functions are easier to jump into and debug()

Defensive programming

3. Avoid non-standard evaluation (NSE) when writing functions

    a. Functions that use NSE like base::subset() or dplyr::filter() are convenient for top-level interactive data analysis, but can be extremely difficult to debug when used inside functions
    b. Same remark applies to the %>% (pipe) operator

4. Avoid functions that may return different types of values

    c. use `lapply()` or `vapply()` rather than `sapply()`
    d. always specify `drop=FALSE` when subsetting with [

5. Wrap error-prone code in `try()` or `tryCatch()`

Note: using `try()` or `tryCatch()` may negatively affect performance of the code.

Profiling and benchmarking

Profiling

- profiling: measuring time and frequency of function calls
- memory profiling; measuring memory usage of function calls
- useful for detecting performance bottlenecks of the code
- can be used on larger pieces of code

There are multiple functions and packages for profiling in R. The `profvis` package provides a very good and intuitive tool.

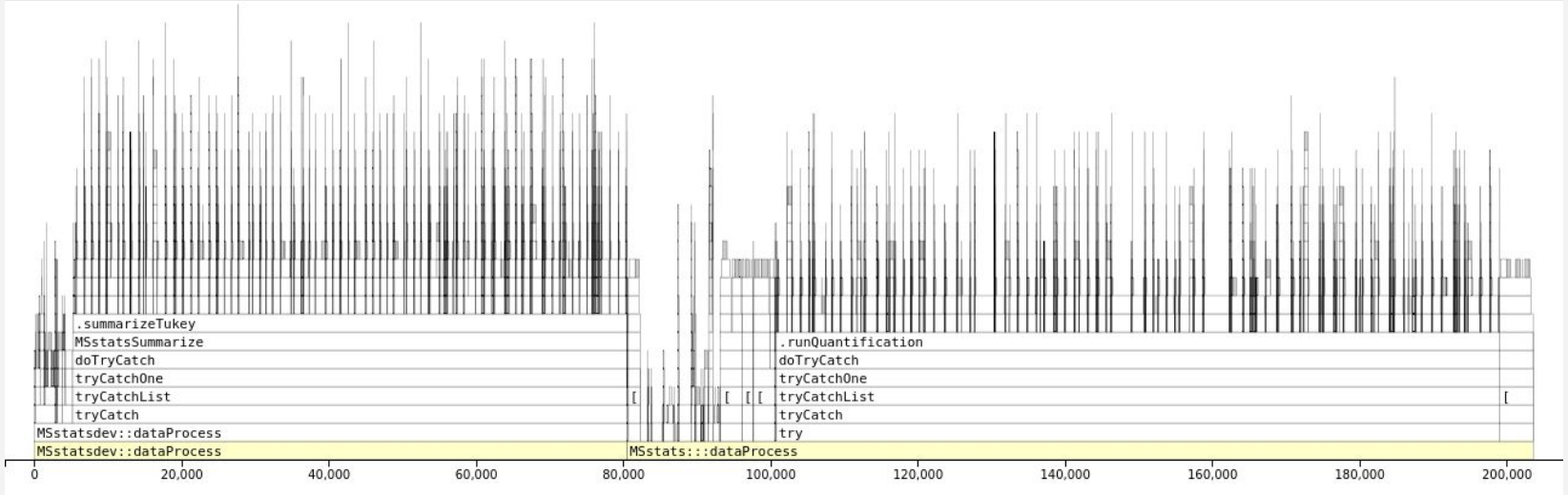
(Next slide: example output)

Flame Graph | Data | Options ▾

<expr>

```
1  profvis::profvis({
2    new = MSstatsdev::dataProcess(sl_v4, featureSubset = "topN", n_top_feature = 100,
3                                  censoredInt = "0")
4    old = MSstats:::dataProcess(as.data.frame(sl_v4), featureSubset = "topN", n_top_feature = 100,
5                                censoredInt = "0")
6  })
7
```

	Memory		Time
new	-3617.8	4027.4	80490
old	-6456.4	6324.9	123080



Profiling notes

- **time profiling**: some of the primitive functions (for example `<-` or `[`) do not show up in the call stack, even though they might take a lot of time to execute; sometimes time may be wrongly attribute to a first line of a multiline expression,
- **memory profiling**: memory profiling is less precise and harder to interpret than time profiling. Memory profiling measures changes in memory allocation. As garbage collection may happen in random moments, memory disallocation may be attributed to a wrong function call,
- results of the profiling are **random**, because profiler takes snapshot of the call stack at random time points,
- R profiling does not work for C/C++/Java code.

Memory and copies in R

Memory allocation

General rules in R:

- call-by-value: changing an argument passed to a function within the function will not affect the original object (with some exceptions)
- copy on modify: a copy of the object will be created if the argument is modified (with some exceptions)

The `pryr` package allows for tracking and understanding copies of R objects;

Tips for avoiding duplication

- Use “primitive” replacement functions when possible
- Always pre-allocate a vector/ matrix before looping over it
- Use numeric() or matrix() before the loop
- Avoid using c() or rbind()/cbind() to append results together
- Modify an vector/matrix as early as possible after creating it
- Use tracemem() to track when objects get copied and rewrite to avoid it
- Use C/C++ code where possible (reference semantics)

Note: R version 4.0.0 introduced proper reference counting. Thanks to this, there is potential for smaller number of unnecessary copies made by R.

Benchmarking

Benchmarking

The `microbenchmark` package in R allows for measuring execution time of pieces of R code.

- it is meant to measure time for small function that run quickly
- the results are very informative when the function is supposed to run many times
- the packages allows to compare multiple implementation of an algorithm

Benchmarking: interpretation

```
microbenchmark::microbenchmark(
    new = MSstatsTMTdev::OpenMStoMSstatsTMTFormat(openms_dataset),
    old = MSstatsTMT::OpenMStoMSstatsTMTFormat(openms_dataset),
    times = 5
)
# Unit: seconds
# expr      min       lq     mean   median       uq      max neval
# new 11.45007 11.57573 12.56472 12.85697 13.07927 13.86157     5
# old 18.15652 18.61845 20.08310 19.51904 21.96909 22.15240     5
```

Code optimization

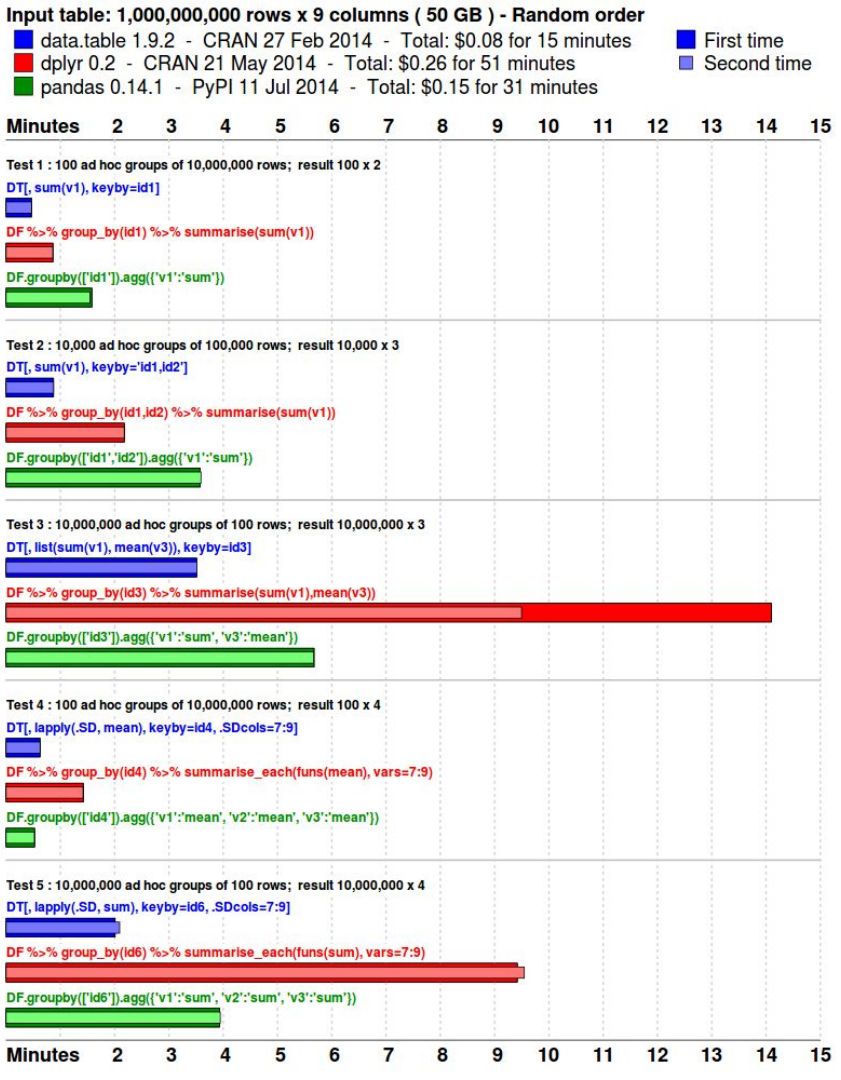
General workflow

1. Make sure that the code is clean
2. Profiling: find slow / memory consuming parts of the code
3. Optimization:
    a. If possible, use existing efficient implementations
    b. If no efficient implementations exist, try alternative implementations (including C/C++)
4. Benchmark alternative solutions

Efficient tabular data manipulation with `data.table` package

data.table package

- top open source package for tabular data manipulation
- extremely useful for working with quantification data,
  in some cases useful with raw spectra, too
- current backed for the `MSstatsConvert` package
- optimized for large datasets
- reference semantics



Basic data manipulation with data.table

- basic operations are done under the [] operator: `dt[i, j, by = ]`,

  i: filtering expression

  j: column selection / modification

  by: grouping variables
- := in j: assign/modify by reference
- special symbol: .SD - **S**ubset of **D**ata
- joins: `merge()` method
- Reshaping data:
  - long format to wide: `dcast()`
  - wide format to long: `melt()`

C/C++ integration with R: Rcpp

Potential R bottlenecks

- loops and recursive functions: large numbers of function calls can negatively affect code performance. On the other hand, the overhead of calling a function is much lower in C/C++
    a. in particular, loops that cannot be vectorized
- problems that require advanced advanced data structures and algorithms that are not available in R
    a. for example, data structures provided by the standard template library in C+++
    b. the same applies to matrix operations (RcppArmadillo)

Rcpp

Rcpp is an R package that provides tools for using C++ code with R.

To export C++ function to R:

1. Add line

   `// [[Rcpp::export]]`

   directly above the C++ function
2. Call the `Rcpp::evalCpp()` function with a path to the C++ file.

R data structures and functions in C++

Rcpp provides several classes that mimic R data structures:

- NumericVector, LogicalVector etc for vectors,
- NumericMatrix for matrices,
- List and DataFrame for lists and data.frame, respectively.

Similarly, many R functions have a counterpart in Rcpp, for example:

- common functions such as `which`
- vectorized operations such as `==`

Parallel data processing

Parallel processing

Some data analysis tasks can be done independently. In this case, many datasets can be processed in parallel.

The easiest problems to parallelize are embarrassingly parallel:

- Independent tasks requiring no communication between them
- Data can be split into similarly-sized, independent subsets
- Almost anything that can be done with lapply() or purrr::map()

Examples: processing raw spectra (each spectrum can be processed separately),

protein-level statistical analysis (each protein can be treated separately)

Parallel processing: cautionary notes

- each parallel process uses memory. In some cases, RAM availability may become the bottleneck,
- communication overhead may negatively affect the performance,
- one should be careful about nested parallelism: some function that we would like to call from parallelized code may require multiple threads.

BiocParallel package

The BiocParallel package on Bioconductor provides a bplapply() function that implements a parallel version of lapply().

BiocParallel allows users to register() different parallel backends, including a fallback to serial evaluation.

Documentation and vignettes:

http://bioconductor.org/packages/BiocParallel/

BiocParallel backends

1. `SerialParam()`: serial evaluation.
2. `SnowParam()`: “simple-network-of-workstations”: creates multiple different R processes.
3. `MulticoreParam()`: creates forks of R process (UNIX only).
4. `DoparParam()`: wrapper around the `foreach` package.
5. `BatchtoolsParam()`: wrapper around the `batchtools` package.

Thank you for your attention!

Thank you to the instructors and to the teaching assistants!

Ryan Benz
Meena Choi
Niyati Chopra
Miguel Cosenza
Matthias Fahrner
Amanda Figueroa-Navedo
Melanie Foell
Omkar Reddy Gojala
Dan Guo
Shubhanshu Gupta
Ting Huang
Maanasa Kaza
Smit Anish Kiri
Devon Kohler
Sai Srikanth Lakkimsetty
Danielle LaMay
Ajeya Makanahalli Kempegowda
Yogesh Nizzer
Harish Ramani
Ruthvik Ravindra
Abdul Rehman
Sai Divya Sangeetha Bhagavatula
Siddarth Sathyanarayanan
Gopalika Shama
Rishabh Rajesh Shanbhag
Sagar Singh
Mateusz Staniak
Sara Taheri
Anuska Tak
Derrie Susan Varghese
Amrutha Vempati

Video of the presentation: https://www.youtube.com/channel/UCnbUMFIlRLaY7fwfSintWuQ/