Computational Statistics

Problems

- Compute nonparametric estimates.
- Compute integrals (and probabilities)

\[ E f(X) = \int f(X) \, dP \quad \left( P(X \in A) = \int_{X \in A} dP \right). \]
- Optimize the likelihood.

Methods

- Numerical linear algebra.
- Monte Carlo integration.
- EM-algorithm, stochastic gradient.
Example, amino acid angles
Ramachandran plot

qplot(phi, psi, data = phipsi)

qplot(phi, psi, data = phipsi2)
Example, amino acid angles

```
hist(phipsi$phi, prob = TRUE)
rug(phipsi$phi)
```

```
hist(phipsi$psi, prob = TRUE)
rug(phipsi$psi)
```
Example, amino acid angles

```r
lines(density(phipsi$phi),
     col = "red", lwd = 2)
```

```r
lines(density(phipsi$psi),
     col = "red", lwd = 2)
```
Statistical topics of the course

- **Smoothing**: what does density do?
  - How do we compute nonparametric estimators?
  - How do we choose tuning parameters?

- **Simulation**: how do we efficiently simulate from a target distribution?
  - How do we assess results from Monte Carlo methods?
  - What if we cannot compute the density?

- **Optimization**: how do we compute the MLE?
  - What if we cannot compute the likelihood?
  - How to deal with very large data sets?
Computational topics of the course

- **Implementation**: writing statistical software.
  - R data structures and functions
  - S3 object oriented programming

- **Correctness**: does the implementation do the right thing?
  - testing
  - debugging
  - accuracy of numerical computations

- **Efficiency**: minimize memory and time usage.
  - benchmark code for comparison
  - profile code for identifying bottlenecks
  - optimize code (Rcpp)
Assignments

The 8 assignments covering 4 topics will form the backbone for the course. Many lectures and practicals will be build around these assignments.

You all need to register (in Absalon) for the presentation of one assignment solution.

- Presentations are done in groups of two-three persons.
- On four Thursdays there will be presentations with discussion and feedback.
- For the exam you need to prepare four *individual* presentations, one for each topic assignment.
Exam

For each of the four topics you choose one out of two assignments to prepare for the exam.

- The exam assessment is based on your presentation *on the basis of the entire content of the course*.
- Get started immediately and work continuously on the assignments as the course progresses.
Prerequisites in R

Good working knowledge of:

- Data structures (vectors, lists, data frames).
- Control structures (loops, if-then-else).
- Function calling.
- Interactive and script usages (source) of R.
- You don't need to be an experienced programmer.
Exercise

With \( x \) a vector of numbers, write code in R to compute a vector \( y \) of logicals such that \( y[i] \) is \texttt{TRUE} whenever \( x[i] > 0 \). E.g.

\[
\begin{align*}
\text{\( x \)} & \quad \text{\( y \)} \\
\text{\# [1] -1 0 1 2} & \quad \text{\# [1] FALSE FALSE TRUE TRUE}
\end{align*}
\]
Vectors in R

Think of a vector as a column with a number of entries. There are two basic types:

- **Atomic vectors**: All components are of the same type, e.g.
  - integers
  - numbers
  - logical values
  - character strings

- **Lists**: May contain components of different types, including components which themselves are lists.

Atomic vectors and lists are important building blocks of all data structures in R.
Atomic vectors and subsetting

Example of a vector.

```r
my_vector <- c(1.1, 3.2, 90, 67.7, 10)
my_vector
```

```
#> [1] 1.1 3.2 90.0 67.7 10.0
```

Extracting components using `[]`.

```r
my_vector[1]
```

```
#> [1] 1.1
```

```r
my_vector[c(1,4)]
```

```
#> [1] 1.1 67.7
```

```r
my_vector[-2]
```

```
#> [1] 1.1 90.0 67.7 10.0
```
Vectors of class `integer`

Example of an integer vector.

```r
integer_vector <- 1L:10L
give_me_a_vector

#> [1]  1  2  3  4  5  6  7  8  9 10

class(integer_vector)

#> [1] "integer"

This is a vector of length ten, i.e. it has ten elements.

length(integer_vector)

#> [1] 10
Vectors of class `numeric`

Example of a vector with non-integer values.

```r
numeric_vector <- seq(0.1, 1, by = 0.1)
numeric_vector
```

```
# [1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```r
class(numeric_vector)
```

```
# [1] "numeric"
```

```r
typeof(numeric_vector)  # Internal storage mode
```

```
# [1] "double"
```
A tricky (non-)integer vector

Example of a vector with only seemingly integer values (and how to fix it).

```r
x <- c(1, 4, 7, 9)
class(x)

## [1] "numeric"

typeof(x)

## [1] "double"

x <- c(1L, 4L, 7L, 9L)
class(x)

## [1] "integer"
```
Comparison of numerical values

Values in a vector of class numeric are approximate.

```r
numeric_vector[2:3]
```

```r
## [1] 0.2 0.3
```

```r
numeric_vector[2:3] == c(0.2, 0.3)
```

```r
## [1]  TRUE FALSE
```

Function `all.equal` has a tolerance.

```r
all.equal(numeric_vector[2:3], c(0.2, 0.3))
```

```r
## [1] TRUE
```
Precision

Note that the internal storage precision is not changed by setting the digits option. It only affects the precision of printed numbers.

The differences arise because not all decimals are represented to arbitrary precision in the binary numeral system.
Vectors of class logical

Example of a logical vector.

```r
logical_vector <- integer_vector > 4
logical_vector
```

```r
# [1] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

```r
class(logical_vector)
```

```r
# [1] "logical"
```

How many percent (%) of the components of `integer_vector` are (strictly) larger than 4?

```r
mean(integer_vector > 4) * 100
```

```r
# [1] 60
```
Vectors of class `character`

Example of a character vector.

```r
character_vector <- c("A", "vector", "of", "length", 6, ".")
character_vector
```

```
[1] "A"  "vector"  "of"  "length"  "6"  "."
```

```r
class(character_vector)
```

```
[1] "character"
```

```r
length(character_vector)
```

```
[1] 6
```

- Observe that the number was coerced to a string.
Vectors of class `factor`

Example of a factor.

```r
factor_vector <- factor(c("m", "m", "f", "m", "f", "m", "m", "f", "f", "f", "m"))
factor_vector
```

```
[1] m m f m f m f f m
Levels: f m
```

```r
class(factor_vector)
```

```
[1] "factor"
```

- Might look like a character vector but based on an integer vector.

```r
typeof(factor_vector)
```

```
[1] "integer"
```

- Can only take a fixed set of values (default: values present in data).
- Possibility of reordering levels convenient e.g. for plots.
Vectors of class `Date`

Example of a date vector.

```r
date_vector <- seq(Sys.Date(), length.out = 4, by = "quarter")
date_vector
```
```
[1] "2021-09-06" "2021-12-06" "2022-03-06" "2022-06-06"
```

```r
class(date_vector)
```
```
[1] "Date"
```

- Date vectors might also look like character vectors, but they are based on numeric vectors.
Example of a list (with named elements).

```r
my_list <- list(my_integers = integer_vector, my_factor = factor_vector,
               my_logicals = logical_vector, my_dates = date_vector)
my_list
```

```r
## $my_integers
## [1] 1 2 3 4 5 6 7 8 9 10
##
## $my_factor
## [1] m m f m f m f f m
## Levels: f m
##
## $my_logicals
## [1] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##
## $my_dates
## [1] "2021-09-06" "2021-12-06" "2022-03-06" "2022-06-06"
```

```r
length(my_list)
```

```r
## [1] 4
```
Subsetting lists

Example of a sublist.

```r
my_first_sublist <- my_list[1:3]
my_first_sublist
```

```r
## $my_integers
##  [1] 1 2 3 4 5 6 7 8 9 10
##
## $my_factor
##  [1] m m f m f m f f f m
## Levels: f m
##
## $my_logicals
##  [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

```r
class(my_first_sublist)
```

```r
## [1] "list"
```
Subsetting lists

Example of another sublist.

```r
my_second_sublist <- my_list[4]
my_second_sublist

#> $my_dates
#> [1] "2021-09-06" "2021-12-06" "2022-03-06" "2022-06-06"

class(my_second_sublist)

#> [1] "list"
```
Subsetting lists

Extracting components from a list.

dates_from_list <- my_list$my_dates
dates_from_list

## [1] "2021-09-06" "2021-12-06" "2022-03-06" "2022-06-06"

class(dates_from_list)

## [1] "Date"

The original date vector and the one extracted from the list are identical:

identical(date_vector, dates_from_list)

## [1] TRUE
Exercise

Run the following code and explain the differences/similarities

```r
x1 <- list(x = c(-1, 0, 1, 2))
x1[1]
x1[[1]]
x1$x
```
Data frames

Example of a data frame.

```
my_data_frame <- as.data.frame(my_first_sublist)
my_data_frame
```

```
##    my_integers my_factor my_logicals
## 1      1     m      FALSE
## 2      2     m      FALSE
## 3      3     f      FALSE
## 4      4     m      FALSE
## 5      5     f       TRUE
## 6      6     m       TRUE
## 7      7     f       TRUE
## 8      8     f       TRUE
## 9      9     f       TRUE
##10     10    m       TRUE
```

```
class(my_data_frame)
```

```
[1] "data.frame"
```

• Why would `as.data.frame(my_list)` give an error?
Subsetting data frames

Extracting one column from a data frame by name.

```
my_data_frame$my_integers
```
```
[1] 1 2 3 4 5 6 7 8 9 10
```

Extracting one column from a data frame using that it is a list.

```
my_data_frame[[1]]
```
```
[1] 1 2 3 4 5 6 7 8 9 10
```

Extracting one column from a data frame using data frame subsetting.

```
my_data_frame[, 1]
```
```
[1] 1 2 3 4 5 6 7 8 9 10
```
Subsetting data frames

You can subset data frames using brackets

```r
my_data_frame[5, 1]
```

```r
[[1] 5
```

... and you can assign values to entries using brackets such as

```r
my_data_frame[5, 1] <- NA
my_data_frame[1, 3] <- NA
```
Functions

We introduced missing values in our data frame above. Suppose we want to impute a value whenever it's missing.

```r
within(my_data_frame, my_integers <- ifelse(is.na(my_integers), 0, my_integers))
```

<table>
<thead>
<tr>
<th>#</th>
<th>my_integers</th>
<th>my_factor</th>
<th>my_logicals</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>m</td>
<td>NA</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>m</td>
<td>FALSE</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>f</td>
<td>FALSE</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>m</td>
<td>FALSE</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>f</td>
<td>TRUE</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>m</td>
<td>TRUE</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>f</td>
<td>TRUE</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>f</td>
<td>TRUE</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>f</td>
<td>TRUE</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>m</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

However, this is a bit cumbersum to copy-paste around, and it's not totally clear what this code does. Let's write a function with a good name.

```r
impute_missing <- function(x)
    ifelse(is.na(x), 0, x)
```
```r
within(my_data_frame, my_integers <- impute_missing(my_integers))
```

<table>
<thead>
<tr>
<th></th>
<th>my_integers</th>
<th>my_factor</th>
<th>my_logicals</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>m</td>
<td>NA</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>m</td>
<td>FALSE</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>f</td>
<td>FALSE</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>m</td>
<td>FALSE</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>f</td>
<td>TRUE</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>m</td>
<td>TRUE</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>f</td>
<td>TRUE</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>f</td>
<td>TRUE</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>f</td>
<td>TRUE</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>m</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
Exercise

Write a function, \texttt{is\_pos}, which takes a vector \texttt{x} as argument and returns a vector of logicals whose i-th entry is \texttt{TRUE} whenever \texttt{x[i] > 0}. E.g.

\begin{verbatim}
\texttt{is\_pos(c(-1, 0, 1, 2))}

\texttt{# [1] FALSE FALSE TRUE TRUE}
\end{verbatim}
Functions

What happens if we apply the `impute_missing` to other columns?

```r
impute_missing

## function(x)
##  ifelse(is.na(x), 0, x)

within(my_data_frame, my_logicals <- impute_missing(my_logicals))

##     my_integers my_factor my_logicals
## 1         1       m         0
## 2         2       m         0
## 3         3       f         0
## 4         4       m         0
## 5        NA       f         1
## 6         6       m         1
## 7         7       f         1
## 8         8       f         1
## 9         9       f         1
## 10       10       m         1
```
Better impute function

```r
impute_missing <- function(x, value = 0) {
    ifelse(is.na(x), value, x)
}

within(my_data_frame, {
    my_integers <- impute_missing(my_integers);
    my逻gicals <- impute_missing(my逻gicals, value = FALSE))
```

<table>
<thead>
<tr>
<th>my_integers</th>
<th>my_factor</th>
<th>my逻gicals</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>m</td>
<td>FALSE</td>
</tr>
<tr>
<td>2</td>
<td>m</td>
<td>FALSE</td>
</tr>
<tr>
<td>3</td>
<td>f</td>
<td>FALSE</td>
</tr>
<tr>
<td>4</td>
<td>m</td>
<td>FALSE</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
<td>TRUE</td>
</tr>
<tr>
<td>6</td>
<td>m</td>
<td>TRUE</td>
</tr>
<tr>
<td>7</td>
<td>f</td>
<td>TRUE</td>
</tr>
<tr>
<td>8</td>
<td>f</td>
<td>TRUE</td>
</tr>
<tr>
<td>9</td>
<td>f</td>
<td>TRUE</td>
</tr>
<tr>
<td>10</td>
<td>m</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
R programming

R functions are fundamental. They don't do anything before they are called and the call is evaluated.

An R function takes a number of arguments, and when a function call is evaluated it computes a return value.

An R program consists of a hierarchy of function calls. When the program is executed, function calls are evaluated and replaced by their return values.

Implementations of R functions are collected into source files, which can be organized into R packages.

An R script (or R Markdown document) is a collection of R function calls, which, when evaluated, compute a desired result.

R programming includes activities at many different levels of sophistication and abstraction.