Rcpp

Lecture 26

Dr. Colin Rundel

Rcpp

> The Rcpp package integrates R and C++ via R functions and a (header-only) C++ library.
>
> All underlying R types and objects, i.e., everything a `SEXP` represents internally in R, are matched to corresponding C++ objects. This covers anything from vectors, matrices or lists to environments, functions and more. Each `SEXP` variant is automatically mapped to a dedicated C++ class. For example, numeric vectors are represented as instances of the `Rcpp::NumericVector` class, environments are represented as instances of `Rcpp::Environment`, functions are represented as `Rcpp::Function`, etc ...

From "Extending R with C++: A Brief Introduction to Rcpp":

> R has always provided an application programming interface (API) for extensions. Based on the C language, it uses a number of macros and other low-level constructs to exchange data structures between the R process and any dynamically-loaded component modules authors added to it. With the introduction of the Rcpp package, and its later refinements, this process has become considerably easier yet also more robust. By now, Rcpp has become the most popular extension mechanism for R.

C++ Types

Type	Size	Description	Value Range
`bool`	1*	Logical value: `true` or `false`	`true` or `false`
`char`	8	Character (ASCII or UTF8)	± 127
`short int`	16	Small integers	$\pm 3.27 \cdot 10^4$
`int`	32	Medium integers	$\pm 2.14 \cdot 10^9$
`long int`	64	Large integers	$\pm 9.22 \cdot 10^1 8$
`float`	32	Small floating point value	$\pm 10^{-38}$ to $\pm 10^{38}$
`double`	64	Large floating point value	$\pm 10^{-308}$ to $\pm 10^{308}$

+ many many more

R types vs C++ types

All of the basic types in R are vectors by default, in C++ the types we just discussed are all scalar. So it is necessary to have one more level of abstraction to translate between the two. Rcpp provides for this with several built in classes:

C++ type (scalar)	Rcpp Class	R type (`typeof`)
`int`	`Rcpp::IntegerVector`	`integer`
`double`	`Rcpp::NumericVector`	`numeric`
`bool`	`Rcpp::LogicalVector`	`logical`
`std::string`	`Rcpp::CharacterVector`	`character`
`char`	`Rcpp::RawVector`	`raw`
`std::complex<double>`	`Rcpp::ComplexVector`	`complex`
	`Rcpp::List`	`list`
	`Rcpp::Environment`	`environment`
	`Rcpp::Function`	`function`
	`Rcpp::XPtr`	`externalptr`
	`Rcpp::S4`	`S4`

Trying things out

Rcpp provides some helpful functions for trying out simple C++ expressions (`evalCpp`), functions (`cppFunction`), or cpp files (`sourceCpp`). It is even possible to include C++ code in Rmd / qmd documents using the Rcpp engine.

```
1 evalCpp("2+2")
```

```
[1] 4
```

```
1 evalCpp("2+2") |> typeof()
```

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

```
1 evalCpp("2+2.") |> typeof()
```

```
[1] "double"
```

What's happening?

```
1 evalCpp("2+2", verbose = TRUE, rebuild = TRUE)
```

```
Generated code for function definition:
------------------------------------------------------

#include <Rcpp.h>

using namespace Rcpp;

// [[Rcpp::export]]
SEXP get_value(){ return wrap( 2+2 ) ; }

Generated extern "C" functions
------------------------------------------------------


#include <Rcpp.h>
#ifdef RCPP_USE_GLOBAL_ROSTREAM
[1] 4
```

C++ functions as R functions

```
cppFunction('
  double cpp_mean(double x, double y) {
    return (x+y)/2;
  }
')
```

```
cpp_mean
```

```
function (x, y)
.Call(<pointer: 0x1080d6694>, x, y)
```

```
cpp_mean(1,2)
```

```
[1] 1.5
```

```
cpp_mean(TRUE,2L)
```

```
[1] 1.5
```

```
cpp_mean(1,"A")
```

```
Error in eval(expr, envir, enclos): Not compatible
with requested type: [type=character;
target=double].
```

```
cpp_mean(c(1,2), c(1,2))
```

```
Error in eval(expr, envir, enclos): Expecting a
single value: [extent=2].
```

Using `sourceCpp`

This allows for an entire `.cpp` source file to be compiled and loaded into R. This is generally the preferred way of working with C++ code and is well supported by RStudio (i.e. provides syntax highlights, tab completion, etc.)

- Make sure to include the Rcpp header

```
#include <Rcpp.h>
```

- If you hate typing `Rcpp::` everywhere, include the namespace

```
using namespace Rcpp;
```

- Specify any desired plugins with

```
// [[Rcpp::plugins(cpp11)]]
```

- Prefix any functions that will be exported with R with

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

- Testing code can be included using an R code block:

```
/*** R
# This R code will be run automatically
*/
```

Example

The following would be avaiable as a file called `mean.cpp` or similar.

```cpp
#include <Rcpp.h>

//[[Rcpp::plugins(cpp11)]]

//[[Rcpp::export]]
double cpp_mean(double x, double y) {
  return (x+y)/2;
}

/*** R
bench::mark(
  cpp_mean(1, 2),
  mean(c(1, 2))
)
*/
```

```r
sourceCpp("mean.cpp")
```

```
> bench::mark(cpp_mean(1, 2), mean(c(1, 2)))
# A tibble: 2 × 13
  expression          min   median `itr/sec` mem_alloc `gc/sec` n_itr  n_gc
  <bch:expr>     <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl> <int> <dbl>
1 cpp_mean(1, 2) 615.02ns 737.96ns  1029492.    2.49KB        0  10000     0
2 mean(c(1, 2))    1.23µs   1.44µs   662329.        0B      132.  9998     2
# ℹ 5 more variables: total_time <bch:tm>, result <list>, memory <list>,
#   time <list>, gc <list>
```

`for` loops

In C & C++ `for` loops are traditionally constructed as,

```
for(initialization; end condition; increment) {
  //...loop code ..
}
```

```
#include <Rcpp.h>

//[[Rcpp::export]]
double cpp_mean(Rcpp::NumericVector x) {
  double sum = 0.0;
  for(int i=0; i != x.size(); i++) {
    sum += x[i];
  }
  return sum/x.size();
}
```

```
cpp_mean(1:10)
```

```
[1] 5.5
```

Range based for loops (C++11)

Since the adoption of the C++11 standard there is an alternative for loop syntax,

```
#include <Rcpp.h>
//[[Rcpp::plugins(cpp11)]]

//[[Rcpp::export]]
double cpp11_mean(Rcpp::NumericVector x) {
  double sum = 0.0;
  for(auto v : x) {
    sum += v;
  }

  return sum/x.size();
}
```

```
cpp11_mean(1:10)
```

```
[1] 5.5
```

Available plugins?

```
1 ls(envir = Rcpp:::.plugins)
```

```
 [1] "cpp0x"          "cpp11"          "cpp14"          "cpp17"
 [5] "cpp1y"          "cpp1z"          "cpp20"          "cpp23"
 [9] "cpp2a"          "cpp2b"          "cpp98"          "openmp"
[13] "unwindProtect"
```

Rcpp Sugar

Rcpp also attempts to provide many of the base R functions within the C++ scope, generally these are referred to as Rcpp Sugar, more can be found here or by examining the Rcpp source.

```
#include <Rcpp.h>
//[[Rcpp::plugins(cpp11)]]

//[[Rcpp::export]]
double rcpp_mean(Rcpp::NumericVector x) {
  return Rcpp::mean(x);
}
```

```
rcpp_mean(1:10)
```

```
[1] 5.5
```

Edge cases

```
1 x = c(1:10,NA)
2 typeof(x)
```

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

```
1 mean(x)
```

```
[1] NA
```

```
1 cpp_mean(x)
```

```
[1] NA
```

```
1 cpp11_mean(x)
```

```
[1] NA
```

```
1 rcpp_mean(x)
```

```
[1] NA
```

```
1 x = c(1:10,NA_real_)
2 typeof(x)
```

```
[1] "double"
```

```
1 mean(x)
```

```
[1] NA
```

```
1 cpp_mean(x)
```

```
[1] NA
```

```
1 cpp11_mean(x)
```

```
[1] NA
```

```
1 rcpp_mean(x)
```

```
[1] NA
```

```
1 y = c(1:10,Inf)
2 typeof(y)
```

```
[1] "double"
```

```
1 mean(y)
```

```
[1] Inf
```

```
1 cpp_mean(y)
```

```
[1] Inf
```

```
1 cpp11_mean(y)
```

```
[1] Inf
```

```
1 rcpp_mean(y)
```

```
[1] Inf
```

Integer mean

```cpp
#include <Rcpp.h>
//[[Rcpp::plugins(cpp11)]]

//[[Rcpp::export]]
double cpp_imean(Rcpp::IntegerVector x) {
  double sum = 0.0;
  for(int i=0; i != x.size(); i++) {
    sum += x[i];
  }

  return sum/x.size();
}

//[[Rcpp::export]]
double cpp11_imean(Rcpp::IntegerVector x) {
  double sum = 0.0;
  for(auto v : x) {
    sum += v;
  }

  return sum/x.size();
}

```

Integer edge cases

```
x = c(1:10,NA)
typeof(x)
```

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

```
mean(x)
```

```
[1] NA
```

```
cpp_imean(x)
```

```
[1] -195225781
```

```
cpp11_imean(x)
```

```
[1] -195225781
```

```
rcpp_imean(x)
```

```
[1] NA
```

```
x = c(1:10,NA_real_)
typeof(x)
```

```
[1] "double"
```

```
mean(x)
```

```
[1] NA
```

```
cpp_imean(x)
```

```
[1] -195225781
```

```
cpp11_imean(x)
```

```
[1] -195225781
```

```
rcpp_imean(x)
```

```
[1] NA
```

```
y = c(1:10,Inf)
typeof(y)
```

```
[1] "double"
```

```
mean(y)
```

```
[1] Inf
```

```
cpp_imean(y)
```

```
[1] -195225781
```

```
cpp11_imean(y)
```

```
[1] -195225781
```

```
rcpp_imean(y)
```

```
[1] NA
```

Missing values - C++ Scalars

From Hadley's Adv-R Rcpp chapter,

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::List scalar_missings() {
  int int_s          = NA_INTEGER;
  Rcpp::String chr_s = NA_STRING;
  bool lgl_s         = NA_LOGICAL;
  double num_s       = NA_REAL;

  return Rcpp::List::create(int_s, chr_s, lgl_s, num_s);
}
```

```r
scalar_missings() |> str()
```

```
List of 4
 $ : int NA
 $ : chr NA
 $ : logi TRUE
 $ : num NA
```

Missing values - Rcpp Vectors

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::List vector_missing() {
  return Rcpp::List::create(
    Rcpp::NumericVector::create(NA_REAL),
    Rcpp::IntegerVector::create(NA_INTEGER),
    Rcpp::LogicalVector::create(NA_LOGICAL),
    Rcpp::CharacterVector::create(NA_STRING)
  );
}
```

```r
vector_missing() |> str()
```

```
List of 4
 $ : num NA
 $ : int NA
 $ : logi NA
 $ : chr NA
```

Performance

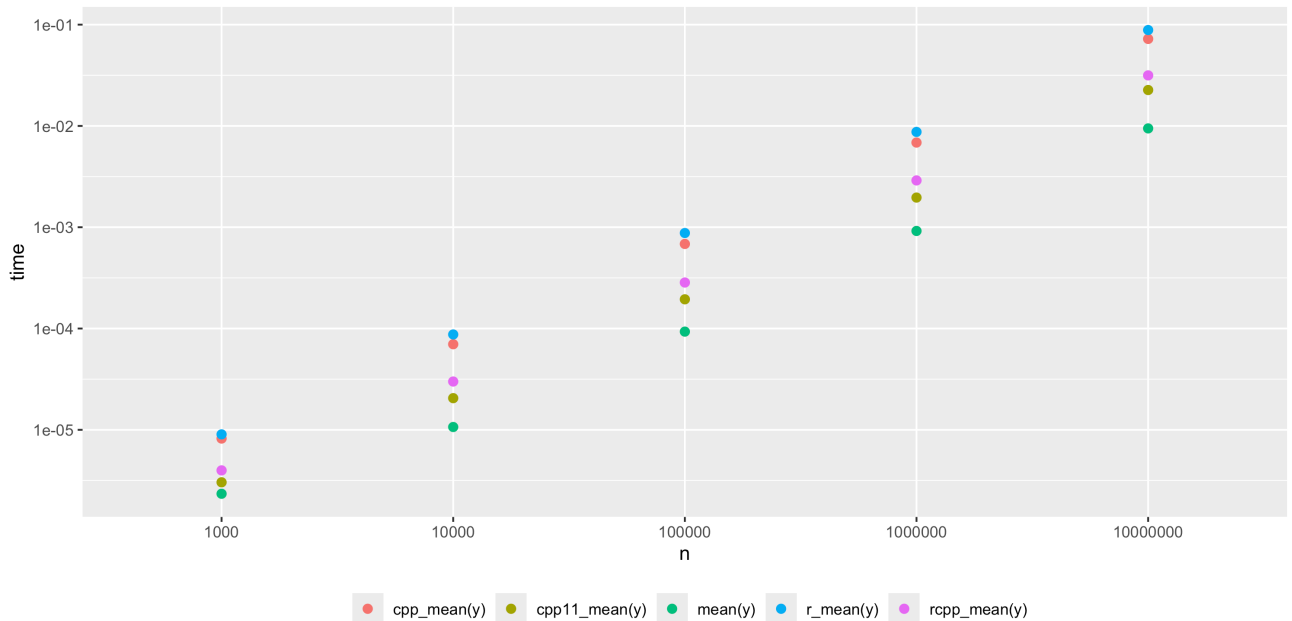
```
r_mean = function(x) {
  sum = 0
  for(v in x) {
    sum = sum + v
  }
  sum / length(x)
}
```

```
y = seq_len(1e6)
bench::mark(
  mean(y),
  cpp_mean(y),
  cpp11_mean(y),
  rcpp_mean(y),
  r_mean(y)
)
```

```
# A tibble: 5 × 6
  expression          min   median `itr/sec` mem_alloc `gc/sec`
  <bch:expr>     <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
1 mean(y)          1.83ms   1.85ms     537.         0B      0
2 cpp_mean(y)      7.31ms   7.44ms     133.      7.63MB     27.2
3 cpp11_mean(y)    1.51ms   1.65ms     598.      7.63MB     87.8
4 rcpp_mean(y)      2.5ms   2.63ms     377.      7.63MB     37.7
5 r_mean(y)        9.81ms  10.25ms      97.7   23.47KB      0
```

bench::press

```
b = bench::press(
  n = 10^c(3:7),
  {
    y = sample(seq_le
    bench::mark(
      mean(y),
      cpp_mean(y),
      cpp11_mean(y),
      rcpp_mean(y),
      r_mean(y)
    )
  }
)
```



Creating a `list`

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::List make_list(int n) {
  return Rcpp::List::create(
    Rcpp::Named("norm") = Rcpp::rnorm(n, 0, 1),
    Rcpp::Named("beta") = Rcpp::rbeta(n, 1, 1),
    Rcpp::IntegerVector::create(1,2,3,4,5, NA_INTEGER)
  );
}
```

```cpp
make_list(10)
```

```
$norm
 [1]  0.98312668  0.21998356  0.01714112  1.29657530 -0.26511154 -1.70247852
 [7]  0.18258180 -0.64859820  0.29851640 -0.71617469

$beta
 [1] 0.76166310 0.76807235 0.75090200 0.58571498 0.06224161 0.96814944
 [7] 0.59443714 0.73810580 0.85443281 0.87490400

[[3]]
[1]  1  2  3  4  5 NA
```

Creating a `data.frame`

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::DataFrame make_df(int n) {
  return Rcpp::DataFrame::create(
    Rcpp::Named("norm") = Rcpp::rnorm(n, 0, 1),
    Rcpp::Named("beta") = Rcpp::rbeta(n, 1, 1)
  );
}
```

```cpp
make_df(10)
```

```
         norm       beta
1  -0.6338848 0.50580132
2   0.3762964 0.91396703
3   1.1612786 0.43635160
4  -1.1622494 0.15703795
5  -0.6473185 0.08762308
6  -1.4911285 0.03178622
7   1.5714742 0.39267113
8  -0.7449894 0.67658924
9  -0.6633967 0.63732888
10  0.4678779 0.50797668
```

Creating a `tbl`

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::DataFrame make_tbl(int n) {
  Rcpp::DataFrame df = Rcpp::DataFrame::create(
    Rcpp::Named("norm") = Rcpp::rnorm(n, 0, 1),
    Rcpp::Named("beta") = Rcpp::rbeta(n, 1, 1)
  );
  df.attr("class") = Rcpp::CharacterVector::create("tbl_df", "tbl", "data.frame");

  return df;
}
```

```cpp
make_tbl(10)
```

```
# A tibble: 10 × 2
     norm  beta
    <dbl> <dbl>
 1  2.62  0.213
 2  0.506 0.220
 3  1.00  0.856
 4 -0.850 0.347
 5  1.85  0.247
 6  1.25  0.394
 7 -0.335 0.388
 8  0.400 0.702
```

Printing

R has some weird behavior when it comes to printing text from C++, Rcpp has function that resolves this, `Rcout`

```
#include <Rcpp.h>

// [[Rcpp::export]]
void n_hello(int n) {
  for(int i=0; i!=n; ++i) {
    Rcpp::Rcout << i+1 << ". Hello world!\n";
  }
}
```

```
n_hello(5)
```

```
1. Hello world!
2. Hello world!
3. Hello world!
4. Hello world!
5. Hello world!
```

Printing NAs

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
void print_na() {
  Rcpp::Rcout << "NA_INTEGER : " << NA_INTEGER << "\n";
  Rcpp::Rcout << "NA_STRING  : " << NA_STRING  << "\n";
  Rcpp::Rcout << "NA_LOGICAL : " << NA_LOGICAL << "\n";
  Rcpp::Rcout << "NA_REAL    : " << NA_REAL    << "\n";
}
```

```cpp
print_na()
```

```
NA_INTEGER : -2147483648
NA_STRING  : 0x13a013400
NA_LOGICAL : -2147483648
NA_REAL    : nan
```

SEXP Conversion

Rcpp attributes provides a bunch of convenience tools that handle much of the conversion from R SEXP's to C++ / Rcpp types and back. Some times it is necessary to handle this directly.

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
SEXP as_wrap(SEXP input) {
  Rcpp::NumericVector r = Rcpp::as<Rcpp::NumericVector>(input);
  Rcpp::NumericVector rev_r = Rcpp::rev(r);

  return Rcpp::wrap(rev_r);
}
```

```r
as_wrap(1:10)
```

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

```r
as_wrap(c(1,2,3))
```

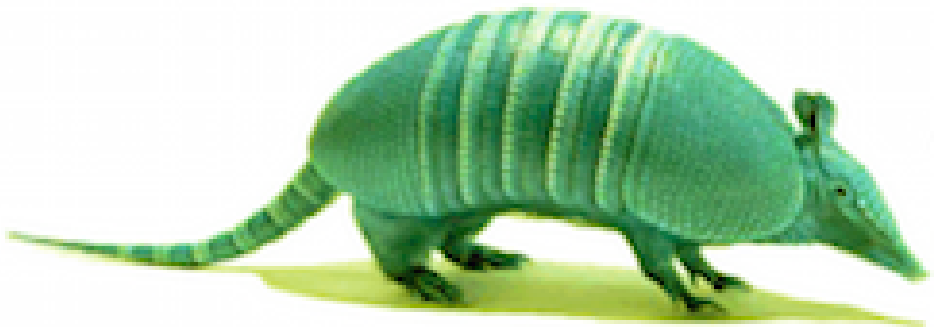
```
[1] 3 2 1
```

```r
as_wrap(c("A","B","C"))
```

```
Error in eval(expr, envir, enclos): Not compatible
with requested type: [type=character;
target=double].
```

RcppArmadillo

Armadillo



Armadillo

C++ linear algebra library

- Developed by Dr. Conrad Sanderson and Dr Ryan Curtin
- Template based linear algebra library with high level syntax (like R or Matlab)
- Heavy lifting is (mostly) handled by LAPACK (i.e. benefits from OpenBLAS)
- Supports vectors, matrices, and cubes in dense or sparse format
- Some builtin expression optimization via template meta-programming
- Header only or shared library versions available

Basic types

Armadillo has 4 basic (dense) templated types:

```
arma::Col<type>,
arma::Row<type>,
arma::Mat<type>,
arma::Cube<type>
```

These types can be specialized using one of the following data types:

```
float, double,
std::complex<float>, std::complex<double>,
short, int, long,
unsigned short, unsigned int, unsigned long
```

typedef Shortcuts

For convenience the following typedefs are defined:

- Vectors:

```
arma::vec     = arma::colvec     =  arma::Col<double>
arma::dvec    = arma::dcolvec    =  arma::Col<double>
arma::fvec    = arma::fcolvec    =  arma::Col<float>
arma::cx_vec  = arma::cx_colvec  =  arma::Col<cx_double>
arma::cx_dvec = arma::cx_dcolvec =  arma::Col<cx_double>
arma::cx_fvec = arma::cx_fcolvec =  arma::Col<cx_float>
arma::uvec    = arma::ucolvec    =  arma::Col<uword>
arma::ivec    = arma::icolvec    =  arma::Col<sword>
```

- Matrices

```
arma::mat     = arma::Mat<double>
arma::dmat    = arma::Mat<double>
arma::fmat    = arma::Mat<float>
arma::cx_mat  = arma::Mat<cx_double>
arma::cx_dmat = arma::Mat<cx_double>
arma::cx_fmat = arma::Mat<cx_float>
arma::umat    = arma::Mat<uword>
arma::imat    = arma::Mat<sword>
```

RcppArmadillo

- Written and maintained by Dirk Eddelbuettel, Romain Francois, Doug Bates and Binxiang Ni
- Provides the header only version of Armadillo along with additional wrappers
  - Wrappers provide easy conversion between Rcpp types and Armadillo types
  - Enables use of Rcpp attributes and related tools
- Requirements - include the following in your C++ code

```
1 // [[Rcpp::depends(RcppArmadillo)]]
2 #include <RcppArmadillo.h>
```

Example Program

```cpp
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>

// [[Rcpp::export]]
arma::mat test_randu(int n, int m) {
  arma::mat A = arma::randu<arma::mat>(n,m);
  return A;
}
```

```r
test_randu(4,5)
```

```
          [,1]      [,2]        [,3]       [,4]
[,5]
[1,] 0.6909937 0.4662218 0.649733796 0.77984086
0.6041882
[2,] 0.9930286 0.3925346 0.888608006 0.54199592
0.3662756
[3,] 0.2877681 0.2688936 0.001710504 0.69397778
0.2114039
[4,] 0.1303068 0.9759141 0.042376122 0.01108976
0.1026822
```

```r
test_randu(3,1)
```

```
           [,1]
[1,] 0.07826737
[2,] 0.45310367
[3,] 0.25307288
```

arma class attributes

Attribute	Description
`.n_rows`	number of rows; present in Mat, Col, Row, Cube, field and SpMat
`.n_cols`	number of columns; present in Mat, Col, Row, Cube, field and SpMat
`.n_elem`	total number of elements; present in Mat, Col, Row, Cube, field and SpMat
`.n_slices`	number of slices; present in Cube and field

```cpp
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>

// [[Rcpp::export]]
void test_attr(arma::mat m) {
  Rcpp::Rcout << "m.n_rows = " << m.n_rows << "\n";
  Rcpp::Rcout << "m.n_cols = " << m.n_cols << "\n";
  Rcpp::Rcout << "m.n_elem = " << m.n_elem << "\n";
}
```

```r
test_attr(matrix(0, 3, 3))
```

```
m.n_rows = 3
m.n_cols = 3
m.n_elem = 9
```

```r
test_attr(matrix(1, 4, 5))
```

```
m.n_rows = 4
m.n_cols = 5
m.n_elem = 20
```

```r
test_attr(1:10)
```

```
Error in eval(expr, envir, enclos):
Not a matrix.
```

```r
test_attr(as.matrix(1:10))
```

```
m.n_rows = 10
m.n_cols = 1
m.n_elem = 10
```

Element access

For an `arma::vec v`,

Call	Description
`v(i)`	Access the `i`-th element with bounds checking
`v.at(i)`	Access the `i`-th element without bounds checking
`v[i]`	Access the `i`-th element without bounds checking

For an `arma::mat m`,

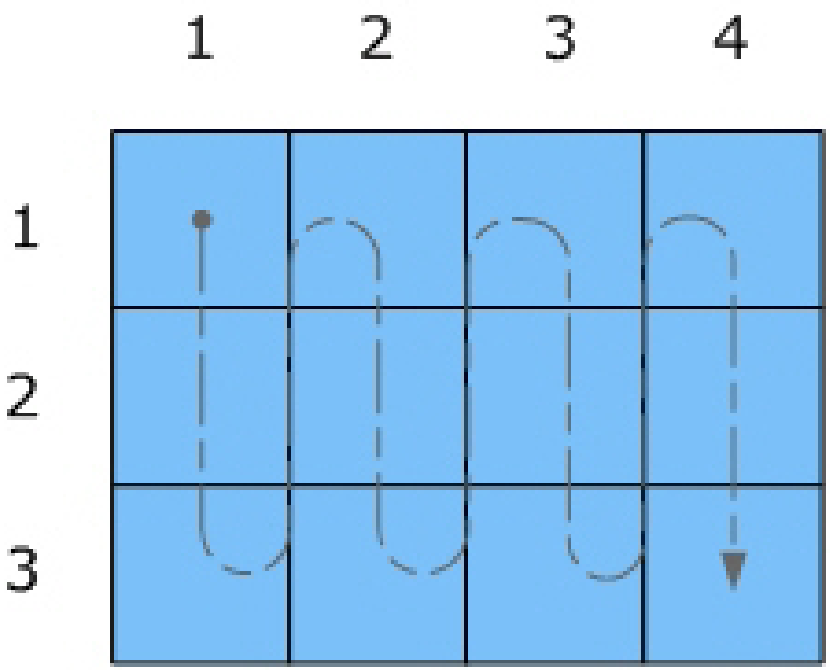
Call	Description
`m(i)`	Access the `i`-th element, treating object as flat and in column major order
`m(i,j)`	Access the element in `i`-th row and `j`-th column with bounds checking
`m.at(i,j)`	Access the element in `i`-th row and `j`-th column without bounds checking

Element access - Cubes

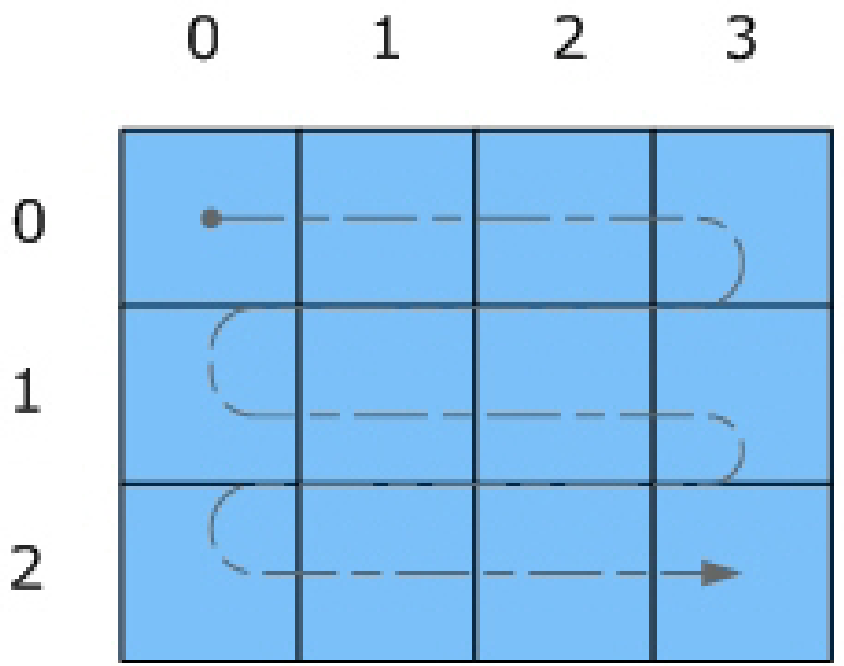
For an `arma::cube c`,

Call	Description
`c(i)`	Access the i-th element, treating object as flat and in column major order
`c(i,j,k)`	Access the element in `i`-th row, `j`-th column, and `k`-th slice with bounds checking
`c.at(i,j,k)`	Access the element in `i`-th row, `j`-th column, and `k`-th slice without bounds checking

Data Organization



A: Column-major order (Fortran-style)



B: Row-major order (C-style)

```cpp
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>

// [[Rcpp::export]]
void test_order(arma::mat m) {
  for(int i=0; i!=m.n_elem; ++i) {
    Rcpp::Rcout << m(i) << " ";
  }
  Rcpp::Rcout << "\n";
}
```

```r
m = matrix(1:9, 3, 3)
```

```r
c(m)
```

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

```r
test_order(m)
```

```
1 2 3 4 5 6 7 8 9
```

```r
c(t(m))
```

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

```r
test_order(t(m))
```

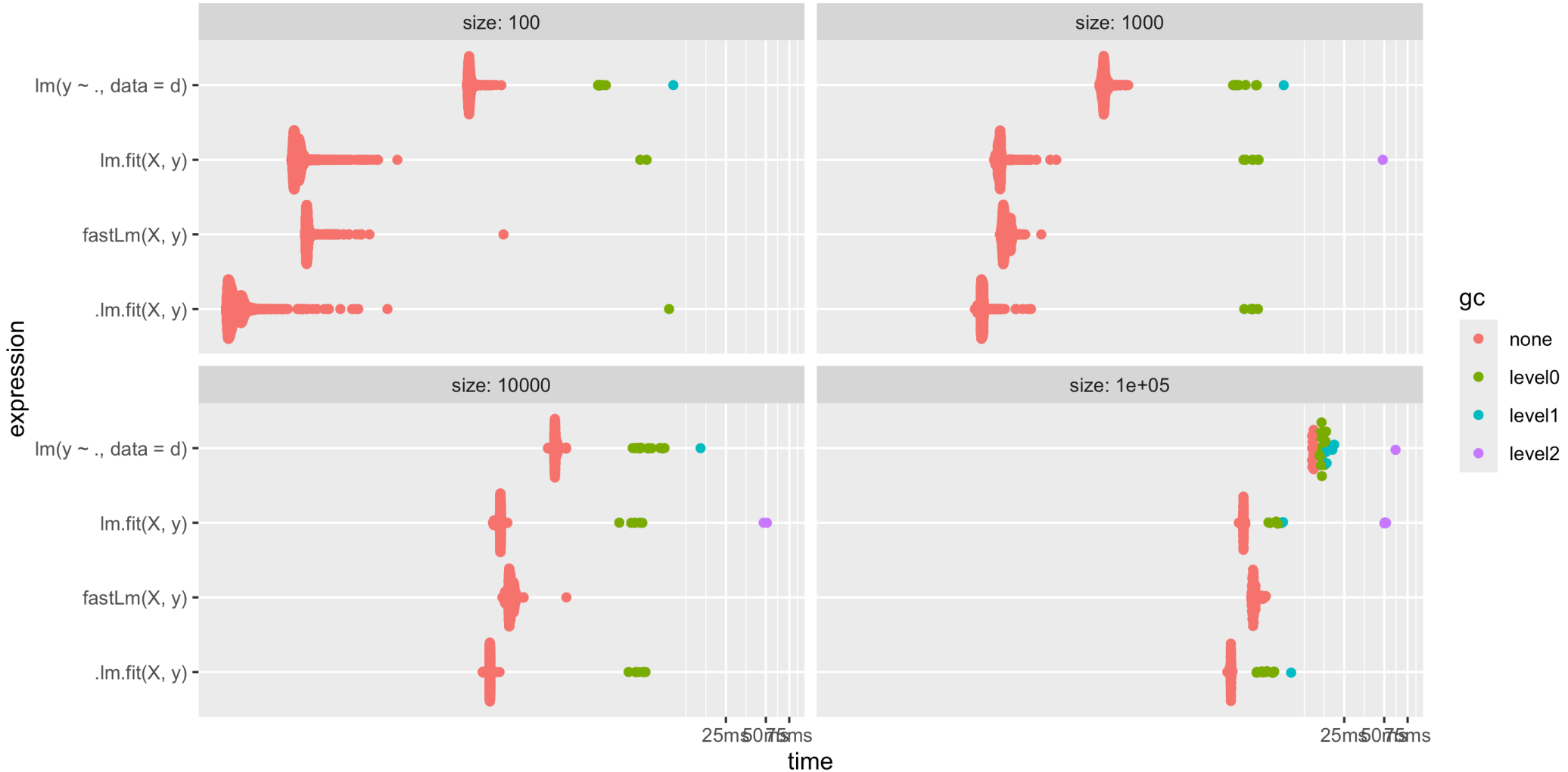
```
1 4 7 2 5 8 3 6 9
```

`fastLm` example

```cpp
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>

// [[Rcpp::export]]
Rcpp::List fastLm(const arma::mat& X, const arma::colvec& y) {
    int n = X.n_rows, k = X.n_cols;

    arma::colvec coef = arma::solve(X, y);    // fit model y ~ X
    arma::colvec res  = y - X*coef;           // residuals

    // std.errors of coefficients
    double s2 = std::inner_product(res.begin(), res.end(), res.begin(), 0.0)/

    arma::colvec std_err = arma::sqrt(s2 * arma::diagvec(arma::pinv(arma::tra

    return Rcpp::List::create(
      Rcpp::Named("coefficients") = coef,
      Rcpp::Named("stderr")       = std_err,
      Rcpp::Named("df.residual")  = n - k
```

```r
library(dplyr)
n=1e5
d = tibble(
  x1 = rnorm(n),
  x2 = rnorm(n),
  x3 = rnorm(n),
  x4 = rnorm(n),
  x5 = rnorm(n),
) %>%
  mutate(
    y = 3 + x1 - x2 + 2*x3 -2*x4 + 3*x5
  )
```

```r
res = bench::press(
  size = c(100, 1000, 10000, 100000),
  {
    d = d[seq_len(size),]
    X = model.matrix(y ~ ., d)
    y = as.matrix(d$y)

    bench::mark(
      lm(y~., data=d),
      lm.fit(X,y),
      .lm.fit(X,y),
      fastLm(X,y),
      check = FALSE
    )
  }
)
```


size: 100
size: 1000
size: 10000
size: 1e+05
lm(y ~ ., data = d)
lm.fit(X, y)
fastLm(X, y)
.lm.fit(X, y)
expression
25ms
50ms
75ms
time
gc
none
level0
level1
level2

MVN Example

Multivariate Normal Distribution - Review

For an n-dimension multivate normal distribution with covariance $\boldsymbol{\Sigma}$ can be written as

$$\underset{n\times 1}{\boldsymbol{Y}} \sim \mathrm{N}(\underset{n\times 1}{\boldsymbol{\mu}}, \underset{n\times n}{\boldsymbol{\Sigma}})$$

where $\left\{\boldsymbol{\Sigma}\right\}_{ij} = \rho_{ij}\sigma_i\sigma_j$

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \sim \mathrm{N}\left(\begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_n \end{pmatrix}, \begin{pmatrix} \rho_{11}\sigma_1\sigma_1 & \rho_{12}\sigma_1\sigma_2 & \cdots & \rho_{1n}\sigma_1\sigma_n \\ \rho_{21}\sigma_2\sigma_1 & \rho_{22}\sigma_2\sigma_2 & \cdots & \rho_{2n}\sigma_2\sigma_n \\ \vdots & \vdots & \ddots & \vdots \\ \rho_{n1}\sigma_n\sigma_1 & \rho_{n2}\sigma_n\sigma_2 & \cdots & \rho_{nn}\sigma_n\sigma_n \end{pmatrix}\right)$$

Multivariate Normal Distribution - Sampling

To generate draws from an n-dimensional multivate normal with mean $\underset{n\times 1}{\boldsymbol{\mu}}$ and covariance matrix $\underset{n\times n}{\boldsymbol{\Sigma}}$,

- Find a matrix $\underset{n\times n}{\boldsymbol{A}}$ such that $\boldsymbol{\Sigma} = \boldsymbol{A}\,\boldsymbol{A}^{t}$
  - most often we use $\boldsymbol{A} = \text{Chol}(\boldsymbol{\Sigma})$ where $\boldsymbol{A}$ is a lower triangular matrix.
- Draw n iid unit normals, $N(0, 1)$, as $\underset{n\times 1}{\boldsymbol{z}}$
- Obtain multivariate normal draws using

$$\underset{n\times 1}{\boldsymbol{y}} = \underset{n\times 1}{\boldsymbol{\mu}} + \underset{n\times n}{\boldsymbol{A}}\ \underset{n\times 1}{\boldsymbol{z}}$$

Multivariate Normal Distribution - Density

For the n dimensional multivate normal given on the last slide, its density is given by

$$\mathrm{f}\left(\boldsymbol{Y}|\boldsymbol{\mu},\boldsymbol{\Sigma}\right) = (2\pi)^{-\mathrm{n}/2} \ \det(\boldsymbol{\Sigma})^{-1/2} \ \exp\left(-\frac{1}{2}\underset{1\times\mathrm{n}}{(\boldsymbol{Y}-\boldsymbol{\mu})'}\underset{\mathrm{n}\times\mathrm{n}}{\boldsymbol{\Sigma}^{-1}}\underset{\mathrm{n}\times 1}{(\boldsymbol{Y}-\boldsymbol{\mu})}\right)$$

and its log density is given by

$$\log \mathrm{f}\left(\boldsymbol{Y}|\boldsymbol{\mu},\boldsymbol{\Sigma}\right) = -\frac{\mathrm{n}}{2}\log 2\pi - \frac{1}{2}\log\det(\boldsymbol{\Sigma}) - \frac{1}{2}\underset{1\times\mathrm{n}}{(\boldsymbol{Y}-\boldsymbol{\mu})'}\underset{\mathrm{n}\times\mathrm{n}}{\boldsymbol{\Sigma}^{-1}}\underset{\mathrm{n}\times 1}{(\boldsymbol{Y}-\boldsymbol{\mu})}$$