

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 (bits)	Description	Value Range
<code>bool</code>	1*	Logical value: <code>true</code> or <code>false</code>	<code>true</code> or <code>false</code>
<code>char</code>	8	Character (ASCII or UTF8)	± 127
<code>short int</code>	16	Small integers	$\pm 3.27 \cdot 10^4$
<code>int</code>	32	Medium integers	$\pm 2.14 \cdot 10^9$
<code>long int</code>	64	Large integers	$\pm 9.22 \cdot 10^{18}$
<code>float</code>	32	Small floating point value	$\pm 10^{-38}$ to $\pm 10^{38}$
<code>double</code>	64	Large floating point value	$\pm 10^{-308}$ to $\pm 10^{308}$

+ many more

R types vs C++ types

All of the basic types in R are vectors by default, the C++ types just shown are all *scalars*. 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)
<code>int</code>	<code>Rcpp::IntegerVector</code>	<code>integer</code>
<code>double</code>	<code>Rcpp::NumericVector</code>	<code>numeric</code>
<code>bool</code>	<code>Rcpp::LogicalVector</code>	<code>logical</code>
<code>std::string</code>	<code>Rcpp::CharacterVector</code>	<code>character</code>
<code>char</code>	<code>Rcpp::RawVector</code>	<code>raw</code>
<code>std::complex<double></code>	<code>Rcpp::ComplexVector</code>	<code>complex</code>
	<code>Rcpp::List</code>	<code>list</code>
	<code>Rcpp::Environment</code>	<code>environment</code>
	<code>Rcpp::Function</code>	<code>function</code>
	<code>Rcpp::XPtr</code>	<code>externalptr</code>
	<code>Rcpp::S4</code>	<code>S4</code>

Trying things out

Rcpp provides some helpful functions for compiling and running 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
Rcpp::Rostream<true>& Rcpp::Rcout = Rcpp::Rcpp_cout_get();
Rcpp::Rostream<false>& Rcpp::Rcerr = Rcpp::Rcpp_cerr_get();
#endif

// get_value
SEXP get_value() {
  [1] 4
```

C++ functions as R functions

```
1 cppFunction('
2   double cpp_mean(double x, double y) {
3       return (x+y)/2;
4   }
5   ')
```

```
1 cpp_mean
```

```
function (x, y)
.Call(<pointer: 0x10ab51a30>, x, y)
```

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

```
[1] 1.5
```

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

```
[1] 1.5
```

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

Error: Not compatible with requested type: [type

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

Error: 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

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

- If you don't like typing `Rcpp::` everywhere, include the namespace

```
1 using namespace Rcpp;
```

- Specify any desired plugins with

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

- Prefix any functions that will be exported with R with

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

- Testing code can be included using an R code block at the end:

```
1 /** R  
2 # This R code will be run automatically  
3 */
```

Example

mean.cpp

```
1  #include <Rcpp.h>
2
3  //[[Rcpp::export]]
4  double cpp_mean(double x, double y) {
5      return (x+y)/2;
6  }
7
8  /*** R
9  bench::mark(
10     cpp_mean(1, 2),
11     mean(c(1, 2))
12 )
13 */
```

```
1 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	total_time
	<bch:expr>	<bch>	<bch:tm>	<dbl>	<bch:byt>	<dbl>	<int>	<dbl>	<bch:tm>
1	cpp_mean(1...	164ns	245.99ns	2944640.	0B	0	10000	0	3.4ms
2	mean(c(1, ...	779ns	1.23µs	792794.	0B	0	10000	0	12.6ms

```
# i 4 more variables: result <list>, memory <list>, time <list>, gc <list>
```

for loops

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

```
1 for(initialization; end condition; increment) {  
2     //...loop code ..  
3 }
```

```
1 #include <Rcpp.h>  
2  
3 //[[Rcpp::export]]  
4 double cpp_mean(Rcpp::NumericVector x) {  
5     double sum = 0.0;  
6     for(int i=0; i != x.size(); i++) {  
7         sum += x[i];  
8     }  
9     return sum/x.size();  
10 }
```

```
1 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,

```
1  #include <Rcpp.h>
2  // [[Rcpp::plugins(cpp11)]]
3
4  // [[Rcpp::export]]
5  double cpp11_mean(Rcpp::NumericVector x) {
6      double sum = 0.0;
7      for(auto v : x) {
8          sum += v;
9      }
10
11     return sum/x.size();
12 }
```

```
1  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] "cpp26"      "cpp2a"      "cpp2b"      "cpp98"  
[13] "openmp"     "unwindProtect"
```

Using `//[[Rcpp::plugins(cpp11)]]` is not strictly necessary as C++17 is now the default standard for R packages as of R

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 details can be found [here](#) or by looking at the Rcpp source.

```
1 #include <Rcpp.h>
2 // [[Rcpp::plugins(cpp11)]]
3
4 // [[Rcpp::export]]
5 double rcpp_mean(Rcpp::NumericVector x) {
6     return Rcpp::mean(x);
7 }
```

```
1 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

```
1  #include <Rcpp.h>
2  //[[Rcpp::plugins(cpp11)]]
3
4  //[[Rcpp::export]]
5  double cpp_imean(Rcpp::IntegerVector x) {
6      double sum = 0.0;
7      for(int i=0; i != x.size(); i++) {
8          sum += x[i];
9      }
10
11     return sum/x.size();
12 }
13
14 //[[Rcpp::export]]
15 double cpp11_imean(Rcpp::IntegerVector x) {
16     double sum = 0.0;
17     for(auto v : x) {
18         sum += v;
19     }
20
21     return sum/x.size();
22 }
23
24 //[[Rcpp::export]]
25 double rcpp_imean(Rcpp::IntegerVector x) {
26     return Rcpp::mean(x);
```

Integer edge cases

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

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

```
1 mean(x)
```

```
[1] NA
```

```
1 cpp_imean(x)
```

```
[1] -195225781
```

```
1 cpp11_imean(x)
```

```
[1] -195225781
```

```
1 rcpp_imean(x)
```

```
[1] NA
```

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

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

```
1 mean(x)
```

```
[1] NA
```

```
1 cpp_imean(x)
```

```
[1] -195225781
```

```
1 cpp11_imean(x)
```

```
[1] -195225781
```

```
1 rcpp_imean(x)
```

```
[1] NA
```

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

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

```
1 mean(y)
```

```
[1] Inf
```

```
1 cpp_imean(y)
```

Warning in cpp_imean(y): NAs in

```
[1] -195225781
```

```
1 cpp11_imean(y)
```

Warning in cpp11_imean(y): NAs

```
[1] -195225781
```

```
1 rcpp_imean(y)
```

Warning in rcpp_imean(y): NAs i

```
[1] NA
```

Missing values - C++ Scalars

From Hadley's Adv-R Rcpp chapter,

```
1 #include <Rcpp.h>
2
3 // [[Rcpp::export]]
4 Rcpp::List scalar_missings() {
5     int int_s      = NA_INTEGER;
6     Rcpp::String chr_s = NA_STRING;
7     bool lgl_s     = NA_LOGICAL;
8     double num_s   = NA_REAL;
9
10    return Rcpp::List::create(int_s, chr_s, lgl_s, num_s);
11 }
```

```
1 scalar_missings() |> str()
```

List of 4

\$: int NA

\$: chr NA

\$: logi TRUE

\$: num NA

Missing values - Rcpp Vectors

```
1 #include <Rcpp.h>
2
3 // [[Rcpp::export]]
4 Rcpp::List vector_missing() {
5     return Rcpp::List::create(
6         Rcpp::NumericVector::create(NA_REAL),
7         Rcpp::IntegerVector::create(NA_INTEGER),
8         Rcpp::LogicalVector::create(NA_LOGICAL),
9         Rcpp::CharacterVector::create(NA_STRING)
10    );
11 }
```

```
1 vector_missing() |> str()
```

List of 4

```
$ : num NA
$ : int NA
$ : logi NA
$ : chr NA
```

Performance

```
1 r_mean = function(x) {  
2   sum = 0  
3   for(v in x) {  
4     sum = sum + v  
5   }  
6   sum / length(x)  
7 }
```

```
1 y = rnorm(1e6)  
2 bench::mark(  
3   mean(y), r_mean(y), cpp_mean(y), cpp11_mean(y), rcpp_mean(y)  
4 )
```

A tibble: 5 × 6

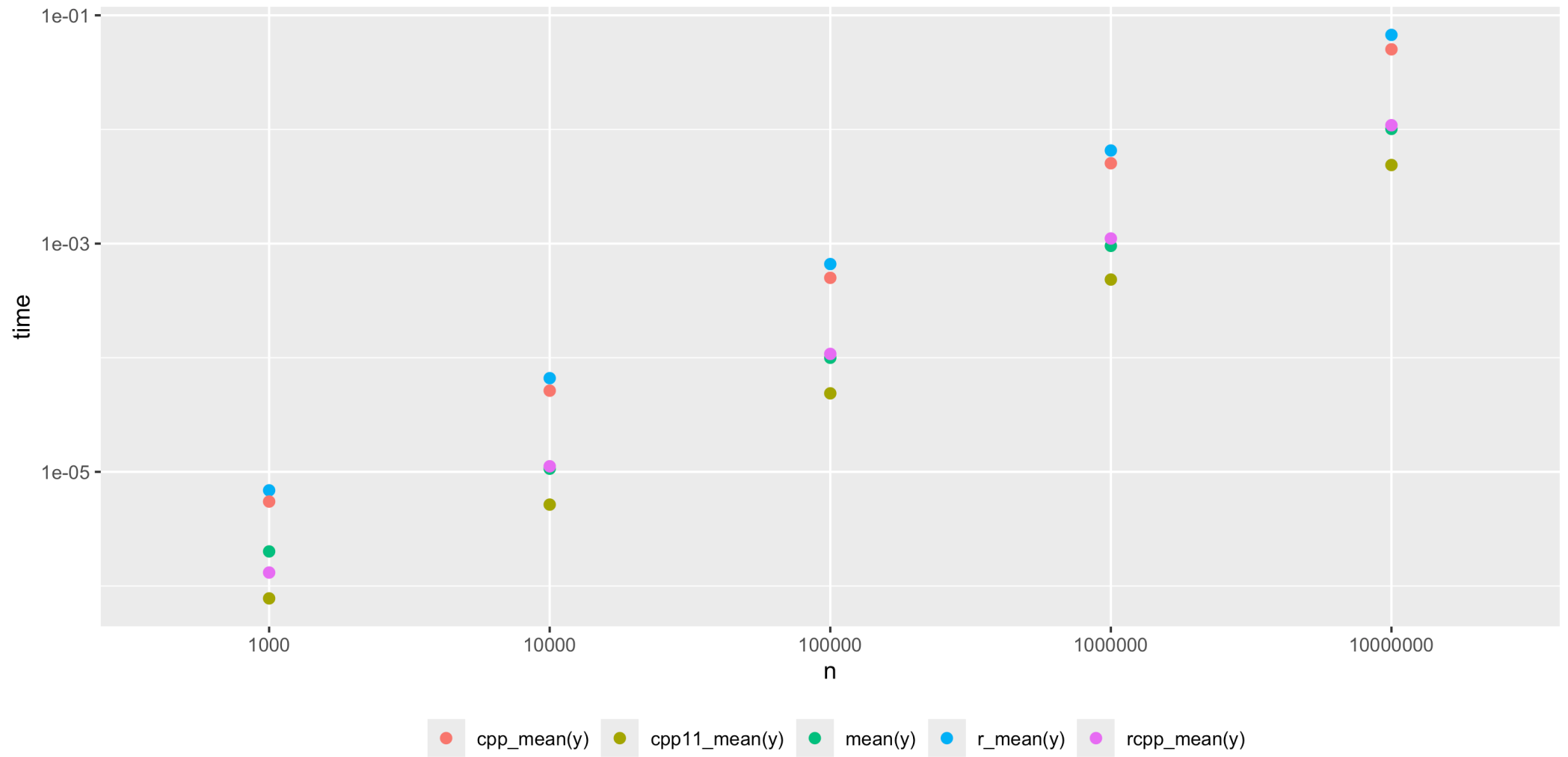
	expression	min	median	`itr/sec`	mem_alloc	`gc/sec`
	<bch:expr>	<bch:tm>	<bch:tm>	<dbl>	<bch:byt>	<dbl>
1	mean(y)	950.42µs	966.49µs	1013.	0B	0
2	r_mean(y)	6.03ms	6.38ms	156.	19.5KB	0
3	cpp_mean(y)	4.63ms	4.83ms	208.	0B	0
4	cpp11_mean(y)	467.48µs	488.02µs	2009.	0B	0
5	rcpp_mean(y)	1.06ms	1.09ms	901.	0B	0

bench::press

```
1 b = bench::press(  
2   n = 10^c(3:7),  
3   {  
4     y = sample(rnorm(n))  
5     bench::mark(  
6       mean(y), r_mean(y), cpp_mean(y), cpp11_mean(y), rcpp_mean(y)  
7     )  
8   }  
9 )
```

Running with:

	n
1	1000
2	10000
3	100000
4	1000000
5	10000000



Creating a list

```
1 #include <Rcpp.h>
2
3 // [[Rcpp::export]]
4 Rcpp::List make_list(int n) {
5     return Rcpp::List::create(
6         Rcpp::Named("norm") = Rcpp::rnorm(n, 0, 1),
7         Rcpp::Named("beta") = Rcpp::rbeta(n, 1, 1),
8         Rcpp::IntegerVector::create(1,2,3,4,5, NA_INTEGER)
9     );
10 }
```

```
1 make_list(10)
```

\$norm

```
[1] 1.9488167 -0.8285304 -1.1097544 -0.5550965 1.2926654 1.2270469
[7] -2.9024726 -0.1560244 -0.7628897 -0.9276166
```

\$beta

```
[1] 0.9387534068 0.5661279624 0.4984754021 0.4521797032 0.7545363114
[6] 0.0000225238 0.0341585791 0.1516264656 0.1550719612 0.5204840924
```

[[3]]

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

Creating a data.frame

```
1 #include <Rcpp.h>
2
3 // [[Rcpp::export]]
4 Rcpp::DataFrame make_df(int n) {
5     return Rcpp::DataFrame::create(
6         Rcpp::Named("norm") = Rcpp::rnorm(n, 0, 1),
7         Rcpp::Named("beta") = Rcpp::rbeta(n, 1, 1)
8     );
9 }
```

```
1 make_df(10)
```

	norm	beta
1	-1.59743101	0.8389083
2	-0.44002009	0.4046016
3	0.04101498	0.4530328
4	-2.22006170	0.9445844
5	0.91717044	0.9382699
6	-0.62993819	0.2152116
7	-2.09899910	0.7313789
8	2.75929305	0.2751544
9	-0.60135051	0.6822893
10	-1.34614376	0.4784344

Creating a tibble

```
1 #include <Rcpp.h>
2
3 // [[Rcpp::export]]
4 Rcpp::DataFrame make_tbl(int n) {
5     Rcpp::DataFrame df = Rcpp::DataFrame::create(
6         Rcpp::Named("norm") = Rcpp::rnorm(n, 0, 1),
7         Rcpp::Named("beta") = Rcpp::rbeta(n, 1, 1)
8     );
9     df.attr("class") = Rcpp::CharacterVector::create("tbl_df", "tbl", "data.frame");
10
11     return df;
12 }
```

```
1 make_tbl(10)
```

A tibble: 10 × 2

	norm	beta
	<dbl>	<dbl>
1	0.0291	0.569
2	-2.18	0.951
3	0.262	0.875
4	0.0244	0.710
5	0.520	0.602
6	-1.20	0.771
7	1.06	0.152

8	-0.400	0.152
9	-0.160	0.436
10	-0.769	0.613

Printing

R has some weird behavior when it comes to printing text from C++, use `Rcout` rather than `std::cout`.

```
1 #include <Rcpp.h>
2
3 // [[Rcpp::export]]
4 void n_hello(int n) {
5     for(int i=0; i!=n; ++i) {
6         Rcpp::Rcout << i+1 << ". Hello world!\n";
7     }
8 }
```

```
1 n_hello(5)
```

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

Printing NAs

```
1 #include <Rcpp.h>
2
3 // [[Rcpp::export]]
4 void print_na() {
5     Rcpp::Rcout << "NA_INTEGER : " << NA_INTEGER << "\n";
6     Rcpp::Rcout << "NA_STRING   : " << NA_STRING   << "\n";
7     Rcpp::Rcout << "NA_LOGICAL  : " << NA_LOGICAL  << "\n";
8     Rcpp::Rcout << "NA_REAL     : " << NA_REAL     << "\n";
9 }
```

```
1 print_na()
```

```
NA_INTEGER : -2147483648
NA_STRING   : 0x120014000
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.

```
1 #include <Rcpp.h>
2
3 // [[Rcpp::export]]
4 SEXP as_wrap(SEXP input) {
5     Rcpp::NumericVector r = Rcpp::as<Rcpp::NumericVector>(input);
6     Rcpp::NumericVector rev_r = Rcpp::rev(r);
7
8     return Rcpp::wrap(rev_r);
9 }
```

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

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

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

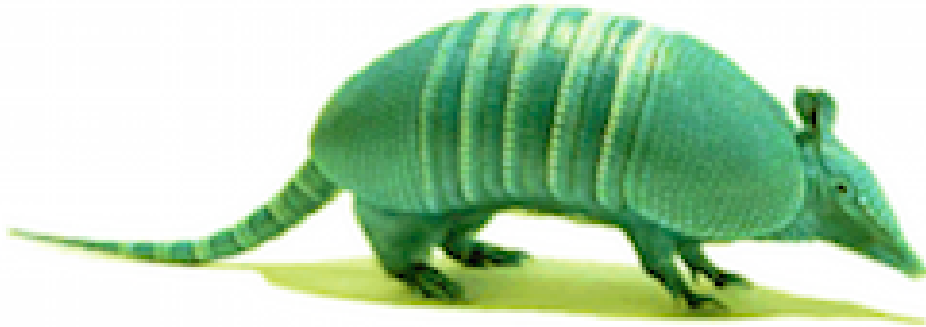
```
Error: Not compatible with requested type:
```

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

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

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:

```
1 arma::Col<type>,  
2 arma::Row<type>,  
3 arma::Mat<type>,  
4 arma::Cube<type>
```

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

```
1 float, double,  
2 std::complex<float>, std::complex<double>,  
3 short, int, long,  
4 unsigned short, unsigned int, unsigned long
```

typedef Shortcuts

For convenience the following typedefs are defined:

- Vectors:

```
1 arma::vec      = arma::colvec      = arma::Col<double>
2 arma::dvec     = arma::dcolvec     = arma::Col<double>
3 arma::fvec     = arma::fcolvec     = arma::Col<float>
4 arma::cx_vec   = arma::cx_colvec   = arma::Col<cx_double>
5 arma::cx_dvec  = arma::cx_dcolvec  = arma::Col<cx_double>
6 arma::cx_fvec  = arma::cx_fcolvec  = arma::Col<cx_float>
7 arma::uvec     = arma::ucolvec     = arma::Col<uword>
8 arma::ivec     = arma::icolvec     = arma::Col<sword>
```

- Matrices

```
1 arma::mat      = arma::Mat<double>
2 arma::dmat     = arma::Mat<double>
3 arma::fmat     = arma::Mat<float>
4 arma::cx_mat   = arma::Mat<cx_double>
5 arma::cx_dmat  = arma::Mat<cx_double>
6 arma::cx_fmat  = arma::Mat<cx_float>
7 arma::umat     = arma::Mat<uword>
8 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
- Include the following in your C++ code:

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

Example Program

```
1 // [[Rcpp::depends(RcppArmadillo)]]
2 #include <RcppArmadillo.h>
3
4 // [[Rcpp::export]]
5 arma::mat test_randu(int n, int m) {
6     arma::mat A = arma::randu<arma::mat>(n,m);
7     return A;
8 }
```

```
1 test_randu(4,5)
```

	[,1]	[,2]	[,3]	[,4]
[1,]	0.1667928	0.6575326	0.5676687	0.60743
[2,]	0.3053677	0.6481933	0.9097105	0.73086
[3,]	0.3001430	0.7471385	0.6986718	0.96816
[4,]	0.8806327	0.5012938	0.3641878	0.60658

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

	[,1]
[1,]	0.9545359
[2,]	0.6865738
[3,]	0.7338450

arma class attributes

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

```

1 // [[Rcpp::depends(RcppArmadillo)]]
2 #include <RcppArmadillo.h>
3
4 // [[Rcpp::export]]
5 void test_attr(arma::mat m) {
6     Rcpp::Rcout << "m.n_rows = " << m.n_rows << "\n";
7     Rcpp::Rcout << "m.n_cols = " << m.n_cols << "\n";
8     Rcpp::Rcout << "m.n_elem = " << m.n_elem << "\n";
9 }

```

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

```

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

```

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

```

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

```

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

Error: Not a matrix.

```
1 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
<code>v(i)</code>	Access the <code>i</code> -th element with bounds checking
<code>v.at(i)</code>	Access the <code>i</code> -th element without bounds checking
<code>v[i]</code>	Access the <code>i</code> -th element without bounds checking

For an `arma::mat m`,

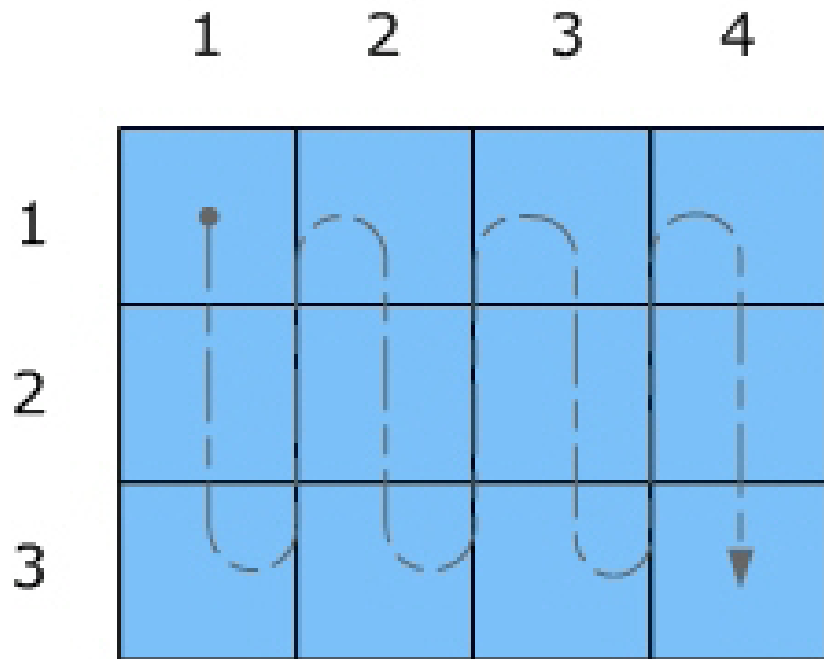
Call	Description
<code>m(i)</code>	Access the <code>i</code> -th element, treating object as flat and in column major order
<code>m(i, j)</code>	Access the element in <code>i</code> -th row and <code>j</code> -th column with bounds checking
<code>m.at(i, j)</code>	Access the element in <code>i</code>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

Element access - Cubes

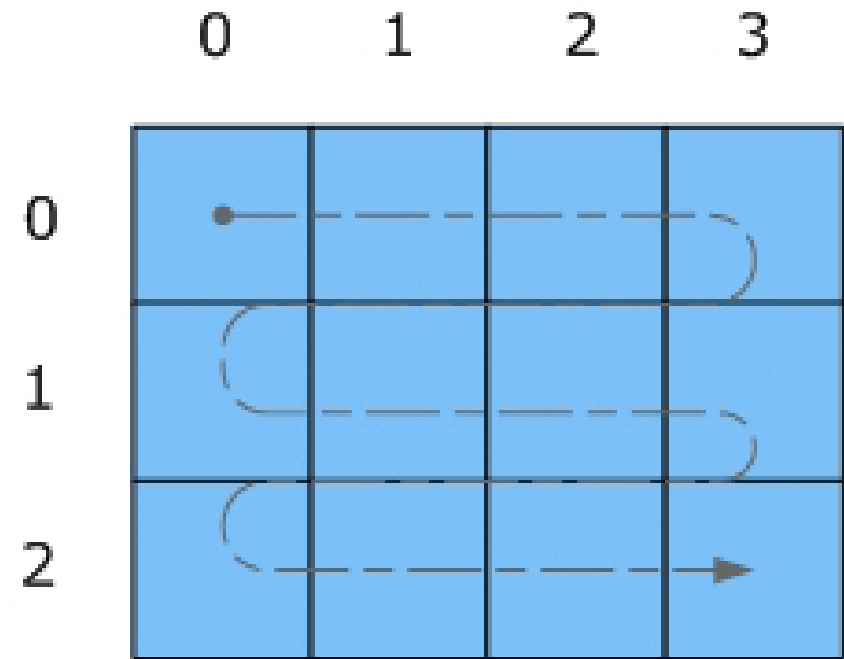
For an `arma::cube c`,

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

Data Organization



A: Column-major order (Fortran-style)



B: Row-major order (C-style)

```

1 // [[Rcpp::depends(RcppArmadillo)]]
2 #include <RcppArmadillo.h>
3
4 // [[Rcpp::export]]
5 void test_order(arma::mat m) {
6     for(int i=0; i!=m.n_elem; ++i) {
7         Rcpp::Rcout << m(i) << " ";
8     }
9     Rcpp::Rcout << "\n";
10 }

```

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

```
1 c(m)
```

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

```
1 test_order(m)
```

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

```
1 c(t(m))
```

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

```
1 test_order(t(m))
```

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

fastLm example

```
1 // [[Rcpp::depends(RcppArmadillo)]]
2 #include <RcppArmadillo.h>
3
4 // [[Rcpp::export]]
5 Rcpp::List fastLm(const arma::mat& X, const arma::colvec& y) {
6     int n = X.n_rows, k = X.n_cols;
7
8     arma::colvec coef = arma::solve(X, y);    // fit model  $y \sim X$ 
9     arma::colvec res = y - X*coef;           // residuals
10
11     // std.errors of coefficients
12     double s2 = std::inner_product(res.begin(), res.end(), res.begin(), 0.0)/(n -
13
14     arma::colvec std_err = arma::sqrt(s2 * arma::diagvec(arma::pinv(arma::trans(X)
15
16     return Rcpp::List::create(
17         Rcpp::Named("coefficients") = coef,
18         Rcpp::Named("stderr")       = std_err,
19         Rcpp::Named("df.residual")   = n - k
20     );
```

```

1 library(dplyr)
2 n=1e5
3 d = tibble(
4   x1 = rnorm(n),
5   x2 = rnorm(n),
6   x3 = rnorm(n),
7   x4 = rnorm(n),
8   x5 = rnorm(n),
9 ) %>%
10 mutate(
11   y = 3 + x1 - x2 + 2*x3 -2*x4 + 3*x5
12 )

```

```

1 res = bench::press(
2   size = c(100, 1000, 10000, 100000),
3   {
4     d = d[seq_len(size),]
5     X = model.matrix(y ~ ., d)
6     y = as.matrix(d$y)
7
8     bench::mark(
9       lm(y~., data=d),
10      lm.fit(X,y),
11      .lm.fit(X,y),
12      fastLm(X,y),
13      check = FALSE
14    )
15  }
16 )

```

Running with:

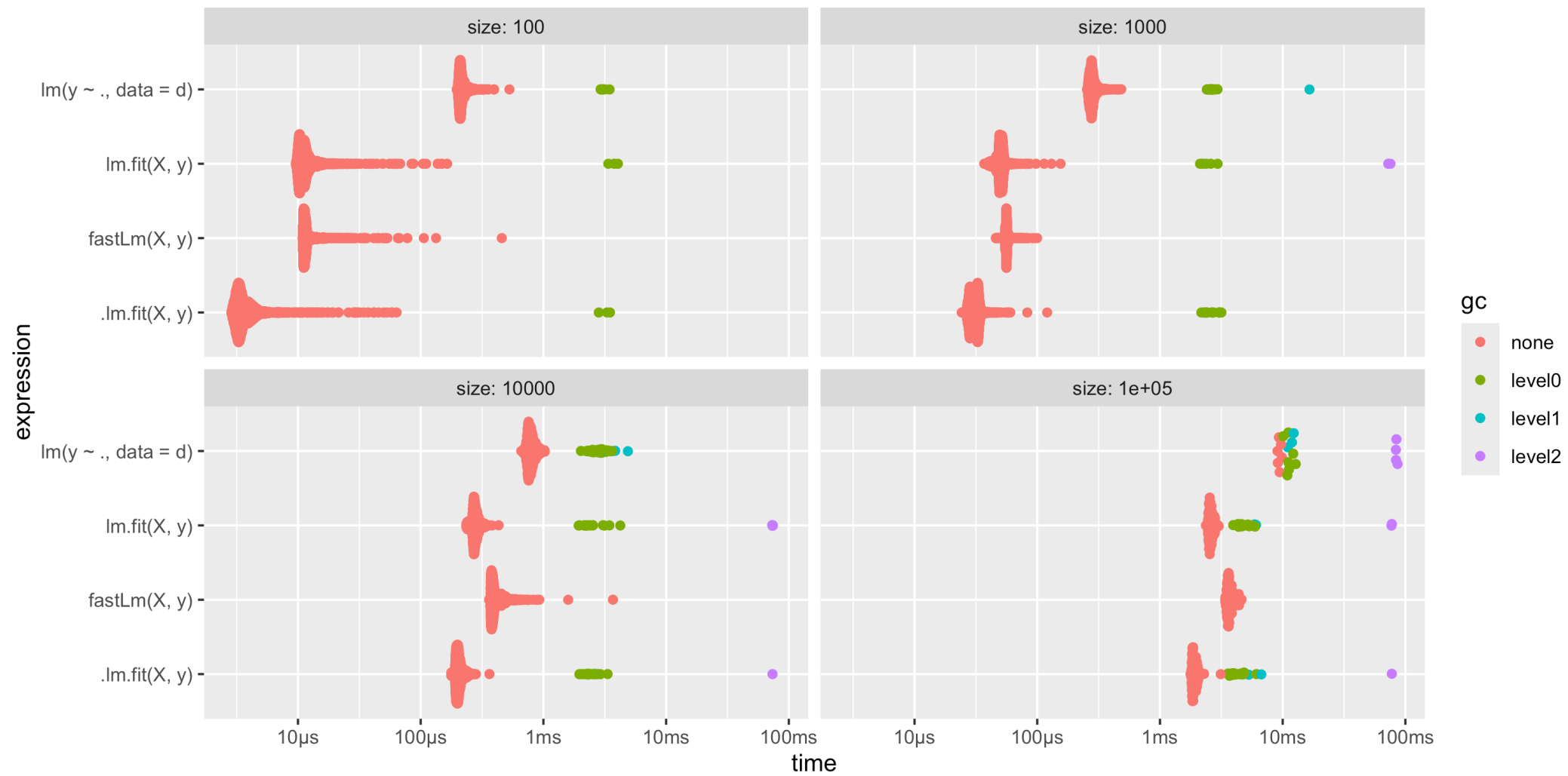
size

1 100

2 1000

3 10000

4 100000



MVN Example

Multivariate Normal Distribution - Review

For an n -dimension multivariate normal distribution with covariance Σ can be written as

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

where *[Math Processing Error]*

[Math Processing Error]

Multivariate Normal Distribution - Sampling

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

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

$$\boldsymbol{y}_{n \times 1} = \boldsymbol{\mu}_{n \times 1} + \boldsymbol{L}_{n \times n} \boldsymbol{z}_{n \times 1}$$

Multivariate Normal Distribution - Density

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

[Math Processing Error]

and its log density is given by

[Math Processing Error]