Rcpp Tutorial
Part III: Advanced Rcpp

Dr. Dirk Eddelbuettel
edd@debian.org
dirk.eddelbuettel@R-Project.org

useR! 2012
Vanderbilt University
June 12, 2012

Outline

1. Syntactic sugar
2. Rcpp Modules
3. Rcpp Classes

Motivating Sugar

Recall the earlier example of a simple (albeit contrived for the purposes of this discussion) R vector expression:

```
ifelse(x < y, x*x, -(y*y))
```

which for a given vector `x` will execute a simple transformation.

We saw a basic C implementation. How would we write it in C++ ?

Motivating sugar

examples/part3/sugarEx1.cpp

Maybe like this.

```
SEXP foo(SEXP xx, SEXP yy) {
  int n = x.size();
  NumericVector res1( n );
  double x_ = 0.0, y_ = 0.0;
  for (int i=0; i<n; i++) {
    x_ = x[i];
    y_ = y[i];
    if (R_IsNA(x_) || R_IsNA(y_)) {
      res1[i] = NA_REAL;
    } else if (x_ < y_) {
      res1[i] = x_ * x_;
    } else {
      res1[i] = -(y_ * y_);
    }
  }
  return(x);
}
```

Motivating sugar

examples/part3/sugarEx2.cpp

But with `sugar` we can simply write it as

```
SEXP foo( SEXP xx, SEXP yy) {
  NumericVector x(xx), y(yy) ;
  return ifelse( x < y, x*x, -(y*y) ) ;
}
```

Sugar: Another example

examples/part3/sugarEx3.cpp

Sugar also gives us things like `sapply` on C++ vectors:

```
double square( double x){
  return x*x ;
}

SEXP foo( SEXP xx ){
  NumericVector x(xx) ;
  return sapply( x, square ) ;
}
```

Sugar: Overview of Contents

logical operators `<, >, <=, >=, ==, !=`

arithmetic operators `+, -, *, /`

functions on vectors `abs, all, any, ceiling, cumsum, diag, diff, exp, head, ifelse, is_na, lapply, mean, pmin, pmax, pow, rep, rep_each, rep_len, rev, sapply, seq_along, seq_len, sd, sign, sum, tail, var,`

functions on matrices `outer, col, row, lower_tri, upper_tri, diag`

statistical functions (dpqr) `rnorm, dpois, qlogis,` etc ...

More information in the `Rcpp-sugar` vignette.

Binary arithmetic operators

Sugar defines the usual binary arithmetic operators : `+`, `-`, `*`, `/`.

```
// two numeric vectors of the same size
NumericVector x ;
NumericVector y ;

// expressions involving two vectors
NumericVector res = x + y ;
NumericVector res = x - y ;
NumericVector res = x * y ;
NumericVector res = x / y ;

// one vector, one single value
NumericVector res = x + 2.0 ;
NumericVector res = 2.0 - x;
NumericVector res = y * 2.0 ;
NumericVector res = 2.0 / y;

// two expressions
NumericVector res = x * y + y / 2.0 ;
NumericVector res = x * ( y - 2.0 ) ;
NumericVector res = x / ( y * y ) ;
```

Binary logical operators

```
// two integer vectors of the same size
NumericVector x ;
NumericVector y ;

// expressions involving two vectors
LogicalVector res = x < y ;
LogicalVector res = x > y ;
LogicalVector res = x <= y ;
LogicalVector res = x >= y ;
LogicalVector res = x == y ;
LogicalVector res = x != y ;

// one vector, one single value
LogicalVector res = x < 2 ;
LogicalVector res = 2 > x;
LogicalVector res = y <= 2 ;
LogicalVector res = 2 != y;

// two expressions
LogicalVector res = ( x + y ) <  ( x*x ) ;
LogicalVector res = ( x + y ) >= ( x*x ) ;
LogicalVector res = ( x + y ) == ( x*x ) ;
```

Unary operators

```
// a numeric vector
NumericVector x ;

// negate x
NumericVector res = -x ;

// use it as part of a numerical expression
NumericVector res = -x * ( x + 2.0 ) ;


// two integer vectors of the same size
NumericVector y ;
NumericVector z ;

// negate the logical expression "y < z"
LogicalVector res = ! ( y < z );
```

Functions producing a single logical result

```
IntegerVector x = seq_len( 1000 ) ;
all( x*x < 3 ) ;

any( x*x < 3 ) ;


// wrong: will generate a compile error
bool res = any( x < y) ) ;

// ok
bool res = is_true( any( x < y ) )
bool res = is_false( any( x < y ) )
bool res = is_na( any( x < y ) )
```

Functions producing sugar expressions

```
IntegerVector x = IntegerVector::create( 0, 1, NA_INTEGER, 3 ) ;

is_na( x )
all( is_na( x ) )
any( ! is_na( x ) )

seq_along( x )
seq_along( x * x * x * x * x * x )

IntegerVector x = seq_len( 10 ) ;

pmin( x, x*x );
pmin( x*x, 2 );

IntegerVector x, y;

ifelse( x < y, x, (x+y)*y );
ifelse( x > y, x, 2 );

sign( xx );
sign( xx * xx );

diff( xx );
```

Mathematical functions

```
IntegerVector x;

abs( x )
exp( x )
log( x )
log10( x )
floor( x )
ceil( x )
sqrt( x )
pow(x, z)      # x to the power of z
```

plus the regular trigonometrics functions and more.

Statistical function d/q/p/r

```
x1 = dnorm(y1, 0, 1);   // density of y1 at m=0, sd=1
x2 = pnorm(y2, 0, 1);   // distribution function of y2
x3 = qnorm(y3, 0, 1);   // quantiles of y3
x4 = rnorm(n, 0, 1);    // 'n' RNG draws of N(0, 1)
```

For beta, binom, caucht, exp, f, gamma, geom, hyper, lnorm, logis, nbeta, nbinom, nbinom_mu, nchisq, nf, norm, nt, pois, t, unif and weibull.

Use something like `RNGScope scope;` to set/reset the RNGs.

Sugar: benchmarks

expression	sugar	R	R / sugar
`any(x*y<0)`	0.000451	5.17	11450
`ifelse(x<y,x*x,-(y*y))`	1.378	13.15	9.54
`ifelse(x<y,x*x,-(y*y))` (⋆)	1.254	13.03	10.39
`sapply(x,square)`	0.220	113.38	515.24

Source: `examples/SugarPerformance/` using R 2.13.0, **Rcpp** 0.9.4, `g++-4.5`, Linux 2.6.32, i7 cpu.

⋆ : version includes optimization related to the absence of missing values

Sugar: benchmarks

Benchmarks of the convolution example from Writing R Extensions.

Implementation	Time in millisec	Relative to R API
R API (as benchmark)	234	
Rcpp sugar	158	0.68
`NumericVector::iterator`	236	1.01
`NumericVector::operator[]`	305	1.30
R API *naively*	2199	9.40

Table: Convolution of *x* and *y* (200 values), repeated 5000 times.

Source: `examples/ConvolveBenchmarks/` using R 2.13.0, **Rcpp** 0.9.4, `g++-4.5`, Linux 2.6.32, i7 cpu.

Sugar: Final Example

examples/part3/sugarExample.R

Consider a simple R function of a vector:

```
foo <- function(x) {

    ## sum of
    ##    -- squares of negatives
    ##    -- exponentials of positives
    s <- sum(ifelse( x < 0,  x*x,  exp(x) ))

    return(s)
}
```

Sugar: Final Example

examples/part3/sugarExample.R

Here is one C++ solution:

```
bar <- cxxfunction(signature(xs="numeric"),
                   plugin="Rcpp", body='

  NumericVector x(xs);

  double s = sum( ifelse( x < 0, x*x, exp(x) ));

  return wrap(s);
')
```

Sugar: Final Example

Benchmark from `examples/part3/sugarExample.R`

```
R> library(compiler)
R> cfoo <- cmpfun(foo)
R> library(rbenchmark)
R> x <- rnorm(1e5)
R> benchmark(foo(x), cfoo(x), bar(x),
+           columns=c("test", "elapsed", "relative",
+                     "user.self", "sys.self"),
+           order="relative", replications=10)
     test elapsed relative user.self sys.self
3  bar(x)   0.033   1.0000      0.03        0
1  foo(x)   0.441  13.3636      0.45        0
2 cfoo(x)   0.463  14.0303      0.46        0
R>
```

Outline

Rcpp Modules - Motivation

The **Rcpp** API makes it easier to write and maintain C++ extension for R.

But we can do better still:

- Even more direct interfaces between C++ and R
- Automatic handling / unwrapping of arguments
- Support exposing C++ functions to R
- Also support exposing C++ classes to R

Standing on the shoulders of `Boost.Python`

Boost.Python is a C++ library which enables seamless interoperability between C++ and the Python programming language.

Rcpp Modules borrows from **Boost.Python** to implement similar interoperability between R and C++.

Rcpp Modules

C++ functions and classes:

```
double square( double x ){
  return x*x;
}

class Foo {
public:
  Foo(double x_) : x(x_) {}

  double bar( double z){
    return pow( x - z, 2.0);
  }

private:
  double x;
};
```

This can be used in R:

```
> square( 2.0 )
[1] 4

> x <- new( Foo, 10 )
> x$bar( 2.0 )
[1] 64
```

Exposing C++ functions

Consider the simple function :

```
double norm( double x, double y ){
    return sqrt( x*x + y*y ) ;
}
```

Exercise : try to expose this function to R with what we have learned this morning. We want an R function that does this:

```
> norm( 2, 3 )
[1] 3.605551
```

Exposing C++ functions

C++ side:

```
#include <Rcpp.h>
double norm( double x, double y ){
    return sqrt( x*x + y*y) ;
}

SEXP norm_wrapper(SEXP x_, SEXP y_) {
    [...]
}
```

Compile with R CMD SHLIB:

```
$ R CMD SHLIB foo.cpp
```

R side:

```
dyn.load( "foo.so" )
norm <- function(x, y){
    .Call( [...] , x, y )
}
```

Exposing C++ functions

With inline

```
inc <- '
double norm( double x, double y ){
    return sqrt( x*x + y*y) ;
}
'
src <- '
   // convert the inputs
   double x = as<double>(x_), y = as<double>(y_);

   // call the function and store the result
   double res = norm( x, y ) ;

   // convert the result
   return wrap(y) ;
'
norm <- cxxfunction(signature(x_ = "numeric",
                              y_ = "numeric" ),
                    body = src, includes = inc,
                    plugin = "Rcpp" )
```

Exposing C++ functions (cont.)

So exposing a C++ function to R is straightforward, yet also somewhat tedious:

- Convert the inputs (from SEXP) to the appropriate types
- Call the function and store the result
- Convert the result to a SEXP

Rcpp Modules use Template Meta Programming (TMP) to replace these steps by a single step:

- Declare which function to expose

Exposing C++ functions with modules

Within a package

C++ side:

```
#include <Rcpp.h>

double norm( double x, double y ){
    return sqrt( x*x + y*y) ;
}
RCPP_MODULE(foo){
    function( "norm", &norm ) ;
}
```

R side:

```
.onLoad <- function(libname, pkgname){
    loadRcppModules()
}
```

(Other details related to module loading to take care of. We will cover them later.)

Exposing C++ functions with modules

Using inline

```
fx <- cxxfunction(, "", includes = '
   double norm( double x, double y ){
     return sqrt( x*x + y*y) ;
   }
   RCPP_MODULE(foo){
      function( "norm", &norm ) ;
   }
', plugin = "Rcpp" )

foo <- Module( "foo", getDynLib(fx) )

norm <- foo$norm
```

Exposing C++ functions

Documentation

`.function` can take an additional argument to document the exposed function:

```
double norm( double x, double y ){
    return sqrt( x*x + y*y) ;
}
RCPP_MODULE(foo){
    function("norm", &norm,
            "Some documentation about the function"
            );
}
```

which can be displayed from the R prompt:

```
R> show( mod$norm )
internal C++ function <0x1c21220>
docstring : Some documentation about the function
signature : double norm(double, double)
```

Exposing C++ functions
Formal arguments

Modules also let you supply formal arguments for more flexibility:

```
using namespace Rcpp;
double norm( double x, double y ) {
    return sqrt( x*x + y*y );
}

RCPP_MODULE(mod_formals2) {
    function( "norm", &norm,
        List::create( _["x"], _["y"] = 0.0 ),
        "Provides a simple vector norm"
        );
}
```

Exposing C++ functions
Formal arguments

Rcpp modules supports different types of arguments:

- Argument without default value : `_["x"]`
- Argument with default value : `_["y"] = 2`
- Ellipsis (...) : `_["..."]`

Exposing C++ classes
Motivation

Motivation: We want to manipulate C++ objects:

- Create instances
- Retrieve/Set data members
- Call methods

External pointers are useful for that, and **Rcpp** modules wraps them in a nice to use abstration.

Exposing C++ classes

A simple C++ class:

```
class Uniform {
public:

    // constructor
    Uniform(double min_, double max_) :
        min(min_), max(max_) {}

    // method
    NumericVector draw(int n) const {
        RNGScope scope;
        return runif( n, min, max );
    }

    // fields
    double min, max;
};
```

Exposing C++ classes

Modules can expose the `Uniform` class to allow this syntax:

```
> u <- new( Uniform, 0, 10 )
> u$draw( 10L )
[1] 3.00874606 7.00303770 6.17387340 0.06449014 7.40344856
[6] 6.48737922 1.73829428 7.53417005 0.38615597 6.66649310
> u$min
[1] 0
> u$max
[1] 10
> u$min <- 5
> u$draw(10)
[1] 7.02818458 8.19557570 5.42092100 6.02311031 8.18770124
[6] 6.18817312 8.60004068 6.60542979 5.41539068 9.96131797
```

Exposing C++ classes

Since C++ does not have reflection capabilities, modules need to declare what to expose:

- Constructors
- Fields or properties
- Methods
- Finalizers

Exposing C++ classes

A simple example

```
class Uniform {
public:
    Uniform(double min_, double max_) : min(min_), max(max_) {}
    NumericVector draw(int n) const {
        RNGScope scope;
        return runif( n, min, max );
    }
    double min, max;
};


RCPP_MODULE(random){
    class_<Uniform>( "Uniform")
    .constructor<double,double>()

    .field( "min", &Uniform::min )
    .field( "max", &Uniform::max  )

    .method( "draw", &Uniform::draw )
    ;
}
```

Exposing C++ classes

... Exposing constructors

```
class Uniform {
public:
    Uniform(double min_, double max_) : min(min_), max(max_) {}
    NumericVector draw(int n) const {
        RNGScope scope;
        return runif( n, min, max );
    }
    double min, max;
};


RCPP_MODULE(random){
    class_<Uniform>( "Uniform")
    .constructor<double,double>()

    .field( "min", &Uniform::min )
    .field( "max", &Uniform::max  )

    .method( "draw", &Uniform::draw )
    ;
}
```

Exposing C++ classes

... Exposing fields

```
class Uniform {
public:
    Uniform(double min_, double max_) : min(min_), max(max_) {}
    NumericVector draw(int n) const {
        RNGScope scope;
        return runif( n, min, max );
    }
    double min, max;
};


RCPP_MODULE(random){
    class_<Uniform>( "Uniform")
    .constructor<double,double>()

    .field( "min", &Uniform::min )
    .field( "max", &Uniform::max  )

    .method( "draw", &Uniform::draw )
    ;
}
```

Exposing C++ classes

... Exposing methods

```
class Uniform {
public:
    Uniform(double min_, double max_) : min(min_), max(max_) {}
    NumericVector draw(int n) const {
        RNGScope scope;
        return runif( n, min, max );
    }
    double min, max;
};


RCPP_MODULE(random){
    class_<Uniform>( "Uniform")
    .constructor<double,double>()

    .field( "min", &Uniform::min )
    .field( "max", &Uniform::max  )

    .method( "draw", &Uniform::draw )
    ;
}
```

Exposing C++ classes
... Constructors

The `.constructor` method of `class_` can expose public constructors taking between 0 and 7 arguments.

The argument types are specified as template parameters of the `.constructor` methods.

It is possible to expose several constructors that take the same number of arguments, but this require the developper to implement dispatch to choose the appropriate constructor.

Exposing C++ classes
... Fields

Public data fields are exposed with the `.field` member function:

```
.field( "x", &Uniform::x )
```

If you do not wish the R side to have write access to a field, you can use the `.field_readonly` field:

```
.field_readonly( "x", &Uniform::x )
```

Exposing C++ classes
... Properties

Properties let the developper associate getters (and optionally setters) instead of retrieving the data directly. This can be useful for:

- Private or protected fields
- To keep track of field access
- To add operations when a field is retrieved or set
- To create a pseudo field that is not directly related to a data member of the class

Exposing C++ classes
... Properties

Properties are declared with one of the `.property` overloads:

```
.property( "z",
           &Foo::get_z,
           &Foo::set_z,
           "Documentation" )
```

This contains

- the R side name of the property (required)
- address of the getter (required)
- address of the setter (optional)
- documentation for the property (optional)

Exposing C++ classes

... Properties, getters

Getters can be:

```
class Foo{
public:
    double get(){ ... }
    ...
} ;

double outside_get( Foo* foo ){ ... }
```

- Public member functions of the target class that take no argument and return something
- Free functions that take a pointer to the target class as unique argument and returns something

Exposing C++ classes

... Properties, setters

Setters can be:

```
class Foo{
public:
    void set(double x){ ... }
    ...
} ;

void outside_set( Foo* foo , double x){ ... }
```

- Public member functions that take exactly one argument (which must match with the type used in the getter)
- Free function that takes exactly two arguments: a pointer to the target class, and another variable (which must match the type used in the getter).

Exposing C++ classes

Fields and properties example

```
class Foo{
public:
    double x, y ;
    double get_z(){ return z; }
    void set_z( double new_z ){ z = new_z ; }
    // ...
private:
    double z ;
};
double get_w(Foo* foo){ ... }
void set_w(Foo* foo, double w ){ ...Â }

RCPP_MODULE(bla){
    class_<Foo>("Foo")
    // ...
    .field( "x", &Foo::x )
    .field_readonly( "y", &Foo::y )
    .property( "z", &Foo::get_z, Foo::set_z )
    .property( "w", &get_w, &set_w )
    ;
}
```

... Methods

The `.method` member function of `class_` is used to expose methods, which can be:

- A public member function of the target class, const or non const, that takes between 0 and 65 parameters and returns either void or something
- A free function that takes a pointer to the target class, followed by between 0 and 65 parameters, and returns either void or something

... Methods, examples

```
class Foo{
public:
    ...
    void bla() ;
    double bar( int x, std::string y ) ;

} ;
double yada(Foo* foo) { ... }

RCPP_MODULE(mod){
    class_<Foo>
    ...
    .method( "bla" , &Foo::bla )
    .method( "bar" , &Foo::bar )
    .method( "yada", &yada )
    ;
}
```

... Finalizers

When the R reference object that wraps the internal C++ object goes out of scope, it becomes candidate for GC.

When it is GC'ed, the destructor of the target class is called.

Finalizers allow the developper to add behavior right before the destructor is called (free resources, etc ...)

Finalizers are associated to exposed classes with the `class_::.finalizer` method. A finalizer is a free function that takes a pointer to the target class as unique argument and returns void.

Exercize : expose this class

```
class Normal{
public:
  // 3 constructors
  Normal() : mean(0.0), sd(1.0){}
  Normal(double mean_) : mean(mean_), sd(1.0){}
  Normal(double mean_, double sd_) :
    mean(mean_), sd(sd_){}

  // one method
  NumericVector draw(int n){
    RNGScope scope ;
    return rnorm( n, mean, sd ) ;
  }

  // two fields (declare them read-only)
  double mean, sd ;
} ;
```

Modules and packages

The best way to use **Rcpp** modules is to embed them in an R package.

The `Rcpp.package.skeleton` (and its `module` argument) creates a package skeleton that has an Rcpp module.

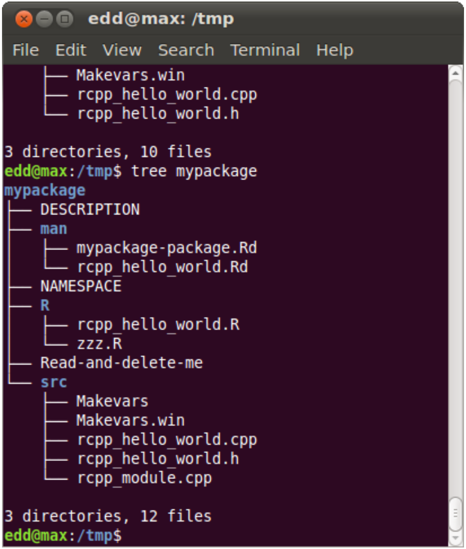
```
> Rcpp.package.skeleton("mypackage",
+                        module = TRUE )
```

Modules and packages

```
> Rcpp.package.skeleton( "mypackage", module=TRUE )
Creating directories ...
Creating DESCRIPTION ...
Creating NAMESPACE ...
Creating Read-and-delete-me ...
Saving functions and data ...
Making help files ...
Done.
Further steps are described in './mypackage/Read-and-delete-me'.

Adding Rcpp settings
>> added RcppModules:  yada
>> added Depends:  Rcpp
>> added LinkingTo:  Rcpp
>> added useDynLib directive to NAMESPACE
>> added Makevars file with Rcpp settings
>> added Makevars.win file with Rcpp settings
>> added example header file using Rcpp classes
>> added example src file using Rcpp classes
>> added example R file calling the C++ example
>> added Rd file for rcpp_hello_world
>> copied the example module
```

Calling `Rcpp.package.skeleton`



```
edd@max: /tmp
File Edit View Search Terminal Help
    ├── Makevars.win
    ├── rcpp_hello_world.cpp
    └── rcpp_hello_world.h

3 directories, 10 files
edd@max:/tmp$ tree mypackage
mypackage
├── DESCRIPTION
├── man
│   ├── mypackage-package.Rd
│   └── rcpp_hello_world.Rd
├── NAMESPACE
├── R
│   ├── rcpp_hello_world.R
│   └── zzz.R
├── Read-and-delete-me
└── src
    ├── Makevars
    ├── Makevars.win
    ├── rcpp_hello_world.cpp
    ├── rcpp_hello_world.h
    └── rcpp_module.cpp

3 directories, 12 files
edd@max:/tmp$
```

We will discuss the individual files in the next few slides.

Also note that the next release will contain two more `cpp` files.

rcpp_module.cpp

```
#include <Rcpp.h>

[...]
int bar( int x){
  return x*2 ;
}
double foo( int x, double y){
  return x * y ;
}
[...]

class World {
public:

  World() :  msg("hello"){}
  void set(std::string msg) { this->msg = msg; }
  std::string greet() { return msg; }

private:
    std::string msg;
};
```

rcpp_module.cpp

```
RCPP_MODULE(yada){
  using namespace Rcpp ;

  [...]

  function( "bar", &bar,
    List::create( _["x"] = 0.0 ),
    "documentation for bar " ) ;

  function( "foo"    , &foo     ,
    List::create( _["x"] = 1, _["y"] = 1.0 ),
    "documentation for foo " ) ;

  class_<World>( "World" )
    .constructor()
    .method( "greet", &World::greet ,  "get the message" )
    .method( "set", &World::set     ,  "set the message" )
  ;
}
```

Modules and packages: DESCRIPTION

```
Package: mypackage
Type: Package
Title: What the package does (short line)
Version: 1.0
Date: 2011-08-15
Author: Who wrote it
Maintainer: Who to complain to <yourfault@somewhere.net>
Description: More about what it does (maybe more than one line)
License: What Licence is it under ?
LazyLoad: yes
Depends: methods, Rcpp (>= 0.9.6)
LinkingTo: Rcpp
RcppModules: yada
```

Modules and packages: zzz.R

The `.onLoad()` function (often in `zzz.R` file) must contain a call to the `loadRcppModules` function.

For the next R version, we can switch to `evalqOnLoad()`.

```
## For R 2.15.0 we still need require(), but note the R CMD check issue
.onLoad <- function(libname, pkgname){
    require("methods")    ## needed, but upsets R CMD check
    loadRcppModules()
}

## For R 2.15.1 and later this will work
#evalqOnLoad({
#       loadModule("NumEx", TRUE)
#       loadModule("yada", TRUE)
#       loadModule("stdVector", TRUE)
#})
```

Modules and packages: NAMESPACE

The NAMESPACE files loads the dyanmic library of the packages, imports from **Rcpp** and exports all local symbols of the package (using regular expression).

```
useDynLib(mypackage)
exportPattern("^[[:alpha:]]+")
import( Rcpp )
```

Modules and packages: Using the package

```
> require( mypackage )
> foo
internal C++ function <0x100612350>
    docstring : documentation for foo
    signature : double foo(int, double)
> foo( 2, 3 )
[1] 6
> World
C++ class 'World' <0x10060edc0>
Constructors:
    World()

Fields: No public fields exposed by this class

Methods:
     std::string greet()
           docstring : get the message
     void set(std::string)
           docstring : set the message
> w <- new( World )
> w$set( "bla bla" )
> w$greet()
[1] "bla bla"
```

stdVector.cpp (in Rcpp's skeleton and unittest)

```
#include <Rcpp.h>                     // need to include the main Rcpp header file only

typedef std::vector<double> vec;   // convenience typedef

void vec_assign( vec* obj, Rcpp::NumericVector data) { // helpers
    obj->assign( data.begin(), data.end() ) ;
}
void vec_insert( vec* obj, int position, Rcpp::NumericVector data) {
    vec::iterator it = obj->begin() + position;
    obj->insert( it, data.begin(), data.end() );
}

Rcpp::NumericVector vec_asR( vec* obj) {
    return Rcpp::wrap( *obj );
}

void vec_set( vec* obj, int i, double value) {
    obj->at( i ) = value;
}

void vec_resize( vec* obj, int n) { obj->resize( n ); }
void vec_push_back( vec* obj, double x ) { obj->push_back( x ); }

// Wrappers for member functions that return a reference -- required on Solaris
double vec_back(vec *obj){ return obj->back() ; }
double vec_front(vec *obj){ return obj->front() ; }
double vec_at(vec *obj, int i){ return obj->at(i) ; }
```

stdVector.cpp cont.

```
RCPP_MODULE(stdVector){
    using namespace Rcpp ;
    // we expose the class std::vector<double> as "vec" on the R side
    class_<vec>("vec")
    // exposing the default constructor
    .constructor()
    // exposing member functions -- taken directly from std::vector<double>
    .method( "size",      &vec::size)
    .method( "max_size",  &vec::max_size)
    .method( "capacity",  &vec::capacity)
    .method( "empty",     &vec::empty)
    .method( "reserve",   &vec::reserve)
    .method( "pop_back",  &vec::pop_back )
    .method( "clear",     &vec::clear )
    // specifically exposing const member functions defined above
    .method( "back",      &vec_back )
    .method( "front",     &vec_front )
    .method( "at",        &vec_at )
    // exposing free functions taking a std::vector<double> *
    // as their first argument
    .method( "assign",    &vec_assign )
    .method( "insert",    &vec_insert )
    .method( "as.vector",&vec_asR )
    .method( "push_back",&vec_push_back )
    .method( "resize",    &vec_resize)
    // special methods for indexing
    .method( "[[",        &vec_at )
    .method( "[[<-",       &vec_set )
    ;
}
```

stdVector.cpp cont.

Usage from R

```
v <- new(vec)                                  # stdVector module

data <- 1:10
v$assign(data)

v[[3]] <- v[[3]] + 1
data[[4]] <- data[[4]] +1

checkEquals( v$as.vector(), data )

v$size()
v$capacity()
```

planar/src/multilayer.cpp

```
#include <RcppArmadillo.h>
#include <iostream>

using namespace Rcpp ;
using namespace RcppArmadillo ;
using namespace arma ;
using namespace std;

Rcpp::List multilayer(const arma::colvec& k0,     \
        const arma::cx_mat& kx,     \
        const arma::cx_mat& epsilon,    \
        const arma::colvec& thickness,    \
        const int& polarisation) {
 [...]
Rcpp::List recursive_fresnel(const arma::colvec& k0,    \
        const arma::cx_mat& kx,    \
        const arma::cx_mat& epsilon,   \
        const arma::colvec& thickness,   \
        const int& polarisation) {

[...]

RCPP_MODULE(planar){
  using namespace Rcpp;

  function( "multilayer", &multilayer,      \
    "Calculates reflection and transmission coefficients of a multilayer stack" );
  function( "recursive_fresnel", &recursive_fresnel,      \
    "Calculates the reflection coefficient of a multilayer stack" );
 }
```

cda/src/{cda,utils}.cpp

```
#include "utils.h"
#include "cda.h"
#include <RcppArmadillo.h>
#include <iostream>

using namespace Rcpp;
using namespace RcppArmadillo;
using namespace std;

arma::mat euler(const double phi, const double theta, const double psi) {
[...]
arma::cx_mat interaction_matrix(const arma::mat& R, const double kn,
                                const arma::cx_mat& invAlpha,
                                const arma::mat& Euler, const int full) {
double extinction(const double kn, const arma::cx_colvec& P,
                  const arma::cx_colvec& Eincident) {
[...]
double absorption(const double kn, const arma::cx_colvec& P,
                  const arma::cx_mat& invpolar) {
[...]

RCPP_MODULE(cda){
   using namespace Rcpp ;

   function( "euler", &euler, "Constructs a 3x3 Euler rotation matrix" ) ;
   function( "extinction", &extinction, "Calculates the extinction cross-section" ) ;
   function( "absorption", &absorption, "Calculates the absorption cross-section" ) ;
   function( "interaction_matrix", &interaction_matrix,
             "Constructs the coupled-dipole interaction matrix" ) ;
}
```

GUTS/src/GUTS_rpp_module.cpp

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

using namespace Rcpp;

RCPP_MODULE(modguts)
{
  class_<GUTS>( "GUTS" )
    .constructor()

    .method( "setConcentrations",    &GUTS::setConcentrations,
             "Set time series vector of concentrations." )
    .method( "setSurvivors",         &GUTS::setSurvivors,
             "Set time series vector of survivors." )
    [...]

    .method( "setSample", &GUTS::setSample, "Set ordered sample vector." )

    .method( "calcLoglikelihood", &GUTS::calcLoglikelihood,
             "Returns calculated log. of likelihood from complete + valid object." )

    .property( "C",      &GUTS::getC, "Vector of concentrations." )
    .property( "Ct",     &GUTS::getCt, "Time vector of concentrations." )
    [...]
    ;
}
```

RcppBDT/src/RcppBDT.cpp

```
RCPP_MODULE(bdt) {

  using namespace boost::gregorian;
  using namespace Rcpp;

  // exposing a class (boost::gregorian::)date as "date" on the R side
  class_<date>("date")

  // constructors
  .constructor("default constructor")
  .constructor<int, int, int>("constructor from year, month, day")

  .method("setFromLocalClock", &date_localDay, "create a date from local clock")
  .method("setFromUTC", &date_utcDay, "create a date from current universal clock")
  [...]
  .method("getYear", &date_year, "returns the year")
  .method("getMonth", &date_month, "returns the month")
  .method("getDay", &date_day, "returns the day")
  .method("getDayOfYear", &date_dayofyear, "returns the day of the year")
  [...]
  .method("getDate", &date_toDate, "returns an R Date object")
  .method("fromDate", &date_fromDate, "sets date from an R Date object")
  [...]
  .const_method("getWeekNumber", &date::week_number, "returns number of week")
  .const_method("getModJulian", &date::modjulian_day, "returns the mod. Julian day")
  .const_method("getJulian", &date::julian_day, "returns the Julian day")
  [...]
  .method("getNthDayOfWeek", &Date_nthDayOfWeek,
          "return nth week's given day-of-week in given month and year")
  [...]
  ;
}
```

Exercise

- Create a package with a module : start by using `Rcpp.package.skeleton`
- Expose two C++ functions
- Expose a C++ class

```
double fun1( NumericVector x){
  return sum(
    head(x,-1) - tail(x,-1)
  ) ;
}

double fun2( NumericVector x ){
  return mean(x) / sd(x) ;
}
```

```
class Normal {
  public:
    Normal(
      double mean_, double sd_
    ) ;
    NumericVector draw( int n ) ;
    int get_ndraws() ;

  private:
    double mean, sd ;
    int ndraws ;
} ;
```

Exercise

```
class Normal;                              // forward declaration
double sumdiff(Normal *obj, NumericVector x) {
    return sum(head(x,-1) - tail(x,-1));
}
double zscore(Normal *obj, NumericVector x ) {
    return mean(x) / sd(x) ;
}
class Normal {
public:
    Normal(double mean_, double sd_) : mean(mean_), sd(sd_), ndraws(0) {};
    NumericVector draw( int n ) {
        ndraws += n;
        RNGScope scope;
        return rnorm(n, mean, sd);
    }
    int get_ndraws() {
        return ndraws;
    }
private:
    double mean, sd ;
    int ndraws ;
} ;
RCPP_MODULE(newmod) {
    class_<Normal>("Normal")
    .constructor<double, double>()
    .method("draw", &Normal::draw)
    .method("get_ndraws", &Normal::get_ndraws)
    .method("sumdiff", &sumdiff)
    .method("zscore", &zscore)
    ;
}
```

Exercise

```
fx <- cxxfunction(signature(), plugin="Rcpp", include=inc)
newmod <- Module("newmod", getDynLib(fx))

nn <- new(newmod$Normal, 10, 10)
set.seed(42)
z <- nn$draw(4)
nn$sumdiff(z)
nn$zscore(z)
```

Outline

1. Syntactic sugar
2. Rcpp Modules
3. Rcpp Classes

Overview

Recently, John Chambers committed some code which will be in the next Rcpp release. This builds on Rcpp Modules, and allows the R side to modify C++ classes.

This is documented in `help(setRcppClass)` as well as in one test package to to support the unit tests.

Example

```
setRcppClass("World", module = "yada", fields = list(more = "character"),
         methods = list(test = function(what) message("Testing: ", what, "; ", more)),
         saveAs = "genWorld")
setRcppClass("stdNumeric", "vec", "stdVector")
evalqOnLoad({ # some methods that use C++ methods
    stdNumeric$methods(
        getEl = function(i) {
            i <- as.integer(i)
            if(i < 1 || i > size())
                NA_real_
            else
                at(i-1L)
        },
        setEl = function(i, value) {
            value <- as.numeric(value)
            if(length(value) != 1)
                stop("Only assigns single values")
            i <- as.integer(i)
            if(i < 1 || i > size())
                stop("index out of bounds")
            else
                set(i-1L, value)
        },
        initialize = function(data = numeric()) {
            callSuper()
            data <- as.double(data)
            n <- as.integer(max(50, length(data) * 2))
            reserve(n)
            assign(data)
        }
    )
})
```

Rcpp Tutorial
Part IV: Applications

Dr. Dirk Eddelbuettel
edd@debian.org
dirk.eddelbuettel@R-Project.org

useR! 2012
Vanderbilt University
June 12, 2012

Outline

The first example

examples/standard/rinside_sample0.cpp

We have seen this first example in part I:

```
#include <RInside.h>                      // embedded R via RInside

int main(int argc, char *argv[]) {

    RInside R(argc, argv);                // create embedded R inst.

    R["txt"] = "Hello, world!\n";         // assign   to 'txt' in R

    R.parseEvalQ("cat(txt)");             // eval string, ignore result

    exit(0);
}
```

Assign a variable, evaluate an expression—easy!

RInside in a nutshell

Key aspects:

- RInside uses the embedding API of R
- An instance of R is launched by the RInside constructor
- It behaves just like a regular R process
- We submit commands as C++ strings which are parsed and evaluated
- Rcpp is to easily get data in and out from the enclosing C++ program.

A second example: part one

examples/standard/rinside_sample1.cpp

```
#include <RInside.h>          // for the embedded R via RInside

Rcpp::NumericMatrix createMatrix(const int n) {
    Rcpp::NumericMatrix M(n,n);
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            M(i,j) = i*10 + j;
        }
    }
    return(M);
}
```

A second example: part two

examples/standard/rinside_sample1.cpp

```
int main(int argc, char *argv[]) {
  RInside R(argc, argv);    // create an embedded R instance

  const int mdim = 4;        // let the matrices be 4 by 4; create, fill
  R["M"] = createMatrix(mdim);   // assign data Matrix to R's 'M' var

  std::string str =
    "cat('Running ls()\n'); print(ls()); "
    "cat('Showing M\n'); print(M); "
    "cat('Showing colSums()\n'); Z <- colSums(M); "
    "print(Z); Z";                    // returns Z
  Rcpp::NumericVector v = R.parseEval(str); // eval, assign
  // now show vector on stdout
  exit(0);
}
```

Other example files provide similar R snippets and interchange.

A third example: Calling R plot functions

examples/standard/rinside_sample11.cpp

```
#include <RInside.h>                    // embedded R via RInside

int main(int argc, char *argv[]) {

  RInside R(argc, argv);                // create an embedded R instance

  // evaluate an R expression with curve()
  std::string cmd = "tmpf <- tempfile('curve'); "
    "png(tmpf); curve(x^2, -10, 10, 200); "
    "dev.off(); tmpf";
  // by running parseEval, we get filename back
  std::string tmpfile = R.parseEval(cmd);

  std::cout << "Could use plot in " << tmpfile << std::endl;
  unlink(tmpfile.c_str());   // cleaning up

  // alternatively, by forcing a display we can plot to screen
  cmd = "x11(); curve(x^2, -10, 10, 200); Sys.sleep(30);";
  R.parseEvalQ(cmd);

  exit(0);
}
```

A fourth example: Using Rcpp modules

examples/standard/rinside_module_sample0.cpp

```
#include <RInside.h>                    // for the embedded R via RInside
// a c++ function we wish to expose to R
const char* hello( std::string who ){
    std::string result( "hello " ) ;
    result += who ;
    return result.c_str() ;
}
RCPP_MODULE(bling){
    using namespace Rcpp ;
    function( "hello", &hello );
}
int main(int argc, char *argv[]) {
    // create an embedded R instance -- and load Rcpp so that modules work
    RInside R(argc, argv, true);
    // load the bling module
    R["bling"] = LOAD_RCPP_MODULE(bling) ;
    // call it and display the result
    std::string result = R.parseEval("bling$hello('world')") ;
    std::cout << "bling$hello('world') = '" << result << "'"
              << std::endl ;
    exit(0);
}
```

Other RInside standard examples

Besides ex0, ex1 and ex11

A quick overview:

- ex2 loads an Rmetrics library and access data
- ex3 run regressions in R, uses coefs and names in C++
- ex4 runs a small portfolio optimisation under risk budgets
- ex5 creates an environment and tests for it
- ex6 illustrations direct data access in R
- ex7 shows `as<>()` conversions from `parseEval()`
- ex8 is another simple bi-directional data access example
- ex9 makes a C++ function accessible to the embedded R
- ex10 creates and alters lists between R and C++
- ex12 uses `sample()` from C++

Outline

Parallel Computing with RInside

R is famously single-threaded.

High-performance Computing with R frequently resorts to fine-grained (**multicore**, **doSMP**) or coarse-grained (**Rmpi**, **pvm**, ...) parallelism. R spawns and controls other jobs.

Jianping Hua suggested to embed R via RInside in MPI applications.

Now we can use the standard and well understood MPI paradigm to launch multiple R instances, each of which is indepedent of the others.

A first example

examples/standard/rinside_sample2.cpp

```
#include <mpi.h>       // mpi header
#include <RInside.h> // for the embedded R via RInside

int main(int argc, char *argv[]) {

  MPI::Init(argc, argv);                          // mpi initialization
  int myrank = MPI::COMM_WORLD.Get_rank();        // current node rank
  int nodesize = MPI::COMM_WORLD.Get_size();      // total nodes running.

  RInside R(argc, argv);                          // embedded R instance

  std::stringstream txt;
  txt << "Hello from node " << myrank              // node information
      << " of " << nodesize << " nodes!" << std::endl;

  R["txt"] = txt.str();                           // assign to R var 'txt'
  R.parseEvalQ("cat(txt)");                       // eval, ignore returns

  MPI::Finalize();                                // mpi finalization
  exit(0);
}
```

A first example: Output

`examples/standard/rinside_sample2.cpp`

```
edd@max:/tmp$ orterun -n 8 ./rinside_mpi_sample2
Hello from node 5 of 8 nodes!
Hello from node 7 of 8 nodes!
Hello from node 1 of 8 nodes!
Hello from node 0 of 8 nodes!
Hello from node 2 of 8 nodes!
Hello from node 3 of 8 nodes!
Hello from node 4 of 8 nodes!
Hello from node 6 of 8 nodes!
edd@max:/tmp$
```

This uses Open MPI just locally, other hosts can be added via `-H node1,node2,node3`.

The other example(s) shows how to gather simulation results from MPI nodes.

Outline

Application example: Qt

RInside `examples/qt/`

The question is sometimes asked how to embed **RInside** in a larger program.

We just added a new example using **Qt**:

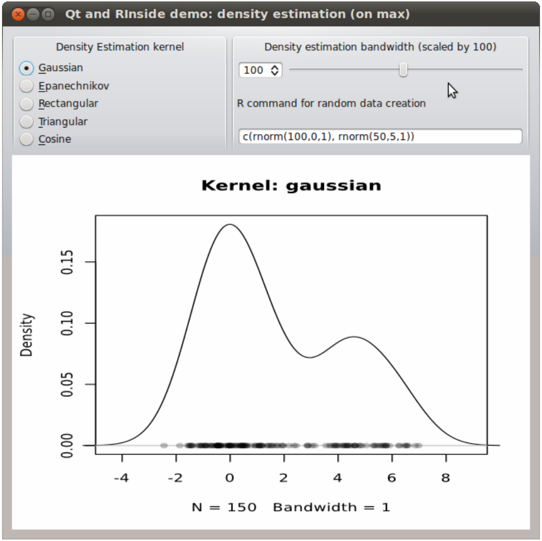
```
#include <QApplication>
#include "qtdensity.h"

int main(int argc, char *argv[])
{
    RInside R(argc, argv);      // create an embedded R instance

    QApplication app(argc, argv);
    QtDensity qtdensity(R);   // pass R inst. by reference
    return app.exec();
}
```

Application example: Qt density slider

RInside `examples/qt/`



This uses standard **Qt** / GUI paradigms of

- radio buttons
- sliders
- textentry

all of which send values to the R process which provides an SVG (or PNG as fallback) image that is plotted.

Application example: Qt density slider

RInside `examples/qt/`

The actual code is pretty standard **Qt** / GUI programming (and too verbose to be shown here).

The `qtdensity.pro` file is interesting as it maps the entries in the `Makefile` (discussed in the next section) to the **Qt** standards.

It may need an update for OS X—we have not tried that yet.

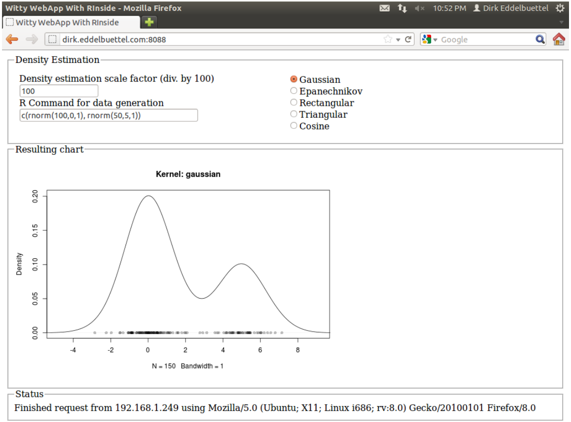
Outline

1. RInside
   - Basics
   - MPI
   - Qt
   - Wt
   - Building with RInside
2. RcppArmadillo
   - Armadillo
   - Example: FastLM
   - Example: VAR(1) Simulation
   - Example: Kalman Filter

Application example: Wt

RInside `examples/wt/`

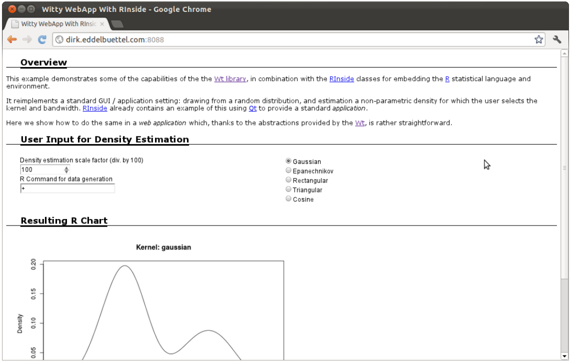
Given the desktop application with **Qt**, the question arises how to deliver something similar "over the web" — and **Wt** helps.



Wt is similar to **Qt** so the code needs only a few changes.
Wt takes care of all browser / app interactions and determines the most featureful deployment.

Application example: Wt

RInside `examples/wt/`



Wt can also be "dressed up" with simple CSS styling (and the text displayed comes from an external XML file, further separating content and presentation).

Outline

Building with RInside

RInside needs headers and libraries from several projects as it

embeds R itself so we need R headers and libraries

uses Rcpp so we need Rcpp headers and libraries

RInside itself so we also need RInside headers and libraries

Building with RInside

Use the Makefile in `examples/standard`

The `Makefile` is set-up to create an binary for example example file supplied. It uses

- **R CMD config** to query all of `-cppflags`, `-ldflags`, `BLAS_LIBS` and `LAPACK_LIBS`
- **Rscript** to query `Rcpp:::CxxFlags` and `Rcpp:::LdFlags`
- **Rscript** to query `RInside:::CxxFlags` and `RInside:::LdFlags`

The `qtdensity.pro` file does the equivalent for **Qt**.

Building with RInside

```
## comment out if you need a different version of R, and set R_HOME
R_HOME :=        $(shell R RHOME)
sources :=       $(wildcard *.cpp)
programs :=      $(sources:.cpp=)

## include headers and libraries for R
RCPPFLAGS :=    $(shell $(R_HOME)/bin/R CMD config --cppflags)
RLDFLAGS :=     $(shell $(R_HOME)/bin/R CMD config --ldflags)
RBLAS :=        $(shell $(R_HOME)/bin/R CMD config BLAS_LIBS)
RLAPACK :=      $(shell $(R_HOME)/bin/R CMD config LAPACK_LIBS)
## if you need to set an rpath to R itself, also uncomment
#RRPATH :=      -Wl,-rpath,$(R_HOME)/lib

## include headers and libraries for Rcpp interface classes
RCPPINCL :=     $(shell echo 'Rcpp:::CxxFlags()' | $(R_HOME)/bin/R --vanilla --slave)
RCPPLIBS :=     $(shell echo 'Rcpp:::LdFlags()'  | $(R_HOME)/bin/R --vanilla --slave)

## include headers and libraries for RInside embedding classes
RINSIDEINCL := $(shell echo 'RInside:::CxxFlags()'|$(R_HOME)/bin/R --vanilla --slave)
RINSIDELIBS := $(shell echo 'RInside:::LdFlags()' |$(R_HOME)/bin/R --vanilla --slave)

## compiler etc settings used in default make rules
CXX :=           $(shell $(R_HOME)/bin/R CMD config CXX)
CPPFLAGS :=      -Wall $(shell $(R_HOME)/bin/R CMD config CPPFLAGS)
CXXFLAGS :=      $(RCPPFLAGS) $(RCPPINCL) $(RINSIDEINCL)
CXXFLAGS +=      $(shell $(R_HOME)/bin/R CMD config CXXFLAGS)
LDLIBS :=        $(RLDFLAGS) $(RRPATH) $(RBLAS) $(RLAPACK) $(RCPPLIBS) $(RINSIDELIBS)

all:            $(programs)
                 @test -x /usr/bin/strip && strip $^
```

Outline

Armadillo

From `arma.sf.net` and slightly edited

What is Armadillo?

Armadillo is a C++ linear algebra library aiming towards a good balance between speed and ease of use. Integer, floating point and complex numbers are supported, as well as a subset of trigonometric and statistics functions. Various matrix decompositions are provided.

A delayed evaluation approach is employed (during compile time) to combine several operations into one and reduce (or eliminate) the need for temporaries. This is accomplished through recursive templates and template meta-programming.

This library is useful if C++ has been decided as the language of choice (due to speed and/or integration capabilities).

Armadillo highlights

- Provides integer, floating point and complex vectors, matrices and fields (3d) with all the common operations.
- Very good documentation and examples at website `http://arma.sf.net`, and a recent technical report (Sanderson, 2010).
- Modern code, building upon and extending from earlier matrix libraries.
- Responsive and active maintainer, frequent updates.

RcppArmadillo highlights

- Template-only builds—no linking, and available whereever R and a compiler work (but **Rcpp** is needed to)!
- Easy to use, just add `LinkingTo: RcppArmadillo, Rcpp` to DESCRIPTION (*i.e.*, no added cost beyond **Rcpp**)
- Really easy from R via **Rcpp**
- Frequently updated, easy to use

Outline

Complete file for fastLM

RcppArmadillo src/fastLm.cpp

```
#include <RcppArmadillo.h>

extern "C" SEXP fastLm(SEXP ys, SEXP Xs) {
  try {
    arma::colvec y = Rcpp::as<arma::colvec>(ys); // direct to arma
    arma::mat X    = Rcpp::as<arma::mat>(Xs);
    int df = X.n_rows - X.n_cols;
    arma::colvec coef = arma::solve(X, y);         // fit model y ~ X
    arma::colvec res  = y - X*coef;                // residuals
    double s2 = std::inner_product(res.begin(), res.end(),
                      res.begin(), 0.0)/df;        // std.errors of coefs
    arma::colvec std_err = arma::sqrt(s2 *
                  arma::diagvec(arma::pinv(arma::trans(X)*X)));
    return Rcpp::List::create(Rcpp::Named("coefsficients")=coef,
                              Rcpp::Named("stderr") = std_err,
                              Rcpp::Named("df")     = df);
  } catch( std::exception &ex ) {
      forward_exception_to_r( ex );
  } catch(...) {
      ::Rf_error( "c++ exception (unknown reason)" );
  }
  return R_NilValue; // -Wall
}
```

Core part of fastLM

RcppArmadillo src/fastLm.cpp

```
arma::colvec y = Rcpp::as<arma::colvec>(ys); // to arma
arma::mat X    = Rcpp::as<arma::mat>(Xs);

int df = X.n_rows - X.n_cols;

arma::colvec coef = arma::solve(X, y);       // fit y ~ X

arma::colvec res  = y - X*coef;              // residuals

double s2 = std::inner_product(res.begin(), res.end(),
                  res.begin(), 0.0)/df;      // std.err coefs

arma::colvec std_err = arma::sqrt(s2 *
        arma::diagvec(arma::pinv(arma::trans(X)*X)));

return Rcpp::List::create(Rcpp::Named("df") = df,
               Rcpp::Named("stderr") = std_err,
               Rcpp::Named("coefficients")     = coef);
```

Easy transfer from (and to) R

RcppArmadillo `src/fastLm.cpp`

```
arma::colvec y = Rcpp::as<arma::colvec>(ys); // to arma
arma::mat X    = Rcpp::as<arma::mat>(Xs);

int df = X.n_rows - X.n_cols;

arma::colvec coef = arma::solve(X, y);         // fit y ~ X

arma::colvec res  = y - X*coef;                // residuals

double s2 = std::inner_product(res.begin(), res.end(),
                res.begin(), 0.0)/df;         // std.err coefs

arma::colvec std_err = arma::sqrt(s2 *
        arma::diagvec(arma::pinv(arma::trans(X)*X)));

return Rcpp::List::create(Rcpp::Named("df") = df,
            Rcpp::Named("stderr") = std_err,
            Rcpp::Named("coefficients")     = coef);
```

Easy linear algebra via Armadillo

```
arma::colvec y = Rcpp::as<arma::colvec>(ys); // to arma
arma::mat X    = Rcpp::as<arma::mat>(Xs);

int df = X.n_rows - X.n_cols;

arma::colvec coef = arma::solve(X, y);        // fit y ~ X

arma::colvec res  = y - X*coef;               // residuals

double s2 = std::inner_product(res.begin(), res.end(),
                 res.begin(), 0.0)/df;        // std.err coefs

arma::colvec std_err = arma::sqrt(s2 *
        arma::diagvec(arma::pinv(arma::trans(X)*X)));

return Rcpp::List::create(Rcpp::Named("df") = df,
             Rcpp::Named("stderr") = std_err,
             Rcpp::Named("coefficients")      = coef);
```

One note on direct casting with Armadillo

The code as just shown:

```
arma::colvec y = Rcpp::as<arma::colvec>(ys);
arma::mat X = Rcpp::as<arma::mat>(Xs);
```

is very convenient, but does incur an additional copy of each object. A lighter variant uses two steps in which only a pointer to the object is copied:

```
Rcpp::NumericVector yr(ys);
Rcpp::NumericMatrix Xr(Xs);
int n = Xr.nrow(), k = Xr.ncol();
arma::mat X(Xr.begin(), n, k, false);
arma::colvec y(yr.begin(), yr.size(), false);
```

If performance is a concern, the latter approach may be preferable.

Performance comparison

Running the script included in the **RcppArmadillo** package:

```
edd@max:~/svn/rcpp/pkg/RcppArmadillo/inst/examples$ r fastLm.r
Loading required package: methods
                        test replications  relative elapsed
1           fLmOneCast(X, y)         5000  1.000000   0.170
2          fLmTwoCasts(X, y)         5000  1.029412   0.175
4    fastLmPureDotCall(X, y)         5000  1.211765   0.206
3           fastLmPure(X, y)         5000  2.235294   0.380
6               lm.fit(X, y)         5000  3.911765   0.665
5 fastLm(frm, data = trees)         5000 40.488235   6.883
7     lm(frm, data = trees)         5000 53.735294   9.135
edd@max:~/svn/rcpp/pkg/RcppArmadillo/inst/examples$
```

NB: This includes a minor change in SVN and not yet in the released package.

Outline

Example: VAR(1) Simulation

examples/part4/varSimulation.r

Lance Bachmeier started this example for his graduate students: Simulate a VAR(1) model row by row:

```
R> ## parameter and error terms used throughout
R> a <- matrix(c(0.5,0.1,0.1,0.5),nrow=2)
R> e <- matrix(rnorm(10000),ncol=2)
R> ## Let's start with the R version
R> rSim <- function(coeff, errors) {
+   simdata <- matrix(0, nrow(errors), ncol(errors))
+   for (row in 2:nrow(errors)) {
+     simdata[row,] = coeff %*% simdata[(row-1),] + errors[row,]
+   }
+   return(simdata)
+ }
R> rData <- rSim(a, e)                              # generated by R
```

Example: VAR(1) Simulation – Compiled R

`examples/part4/varSimulation.r`

With R 2.13.0, we can also compile the R function:

```
R> ## Now let's load the R compiler (requires R 2.13 or later)
R> suppressMessages(require(compiler))
R> compRsim <- cmpfun(rSim)
R> compRData <- compRsim(a,e)                # gen. by R 'compiled'
R> stopifnot(all.equal(rData, compRData))   # checking results
```

Example: VAR(1) Simulation – RcppArmadillo

examples/part4/varSimulation.r

```
R> ## Now load 'inline' to compile C++ code on the fly
R> suppressMessages(require(inline))
R> code <- '
+    arma::mat coeff = Rcpp::as<arma::mat>(a);
+    arma::mat errors = Rcpp::as<arma::mat>(e);
+    int m = errors.n_rows; int n = errors.n_cols;
+    arma::mat simdata(m,n);
+    simdata.row(0) = arma::zeros<arma::mat>(1,n);
+    for (int row=1; row<m; row++) {
+      simdata.row(row) = simdata.row(row-1) *
+                         trans(coeff)+errors.row(row);
+    }
+    return Rcpp::wrap(simdata);
+ '
R> ## create the compiled function
R> rcppSim <- cxxfunction(signature(a="numeric",e="numeric"),
+                         code,plugin="RcppArmadillo")
R> rcppData <- rcppSim(a,e)                   # generated by C++ code
R> stopifnot(all.equal(rData, rcppData))     # checking results
```

Example: VAR(1) Simulation – RcppArmadillo

examples/part4/varSimulation.r

```
R> ## now load the rbenchmark package and compare all three
R> suppressMessages(library(rbenchmark))
R> res <- benchmark(rcppSim(a,e),
+                  rSim(a,e),
+                  compRsim(a,e),
+                  columns=c("test", "replications",
+                            "elapsed", "relative"),
+                  order="relative")
R> print(res)
            test replications elapsed relative
1  rcppSim(a, e)          100   0.038   1.0000
3 compRsim(a, e)          100   2.011  52.9211
2     rSim(a, e)          100   4.148 109.1579
R>
```

So more than fifty times faster than byte-compiled R and more than hundred times faster than R code.

Example: VAR(1) Simulation – RcppArmadillo

examples/part4/varSimulation.r

```
R> ## now load the rbenchmark package and compare all three
R> suppressMessages(library(rbenchmark))
R> res <- benchmark(rcppSim(a,e),
+                  rSim(a,e),
+                  compRsim(a,e),
+                  columns=c("test", "replications",
+                            "elapsed", "relative"),
+                  order="relative")
R> print(res)
           test replications elapsed relative
1  rcppSim(a, e)          100   0.038   1.0000
3 compRsim(a, e)          100   2.011  52.9211
2     rSim(a, e)          100   4.148 109.1579
R>
```

Outline

Kalman Filter

The Mathworks has a nice example[1] of a classic 'object tracking' problem showing gains from going from Matlab code to compiled C code.

The example is short:

```
% Copyright 2010 The MathWorks, Inc.
function y = kalmanfilter(z)
% #codegen
    dt=1;
    % Initialize state transition matrix
    A=[1 0 dt 0 0 0;...       % [x  ]
       0 1 0 dt 0 0;...       % [y  ]
       0 0 1 0 dt 0;...       % [Vx]
       0 0 0 1 0 dt;...       % [Vy]
       0 0 0 0 1 0 ;...       % [Ax]
       0 0 0 0 0 1 ];         % [Ay]
    H = [ 1 0 0 0 0 0; 0 1 0 0 0 0 ];
    Q = eye(6);
    R = 1000 * eye(2);
    persistent x_est p_est
    if isempty(x_est)
        x_est = zeros(6, 1);
        p_est = zeros(6, 6);
    end

    % Predicted state and covariance
    x_prd = A * x_est;
    p_prd = A * p_est * A' + Q;
    % Estimation
    S = H * p_prd' * H' + R;
    B = H * p_prd';
    klm_gain = (S \ B)';
    % Estimated state and covariance
    x_est = x_prd+klm_gain*(z-H*x_prd);
    p_est = p_prd-klm_gain*H*p_prd;
    % Compute the estimated measurements
    y = H * x_est;
end                     % of the function
```

[1] http://www.mathworks.com/products/matlab-coder/demos.html?file=/products/demos/shipping/coder/coderdemo_kalman_filter.html

Kalman Filter: In R

Easy enough – first naive solution

```r
FirstKalmanR <- function(pos) {

  kf <- function(z) {
    dt <- 1

    A <- matrix(c(1, 0, dt, 0, 0, 0,   # x
                  0, 1, 0, dt, 0, 0,   # y
                  0, 0, 1, 0, dt, 0,   # Vx
                  0, 0, 0, 1, 0, dt,   # Vy
                  0, 0, 0, 0, 1,  0,   # Ax
                  0, 0, 0, 0, 0,  1),  # Ay
                6, 6, byrow=TRUE)
    H <- matrix( c(1, 0, 0, 0, 0, 0,
                   0, 1, 0, 0, 0, 0),
                2, 6, byrow=TRUE)
    Q <- diag(6)
    R <- 1000 * diag(2)

    N <- nrow(pos)
    y <- matrix(NA, N, 2)

    ## predicted state and covriance
    xprd <- A %*% xest
    pprd <- A %*% pest %*% t(A) + Q

    ## estimation
    S <- H %*% t(pprd) %*% t(H) + R
    B <- H %*% t(pprd)
    ##  kalmangain <- (S \ B)'
    kg <- t(solve(S, B))

    ## est. state and cov, assign to vars in parent env
    xest <<- xprd + kg %*% (z-H%*%xprd)
    pest <<- pprd - kg %*% H %*% pprd

    ## compute the estimated measurements
    y <- H %*% xest
  }

  xest <- matrix(0, 6, 1)
  pest <- matrix(0, 6, 6)

  for (i in 1:N) {
      y[i,] <- kf(t(pos[i,,drop=FALSE]))
  }

  invisible(y)
}
```

Kalman Filter: In R

Easy enough – with some minor refactoring

```r
KalmanR <- function(pos) {

  kf <- function(z) {
    ## predicted state and covriance
    xprd <- A %*% xest
    pprd <- A %*% pest %*% t(A) + Q

    ## estimation
    S <- H %*% t(pprd) %*% t(H) + R
    B <- H %*% t(pprd)
    ##   kg <- (S \ B)'
    kg <- t(solve(S, B))

    ## estimated state and covariance
    ## assigned to vars in parent env
    xest <<- xprd + kg %*% (z-H%*%xprd)
    pest <<- pprd - kg %*% H %*% pprd

    ## compute the estimated measurements
    y <- H %*% xest
  }
  dt <- 1
  A <- matrix(c(1, 0, dt, 0, 0, 0, # x
                0, 1, 0, dt, 0, 0, # y
                0, 0, 1, 0, dt, 0, # Vx
                0, 0, 0, 1, 0, dt, # Vy
                0, 0, 0, 0, 1,  0, # Ax
                0, 0, 0, 0, 0,  1),# Ay
                6, 6, byrow=TRUE)
  H <- matrix(c(1, 0, 0, 0, 0, 0,
                0, 1, 0, 0, 0, 0),
              2, 6, byrow=TRUE)
  Q <- diag(6)
  R <- 1000 * diag(2)

  N <- nrow(pos)
  y <- matrix(NA, N, 2)

  xest <- matrix(0, 6, 1)
  pest <- matrix(0, 6, 6)

  for (i in 1:N) {
    y[i,] <- kf(t(pos[i,,drop=FALSE]))
  }
  invisible(y)
}
```

Kalman Filter: In C++

Using a simple class

```cpp
using namespace arma;

class Kalman {
private:
  mat A, H, Q, R, xest, pest;
  double dt;

public:
  // constructor, sets up data structures
  Kalman() : dt(1.0) {
    A.eye(6,6);
    A(0,2)=A(1,3)=A(2,4)=A(3,5)=dt;
    H.zeros(2,6);
    H(0,0) = H(1,1) = 1.0;
    Q.eye(6,6);
    R = 1000 * eye(2,2);
    xest.zeros(6,1);
    pest.zeros(6,6);
  }

// sole member function: estimate model
mat estimate(const mat & Z) {
  unsigned int n = Z.n_rows,
               k = Z.n_cols;
  mat Y = zeros(n, k);

  for (unsigned int i = 0; i<n; i++) {
    colvec z = Z.row(i).t();

    // predicted state and covriance
    mat xprd = A * xest;
    mat pprd = A * pest * A.t() + Q;

    // estimation
    mat S = H * pprd.t() * H.t() + R;
    mat B = H * pprd.t();
    mat kg = trans(solve(S, B));

    // estimated state and covariance
    xest = xprd + kg * (z - H * xprd);
    pest = pprd - kg * H * pprd;

    // compute the estimated measurements
    colvec y = H * xest;
    Y.row(i) = y.t();
  }
  return Y;
 }
};
```

Kalman Filter in C++

Trivial to use from R

Given the code from the previous slide in a text variable `kalmanClass`, we just do this

```
kalmanSrc <- '
  mat Z = as<mat>(ZS);        // passed from R
  Kalman K;
  mat Y = K.estimate(Z);
  return wrap(Y);
'

KalmanCpp <- cxxfunction(signature(ZS="numeric"),
                         body=kalmanSrc,
                         include=kalmanClass,
                         plugin="RcppArmadillo")
```

Kalman Filter: Performance

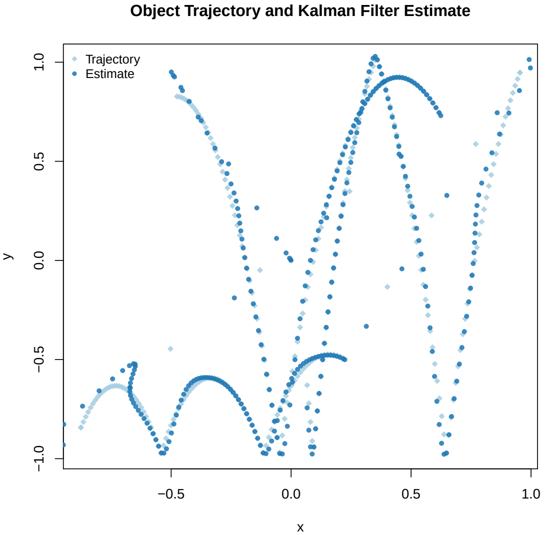
Quite satisfactory relative to R

Even byte-compiled 'better' R version is 66 times slower:

```
R> FirstKalmanRC <- cmpfun(FirstKalmanR)
R> KalmanRC <- cmpfun(KalmanR)
R>
R> stopifnot(identical(KalmanR(pos), KalmanRC(pos)),
+           all.equal(KalmanR(pos), KalmanCpp(pos)),
+           identical(FirstKalmanR(pos), FirstKalmanRC(pos)),
+           all.equal(KalmanR(pos), FirstKalmanR(pos)))
R>
R> res <- benchmark(KalmanR(pos), KalmanRC(pos),
+                  FirstKalmanR(pos), FirstKalmanRC(pos),
+                  KalmanCpp(pos),
+                  columns = c("test", "replications",
+                              "elapsed", "relative"),
+                  order="relative",
+                  replications=100)
R>
R> print(res)
                test replications elapsed relative
5     KalmanCpp(pos)          100   0.087   1.0000
2      KalmanRC(pos)          100   5.774  66.3678
1       KalmanR(pos)          100   6.448  74.1149
4 FirstKalmanRC(pos)          100   8.153  93.7126
3  FirstKalmanR(pos)          100   8.901 102.3103
```

Kalman Filter: Figure

Last but not least we can redo the plot as well



Outline

RcppEigen

RcppEigen wraps the Eigen library for linear algebra.

Eigen is similar to Armadillo, and very highly optimised—by internal routines replacing even the BLAS for performance.

Eigen is also offering a more complete API than Armadillo (but I prefer to work with the simpler Armadillo, most of the time).

RcppEigen is written mostly by Doug Bates who needs sparse matrix support for his C++ rewrite of **lme4** (e.g. **lme4eigen**).

Eigen can be faster than Armadillo. Andreas Alfons' CRAN package **robustHD** (using Armadillo) with a drop-in replacement **sparseLTSEigen** sees gain of 1/4 to 1/3.

However, Eigen is not always available on all platforms as there can be issues with older compilers (eg on OS X).

Outline

RcppEigen's fastLm

Slightly simplified / shortened

```
const MMatrixXd        X(as<MMatrixXd>(Xs));
const MVectorXd        y(as<MVectorXd>(ys));
Index                  n = X.rows(), p = X.cols();
lm                   ans = do_lm(X, y, ::Rf_asInteger(type));
NumericVector       coef = wrap(ans.coef());
List            dimnames = NumericMatrix(Xs).attr("dimnames");
VectorXd           resid = y - ans.fitted();
double                s2 = resid.squaredNorm()/ans.df();
PermutationType     Pmat = PermutationType(p);
Pmat.indices()           = ans.perm();
VectorXd              dd = Pmat * ans.unsc().diagonal();
ArrayXd               se = (dd.array() * s2).sqrt();
return List::create(_["coefficients"]  = coef,
                    _["se"]            = se,
                    _["rank"]          = ans.rank(),
                    _["df.residual"]   = ans.df(),
                    _["perm"]          = ans.perm(),
                    _["residuals"]     = resid,
                    _["s2"]            = s2,
                    _["fitted.values"] = ans.fitted(),
                    _["unsc"]          = ans.unsc());
```

RcppEigen's fastLm (cont.)
The lm alternatives

Doug defines a base class `lm` from which the following classes derive:

- LLt (standard Cholesky decomposition)
- LDLt (robust Cholesky decompostion with pivoting)
- SymmEigen (standard Eigen-decomposition)
- QR (standard QR decomposition)
- ColPivQR (Householder rank-revealing QR decomposition with column-pivoting)
- SVD (standard SVD decomposition)

The example file `lmBenchmark.R` in the package runs through these.

RcppEigen's fastLm (cont.)

The benchmark results

```
lm benchmark for n = 100000 and p = 40: nrep = 20
       test   relative elapsed user.self sys.self
3      LDLt   1.000000   0.911      0.91     0.00
7       LLt   1.000000   0.911      0.91     0.00
5  SymmEig   2.833150   2.581      2.17     0.40
6        QR   5.050494   4.601      4.17     0.41
2 ColPivQR   5.102086   4.648      4.20     0.43
8      arma   6.837541   6.229      6.00     0.00
1    lm.fit   9.189901   8.372      7.12     1.14
4       SVD  32.183315  29.319     28.44     0.76
9       GSL 113.680571 103.563    102.42     0.53
```

This improves significantly over the Armadillo-based solution.

One last remark on the fastLm routines

Doug sometimes reminds us about the occassional fine differences between *statistical* numerical analysis and standard numerical analysis.

Pivoting schemes are a good example. R uses a custom decomposition (with pivoting) inside of `lm()` which makes it both robust and precise, particularly for rank-deficient matrices.

The example for `fastLm` in both **RcppArmadillo** and **RcppEigen** provides an illustration.

If you are *really* sure your data is well-behaved, then using a faster (non-pivoting) scheme as in **RcppArmadillo**

Outline

3 RcppEigen
- Overview
- Example

4 RcppGSL
- Overview
- Example

RcppGSL

RcppGSL is a convenience wrapper for accessing the **GNU GSL**, particularly for vector and matrix functions.

Given that the **GSL** is a C library, we need to

- do memory management and free objects
- arrange for the GSL linker to be found

RcppGSL may still be a convenient tool for programmers more familiar with C than C++ wanting to deploy GSL algorithms.

Outline

Vector norm example—*c.f.* GSL manual

examples/part4/gslNorm.cpp

```
#include <RcppGSL.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>

extern "C" SEXP colNorm(SEXP sM) {
  try {
    RcppGSL::matrix<double> M = sM;      // SEXP to gsl data structure
    int k = M.ncol();
    Rcpp::NumericVector n(k);            // to store results
    for (int j = 0; j < k; j++) {
      RcppGSL::vector_view<double> colview =
                               gsl_matrix_column (M, j);
      n[j] = gsl_blas_dnrm2(colview);
    }
    M.free();
    return n;                            // return vector
  } catch( std::exception &ex ) {
    forward_exception_to_r( ex );
  } catch(...) {
    ::Rf_error( "c++ exception (unknown reason)" );
  }
  return R_NilValue; // -Wall
}
```

Core part of example

`examples/part4/gslNorm.cpp`

```
RcppGSL::matrix<double> M = sM;      // SEXP to GSL data
int k = M.ncol();
Rcpp::NumericVector n(k);            // to store results

for (int j = 0; j < k; j++) {
  RcppGSL::vector_view<double> colview =
                              gsl_matrix_column (M, j);
  n[j] = gsl_blas_dnrm2(colview);
}
M.free();
return n;                            // return vector
```

Core part of example

Using standard GSL functions: `examples/part4/gslNorm.cpp`

```
RcppGSL::matrix<double> M = sM;        // SEXP to GSL data
int k = M.ncol();
Rcpp::NumericVector n(k);              // to store results

for (int j = 0; j < k; j++) {
  RcppGSL::vector_view<double> colview =
                         gsl_matrix_column (M, j);
  n[j] = gsl_blas_dnrm2(colview);
}
M.free();
return n;                              // return vector
```

Outline

Gibbs Sampler Example

Darren Wilkinson wrote a couple of blog posts illustrating the performance of different implementations (C, Java, Python, ...) for a simple MCMC Gibbs sampler of this bivariate density::

$$f(x, y) = kx^2 \exp(-xy^2 - y^2 + 2y - 4x)$$

with conditional distributions

$$\begin{aligned} f(x|y) &\sim \text{Gamma}(3, y^2 + 4) \\ f(y|x) &\sim N\left(\frac{1}{1+x}, \frac{1}{2(1+x)}\right) \end{aligned}$$

i.e. we need repeated RNG draws from both a Gamma and a Gaussian distribution.

Outline

Gibbs Sampler Example

Sanjog Misra then sent me working R and C++ versions which I extended. In R we use:

```
Rgibbs <- function(N,thin) {
    mat <- matrix(0,ncol=2,nrow=N)
    x <- 0
    y <- 0
    for (i in 1:N) {
        for (j in 1:thin) {
            x <- rgamma(1,3,y*y+4)
            y <- rnorm(1,1/(x+1),1/sqrt(2*(x+1)))
        }
        mat[i,] <- c(x,y)
    }
    mat
}
```

Outline

Gibbs Sampler Example (cont.)

The C++ version using **Rcpp** closely resembles the R version::

```
gibbscode <- '
  // n and thin are SEXPs which the Rcpp::as function maps to C++ vars
  int N   = as<int>(n);
  int thn = as<int>(thin);
  int i,j;
  NumericMatrix mat(N, 2);
  RNGScope scope;           // Initialize Random number generator
  double x=0, y=0;
  for (i=0; i<N; i++) {
    for (j=0; j<thn; j++) {
      x = ::Rf_rgamma(3.0,1.0/(y*y+4));
      y = ::Rf_rnorm(1.0/(x+1),1.0/sqrt(2*x+2));
    }
    mat(i,0) = x;
    mat(i,1) = y;
  }
  return mat;               // Return to R
'
```

Gibbs Sampler Example (cont.)

We compile the C++ function:

```
# Compile and Load
RcppGibbs <- cxxfunction(signature(n="int",
                                   thin = "int"),
                         gibbscode, plugin="Rcpp")
```

Outline

Gibbs Sampler Example (cont.)

We also create a similar variant using the GSL's random number generators (as in Darren's example):

```
gslgibbscode <- '
  int N = as<int>(ns);
  int thin = as<int>(thns);
  int i, j;
  gsl_rng *r = gsl_rng_alloc(gsl_rng_mt19937);
  double x=0, y=0;
  NumericMatrix mat(N, 2);
  for (i=0; i<N; i++) {
    for (j=0; j<thin; j++) {
      x = gsl_ran_gamma(r,3.0,1.0/(y*y+4));
      y = 1.0/(x+1)+gsl_ran_gaussian(r,1.0/sqrt(2*x+2));
    }
    mat(i,0) = x;
    mat(i,1) = y;
  }
  gsl_rng_free(r);
  return mat;              // Return to R
'
```

Gibbs Sampler Example (cont.)

We compile the GSL / C function:

```
gslgibbsincl <- '
  #include <gsl/gsl_rng.h>
  #include <gsl/gsl_randist.h>

  using namespace Rcpp;  // just to be explicit
'

## Compile and Load
GSLGibbs <- cxxfunction(signature(ns="int",
                                  thns = "int"),
                        body=gslgibbscode,
                        includes=gslgibbsincl,
                        plugin="RcppGSL")
```

Outline

Gibbs Sampler Example (cont.)

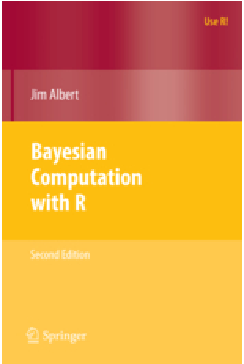
The result show a dramatic gain from the two compiled version relative to the R version, and the byte-compiled R version:

```
                  test repl. elapsed  relative user.self sys.self
4   GSLGibbs(N, thn)    10   7.918  1.000000      7.87     0.00
3  RcppGibbs(N, thn)    10  12.300  1.553423     12.25     0.00
2    RCgibbs(N, thn)    10 306.349 38.690200    305.07     0.11
1     Rgibbs(N, thn)    10 412.467 52.092321    410.76     0.18
```

The gain of the **GSL** version relative to the **Rcpp** is due almost entirely to a much faster RNG for the gamma distribution as shown by `timeRNGs.R`.

Outline

Accelerating Monte Carlo



Albert. *Bayesian Computation with R*, 2nd ed. Springer, 2009

Albert introduces simulations with a simple example in the first chapter.

We will study this example and translate it to R using RcppArmadillo (and Rcpp).

The idea is to, for a given level α, and sizes *n* and *m*, draw a number *N* of samples at these sizes, compoute a *t*-statistic and record if the test statistic exceeds the theoretical critical value given the parameters.

This allows us to study the impact of varying α, *N* or *M* — as well as varying parameters or even families of the random vectors.

Restating the problem

- With two samples $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ we can test

$$H_0 : \mu_x = \mu_y$$

- With sample means $\bar{X}$ and $\bar{Y}$, and s_x and $_y$ as respective standard deviations, the standard test is

$$T = \frac{\bar{X} - \bar{Y}}{s_P \sqrt{1/m + 1/n}}$$

whew s_p is the pooled standard deviation

$$s_p = \sqrt{\frac{(m-1)s_x^2 + (n-1)s_y^2}{m+n-2}}$$

Restating the problem

- Under H_0, we have $T \sim t(m+n-2)$ provided that
  - x_i and $x+i$ are NID
  - the standard deviations of populations x and y are equal.
- For a given level α, we can reject H if

$$|T| \geq t_{n+m-2,\alpha/2}$$

- But happens when we have
  - unequal population variances, or
  - non-normal distributions?
- Simulations can tell us.

Outline

Basic R version

Core function: `examples/part4/montecarlo.r`

```
## Section 1.3.3
## simulation algorithm for normal populations
sim1_3_3_R <- function() {
    alpha <- .1; m <- 10; n <- 10 # sets alpha, m, n
    N <- 10000                    # sets nb of sims
    n.reject <- 0                 # number of rejections
    crit <- qt(1-alpha/2,n+m-2)
    for (i in 1:N) {
        x <- rnorm(m,mean=0,sd=1) # simulates xs from population 1
        y <- rnorm(n,mean=0,sd=1) # simulates ys from population 2
        t.stat <- tstatistic(x,y) # computes the t statistic
        if (abs(t.stat)>crit)
            n.reject=n.reject+1   # reject if |t| exceeds critical pt
    }
    true.sig.level <- n.reject/N  # est. is proportion of rejections
}
```

Basic R version

Helper function for *t*-statistic: `examples/part4/montecarlo.r`

```r
## helper function
tstatistic <- function(x,y) {
    m <- length(x)
    n <- length(y)
    sp <- sqrt(((m-1)*sd(x)^2 + (n-1)*sd(y)^2) / (m+n-2))
    t.stat <- (mean(x) - mean(y)) / (sp*sqrt(1/m + 1/n))
    return(t.stat)
}
```

Outline

RcppArmadillo version

Main function: examples/part4/montecarlo.r

```
sim1_3_3_arma <- cxxfunction(, plugin="RcppArmadillo",
                             inc=tstat_arma, body='
  RNGScope scope;        // properly deal with RNGs
  double alpha = 0.1;
  int m = 10, n = 10;    // sets alpha, m, n
  int N = 10000;         // sets the number of sims
  double n_reject = 0;   // counter of num. of rejects
  double crit = ::Rf_qt(1.0-alpha/2.0, n+m-2.0,true,false);
  for (int i=0; i<N; i++)  {
    NumericVector x = rnorm(m, 0, 1);    // sim xs from pop 1
    NumericVector y = rnorm(n, 0, 1);    // sim ys from pop 2
    double t_stat = tstatistic(Rcpp::as<arma::vec>(x),
                               Rcpp::as<arma::vec>(y));
    if (fabs(t_stat) > crit)
      n_reject++;                 // reject if |t| exceeds critical pt
  }
  double true_sig_level = 1.0*n_reject / N;  // est. prop rejects
  return(wrap(true_sig_level));
')
```

RcppArmadillo version

Helper function for *t*-statistic: : `examples/part4/montecarlo.r`

```
tstat_arma <- '
 double tstatistic(const NumericVector &x,
                   const NumericVector &y) {
   int m = x.size();
   int n = y.size();
   double sp = sqrt( ( (m-1.0)*pow(sd(x),2) +
                 (n-1)*pow(sd(y),2) ) / (m+n-2.0) );
   double t_stat = (mean(x)-mean(y))/(sp*sqrt(1.0/m+1.0/n));
   return(t_stat);
 }
'
```

Outline

Rcpp version—using sugar functions `mean`, `sd`, ...

Main function: `examples/part4/montecarlo.r`

```
sim1_3_3_rcpp <- cxxfunction(, plugin="Rcpp",
                                 inc=tstat_rcpp, body='
  RNGScope scope;       // properly deal with RNG settings
  double alpha = 0.1;
  int m = 10, n = 10; // sets alpha, m, n
  int N = 10000;        // sets the number of simulations
  double n_reject = 0;  // counter of num. of rejections
  double crit = ::Rf_qt(1.0-alpha/2.0, n+m-2.0, true, false);
  for (int i=0; i<N; i++)  {
    NumericVector x = rnorm(m, 0, 1);     // sim xs from pop 1
    NumericVector y = rnorm(n, 0, 1);     // sim ys from pop 2
    double t_stat = tstatistic(x, y);
    if (fabs(t_stat) > crit)
      n_reject++;           // reject if |t| exceeds critical pt
  }
  double true_sig_level = 1.0*n_reject / N;  // est. prop rejects
  return(wrap(true_sig_level));
')
```

Rcpp version—using SVN version with `mean`, `sd`, ...

Helper function: `examples/part4/montecarlo.r`

```
tstat_rcpp <- '
 double tstatistic(const NumericVector &x,
                   const NumericVector &y) {
   int m = x.size();
   int n = y.size();
   double sp = sqrt( ( (m-1.0)*pow(sd(x),2) +
                       (n-1)*pow(sd(y),2) ) / (m+n-2.0) );
   double t_stat = (mean(x)-mean(y))/(sp*sqrt(1.0/m+1.0/n));
   return(t_stat);
 }
'
```

Outline

Benchmark results

examples/part4/montecarlo.r

```
R> library(rbenchmark)
R> res <- benchmark(sim1_3_3_R(),
+                  sim1_3_3_Rcomp(),
+                  sim1_3_3_arma(),
+                  sim1_3_3_rcpp(),
+                  columns=c("test", "replications",
+                            "elapsed", "relative",
+                            "user.self"),
+                  order="relative")
R> res
              test replications elapsed relative user.self
3  sim1_3_3_arma()          100   2.118  1.00000      2.12
4  sim1_3_3_rcpp()          100   2.192  1.03494      2.19
1     sim1_3_3_R()          100 153.772 72.60246    153.70
2 sim1_3_3_Rcomp()          100 154.251 72.82861    154.19
R>
```

Benchmark results

```
R> res
                test replications elapsed relative user.self
3  sim1_3_3_arma()          100   2.118  1.00000      2.12
4  sim1_3_3_rcpp()          100   2.192  1.03494      2.19
1     sim1_3_3_R()          100 153.772 72.60246    153.70
2 sim1_3_3_Rcomp()          100 154.251 72.82861    154.19
R>
```

In this example, the R compiler does not help at all. The difference between **RcppArmadillo** and **Rcpp** is neglible.

Suggestions (by Albert): replace *n*, *m*, standard deviations of Normal RNG, replace Nornal RNG, ... which, thanks to **Rcpp** and 'Rcpp sugar' is a snap.

Simulation results

examples/part4/montecarlo.r

Albert reports this table:

Populations	True Sign. Level
Normal pop. with equal spreads	0.0986
Normal pop. with unequal spreads	0.1127
$t(4)$ distr. with equal spreads	0.0968
Expon. pop. with equal spreads	0.1019
Normal + exp. pop. with unequal spreads	0.1563

Table: True significance level of t-test computed by simulation; standard error of each estimate is approximately 0.003.

Our simulations are $\approx$ 70-times faster, so we can increase the number of simulation by 100 and reduce the standard error to $\sqrt{0.1 \times 0.9/1,000,000} = 0.0003$.

That's it, folks!

Want to learn more ?

- Check all the vignettes
- Ask questions on the `Rcpp-devel` mailing list
- Hands-on training courses and consulting are available

Romain François **romain@r-enthusiasts.com**
Dirk Eddelbuettel **edd@debian.org**