Compiling parts of R using the NIMBLE system for programming algorithms

Christopher Paciorek UC Berkeley Statistics

Joint work with:

Perry de Valpine (PI) UC Berkeley Environmental Science, Policy and Managem't
Daniel Turek UC Berkeley Statistics and ESPM
Cliff Anderson-Bergman Lawrence Livermore Lab (alumnus)
Duncan Temple Lang UC Davis Statistics

http://r-nimble.org

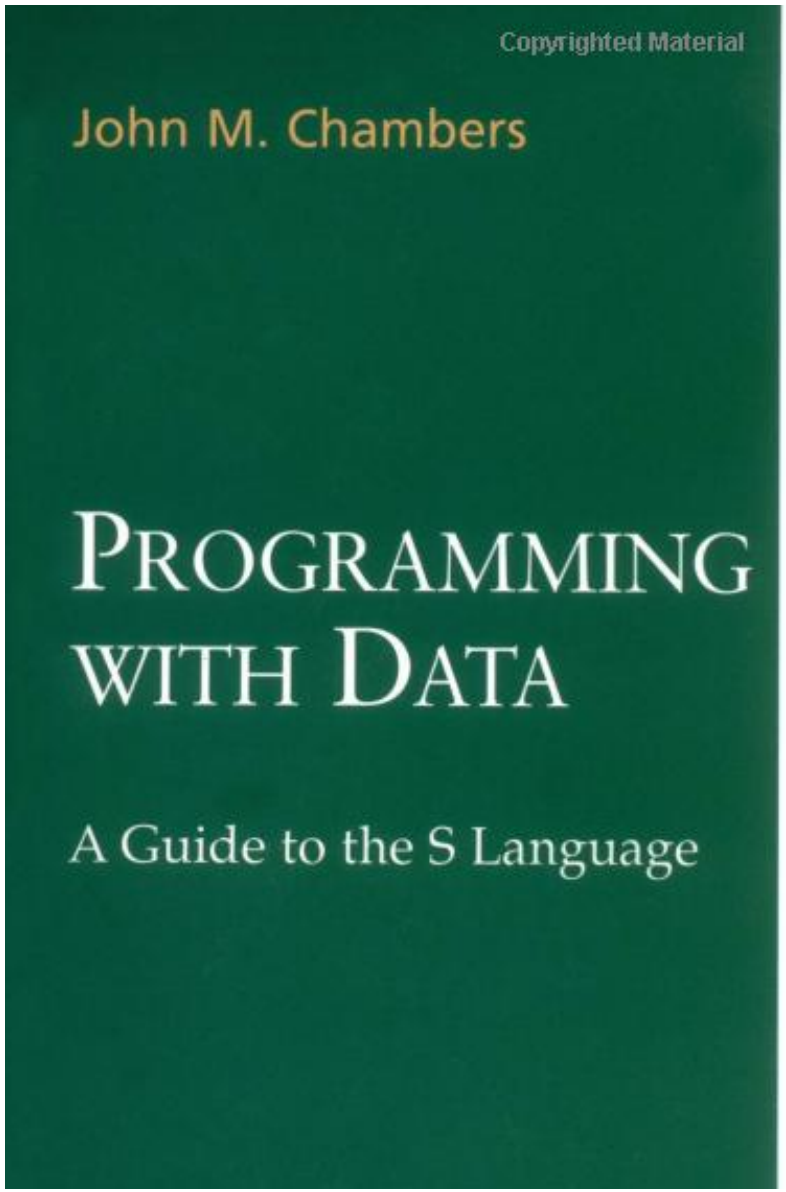
useR Conference
June 2016

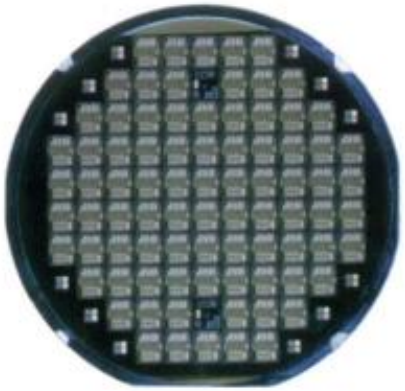
Funded by NSF DBI-1147230

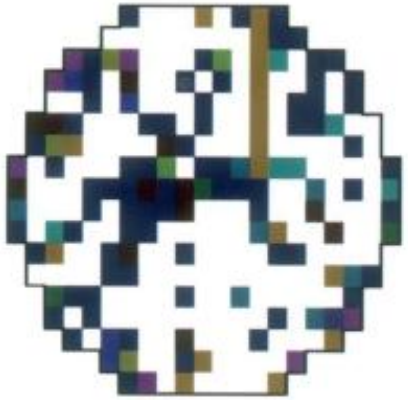
NIMBLE Background and Goals

- Software for fitting hierarchical models has opened their use to a wide variety of communities
- Most software for fitting such models is either model-specific or algorithm-specific
- Software for general models such as BUGS/JAGS is a black box and hard to extend
- Our goal is to combine flexible model specification with flexible algorithm programming, while
  - Retaining BUGS compatibility
  - Providing a variety of standard algorithms
  - Allowing developers to add new algorithms (including modular combination of algorithms)
  - **Allowing users to operate within R**
  - **Providing speed via compilation to C++, with R wrappers**

The Success of R







NIMBLE System Components

1. Hierarchical model specification

   BUGS language → R/C++ model object

2. Algorithm programming via nimbleFunctions

   NIMBLE programming language (DSL) within R → R/C++ algorithm object

3. Algorithm library

   MCMC, Particle Filter/Sequential MC, MCEM, etc.

Using nimbleFunctions to compile R

R code for a Markov chain

```
mc <- function(n, rho1, rho2) {
   path <- rep(0, n)
   path[1:2] <- rnorm(2)
   for(i in 3:n)
      path[i] <- rho1*path[i-1] + rho2*path[i-2] + rnorm(1)
   return(path)
}
```

NIMBLE code

```
nim_mc <- nimbleFunction(
  run = function(n = double(0), rho1 = double(0), rho2 = double(0)) {
     returnType(double(1))
     path <- numeric(n, init = FALSE)
     path[1] <- rnorm(1)
     path[2] <- rnorm(1)
     for(i in 3:n)
          path[i] <- rho1*path[i-1] + rho2*path[i-2] + rnorm(1)
     return(path)
})
```

NIMBLE DSL

Compile to C++ (and then to binary)

```
cnim_mc <- compileNimble(nim_mc)
```

Using nimbleFunctions to compile R

```
cnim_mc<- compileNimble(nim_mc)
#g++ -I/usr/share/R/include -DNDEBUG -DEIGEN_MPL2_ONLY=1 -I"/home/paciorek/R/
x86_64/3.2/nimble/include"     -fpic  -g -O2 -fstack-protector --param=ssp-buffer-size=4 -
Wformat -Werror=format-security -D_FORTIFY_SOURCE=2 -g  -c P_1_rcFun_4.cpp -o
P_1_rcFun_4.o
#g++ -shared -L/usr/lib/R/lib -Wl,-Bsymbolic-functions -Wl,-z,relro -o
P_1_rcFun_09_02_02.so P_1_rcFun_4.o -L/home/paciorek/R/x86_64/3.2/nimble/CppCode -
Wl,-rpath=/home/paciorek/R/x86_64/3.2/nimble/CppCode -lnimble -L/usr/lib/R/lib -lR

n <- 1e6
rho1 <- .8; rho2 <- .1
set.seed(0)
system.time( path1 <- mc(n, rho1, rho2) )    # original R version
#   user  system elapsed
#  3.883   0.001   3.883
set.seed(0)
system.time( path2 <- cnim_mc(n, rho1, rho2) )   # compiled version
#   user  system elapsed
#  0.070   0.004   0.074
> identical(path1, path2)
[1] TRUE
```

Using nimbleFunctions for Algorithms

Users can write nimbleFunctions for use with statistical models to:

- Code their own algorithms
- Create user-defined MCMC samplers for use in NIMBLE's MCMC engine
- Write distributions and functions for use in BUGS code

nimbleFunctions that work with models have two components:

- setup function that is written in R and provides information to specialize an algorithm to a model
- run function (as we have seen) that encodes generic execution of algorithm on arbitrary model

Using nimbleFunctions for Algorithms

```
sampler_RW <- nimbleFunction(
   contains = sampler_BASE,
    setup = function(model, mvSaved, targetNode) {
        calculationNodes  <- model$getDependencies(targetNode)
    },
    run = function(scale = double(0)) {
        ## get current log probabilities
        logProb_current <- getLogProb(model, calculationNodes)
        ## Make proposal and put in model
        proposalValue <- rnorm(1, mean = model[[targetNode]],
            sd = scale)
        model[[targetNode]] <<- proposalValue
        ## Calculate proposal log probabilities
        logProb_proposal <- calculate(model, calculationNodes)
        ## accept or reject
        log_Metropolis_Hastings_ratio <- logProb_proposal –
            logProb_current
        accept <- decide(log_Metropolis_Hastings_ratio)
        ## update saved states of model
        if(accept)
            copy(from = model, to = mvSaved, row = 1,
                 nodes = calculationNodes, logProb = TRUE)
        else
            copy(from = mvSaved, to = model, row = 1,
                 nodes = calculationNodes, logProb = TRUE)
        returnType(integer(0))
        return(accept)
    })
```

R code

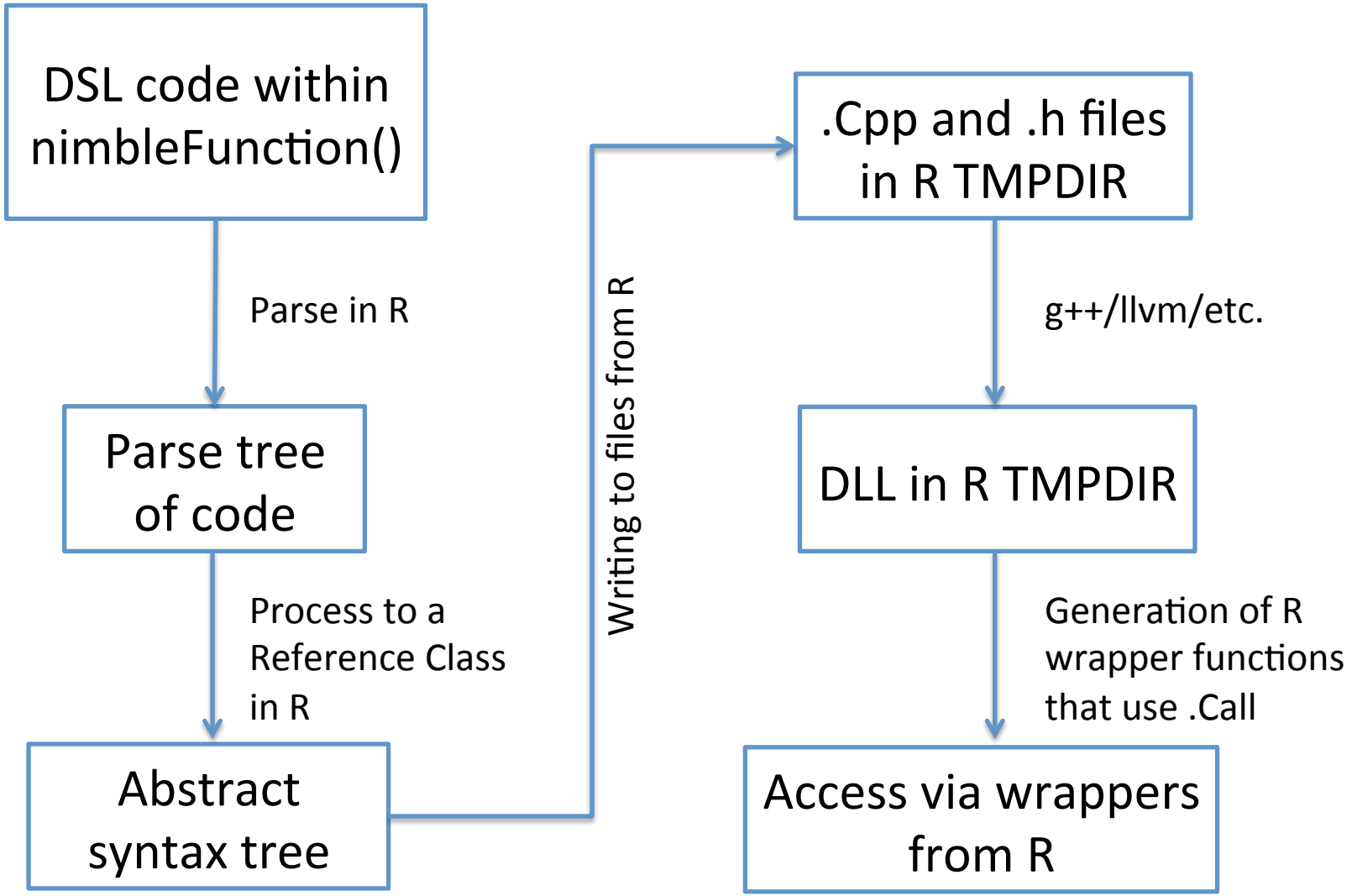
NIMBLE DSL

The NIMBLE compiler

Feature summary:

- R-like matrix algebra (using Eigen library)
- R-like indexing (e.g. X[1:5,])
- Sequential integer iteration
- if-then-else, while
- Handles multiple and nested functions
- Instantiation of variables
- Access to much of Rmath.h (e.g. distributions)
- Automatic R interface / wrapper
- Many improvements / extensions planned
- Use of model variables and nodes
- Model calculate (logProb) and simulate functions

How R code is Compiled in NIMBLE



Key steps in compiling R -> C++

nf <- nimbleFunction(…)

Generate **custom reference class definition**

nfOneCase <- nf(setup arguments…)

Evaluate setup code in R (possible for multiple cases)

Symbol table initiated from setup code results

Run function and other member functions converted to **Abstract Syntax Tree (AST)**.

Partial evaluation of some functions (mostly for generic model uses).

AST transformed and annotated:
- Types inferred
- Symbol table populated
- Sizes tracked as expressions
- Resizing and size-checking calls inserted
- Intermediate variables inserted
- Labeling for Eigen compatibility
- Insertion of Eigen matrix / map setup

Creation of reference class object to **manage C ++ function/class content**.
- Also creates AST for C function for .C()
- Includes generic void* system to access any member data easily from R.

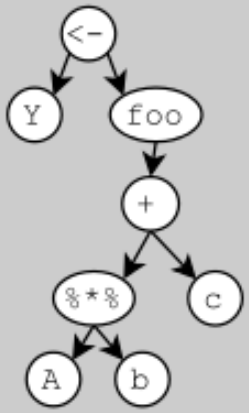
Write .cpp and .h files and compile them

Generate reference class definition to **access function or object(s) of compiled code**
- creates natural R calls
- allows natural access to C++ member data

Compilation steps

(a) Original NIMBLE code: `Y <- foo(A %*% b + c) ## %*% is matrix multiplication in R`

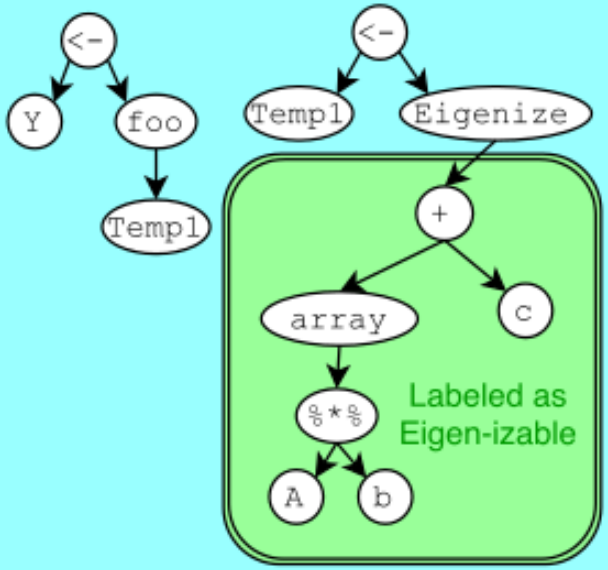
(b): Create Abstract Syntax Tree (AST)



(c): Label types at every AST vertex (not shown)

(d). Add Y to symbol table if needed

(e). Label for Eigen and transform as needed



(f). Add Temp1 and necessary Eigen variables to symbol table.

Future

Annotate and transform AST for

- distributed processing
- automatic differentiation

(g) Final C++

```
double Y;
NimbleArray<2, double> Temp1;
EigenMap Eig_Temp1, Eig_A, Eig_b, Eig_c;
// pointer and resizing details omitted
Temp1 = (Eig_A * Eig_b).array() + Eig_c;
Y = foo(Temp1);
```

Basic example: user experience

```
example <- nimbleFunction(
    run = function(x = double(1), max = double(0), prob = double(0)) {
        # type/size information
        returnType(double(1))
        n <- length(x)
        out <- numeric(n, init = FALSE)
        # core computation
        for( i in 1:n) {
            out[i] <- exp(x[i])
            if(out[i] > max)
                if(runif(1) < prob)
                    out[i] <- max
        }
        return(out)
})

cExample <- compileNimble(example)
# g++ -I/usr/share/R/include -DNDEBUG -DEIGEN_MPL2_ONLY=1 ……
input <- rnorm(4)
set.seed(0);    example(input, 1, 0.8)      # run R-based nimbleFunction
# [1] 1.0369570 0.7250988 1.0000000 0.1078377
set.seed(0);   cExample(input, 1, 0.8)     # run compiled nimbleFunction
# [1] 1.0369570 0.7250988 1.0000000 0.1078377
```

Basic example: calls from R

```
> example
function (x, max, prob)
{
    n <- length(x)
    out <- nimNumeric(n, init = FALSE)
    for (i in 1:n) {
        out[i] <- exp(x[i])
        if (out[i] > max)
            if (runif(1) < prob)
                out[i] <- max
    }
    return(out)
}
<environment: 0x533a330>

> cExample
function (x, max, prob)
{
    ans <- .Call(list(name = "CALL_rcFun_4", address = <pointer: 0x7f46cdc09d60>,
        package = NULL), x, max, prob)
    ans <- ans[[4]]
    ans
}
```

Basic example: generated C++ code

```
NimArr<1, double>  rcFun_4 ( NimArr<1, double> & ARG1_x_, double ARG2_max_, double
ARG3_prob_ ) {
	int n;
	NimArr<1, double> out;
	int i;
	n = ARG1_x_.size();
	out.initialize(0, 0, n);
	for(i=1; i<= static_cast<int>(n); ++i) {
		out[(i) - 1] = exp(ARG1_x_[(i) - 1]);
		if(out[(i) - 1] > ARG2_max_) {
			if(runif(0, 1) < ARG3_prob_) {
				out[(i) - 1] = ARG2_max_;
			}
		}
	}
	return(out);
}

SEXP  CALL_rcFun_4 ( SEXP S_ARG1_x_, SEXP S_ARG2_max_, SEXP S_ARG3_prob_ ) {
  // ...
}
```

Basic example using Eigen for vectorization

Uncompiled nimbleFunction (DSL) code

```
example_vec <- nimbleFunction(
    run = function(x = double(1)) {
        returnType(double(1))
        out <- acos(tanh(x))
        return(out)
})
```

Compiled C++ code

```
NimArr<1, double>  rcFun_5 ( NimArr<1, double> & ARG1_x_ ) {
        NimArr<1, double> out;
        Map<MatrixXd> Eig_out(0,0,0);
        EigenMapStr Eig_ARG1_x_Interm_1(0,0,0, EigStrDyn(0, 0));
        out.setSize(ARG1_x_.dim()[0]);
        new (&Eig_out) Map< MatrixXd >(out.getPtr(),ARG1_x_.dim()[0],1);
        new (&Eig_ARG1_x_Interm_1) EigenMapStr(ARG1_x_.getPtr() +
        static_cast<int>(ARG1_x_.getOffset() + static_cast<int>(0)),ARG1_x_.dim()[0],1,EigStrDyn(0,
                        ARG1_x_.strides()[0]));
        Eig_out = (((Eig_ARG1_x_Interm_1).array()).unaryExpr(std::ptr_fun<double, double>(tanh))).acos();
return(out);
}
```

Compiler Extensibility

- Compiler is written in R with extensibility in mind.
- Adding new functions requires:
  - Possible syntax modification
  - A function to annotate AST with appropriate sizes and types (can be an existing function or a new one)
  - Determination of C++ output format
  - Other details
- Adding new types is more involved.
- Goal is to automate /isolate some extensibility steps.

Goals for extending NIMBLE

- Advanced math
  - Automatic differentiation (generate code to use existing C++ library)
  - More linear algebra (sparsity and more)
- Advanced computing
  - Distributed computing (in particular openMP, GPUs)
  - More modular compilation units
  - More native use of R objects:
    - Less copying
    - Access to lists
  - More generic interfaces to compiled code from languages other than R
- Faster R processing
  - Some R steps of compilation process are slow
- Better support & extensions for BUGS language
  - Stochastic indexing
  - More algorithms
  - Faster processing

Interested?

- First release was June 2014; version 0.6 in process of being released on CRAN.
- Lots of information (manual, examples, etc.) on r-nimble.org
- Announcements: nimble-announce Google site
- User support/discussion: nimble-users Google site
- Write an algorithm using NIMBLE!
- Help with development of NIMBLE: email nimble.stats@gmail.com or see github.com/nimble-dev