Flexible Programming of Hierarchical Modeling Algorithms and Compilation of R Using NIMBLE

Christopher Paciorek  UC Berkeley Statistics

Joint work with:
- Perry de Valpine (PI)  UC Berkeley Environmental Science, Policy and Managem’t
- Daniel Turek  Williams College Math and Statistics
- Nick Michaud  UC Berkeley ESPM and Statistics
- Duncan Temple Lang  UC Davis Statistics

Bayesian nonparametrics development with:
- Claudia Wehrhahn Cortes  UC Santa Cruz Applied Math and Statistics
- Abel Rodriguez  UC Santa Cruz Applied Math and Statistics

http://r-nimble.org

October 2017

Funded by NSF DBI-1147230, ACI-1550488, DMS-1622444; Google Summer of Code 2015, 2017
Hierarchical statistical models

A basic random effects / Bayesian hierarchical model

Probabilistic model

\[
\begin{align*}
\alpha & \sim \text{Exp}(1) \\
\beta & \sim \text{Gamma}(0.1, 1.0) \\
\theta_i & \sim \text{Gamma}(\alpha, \beta) \\
\lambda_i & \sim \text{Poisson}(\lambda_i) \\
\end{align*}
\]
Hierarchical statistical models

A basic random effects / Bayesian hierarchical model

BUGS DSL code

```r
# priors on hyperparameters
alpha ~ dexp(1.0)
beta ~ dgamma(0.1,1.0)
for (i in 1:N){
  # latent process (random effects)
  # random effects distribution
  theta[i] ~ dgamma(alpha,beta)
  # linear predictor
  lambda[i] <- theta[i]*t[i]
  # likelihood (data model)
x[i] ~ dpois(lambda[i])
}
```

Probabilistic model

\[
\begin{align*}
\alpha & \sim \text{Exp}(1) \\
\beta & \sim \text{Gamma}(0.1, 1.0) \\
\theta_i & \sim \text{Gamma}(\alpha, \beta) \\
\lambda_i & \leftarrow \theta_i t_i \\
x_i & \sim \text{Poisson}(\lambda_i)
\end{align*}
\]
Divorcing model specification from algorithm

Your new method

Variational Bayes

MCEM

Quadrature

Maximum likelihood

MCMC Flavor 1

MCMC Flavor 2

Particle Filter

Importance Sampler

Flexible programming of hierarchical modeling algorithms using NIMBLE (r-nimble.org)
What can a practitioner do with hierarchical models?

Two basic software designs:

1. Typical R/Python package = Model family + 1 or more algorithms
   - GLMMs: lme4, MCMCglmm
   - GAMMs: mgcv
   - spatial models: spBayes, INLA
What can a practitioner do with hierarchical models?

Two basic software designs:

1. Typical R/Python package = Model family + 1 or more algorithms
   - GLMMs: lme4, MCMCglmm
   - GAMMns: mgcv
   - spatial models: spBayes, INLA

2. Flexible model + black box algorithm
   - BUGS: WinBUGS, OpenBUGS, JAGS
   - PyMC
   - INLA
   - Stan

Flexible programming of hierarchical modeling algorithms using NIMBLE (r-nimble.org)
Existing software

Examples: BUGS (WinBUGS, OpenBUGS, JAGS), INLA, Stan

Widely used in various disciplines: environmental sciences, social sciences, biomedical/health sciences, statistics

Flexible programming of hierarchical modeling algorithms using NIMBLE (r-nimble.org)
NIMBLE: The Goal

Model

\[ X(1) \rightarrow X(2) \rightarrow X(3) \]

\[ Y(1) \rightarrow Y(2) \rightarrow Y(3) \]

Algorithm language

Flexible programming of hierarchical modeling algorithms using NIMBLE (r-nimble.org)
NIMBLE philosophy

• Combine flexible model specification with flexible algorithm programming, while
  – Retaining BUGS DSL 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
NIMBLE system components

1. Hierarchical model specification

   BUGS language \(\rightarrow\) R/C++ model object

2. Algorithm library

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

3. Algorithm programming via nimbleFunctions

   NIMBLE programming language (DSL) within R \(\rightarrow\) R/C++
   algorithm object
You give NIMBLE BUGS DSL code:

```r
pumpCode <- nimbleCode( 
  # priors on hyperparameters
  alpha ~ dexp(1.0)
  beta ~ dgamma(0.1,1.0)
  for (i in 1:N){
    theta[i] ~ dgamma(alpha,beta)
    lambda[i] <- theta[i]*t[i]
    x[i] ~ dpois(lambda[i])
  }
}
)
```

You get a programmable model object:

```r
> pumpModel$theta[1] <- 5  # set values in model
> simulate(pumpModel, 'theta')  # simulate from prior
> beta_deps <- pumpModel$getDependencies('beta')  # model structure
> calculate(pumpModel, beta_deps)  # calculate probability density
> getLogProb(pumpModel, 'theta')
```

Flexible programming of hierarchical modeling algorithms using NIMBLE (r-nimble.org)
pumpCode <- nimbleCode( 
  alpha ~ dexp(1.0)
  beta ~ dgamma(0.1, 1.0)
  for (i in 1:N){
    theta[i] ~ dgamma(alpha, beta)
    lambda[i] <- theta[i]*t[i]
    x[i] ~ dpois(lambda[i])
  }
)

> pumpMCMCconf <- configureMCMC(pumpModel)
> pumpMCMCconf$printSamplers()
[1] RW sampler: alpha       
[...snip...]
> pumpMCMCconf$addSampler('alpha', 'slice', list(adaptInterval = 100))
> pumpMCMCconf$removeSamplers('beta')
> pumpMCMCconf$addSampler('beta', 'slice', list(adaptInterval = 100))
> pumpMCMCconf$addMonitors('theta')
> pumpMCMC <- buildMCMC(pumpMCMCspec)
> pumpMCMC_Cpp <- compileNimble(pumpMCMC, project = pumpModel)
> pumpMCMC_Cpp$run(20000)
NIMBLE system components

1. Hierarchical model specification

   BUGS language ➔ R/C++ model object

2. Algorithm library

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

3. Algorithm programming via nimbleFunctions

   NIMBLE programming language (DSL) within R ➔ R/C++ algorithm object
NIMBLE’s algorithm library

– MCMC samplers:
  • Conjugate, adaptive Metropolis, adaptive blocked Metropolis, slice, elliptical slice sampler, particle MCMC, specialized samplers for particular distributions (Dirichlet, CAR)
  • Flexible choice of sampler for each parameter
  • User-specified blocks of parameters

– Sequential Monte Carlo (particle filters)
  • Various flavors

– MCEM

– Write your own
NIMBLE system components

1. Hierarchical model specification
   
   BUGS language $\Rightarrow$ R/C++ model object

2. Algorithm library
   
   MCMC, Particle Filter/Sequential MC, MCEM, etc.

3. Algorithm programming via nimbleFunctions
   
   NIMBLE programming language (DSL) within R $\Rightarrow$ R/C++ algorithm object
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 that encodes generic execution of algorithm on arbitrary model
NIMBLE: programming with models

sampler_myMetropolis_RandomWalk <- nimbleFunction(

calculate

setup = function(model, mvSaved, targetNode, scale) {
    calcNodes <- model$getDependencies(targetNode)
},

run = function() {
    model_lp_initial <- calculate(model, calcNodes)
    proposal <- rnorm(1, model[[targetNode]], scale)
    model[[targetNode]] <<- proposal
    model_lp_proposed <- calculate(model, calcNodes)
    log_MH_ratio <- model_lp_proposed - model_lp_initial

    if(decide(log_MH_ratio)) jump <- TRUE
    else jump <- FALSE
# .... Various bookkeeping operations ...
})
sampler_myRW <- nimbleFunction(

setup = function(model, mvSaved, targetNode, scale) {
  calcNodes <- model$getDependencies(targetNode)
},

run = function() {
  model_lp_initial <- calculate(model, calcNodes)
  proposal <- rnorm(1, model[[targetNode]], scale)
  model[[targetNode]] <<- proposal
  model_lp_proposed <- calculate(model, calcNodes)
  log_MH_ratio <- model_lp_proposed - model_lp_initial
  if(decide(log_MH_ratio)) jump <- TRUE
  else jump <- FALSE
  # .... Various bookkeeping operations ... #
})
NIMBLE: programming with models

sampler_myRW <- nimbleFunction(

setup = function(model, mvSaved, targetNode, scale) {
  calcNodes <- model$getDependencies(targetNode)
},
run = function() {
  model_lp_initial <- calculate(model, calcNodes)
  proposal <- rnorm(1, model[[targetNode]], scale)
  model[[targetNode]] <<- proposal
  model_lp_proposed <- calculate(model, calcNodes)
  log_MH_ratio <- model_lp_proposed - model_lp_initial

  if(decide(log_MH_ratio)) jump <- TRUE
  else jump <- FALSE

  # .... Various bookkeeping operations ... #
})
Using nimbleFunctions to compile R

R code for a Markov chain

```r
cmc <- function(n, rho1, rho2) {
  path <- rep(0, n)  # initialize
  path[1:2] <- rnorm(2)
  for(i in 3:n)  # propagate forward in time
    path[i] <- rho1*path[i-1] + rho2*path[i-2] + rnorm(1)
  return(path)
}
```

NIMBLE code

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

Compile to C++ (and then to executable)

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

Flexible programming of hierarchical modeling algorithms using NIMBLE (r-nimble.org)
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,-Bs symbolic-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 -

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
The NIMBLE compiler (NIMBLE DSL code)

Feature summary:

- R-like matrix algebra (using Eigen library)
- R-like indexing (e.g. x[1:5,])
- Use of model variables and nodes
- Model calculate (logProb) and simulate functions
- Sequential integer iteration
- If-then-else, do-while
- Access to much of Rmath.h (e.g. distributions)
- Automatic R interface / wrapper
- Call out to your own C/C++ or back to R
- Many improvements / extensions planned
How DSL code is compiled in NIMBLE

- DSL code within nimbleFunction()
  - Parse in R
    - Parse tree of code
      - Process in R
        - Abstract syntax tree
  - .Cpp and .h files in R TMPDIR
    - g++/llvm/etc.
      - Shared library in R TMPDIR
        - Generation of R wrapper functions that use .Call
          - Access via wrappers from R

Flexible programming of hierarchical modeling algorithms using NIMBLE (r-nimble.org)
### Key steps in compiling R -> C++

- **Generate custom class definition**
- **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 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 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: \( Y \leftarrow \text{foo}(A \times b + c) \) \# \# \# \# 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.

(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: calls from R

> nim_mc
function (n, rho1, rho2)
{
  path <- nimNumeric(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)
}

> cnim_mc
function (n, rho1, rho2)
{
  if (is.null(CnativeSymbolInfo_)) {
    warning("Trying to call compiled nimbleFunction that does not exist (may have been cleared).")
    return(NULL)
  }
  ans <- .Call(CnativeSymbolInfo_, n, rho1, rho2)
  ans <- ans[[4]]
  ans
}
Basic example: generated C++ code

NimArr<1, double> rcFun_2 ( double ARG1_n_, double ARG2_rho1_, double ARG3_rho2_ ) {
    NimArr<1, double> path;
    double i;
    path.initialize(0, false, true, true, ARG1_n_);
    path[0] = rnorm(0, 1);
    path[1] = rnorm(0, 1);
    for(i=3; i<= static_cast<int>(ARG1_n_); ++i) {
        path[(i) - 1] = (ARG2_rho1_ * path[(i - 1) - 1] + ARG3_rho2_ * path[(i - 2) - 1]) + rnorm(0, 1);
    }
    return(path);
}

SEXP CALL_rcFun_4 ( SEXP S_ARG1_n_, SEXP S_ARG2_rho1_, SEXP S_ARG3_rho2_ ) {
    // ...
}
Basic example using Eigen for vectorization

Uncompiled nimbleFunction (DSL) code

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

Compiled C++ code

```cpp
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/allows:
  – 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++ CppAD library): well underway
  – More linear algebra (sparsity and more)

• Advanced computing
  – Parallelization via compilation to Tensorflow (in place of Eigen): initial steps done
  – More modular compilation units
  – More native use of R objects in C++ (less copying)

• Scalability
  – Faster R processing of model and algorithm code
  – Vectorization of algorithms for replicated model nodes

• More algorithms
Interested?

- Version 0.6-6 on R package repository (CRAN)
- Lots of information (manual, examples, etc.) on r-nimble.org
- Development: github.com/nimble-dev/nimble
- Announcements: nimble-announce Google site
- User support/discussion: nimble-users Google site