nimble models: : CHEAT SHEET

NIMBLE workflow

required arguments
optional arguments

Write model code

`nimbleCode()` NIMBLE lang
`readBUGSmodel()` read BUGS

Create model object

`nimbleModel()`

from `nimbleCode()`
constants of the model e.g. for-loop ranges, known index vectors
values to label as data nodes
starting values for the parameters

```
myModel <- nimbleModel(
  code = modelCode,
  constants = list(N = 8,
  numGroups = 4, ...),
  data = list(y = myData,
    X = myCovs),
  inits = list(beta0 = 0,
    beta1 = 0, ...))
```

`constants` **can't be** changed after creating a model
`data & inits` **can** be changed

Configure an MCMC algorithm

`configureMCMC()`

Advanced customization

Build an MCMC object

`buildMCMC()`

model object or **MCMC configuration**
variables names for MCMC output
thinning interval
a second set of variables with its own thinning

```
myMCMC <- buildMCMC(
  conf = [myModel|myConf],
  monitors = c("beta0",...),
  thin = 10,
  monitors2, thin2)
```

Run MCMC

`runMCMC()` using MCMC object
`nimbleMCMC()` one-line invocation

Compile in C++ for faster execution
MCMC from `buildMCMC()` compiled or uncompiled
number of MCMC iterations & burnin

```
compMCMC <- compileNimble(
myMCMC,project = myModel)

samples <- runMCMC(
  mcmc = compMCMC,
  iter=1000, nburnin=100)
```

from `nimbleCode()`
as in `nimbleModel()`
number of MCMC iterations & burnin

```
samples <- nimbleMCMC(
  code = modelCode,
  data, constants, inits,
  niter, nburnin)
```

Writing model code

Split code over multiple lines to help people read it.

```
modelCode <- nimbleCode({
  beta0 ~ dnorm(0, sd = 1000)
  beta1 ~ dnorm(0, 1E-6)
  sdGroups ~ dunif(0, 100)
  fixed_effects[1:N] <- beta0 + beta1 * X[1:N]
  for(i in 1:N) {
    log(eta[i]) <- fixed_effects[i] +
                   alpha[groupID[i]]
    y[i] ~ dpois(eta[i])
  }
  for(iGroup in 1:numGroups) {
    alpha[iGroup] ~ dnorm(0, sd = sdGroups)
  }
})
```

Use named arguments for non-default parameterization e.g. `beta0` and `beta1` follow equivalent distributions (default is precision, `tau`).

Link functions can be declared on the left-hand side.

Order of declaration does not matter `alpha[iGroup]` can be declared after being used in other declarations.

Vectorized declarations create vector nodes. This means `fixed_effects[1:N]` will be a single node. One vector node vs. multiple scalar nodes give different model graphs, so use with care.

Provide explicit index ranges or use empty brackets (`[]`) and provide the `dimensions` argument to `nimbleModel()`.

Nested indexing is a good way to implement experimental groups or factor levels. If groups are known from the design, include them in `constants`.

Using models

Models can be compiled.
`cModel <- compileNimble(myModel)`
In methods below, "`model`" can be `cModel` or `myModel`.

Models can access and change variables.
`model$beta0 <- 5`
`model[["beta0"]] <- 5`

Models can simulate or calculate log-probabilities.

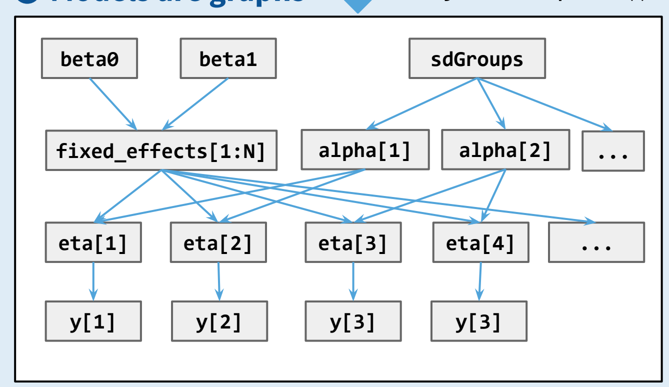
`model$calculate(nodes)`
returns sum of log probability densities.

`model$calculateDiff(nodes)`
returns difference in sum of log probability densities between current and previous node values.

`model$getLogProb(nodes)`
returns sum of most recently calculated log probability densities.

`model$simulate(nodes, includeData = FALSE)`
simulates into stochastic nodes. `includeData = FALSE` protects data.

Models are graphs `myModel$plot()`



Models know about nodes, variables and relationships.

`model$getNodesNames()`
returns node names
e.g. `"eta[1]", "eta[2]",...`

`model$getVarNames()`
returns variable names
e.g. `"eta"`

`model$expandNodeNames(nodes)`
e.g. `"y"` is expanded to `"y[1]", "y[2]",...`

`model$getDependencies(nodes, ...)`
returns nodes that depend on input nodes.

Uncompiled models can be debugged, updated, and copied.

Flag nodes as data and set inits
`myModel$setData("y")`
`myModel$setInits(inits)`

Debug model errors
`myModel$check()`
check for missing/invalid values.
`myModel$initializeInfo()`
which nodes are not fully initialized?
`myModel$checkBasics()`
check for size/dimension mismatches and `NA`.

Make a copy
`myModel$newModel(replicate = TRUE)`

Models know properties of nodes.

`model$getDimension(node)`
`model$getDistribution(nodes)`
`model$isDeterm(nodes)`
`model$isStoch(nodes)`
`model$isData(nodes)`
`model$isDiscrete(nodes)`
`model$isMultivariate(nodes)`
`model$isBinary(nodes)`
`model$isEndNode(nodes)`
`model$isTruncated(nodes)`

NIMBLE

nimble distributions and functions: : CHEAT SHEET

Declarations

STOCHASTIC
`x ~ ddist(args)`
DETERMINISTIC
`z <- fn(args)`
TRUNCATED STOCHASTIC
`x ~ T(ddist(args), min, max)`
CENSORED STOCHASTIC
`seg ~ dinterval(t, c[1:nSegments])`
`t ~ ddist(args)`
CONSTRAINT
`one ~ dconstraint(condition)`

Deterministic Functions

SCALAR or COMPONENT-WISE

Logical: `|, &, !, >, >=, <, <=, !=, ==, equals, step`
Arithmetic: `+, -, *, /, ^, pow(x, y) %%, exp, log, sqrt, abs, cube`
Trigonometric: `sin, cos, tan, asin, acos, atan, asinh, acosh, atanh`
Links: `logit, probit, cloglog`
(links can also be used on left-hand side of a declaration)
Inverse links: `ilogit/expit, iprobit/phi, icloglog`
Rounding: `ceiling, floor, round, trunc`
Specials: `lgamma/loggam, besselK, log1p, lfactorial, logfact`
Distributions: `d, p, q, r` forms of available distributions can be used as deterministic functions.

VECTOR and/or MATRIX
Returning scalar: `inprod, logdet, sum, mean, sd, prod, min, max`
Returning vector: `pmin, pmax, eigen(x)$values, svd(x)$d`
Returning matrix: `inverse, chol, %*%, t, solve, forwardsolve, backsolve, eigen(x)$vectors, svd(x)$u, svd(x)$v`

Write you own!

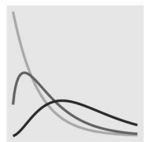
See Ch 12 of User Manual

NIMBLE allows you to write **new distributions and functions** using `nimbleFunction()`.

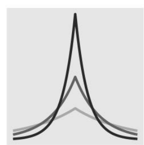
Univariate Distributions

Continuous

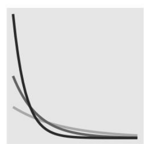
BETA
`y ~ dbeta([shape1,shape2 | mean, sd])`
`shape1=mean^2*(1-mean)/sd^2-mean`
`shape2=mean*(1-mean)^2/sd^2+mean-1`



CHI-SQUARE
`y ~ dchisq(df)`

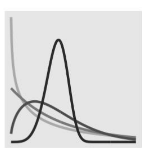


DOUBLE EXPONENTIAL (LAPLACE)
`y ~ ddexp(location, [scale|rate|var])`
`scale = 1/rate`
`scale = sqrt(var/2)`



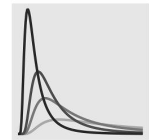
EXPONENTIAL
`y ~ dexp([rate|scale])`
`rate = 1/scale`

FLAT (improper)
`y ~ dflat()`

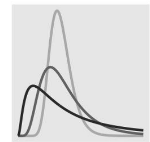


GAMMA
`y ~ dgamma([shape,[rate|scale]] |[mean,sd])`
`scale = 1/rate`
`shape = mean^2/sd^2`
`scale = sd^2/mean`

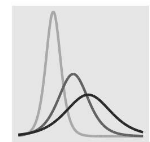
HALF FLAT (improper)
`y ~ dhalfflat()`



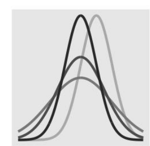
INVERSE GAMMA
`y ~ dinvgamma(shape, [rate|scale])`
`rate = 1/scale`



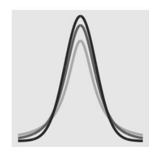
LOGISTIC
`y ~ dlogis(location, [rate|scale])`
`scale = 1/rate`



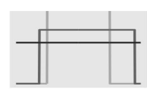
LOG-NORMAL
`y ~ dlnorm(meanlog, [taulog|sdlog|varlog])`
`sdlog = 1/sqrt(taulog)`
`sdlog = sqrt(varlog)`



NORMAL
`y ~ dnorm(mean, [tau|sd|var])`
`sd = 1/sqrt(tau)`
`sd = sqrt(var)`



STUDENT T
`y ~ dt(mu, [tau|sigma|sigma2], df)`
`sigma = 1/sqrt(tau)`
`sigma = sqrt(sigma2)`



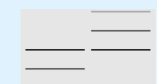
UNIFORM
`y ~ dunif(min, max)`

WEIBULL
`y ~ dweib(shape, [lambda|scale|rate])`
`scale = lambda^(-1/shape)`
`scale = 1/rate`

DISTRIBUTION NAME
`y ~ ddist([default|alternative])`
`canonical = fn(provided)`

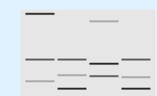
Lifted nodes are inserted when non-canonical parameters are used. Default parameters are not necessarily canonical.

Discrete

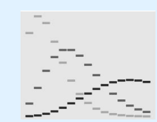


BERNOULLI
`y ~ dbern(prob)`

BINOMIAL
`y ~ dbinom(prob, size)`



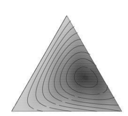
CATEGORICAL
`y ~ dcat(prob)`



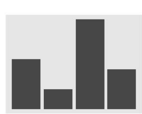
NEGATIVE BINOMIAL
`y ~ dnegbin(prob, size)`

POISSON
`y ~ dpois(lambda)`

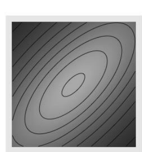
Multivariate distributions



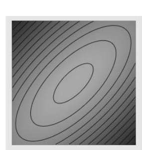
DIRICHLET
`y[] ~ ddirch(alpha[])`



MULTINOMIAL
`y[] ~ dmulti(prob[], size)`

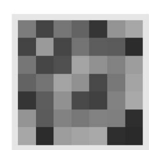


MULTIVARIATE NORMAL
`y[] ~ dmnorm(mean[], [prec[,] | cov[,] | cholesky[,]], prec_param)`



MULTIVARIATE STUDENT T
`y[] ~ dmvt(mu[], [prec[,] | scale[,] | cholesky[,]], df, prec_param)`

cholesky = `chol(prec)` : `prec_param=1`
cholesky = `chol(cov)` : `prec_param=0` for `dmnorm`
cholesky = `chol(scale)`: `prec_param=0` for `dmvt`
cholesky is `chol(prec)` when `prec_param=1`,
`chol(cov)|chol(scale)` when `prec_param=0`

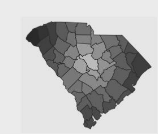


WISHART
`y[,] ~ dwish( [R[,] | S[,] | cholesky[,]], df, scale_param)`

INVERSE WISHART
`y[,] ~ dinvwish([S[,] | R[,] | cholesky[,]], df,scale_param)`

cholesky = `chol(R)`: `scale_param=0`
cholesky = `chol(S)`: `scale_param=1`
cholesky is `chol(S)` when `scale_param=1`,
`chol(R)` when `scale_param=0`

Distributions for spatial models

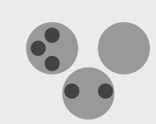


CONDITIONAL AUTOREGRESSIVE
intrinsic (improper)
`y[] ~ dcar_normal(adj[], weights[], num[], tau, c, zero_mean)`
proper
`y[] ~ dcar_proper(mu[], C[], adj[], num[], M[], tau, gamma)`

See Ch 9 of User Manual

Bayesian nonparametric distributions

See Ch 10 of User Manual



CHINESE RESTAURANT PROCESS
`y[] ~ dCRP(conc, size)`
conc= concentration parameter

STICK BREAKING PROCESS
`y[] ~ stick_breaking(z[])`
z = vector of breaking points