Package: nimbleQuad (via r-universe)

May 15, 2026

Title Laplace Approximation, Quadrature, and Nested Deterministic Approximation Methods for 'nimble'

Description Provides deterministic approximation methods for use with the 'nimble' package. These include Laplace approximation and higher-order extension of Laplace approximation using adaptive Gauss-Hermite quadrature (AGHQ), plus nested deterministic approximation methods related to the 'INLA' approach. Additional information is available in the NIMBLE User Manual and a 'nimbleQuad' tutorial, both available at <https://r-nimble.org/documentation.html>.

Version 1.4.0

Date 2026-01-08

Maintainer Christopher Paciorek <paciorek@stat.berkeley.edu>

Depends R (>= 3.5.0), nimble (>= 1.4.0)

Imports methods, R6, pracma, polynom

Suggests testthat, mvQuad, faraway

License BSD_3_clause + file LICENSE | GPL (>= 2)

Copyright See COPYRIGHTS file.

Encoding UTF-8

RoxygenNote 7.3.2

Collate quadratureRules.R quadratureGrids.R Laplace.R buildNestedApprox.R runNestedApprox.R summaryUtils.R

NeedsCompilation no

Author Paul van Dam-Bates [aut], Perry de Valpine [aut], Wei Zhang [aut], Christopher Paciorek [aut, cre], Daniel Turek [aut]

Config/pak/sysreqs libglpk-dev make libxml2-dev

Repository https://paciorek.r-universe.dev

Date/Publication 2026-01-14 09:10:08 UTC

RemoteUrl https://github.com/cran/nimbleQuad

RemoteRef HEAD

RemoteSha f23664a8998f2e3c1a1810b24e5689f778df3fc5

Contents

`approxSummary` *Main class for nested approximation information*

Description

This class holds the result of `runNestedApprox` and provides methods for improving and extending inference using the nested approximation.

Details

See `runNestedApprox` for an overview of usage, including example usage of the methods. Each method has an accompanying wrapper function (with the same name) that takes the `approxSummary` object as its first argument, with the remaining arguments the same as for the method. See the help information on the accompanying function for more detailed information, e.g., `improveParamMarginals`.

Methods

Public methods:

- approxSummary$new()
- approxSummary$generateParamsMatrix()
- approxSummary$print()
- approxSummary$setParamGrid()
- approxSummary$improveParamMarginals()
- approxSummary$calcMarginalLogLikImproved()
- approxSummary$sampleParams()
- approxSummary$sampleLatents()
- approxSummary$qmarginal()
- approxSummary$rmarginal()
- approxSummary$dmarginal()
- approxSummary$emarginal()
- approxSummary$plotMarginal()
- approxSummary$clone()

Method `new()`:

Usage:

```
approxSummary$new(
  approx,
  quantiles,
  expectations,
  marginalsApprox,
  marginalsRaw,
  indivParamTransforms,
  originalScale,
  marginalLogLik,
  marginalLogLikImproved,
  samples,
  paramSamples
)
```

Method `generateParamsMatrix()`:

Usage:

```
approxSummary$generateParamsMatrix()
```

Method `print()`:

Usage:

```
approxSummary$print()
```

Method `setParamGrid()`:

Usage:

```
approxSummary$setParamGrid(summary, quadRule = "NULL", nQuad = -1, prune = -1)
```

Method `improveParamMarginals()`**:**

Usage:

```
approxSummary$improveParamMarginals(
  nodes,
  nMarginalGrid = 5,
  nQuad = 3,
  quadRule = "NULL",
  prune = -1,
  transform = "spectral"
)
```

Method `calcMarginalLogLikImproved()`**:**

Usage:

```
approxSummary$calcMarginalLogLikImproved()
```

Method `sampleParams()`**:**

Usage:

```
approxSummary$sampleParams(n = 1000, matchMarginals = TRUE)
```

Method `sampleLatents()`**:**

Usage:

```
approxSummary$sampleLatents(n = 1000, includeParams = FALSE)
```

Method `qmarginal()`**:**

Usage:

```
approxSummary$qmarginal(node, quantiles = c(0.025, 0.25, 0.5, 0.75, 0.975))
```

Method `rmarginal()`**:**

Usage:

```
approxSummary$rmarginal(node, n = 1000)
```

Method `dmarginal()`**:**

Usage:

```
approxSummary$dmarginal(node, x, log = FALSE)
```

Method `emarginal()`**:**

Usage:

```
approxSummary$emarginal(node, functional, ...)
```

Method `plotMarginal()`**:**

Usage:

```
approxSummary$plotMarginal(
  node,
  log = FALSE,
  xlim = NULL,
  ngrid = 200,
  add = FALSE,
  ...
)
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
approxSummary$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`buildLaplace` *Laplace approximation and adaptive Gauss-Hermite quadrature*

Description

Build a Laplace or AGHQ approximation algorithm for a given NIMBLE model.

Usage

```
buildLaplace(
  model,
  paramNodes,
  randomEffectsNodes,
  calcNodes,
  calcNodesOther,
  control = list()
)

buildAGHQ(
  model,
  nQuad = 1,
  paramNodes,
  randomEffectsNodes,
  calcNodes,
  calcNodesOther,
  control = list()
)
```

Arguments

`model`	a NIMBLE model object, such as returned by `nimbleModel`. The model must have automatic derivatives (AD) turned on, e.g. by using `buildDerivs=TRUE` in `nimbleModel`.
`paramNodes`	a character vector of names of parameter nodes in the model; defaults are provided by `setupMargNodes`. Alternatively, `paramNodes` can be a list in the format returned by `setupMargNodes`, in which case `randomEffectsNodes`, `calcNodes`, and `calcNodesOther` are not needed (and will be ignored).
`randomEffectsNodes`	a character vector of names of continuous unobserved (latent) nodes to marginalize (integrate) over using Laplace/AGHQ approximation; defaults are provided by `setupMargNodes`.

`calcNodes` a character vector of names of nodes for calculating the integrand for Laplace/AGHQ approximation; defaults are provided by `setupMargNodes`. There may be deterministic nodes between `paramNodes` and `calcNodes`. These will be included in calculations automatically and thus do not need to be included in `calcNodes` (but there is no problem if they are).

`calcNodesOther` a character vector of names of nodes for calculating terms in the log-likelihood that do not depend on any `randomEffectsNodes`, and thus are not part of the marginalization, but should be included for purposes of finding the MLE. This defaults to stochastic nodes that depend on `paramNodes` but are not part of and do not depend on `randomEffectsNodes`. There may be deterministic nodes between `paramNodes` and `calcNodesOther`. These will be included in calculations automatically and thus do not need to be included in `calcNodesOther` (but there is no problem if they are).

`control` a named list for providing additional settings used in Laplace/AGHQ approximation. See `control` section below. Most of these can be updated later with the 'updateSettings' method.

`nQuad` number of quadrature points for AGHQ (in one dimension). Laplace approximation is AGHQ with 'nQuad=1'. Only odd numbers of nodes really make sense. Often only one or a few nodes can achieve high accuracy. A maximum of 35 nodes is supported. Note that for multivariate quadratures, the number of nodes will be (number of dimensions)^nQuad.

`buildLaplace` **and** `buildAGHQ`

`buildLaplace` creates an object that can run Laplace approximation for a given model or part of a model. `buildAGHQ` creates an object that can run adaptive Gauss-Hermite quadrature (AGHQ, sometimes called "adaptive Gaussian quadrature") for a given model or part of a model. Laplace approximation is AGHQ with one quadrature point, hence 'buildLaplace' simply calls 'buildAGHQ' with 'nQuad=1'. These methods approximate the integration over continuous random effects in a hierarchical model to calculate the (marginal) likelihood.

`buildAGHQ` and `buildLaplace` will by default (unless changed manually via 'control$split') determine from the model which random effects can be integrated over (marginalized) independently. For example, in a GLMM with a grouping factor and an independent random effect intercept for each group, the random effects can be marginalized as a set of univariate approximations rather than one multivariate approximation. On the other hand, correlated or nested random effects would require multivariate marginalization.

Maximum likelihood estimation is available for Laplace approximation ('nQuad=1') with univariate or multivariate integrations. With 'nQuad > 1', maximum likelihood estimation is available only if all integrations are univariate (e.g., a set of univariate random effects). If there are multivariate integrations, these can be calculated at chosen input parameters but not maximized over parameters. For example, one can find the MLE based on Laplace approximation and then increase 'nQuad' (using the 'updateSettings' method below) to check on accuracy of the marginal log likelihood at the MLE.

Beware that quadrature will use 'nQuad^k' quadrature points, where 'k' is the dimension of each integration. Therefore quadrature for 'k' greater that 2 or 3 can be slow. As just noted, 'buildAGHQ' will determine independent dimensions of quadrature, so it is fine to have a set of univariate random

effects, as these will each have k=1. Multivariate quadrature (k>1) is only necessary for nested, correlated, or otherwise dependent random effects.

The recommended way to find the maximum likelihood estimate and associated outputs is by calling runLaplace or runAGHQ. The input should be the compiled Laplace or AGHQ algorithm object. This would be produced by running compileNimble with input that is the result of buildLaplace or buildAGHQ.

For more granular control, see below for methods findMLE and summary. See function summaryLaplace for an easier way to call the summary method and obtain results that include node names. These steps are all done within runLaplace and runAGHQ.

The NIMBLE User Manual at r-nimble.org also contains an example of Laplace approximation.

How input nodes are processed

buildLaplace and buildAGHQ make good tries at deciding what to do with the input model and any (optional) of the node arguments. However, random effects (over which approximate integration will be done) can be written in models in multiple equivalent ways, and customized use cases may call for integrating over chosen parts of a model. Hence, one can take full charge of how different parts of the model will be used.

Any of the input node vectors, when provided, will be processed using `nodes <- model$expandNodeNames(nodes)`, where nodes may be paramNodes, randomEffectsNodes, and so on. This step allows any of the inputs to include node-name-like syntax that might contain multiple nodes. For example, `paramNodes = 'beta[1:10]'` can be provided if there are actually 10 scalar parameters, 'beta[1]' through 'beta[10]'. The actual node names in the model will be determined by the exapndNodeNames step.

In many (but not all) cases, one only needs to provide a NIMBLE model object and then the function will construct reasonable defaults necessary for Laplace approximation to marginalize over all continuous latent states (aka random effects) in a model. The default values for the four groups of nodes are obtained by calling setupMargNodes, whose arguments match those here (except for a few arguments which are taken from control list elements here).

setupMargNodes tries to give sensible defaults from any combination of paramNodes, randomEffectsNodes, calcNodes, and calcNodesOther that are provided. For example, if you provide only randomEffectsNodes (perhaps you want to marginalize over only some of the random effects in your model), setupMargNodes will try to determine appropriate choices for the others.

setupMargNodes also determines which integration dimensions are conditionally independent, i.e., which can be done separately from each other. For example, when possible, 10 univariate random effects will be split into 10 univariate integration problems rather than one 10-dimensional integration problem.

The defaults make general assumptions such as that randomEffectsNodes have paramNodes as parents. However, The steps for determining defaults are not simple, and it is possible that they will be refined in the future. It is also possible that they simply don't give what you want for a particular model. One example where they will not give desired results can occur when random effects have no prior parameters, such as 'N(0,1)' nodes that will be multiplied by a scale factor (e.g. sigma) and added to other explanatory terms in a model. Such nodes look like top-level parameters in terms of model structure, so you must provide a randomEffectsNodes argument to indicate which they are.

It can be helpful to call setupMargNodes directly to see exactly how nodes will be arranged for Laplace approximation. For example, you may want to verify the choice of randomEffectsNodes

or get the order of parameters it has established to use for making sense of the MLE and results from the summary method. One can also call setupMargNodes, customize the returned list, and then provide that to buildLaplace as paramNodes. In that case, setupMargNodes will not be called (again) by buildLaplace.

If setupMargNodes is emitting an unnecessary warning, simply use control=list(check=FALSE).

Managing parameter transformations that may be used internally

If any paramNodes (parameters) or randomEffectsNodes (random effects / latent states) have constraints on the range of valid values (because of the distribution they follow), they will be used on a transformed scale determined by parameterTransform. This means the Laplace approximation itself will be done on the transformed scale for random effects and finding the MLE will be done on the transformed scale for parameters. For parameters, prior distributions are not included in calculations, but they are used to determine valid parameter ranges and hence to set up any transformations. For example, if sigma is a standard deviation, you can declare it with a prior such as sigma ~ dhalfflat() to indicate that it must be greater than 0.

For default determination of when transformations are needed, all parameters must have a prior distribution simply to indicate the range of valid values. For a param p that has no constraint, a simple choice is p ~ dflat().

Understanding inner and outer optimizations

Note that there are two numerical optimizations when finding maximum likelihood estimates with a Laplace or (1D) AGHQ algorithm: (1) maximizing the joint log-likelihood of random effects and data given a parameter value to construct the approximation to the marginal log-likelihood at the given parameter value; (2) maximizing the approximation to the marginal log-likelihood over the parameters. In what follows, the prefix 'inner' refers to optimization (1) and 'outer' refers to optimization (2). Currently both optimizations default to using method "nlminb". However, one can use other optimizers or simply run optimization (2) manually from R; see the example below. In some problems, choice of inner and/or outer optimizer can make a big difference for obtaining accurate results, especially for standard errors. Hence it is worth experimenting if one is in doubt.

control list arguments

The control list allows additional settings to be made using named elements of the list. Most (but not all) of these can be updated later using the 'updateSettings' method. Supported elements include:

- split. If TRUE (default), randomEffectsNodes will be split into conditionally independent sets if possible. This facilitates more efficient Laplace or AGHQ approximation because each conditionally independent set can be marginalized independently. If FALSE, randomEffectsNodes will be handled as one multivariate block, with one multivariate approximation. If split is a numeric vector, randomEffectsNodes will be split by calling split(randomEffectsNodes, control$split). The last option allows arbitrary control over how randomEffectsNodes are blocked.
- check. If TRUE (default), a warning is issued if paramNodes, randomEffectsNodes and/or calcNodes are provided but seem to have missing or unnecessary elements based on some default inspections of the model. If unnecessary warnings are emitted, simply set check=FALSE.

- innerOptimControl. An 'optimControlNimbleList' list of control parameters (an R list is sufficient for uncompiled operation) for the inner optimization of Laplace approximation using nimOptim. See 'Details' of nimOptim for further information. Default is 'nimOptimDefaultControl()'.
- innerOptimMethod. Optimization method to be used in nimOptim for the inner optimization. See 'Details' of nimOptim. Currently nimOptim in NIMBLE supports: "Nelder-Mead"", "BFGS", "CG", "L-BFGS-B", "nlminb", "bobyqa", and user-provided optimizers. By default, method "nlminb" is used for both univariate and multivariate cases. For "nlminb", "bobyqa", or user-provided optimizers, only a subset of elements of the innerOptimControlList are supported. (Note that control over the outer optimization method is available as an argument to 'findMLE'). Choice of optimizers can be important and so can be worth exploring.
- innerOptimStart. Method for determining starting values for the inner optimization. Options are:
  - "last.best" (default): use optimized random effects values corresponding to the best outer optimization (i.e. the largest marginal log likelihood value) so far for each conditionally independent part of the approximation;
  - "last": use the result of the last inner optimization;
  - "zero": use all zeros;
  - "constant": always use the same values, determined by innerOptimStartValues;
  - "random": randomly draw new starting values from the model (i.e., from the prior);
  - "model": use values for random effects stored in the model, which are determined from the first call.

  Note that "model" and "zero" are shorthand for "constant" with particular choices of innerOptimStartValues. Note that "last" and "last.best" require a choice for the very first values, which will come from innerOptimStartValues. The default is innerOptimStart="zero" and may change in the future.
- innerOptimStartValues. Values for some of innerOptimStart approaches. If a scalar is provided, that value is used for all elements of random effects for each conditionally independent set. If a vector is provided, it must be the length of *all* random effects. If these are named (by node names), the names will be used to split them correctly among each conditionally independent set of random effects. If they are not named, it is not always obvious what the order should be because it may depend on the conditionally independent sets of random effects. It should match the order of names returned as part of 'summaryLaplace'.
- innerOptimWarning. If FALSE (default), do not emit warnings from the inner optimization. Optimization methods may sometimes emit a warning such as for bad parameter values encountered during the optimization search. Often, a method can recover and still find the optimum. In the approximations here, sometimes the inner optimization search can fail entirely, yet the outer optimization see this as one failed parameter value and can recover. Hence, it is often desirable to silence warnings from the inner optimizer, and this is done by default. Set innerOptimWarning=TRUE to see all warnings.
- useInnerCache. If TRUE (default), use caching system for efficiency of inner optimizations. The caching system records one set of previous parameters and uses the corresponding results if those parameters are used again (e.g., in a gradient call). This should generally not be modified.
- outerOptimMethod. Optimization method to be used in nimOptim for the outer optimization. See 'Details' of nimOptim. Currently nimOptim in NIMBLE supports: "Nelder-Mead"",

"BFGS", "CG", "L-BFGS-B", "nlminb", "bobyqa", and user-provided optimizers. By default, method "nlminb" is used for both univariate and multivariate cases, although some problems may benefit from other choices. For "nlminb", "bobyqa", or user-provided optimizers, only a subset of elements of the innerOptimControlList are supported. (Note that control over the outer optimization method is available as an argument to 'findMLE'). Choice of optimizers can be important and so can be worth exploring.

- outerOptimControl. An 'optimControlNimbleList' of control parameters (an R list is sufficient for uncompiled operation) for maximizing the Laplace log-likelihood using nimOptim. See 'Details' of nimOptim for further information.
- quadTransform (relevant only nQuad>1). For multivariate AGHQ, a grid must be constructed based on the Hessian at the inner mode. Options include "cholesky" (default) and "spectral" (i.e., eigenvectors and eigenvalues) for the corresponding matrix decompositions on which the grid can be based.
- ADuseNormality. For random effects nodes that are distributed univariate or multivariate normal (Gaussian), the derivatives with respect to those nodes are known in closed form. By default, the approximation for multivariate Laplace/AGHQ will make use of this closed form. Set to FALSE to have the derivatives determined entirely using AD. Doing so may use more memory but may be faster for low-dimensional cases.
- outerOptimUseAD. The optimization of the (hyper)parameters (the "outer" optimization can provide an AD-based gradient to the chosen outer optimization function or can omit this, causing any derivative-based optimization method to use finite differences. Turning this off allows one to avoid any complexity associated with use of AD applied to the inner Laplace/AGHQ approximation. This option is not active when when ADuseNormality = TRUE, as is the case by default, because outer optimization does not (and cannot because of limitations in NIMBLE's AD implementation) use the AD-based gradient in that situation.

end itemize

Available methods

The object returned by buildLaplace or buildAGHQ is a nimbleFunction object with numerous methods (functions). Here these are described in three tiers of user relevance.

Most useful methods

The most relevant methods to a user are:

- calcLogLik(p, trans=FALSE). Calculate the approximation to the marginal log-likelihood function at parameter value p, which (if trans is FALSE) should match the order of paramNodes. For any non-scalar nodes in paramNodes, the order within the node is column-major. The order of names can be obtained from method getNodeNamesVec(TRUE). Return value is the scalar (approximate, marginal) log likelihood.

  If trans is TRUE, then p is the vector of parameters on the transformed scale, if any, described above. In this case, the parameters on the original scale (as the model was written) will be determined by calling the method pInverseTransform(p). Note that the length of the parameter vector on the transformed scale might not be the same as on the original scale (because some constraints of non-scalar parameters result in fewer free transformed parameters than original parameters).

- `calcLaplace(p, trans)`. This is the same as `calcLogLik` but requires that the approximation be Laplace (i.e `nQuad` is 1), and results in an error otherwise.
- `findMLE(pStart, hessian)`. Find the maximum likelihood estimates of parameters using the approximated marginal likelihood. This can be used if `nQuad` is 1 (Laplace case) or if `nQuad>1` and all marginalizations involve only univariate random effects. Arguments are `pStart`: initial parameter values (defaults to parameter values currently in the model); and `hessian`: whether to calculate and return the Hessian matrix (defaults to `TRUE`, which is required for subsequent use of `summary` method). Second derivatives in the Hessian are determined by finite differences of the gradients obtained by automatic differentiation (AD). Return value is a nimbleList of type `optimResultNimbleList`, similar to what is returned by R's optim. See `help(nimOptim)`. Note that parameters (`par`) are returned for the natural parameters, i.e. how they are defined in the model. But the `hessian`, if requested, is computed for the parameters as transformed for optimization if necessary. Hence one must be careful interpreting 'hessian' if any parameters have constraints, and the safest next step is to use the `summary` method or `summaryLaplace` function.
- `findMAP(pStart, hessian)`. Find the maximum a posteriori estimates (posterior mode) of parameters using the approximated marginal likelihood (and parameter priors). See information above regarding `findMLE` for details.
- `optimize(pStart, includePrior, includeJacobian, hessian, parscale, keepOneFixed)`. Optimize the approximated marginal likelihood with flexibility to specify whether to include the parameter prior. `findMLE` and `findMAP` are simple wrappers around this method. Note that one can fit a regularized model that uses the prior as a penalty but excludes the Jacobian of the transformation.
- `summary(MLEoutput, originalScale, randomEffectsStdError, jointCovariance)`. Summarize the maximum likelihood estimation results, given object `MLEoutput` that was returned by `findMLE`. The summary can include a covariance matrix for the parameters, the random effects, or both), and these can be returned on the original parameter scale or on the (potentially) transformed scale(s) used in estimation. It is often preferred instead to call function (not method) 'summaryLaplace' because this will attach parameter and random effects names (i.e., node names) to the results.

  In more detail, `summary` accepts the following optional arguments:

  - `originalScale`. Logical. If TRUE, the function returns results on the original scale(s) of parameters and random effects; otherwise, it returns results on the transformed scale(s). If there are no constraints, the two scales are identical. Defaults to TRUE.
  - `randomEffectsStdError`. Logical. If TRUE, standard errors of random effects will be calculated. Defaults to TRUE.
  - `jointCovariance`. Logical. If TRUE, the joint variance-covariance matrix of the parameters and the random effects will be returned. If FALSE, the variance-covariance matrix of the parameters will be returned. Defaults to FALSE.

  The object returned by `summary` is an `AGHQuad_summary` nimbleList with elements:

  - `params`. A nimbleList that contains estimates and standard errors of parameters (on the original or transformed scale, as chosen by `originalScale`).
  - `randomEffects`. A nimbleList that contains estimates of random effects and, if requested (`randomEffectsStdError=TRUE`) their standard errors, on original or transformed scale. Standard errors are calculated following the generalized delta method of Kass and Steffey (1989).

  - vcov. If requested (i.e. jointCovariance=TRUE), the joint variance-covariance matrix of the parameters and random effects, on original or transformed scale. If jointCovariance=FALSE, the covariance matrix of the parameters, on original or transformed scale.
  - scale. "original" or "transformed", the scale on which results were requested.

Methods for more advanced uses

Additional methods to access or control more details of the Laplace/AGHQ approximation include:

- updateSettings. This provides a single function through which many of the settings described above (mostly for the control list) can be later changed. Options that can be changed include: innerOptimMethod, innerOptimStart, innerOptimStartValues, useInnerCache, nQuad, quadTransform, innerOptimControl, and outerOptimControl. For innerOptimStart, method "zero" cannot be specified but can be achieved by choosing method "constant" with innerOptimStartValues=0. Only provided options will be modified. The exceptions are innerOptimControl, outerOptimControl, which are replaced only when replace_innerOptimControl=TRUE or replace_outerOptimControl=TRUE, respectively.
- getNodeNamesVec(returnParams). Return a vector (>1) of names of parameters/random effects nodes, according to returnParams = TRUE/FALSE. Use this if there is more than one node.
- getNodeNameSingle(returnParams). Return the name of a single parameter/random effect node, according to returnParams = TRUE/FALSE. Use this if there is only one node.
- checkInnerConvergence(message). Checks whether all internal optimizers converged. Returns a zero if everything converged and one otherwise. If message = TRUE, it will print more details about convergence for each conditionally independent set.
- gr_logLik(p, trans). Gradient of the (approximated) marginal log-likelihood at parameter value p. Argument trans is similar to that in calcLaplace. If there are multiple parameters, the vector p is given in the order of parameter names returned by getNodeNamesVec(returnParams=TRUE).
- gr_Laplace(p, trans). This is the same as gr_logLik.
- otherLogLik(p). Calculate the calcNodesOther nodes, which returns the log-likelihood of the parts of the model that are not included in the Laplace or AGHQ approximation.
- gr_otherLogLik(p). Gradient (vector of derivatives with respect to each parameter) of otherLogLik(p). Results should match gr_otherLogLik_internal(p) but may be more efficient after the first call.

Internal or development methods

Some methods are included for calculating the (approximate) marginal log posterior density by including the prior distribution of the parameters. This is useful for finding the maximum a posteriori probability (MAP) estimate. Currently these are provided for point calculations without estimation methods.

- calcPrior_p(p). Log density of prior distribution.
- calcPrior_pTransformed(pTransform). Log density of prior distribution on transformed scale, includes the Jacobian.
- calcPostLogDens(p). Marginal log posterior density in terms of the parameter p.

- `calcPostLogDens_pTransformed (pTransform)`. Marginal log posterior density in terms of the transformed parameter, which includes the Jacobian transformation.
- `gr_postLogDens_pTransformed(pTransform)`. Graident of marginal log posterior density on the transformed scale. Other available options that are used in the derivative for more flexible include `logDetJacobian(pTransform)` and `gr_logDeJacobian(pTransform)`, as well as `gr_prior(p)`.

Finally, methods that are primarily for internal use by other methods include:

- `gr_logLik_pTransformed`. Gradient of the Laplace approximation (`calcLogLik_pTransformed(pTransform)`) at transformed (unconstrained) parameter value `pTransform`.
- `pInverseTransform(pTransform)`. Back-transform the transformed parameter value `pTransform` to original scale.
- `derivs_pInverseTransform(pTransform, order)`. Derivatives of the back-transformation (i.e. inverse of parameter transformation) with respect to transformed parameters at `pTransform`. Derivative order is given by `order` (any of 0, 1, and/or 2).
- `reInverseTransform(reTrans)`. Back-transform the transformed random effects value `reTrans` to original scale.
- `derivs_reInverseTransform(reTrans, order)`. Derivatives of the back-transformation (i.e. inverse of random effects transformation) with respect to transformed random effects at `reTrans`. Derivative order is given by `order` (any of 0, 1, and/or 2).
- `optimRandomEffects(pTransform)`. Calculate the optimized random effects given transformed parameter value `pTransform`. The optimized random effects are the mode of the conditional distribution of random effects given data at parameters `pTransform`, i.e. the calculation of `calcNodes`.
- `inverse_negHess(p, reTransform)`. Calculate the inverse of the negative Hessian matrix of the joint (parameters and random effects) log-likelihood with respect to transformed random effects, evaluated at parameter value `p` and transformed random effects `reTransform`.
- `hess_logLik_wrt_p_wrt_re(p, reTransform)`. Calculate the Hessian matrix of the joint log-likelihood with respect to parameters and transformed random effects, evaluated at parameter value `p` and transformed random effects `reTransform`.
- `one_time_fixes()`. Users never need to run this. Is is called when necessary internally to fix dimensionality issues if there is only one parameter in the model.
- `calcLogLik_pTransformed(pTransform)`. Laplace approximation at transformed (unconstrained) parameter value `pTransform`. To make maximizing the Laplace likelihood unconstrained, an automated transformation via `parameterTransform` is performed on any parameters with constraints indicated by their priors (even though the prior probabilities are not used).
- `gr_otherLogLik_internal(p)`. Gradient (vector of derivatives with respect to each parameter) of `otherLogLik(p)`. This is obtained using automatic differentiation (AD) with single-taping. First call will always be slower than later calls.
- `cache_outer_logLik(logLikVal)`. Save the marginal log likelihood value to the inner Laplace mariginlization functions to track the outer maximum internally.
- `reset_outer_inner_logLik()`. Reset the internal saved maximum marginal log likelihood.

- `get_inner_cholesky(atOuterMode = integer(0, default = 0))`. Returns the cholesky of the negative Hessian with respect to the random effects. If `atOuterMode = 1` then returns the value at the overall best marginal likelihood value, otherwise `atOuterMode = 0` returns the last.
- `get_inner_mode(atOuterMode = integer(0, default = 0))`. Returns the mode of the random effects for either the last call to the innner quadrature functions (`atOuterMode = 0`), or the last best value for the marginal log likelihood, `atOuterMode = 1`.

Author(s)

Wei Zhang, Perry de Valpine, Paul van Dam-Bates

References

Kass, R. and Steffey, D. (1989). Approximate Bayesian inference in conditionally independent hierarchical models (parametric empirical Bayes models). *Journal of the American Statistical Association*, 84(407), 717-726.

Liu, Q. and Pierce, D. A. (1994). A Note on Gauss-Hermite Quadrature. *Biometrika*, 81(3) 624-629.

Jackel, P. (2005). A note on multivariate Gauss-Hermite quadrature. London: *ABN-Amro. Re.*

Skaug, H. and Fournier, D. (2006). Automatic approximation of the marginal likelihood in non-Gaussian hierarchical models. *Computational Statistics & Data Analysis*, 56, 699-709.

Examples

```
pumpCode <- nimbleCode({
  for (i in 1:N){
    theta[i] ~ dgamma(alpha, beta)
    lambda[i] <- theta[i] * t[i]
    x[i] ~ dpois(lambda[i])
  }
  alpha ~ dexp(1.0)
  beta ~ dgamma(0.1, 1.0)
})
pumpConsts <- list(N = 10, t = c(94.3, 15.7, 62.9, 126, 5.24, 31.4, 1.05, 1.05, 2.1, 10.5))
pumpData <- list(x = c(5, 1, 5, 14, 3, 19, 1, 1, 4, 22))
pumpInits <- list(alpha = 0.1, beta = 0.1, theta = rep(0.1, pumpConsts$N))
pump <- nimbleModel(code = pumpCode, name = "pump", constants = pumpConsts,
                    data = pumpData, inits = pumpInits, buildDerivs = TRUE)

# Build Laplace approximation
pumpLaplace <- buildLaplace(pump)


# Compile the model
Cpump <- compileNimble(pump)
CpumpLaplace <- compileNimble(pumpLaplace, project = pump)
# Calculate MLEs of parameters
MLEres <- CpumpLaplace$findMLE()
# Calculate estimates and standard errors for parameters and random effects on original scale
```

```
allres <- CpumpLaplace$summary(MLEres, randomEffectsStdError = TRUE)

# Change the settings and also illustrate runLaplace
innerControl <- nimOptimDefaultControl()
innerControl$maxit <- 1000
CpumpLaplace$updateSettings(innerOptimControl = innerControl,
                            replace_innerOptimControl = TRUE)
newres <- runLaplace(CpumpLaplace)

# Illustrate use of the component log likelihood and gradient functions to
# run an optimizer manually from R.
# Use nlminb to find MLEs
MLEres.manual <- nlminb(c(0.1, 0.1),
                        function(x) -CpumpLaplace$calcLogLik(x),
                        function(x) -CpumpLaplace$gr_Laplace(x))
```

`buildNestedApprox` *Build Nested Bayesian Approximation Using Quadrature-based Methods*

Description

Build a nested approximation for a given NIMBLE model, providing parameter estimation and sampling for latent nodes. The approximation uses an inner Laplace (or AGHQ) approximation to approximately marginalize over latent nodes and an outer quadrature grid on the parameters.

Usage

```
buildNestedApprox(
  model,
  paramNodes,
  latentNodes,
  calcNodes,
  calcNodesOther,
  control = list()
)
```

Arguments

`model`	a NIMBLE model object created by calling `nimbleModel`. The model must have automatic derivatives (AD) turned on, e.g., by using `buildDerivs=TRUE` in `nimbleModel`.
`paramNodes`	optional character vector of (hyper)parameter nodes in the model. If missing, this will be the stochastic non-data nodes with no parent stochastic nodes.
`latentNodes`	optional character vector of latent nodes (e.g., random and fixed effects) in the model. If missing this will be the stochastic non-data nodes that are not determined to be parameter nodes.

`calcNodes` optional character vector of names of nodes for calculating the integrand for Laplace/AGHQ approximation over the latent nodes; defaults are provided by `setupMargNodes`. Note that users will generally not need to provide this. There may be deterministic nodes between `paramNodes` and `calcNodes`. These will be included in calculations automatically and thus do not need to be included in `calcNodes` (but there is no problem if they are).

`calcNodesOther` optional character vector of names of nodes for calculating terms in the log-likelihood that do not depend on any `randomEffectsNodes`, and thus are not part of the marginalization, but should be included for purposes of finding the approximation. Note that users will generally not need to provide this. This defaults to stochastic nodes that depend on `paramNodes` but are not part of and do not depend on `latentNodes`. There may be deterministic nodes between `paramNodes` and `calcNodesOther`. These will be included in calculations automatically and thus do not need to be included in `calcNodesOther` (but there is no problem if they are).

`control` a named list for providing additional settings for the approximation, including settings for the inner Laplace/AGHQ approximation. See `control` section below.

Details

This function builds a nested Bayesian approximation for the provided model. NIMBLE's nested approximation provides approximate posterior inference using methodology similar to the well-known INLA approach (Rue et al. 2009), implemented in the R-INLA package and to the related methods for extended Gaussian latent models (EGLMs) of Stringer et al. (2023), implemented in the 'aghq' R package. For more details on the nested approximation algorithms, see the NIMBLE User Manual.

Unlike Laplace/AGHQ approximation, the nested approximation is Bayesian, requiring prior distributions for all parameters and providing functionality to estimate the marginal posterior distributions of the individual parameters and to sample from the marginal joint posterior distribution of the latent nodes. Similarly to Laplace, the nested approximation uses Laplace/AGHQ to approximately marginalize over the latent nodes. However, instead of then maximizing the approximate marginalized posterior density, the nested approximation uses a quadrature grid on the parameters to perform approximate Bayesian inference on the parameters and latent nodes.

The recommended way to use the nested approximation once it is built is to call `runNestedApprox` on the returned object, and then to call additional approximation functions on the output of `runNestedApprox` as needed. For details see `runNestedApprox`. However, for more granular control, one can also call the internal methods of the nested approximation, discussed briefly below.

In general, the theory that underpins these approximations assumes that the latent nodes (fixed and random effects) are Gaussian. `buildNestedApprox` makes no such assumptions, allowing the user to extend these approximations to any imaginable set of models in NIMBLE. However, the accuracy of the approximation is then not supported theoretically, and it is up to the user to determine whether or not the posterior approximation is valid.

Computational considerations

The computational cost of the nested approximation can vary depending on what nodes are considered as parameter nodes and what nodes as latent nodes, as well as by the number of quadrature

points (for both the latent and parameter nodes) and type of grid used for the parameter nodes. Some details are provided in the User Manual.

`buildNestedApprox` will by default (unless changed manually by specifying sets of nodes) determine from the model which latent nodes can be integrated over (marginalized) independently. For example, in a GLMM with a grouping factor and an independent random effect intercept for each group (and no fixed effects), the random effects can be marginalized as a set of univariate approximations rather than one multivariate approximation. On the other hand, correlated or nested random effects would require multivariate marginalization, as would the presence of fixed effects (since they affect all the observations). Independent marginalizations result in lower-dimensional calculations (essentially exploiting sparsity in the covariance structure of the latent nodes) and therefore improve computational efficiency. Note that at this time, the nested approximation cannot otherwise take advantage of sparsity in the covariance structure of the latent nodes.

How input nodes are processed

In many cases, the selection of parameter and latent nodes will be handled automatically in a reasonable fashion. However, random effects can be written in models in multiple equivalent ways, and customized use cases may call for integrating over chosen parts of a model. Hence, one can take full charge of how different parts of the model will be used, specifying explicitly the `paramNodes` and `latentNodes`. The User Manual provides more details on situations in which one may want to specify these nodes explicitly.

Any of the input node vectors, when provided, will be processed using `nodes <- model$expandNodeNames(nodes)`, where `nodes` may be `paramNodes`, `latentNodes`, and so on. This step allows any of the inputs to include node-name-like syntax that might contain multiple nodes. For example, `paramNodes = 'beta[1:10]'` can be provided if there are actually 10 scalar parameters, 'beta[1]' through 'beta[10]'. The actual node names in the model will be determined by the `expandNodeNames` step.

In many (but not all) cases, one only needs to provide a NIMBLE model object and then the function will construct reasonable defaults necessary for Laplace approximation to marginalize over all continuous latent nodes (both random and fixed effects) in a model.

`buildNestedApprox` uses `setupMargNodes` (in a multi-step process) to try to give sensible defaults from any combination of `paramNodes`, `latentNodes`, `calcNodes`, and `calcNodesOther` that are provided.

`setupMargNodes` also determines which integration dimensions are conditionally independent, i.e., which can be done separately from each other. For example, when possible, 10 univariate random effects will be split into 10 univariate integration problems rather than one 10-dimensional integration problem. Note that models that include fixed effects as latent nodes often prevent this splitting into conditionally independent sets.

The defaults make general assumptions such as that `latentNodes` have `paramNodes` as parents or (for fixed effects) are also components of a linear predictor expression. However, the steps for determining defaults are not simple, and it is possible that they will be refined in the future. It is also possible that they simply don't give what you want for a particular model. One example where they will not give desired results can occur when random effects have no prior parameters, such as 'N(0,1)' nodes that will be multiplied by a scale factor (e.g., 'sigma") and added to other explanatory terms in a model. Such nodes look like top-level parameters in terms of model structure, so you must provide a `latentNodes` argument to indicate which they are.

`control` list arguments

The `control` list allows additional settings to be made using named elements of the list. Any elements in addition to those below are passed along as the `control` list for the inner Laplace/AGHQ approximation (see `buildAGHQ`). Below, 'd' refers to the number of parameter nodes. Supported elements include:

- `nQuadOuter`. Number of outer quadrature points in each dimension (for parameter nodes). Default is 3 for d > 1, 5 for d = 1. Not used with CCD grid.
- `nQuadInner`. Number of inner quadrature points in each dimension (for latent nodes). Default is 1, corresponding to Laplace approximation.
- `paramGridRule`. Quadrature rule for the parameter grid. Defaults to `"CCD"` for d > 2 and to `"AGHQ"` otherwise. Can also be `"AGHQSPARSE"` or (for user-defined grids) a user-defined nimbleFunction generator (created by calling 'nimbleFunction') with an appropriate 'buildGrid' method that has arguments `levels` and `d` and that returns a matrix.
- `paramGridRule_userType`. If `paramGridRule` is a user-defined rule, this optional element can be used to indicate that the provided rule constructs a univariate rule rather than directly constructing a multivariate rule and that a multivariate rule should be constructed from the univariate rule as either a product rule (by specifying "PRODUCT") or a sparse rule (by specifying "SPARSE").
- `innerOptimWarning`. Whether to show inner optimization warnings. Default is `FALSE`.
- `marginalGridRule`. Rule for the grid for parameter marginalization. Default is `"AGHQ"`. Can also be `"AGHQSPARSE"`. At present, user-defined grids are not allowed.
- `marginalGridPrune`. Pruning parameter for marginal grid. Default is 0, corresponding to no pruning.
- `quadTransform`. Quadrature transformation method. Default is `"spectral"`, with `"cholesky"` as the other option.

Parameter transformations used internally

If any `paramNodes` (parameters) or `latentNodes` have constraints on the range of valid values (because of the distribution they follow), they will be used on a transformed scale determined by `parameterTransform`. This means that internally the Laplace/AGHQ approximation itself will be done on the transformed scale for latent nodes and that the grid-based computation on the parameters will be done on the transformed scale.

Available methods for advanced/development use

Additional methods to access or control the Laplace/AGHQ approximation directly (as an alternative to the recommended use of `runNestedApprox`) include the following, described only briefly:

- `findMode`: Find the posterior mode for hyperparameters.
- `buildParamGrid`: Build the parameter grid using specified quadrature rule and settings.
- `setParamGridRule`: Set the quadrature rule for the parameter grid (AGHQ, CCD, USER, AGHQSPARSE).
- `calcEigen`: Calculate eigendecomposition of the negative Hessian for spectral transformations.

- `calcCholesky`: Calculate Cholesky decomposition of the negative Hessian for Cholesky transformations.
- `setTransformations`: Set transformation method between spectral and Cholesky approaches.
- `z_to_paramTrans`: Transform from standard (z) scale to parameter transform scale.
- `paramTrans_to_z`: Transform from parameter transform scale to standard (z) scale.
- `calcSkewedSD`: Calculate skewed standard deviations for asymmetric Gaussian approximations.
- `getSkewedStdDev`: Retrieve the calculated skewed standard deviations.
- `calcMarginalLogLikApprox`: Calculate INLA-like approximation of marginal log-likelihood using the approximate Gaussian.
- `calcParamGrid`: Calculate inner approximation at parameter grid values and cache results. Required only for latent node simulation and quadrature-based marginal log-likelihood.
- `calcMarginalLogLikQuad`: Calculate quadrature-based marginal log-likelihood.
- `calcMarginalParamQuad`: Calculate univariate marginal parameter distributions using selected quadrature rule.
- `calcMarginalParamIntegFree`: Calculate univariate marginal parameter distributions using INLA-like integration-free method based on approximate Gaussian approximations.
- `simulateLatents`: Simulate from the posterior distribution of (transformed) latent nodes.
- `simulateParams`: Simulate from the marginal posterior of (transformed) parameters using skewed normal.
- `getParamGrid`: Retrieve the parameter grid points on the transformed scale.

Author(s)

Paul van Dam-Bates, Christopher Paciorek, Perry de Valpine

References

Rue, H., Martino, S., and Chopin, N. (2009). Approximate Bayesian inference for latent Gaussian models by using integrated nested Laplace approximations. Journal of the Royal Statistical Society: Series B (Statistical Methodology), 71(2), 319-392.

Stringer, A., Brown, P., and Stafford, J. (2023). Fast, scalable approximations to posterior distributions in extended latent Gaussian models. Journal of Computational and Graphical Statistics, 32(1), 84-98.

Examples

```
data(penicillin, package="faraway")
code <- nimbleCode({
    for(i in 1:n) {
        mu[i] <- inprod(b[1:nTreat], x[i, 1:nTreat]) + re[blend[i]]
        y[i] ~ dnorm(mu[i], tau = Tau)
    }
    # Priors corresponding simply to INLA defaults and not being recommended.
    # Instead consider uniform or half-t distributions on the standard deviation scale
    # or penalized complexity priors.
```

```
    Tau ~ dgamma(1, 5e-05)
    Tau_re ~ dgamma(1, 5e-05)
    for( i in 1:nTreat ){ b[i] ~ dnorm(0, tau = 0.001) }
    for( i in 1:nBlend ){ re[i] ~ dnorm(0, tau = Tau_re) }
})
X <- model.matrix(~treat, data = penicillin)
data = list(y = penicillin$yield)
constants = list(nTreat = 4, nBlend = 5, n = nrow(penicillin),
                 x = X, blend = as.numeric(penicillin$blend))
inits <- list(Tau = 1, Tau_re = 1, b = c(mean(data$y), rep(0,3)), re = rep(0,5))

model <- nimbleModel(code, data = data, constants = constants,
                 inits = inits, buildDerivs = TRUE)
approx <- buildNestedApprox(model = model)
```

`calcMarginalLogLikImproved` *Calculate improved marginal log-likelihood using grid-based quadrature*

Description

Uses quadrature (by default AGHQ) to get an improved estimate of the marginal log-likelihood.

Usage

```
calcMarginalLogLikImproved(summary)
```

Arguments

`summary`	an approxSummary object, returned by `runNestedApprox`.

Details

Users will not generally need to call this function directly, as it is called automatically when sampling from the posterior of the latent nodes, since its computation comes for free in that case.

Warning: the marginal log-likelihood is invalid for improper priors and may not be useful for non-informative priors, because it averages the log-likelihood (approximately marginalized with respect to the latent nodes) over the prior distribution, thereby including log-likelihood values corresponding to parameter values that are inconsistent with the data.

Value

The improved marginal log-likelihood.

configureQuadGrid *Configure Quadrature Grids*

Description

Takes requested quadrature rules, builds the associated quadrature grids, and caches them. Updates all the features of the grids on call and returns nodes and weights.

Usage

```
configureQuadGrid(d = 1, levels = 3, quadRule = "AGHQ", control = list())
```

Arguments

d	Number of dimensions.
levels	Number of quadrature points to generate in each dimension (or the level of accuracy of a sparse grid).
quadRule	Default quadRule to be used. Options are "AGHQ" or "CCD". May also be a user supplied quadrature rule as a nimbleFunction.
control	list to control how the quadrature rules are built. See details for options.

Details

Different options for building quadrature rules can be specified by 'control' list. These include

- quadRules which includes all the different rules that are being requested.
- constructionRule How each quadrature rule should be combined into multiple dimension. Currently possible to choose "product" which repeats creates a grid of repeated nodes in each dimension. Alternatively, can use "sparse" to apply standard sparse construction.
- CCD_f0 multiplier for the CCD grid for how much past the radius sqrt(d) to extend the nodes. Unless an advanced user, keep at default of 1.1.
- prune the proportion of quadrature points (when generated by the product rule) to keep based on the weights for integration over a multivariate normal.
- userConstruction choose method to construct multivariate grid. If "MULTI" then user provided a multivariate construction in the provided function. If "PRODUCT" then a product rule construction is used to generate quadrature points in each dimension. If "SPARSE", then a Smolyak rule is applied.

Quadrature grids are generally based on adaptive Gauss-Hermite (GH) quadrature which is expanded via a product or sparse rule into multiple dimensions. Sparse grids are built following the Smolyak rule (Heiss and Winschel, 2008) and demonstrated in the package **mvQuad** (Weiser, 2023). Pruning is also implemented as described in Jäckel (2005), where weights are adjusted by the value of a standard multivariate normal at that node, and nodes are removed until some threshold is met.

The available methods that can called by this function once it is setup are:

- `buildGrid` method for creating the quadrature grid. Inputs are `method` includes ("AGHQ", "CCD", "USER") to choose the active quadrature rule. `nQuad` number of quadrature points (nQuad) per dimension, `prune` proportion of points in the product construction to use, and `constructionRule` includes ("product", "sparse"). Default behaviour for all input is to use values that were last requested.
- `setDim` Allows the user to change the dimension of the quadrature grids which will reset all grids in use.
- `nodes`, `weights`, `gridSize`, and `modeIndex` give access to the user for the details of the quadrature grid in use. This is either based on the last call to `buildGrid`, or by choosing a different grid with `setMethod`. `nodes` and `weights` return all nodes and weights if no values are passed, or if an index is passed, the node and weight associated with that index. Passing -1 indicates that the mode should be returned which in this case is all zeros.

Author(s)

Paul van Dam-Bates

References

Heiss, F. and Winschel V. (2008). Likelihood approximation by numerical integration on sparse grids. Journal of Econometrics 144 (1), 62–80.

Weiser, C. (2023). _mvQuad: Methods for Multivariate Quadrature._. (R package version 1.0-8), <https://CRAN.R-project.org/package=mvQuad>.

Jäckel, P. (2005). A note on multivariate gauss-hermite quadrature. London: ABN-Amro. Re.

Examples

```
library(mvQuad)
RmvQuad <- function(levels, d) {
  out <- mvQuad::createNIGrid(dim=d, type = "GHe", level=levels, ndConstruction = "sparse")
   cbind(out$weights, out$nodes)
}
nimMVQuad <- nimbleRcall(function(levels = integer(), d = integer()){},
                         Rfun = "RmvQuad", returnType = double(2))
myQuadRule <- nimbleFunction(
     contains = QUAD_RULE_BASE,
     name = "quadRule_USER",
     setup = function() {},
     run = function() {},
     methods = list(
       buildGrid = function(levels = integer(0, default = 0), d = integer(0, default = 1)) {
              output <- nimMVQuad(levels, d)
              returnType(double(2))
              return(output)
          }
     )
 )

quadGrid_user <- configureQuadGrid(d=2, levels=3, quadRule = myQuadRule,
                   control = list(quadRules = c("AGHQ", "CCD", "AGHQSPARSE"),
```

```
                                    userConstruction = "MULTI"))
```

`dmarginal` *Evaluate the marginal posterior density for a parameter.*

Description

Density evaluation for univariate parameter marginals.

Usage

```
dmarginal(summary, node, x, log = FALSE)
```

Arguments

`summary`	an approxSummary object, returned by `runNestedApprox`.
`node`	parameter node of interest. Specified as character (when using original scale) or integer (when using transformed scale), where the scale was specified in `runNestedApprox`.
`x`	numeric vector of values at which to evaluate the density.
`log`	logical; if `TRUE`, return log-density. Default is `FALSE`.

Details

Uses a spline approximation to the log-density of the marginal posterior distribution, based on a cached approximation of the marginal density on a fine grid.

Value

Numeric vector of (log-)density values.

`drop_algorithm` *Drop Algorithm to generate permutations of dimension d with a fixed sum.*

Description

Generates a matrix of all permutations of 'd' cols that sum to 'order' with no zeros.

Usage

```
drop_algorithm(d, order)
```

Arguments

d	Number of columns (dimensions)
order	Row sum to permute over.

Details

This function generates permutation matrix in order to be used for sparse grid quadrature building. It is adapted from the library ‘mvQuad‘ (Weiser, 2023).

Author(s)

Paul van Dam-Bates

References

Weiser, C. (2023). _mvQuad: Methods for Multivariate Quadrature._. (R package version 1.0-8), <https://CRAN.R-project.org/package=mvQuad>.

---

emarginal *Compute the expectation of a function of a parameter under the marginal posterior distribution*

Description

Posterior expectations for univariate parameter marginals.

Usage

```
emarginal(summary, node, functional, ...)
```

Arguments

summary	an approxSummary object, returned by runNestedApprox.
node	parameter node of interest. Specified as character (when using original scale) or integer (when using transformed scale), where the scale was specified in runNestedApprox.
functional	function to compute the expectation of.
...	Additional arguments passed to the function.

Details

Estimate the expectation of a function of a parameter using univariate numerical integration based on a cached approximation of the marginal density on a fine grid.

See runNestedApprox for example usage.

Value

Numeric value of the expectation.

improveParamMarginals *Improve univariate parameter marginals using grid-based quadrature*

Description

Uses d-1 dimensional quadrature (by default AGHQ) to get improved univariate marginal estimates for parameters. Users can select to apply to specific nodes of interest to limit computation.

Usage

```
improveParamMarginals(
  summary,
  nodes,
  nMarginalGrid = 5,
  nQuad,
  quadRule = "NULL",
  prune = -1,
  transform = "spectral"
)
```

Arguments

summary	an approxSummary object, returned by runNestedApprox.
nodes	parameter nodes to improve inference for. Specified as character (when using original scale) or integer (when using transformed scale), where the scale was specified in runNestedApprox.
nMarginalGrid	number of grid points for marginal calculations. Default is 5.
nQuad	number of AGHQ quadrature points. Default is 5 if d=2 and 3 otherwise.
quadRule	quadrature rule to use for the parameter grid. Can be any of "AGHQ", "CCD", "AGHQSPARSE" or "USER", the latter for user-defined grids, but standard use will be of "AGHQ".
prune	pruning parameter for removing AGHQ points at low-density points.
transform	grid transformation method for internal AGHQ. Default is "spectral".

Details

See runNestedApprox for example usage.

Value

The modified approxSummary object with improved marginals.

logSumExp *Log sum exponential.*

Description

Compute the sum of two log values on the real scale and return on log scale.

Usage

```
logSumExp(log1, log2)
```

Arguments

log1	scalar of value 1 to exponentiate and sum.
log2	scalar of value 2 to exponentiate and sum with value 1.

Details

Adds two values from the log scale at the exponential scale, and then logs it. When values are really negative, this function is numerically stable and reduces the chance of underflow. It is a two value version of a log-sum-exponential.

Value

logSumExp returns `log(exp(log1) + exp(log2))`

plotMarginal *Plot the marginal posterior for a parameter*

Description

Univariate marginal posterior plotting for parameters.

Usage

```
plotMarginal(
  summary,
  node,
  log = FALSE,
  xlim = NULL,
  ngrid = 200,
  add = FALSE,
  ...
)
```

Arguments

summary	an approxSummary object, returned by runNestedApprox.
node	parameter node of interest. Specified as character (when using original scale) or integer (when using transformed scale), where the scale was specified in runNestedApprox.
log	logical; if TRUE, plot log-density. Default is FALSE.
xlim	(optional) range of x values to use. Default is the .001 and .999 quantiles.
ngrid	number of grid points at which to plot. Default is 200.
add	logical; if TRUE, add to existing plot. Default is FALSE.
...	Additional arguments passed to plotting function.

Value

None. Produces a plot.

---

qmarginal *Compute quantiles for a parameter*

Description

Quantile estimation for univariate parameter marginals.

Usage

```
qmarginal(summary, node, quantiles = c(0.025, 0.25, 0.5, 0.75, 0.975))
```

Arguments

summary	an approxSummary object, returned by runNestedApprox.
node	parameter node of interest. Specified as character (when using original scale) or integer (when using transformed scale), where the scale was specified in runNestedApprox.
quantiles	numeric vector of quantiles to compute. Default is c(0.025, 0.25, 0.5, 0.75, 0.975).

Details

Uses a spline approximation to the quantile function of the marginal posterior distribution, based on a cached approximation of the marginal density on a fine grid.

See runNestedApprox for example usage.

Value

Named vector of quantile estimates.

QUAD_RULE_BASE *Base class for nimble function list quadrature rules.*

Description

Base class for nimble function list quadrature rules.

Usage

```
QUAD_RULE_BASE()
```

Details

This is a class definition that must be included via `contains = QUAD_RULE_BASE` in a `nimbleFunction` if intending to make a quadrature rule to be used.

Author(s)

Paul van Dam-Bates

quadGH *Gauss-Hermite Quadrature Points in one dimension*

Description

Generates GH quadrature weights and nodes for integrating a general univariate function from `-Inf` to `Inf`.

Usage

```
quadGH(levels = 1, type = "GHe")
```

Arguments

`levels`	How many quadrature points to generate.
`type`	Choose type of Gauss-Hermite nodes and weights. Defaults to `"GHe"`.

Details

This function generates Gauss-Hermite (GH) points and returns a matrix with the first column as weights and the second nodes. Some numerical issues occur in Eigen decomposition making the grid weights only accurate up to 35 quadrature nodes. GH nodes approximately integrate the function g(x) = f(x)*exp(-x^2). If the type is chosen as 'type = "GHe"', the nodes are adjusted to integrate a general function, f(x), adjusting the nodes by the sqrt(2) and the weights by sqrt(2) * exp(x^2).

Author(s)

Paul van Dam-Bates

References

Golub, G. H. and Welsch, J.H. (1969). Calculation of Gauss Quadrature Rules. Mathematics of Computation 23 (106): 221-230.

Liu, Q. and Pierce, D.A. (1994). A Note on Gauss-Hermite Quadrature. Biometrika, 81(3) 624-629.

Jäckel, P. (2005). A note on multivariate Gauss-Hermite quadrature. London: ABN-Amro. Re.

---

quadGridCache *Caching system for building multiple quadrature grids.*

Description

Save the quadrature grid generated by a chosen rule and return it upon request.

Usage

```
quadGridCache()
```

Details

This function is intended to be used within another NIMBLE function to conveninetly cache multiple quadrature grids. It cannot be compiled without being included within a virtual nimble list "QUAD_CACHE_BASE".

Author(s)

Paul van Dam-Bates

---

quadRule_CCD *Central Composite Design (CCD) used for approximate posterior distributions.*

Description

Generate a d dimension CCD grid via a nimble function list.

Usage

```
quadRule_CCD(f0 = 1.1)
```

Arguments

f0	multiplier for the how far to extend nodes (default = 1.1).

Details

This function generates a Central Composite Design (CCD) grid to be used in approximate posteriors. It cannot be compiled without being included within a virtual nimble list "QUAD_RULE_BASE". On setup, f0 multiplier for the CCD grid for how much past the radius sqrt(d) to extend the nodes. Unless an advanced user, keep at default of 1.1.

Once the function is setup, it has a method 'buildGrid' which can be called to build the CCD grid. Input is d, the number of dimensions and nQuad, which is ignored but part of the default quadrature methods. Details of how the CCD grid works can be found in Rue et al. (2009). Full details for CCD as a quadrature tool are described in the thesis by Pietiläinen (2010).

Author(s)

Paul van Dam-Bates

References

Rue, H., Martino, S., and Chopin, N. (2009). Approximate Bayesian Inference for Latent Gaussian Models by Using Integrated Nested Laplace Approximations. Journal of the Royal Statistical Society, Series B 71 (2): 319–92.

Pietiläinen, V. (2010). Approximations for Integration over the Hyperparameters in Gaussian Processes. [Master's Thesis]

`quadRule_GH` *Gauss-Hermite Quadrature Rule for Laplace and Approx Posteriors*

Description

Generate a 1 dimension GHQ grid via a nimble function list.

Usage

```
quadRule_GH(type = "GHe")
```

Arguments

`type`	Choose type of Gauss-Hermite nodes and weights. Defaults to `"GHe"`.

Details

This function a 1D Gauss-Hermite Quadrature Grid (nodes and weights). When choosing 'type = "GHe"', the nodes and weights are to integrate a general function. If 'type = "GHN"', the weights are multiplied by a standard normal. It cannot be compiled without being included within a virtual nimble list "QUAD_RULE_BASE".

Author(s)

Paul van Dam-Bates

References

Jäckel, P. (2005). A note on multivariate Gauss-Hermite quadrature. London: ABN-Amro. Re. Liu, Q. and Pierce, D. (1994) A Note on Gauss-Hermite Quadrature. Biometrika, 83, 624-629.

`rmarginal` *Draw random samples from the marginal posterior of a parameter*

Description

Random sampling for univariate parameter marginals.

Usage

```
rmarginal(summary, node, n = 1000)
```

Arguments

`summary`	an approxSummary object, returned by `runNestedApprox`.
`node`	parameter node of interest. Specified as character (when using original scale) or integer (when using transformed scale), where the scale was specified in `runNestedApprox`.
`n`	number of samples to draw. Default is 1000.

Details

Uses the inverse CDF method applied to the quantile function of the marginal posterior distribution.

Value

Numeric vector of samples.

`runLaplace` *Combine steps of running Laplace or adaptive Gauss-Hermite quadrature approximation*

Description

Use an approximation (compiled or uncompiled) returned from ‘buildLaplace‘ or ‘buildAGHQ‘ to find the maximum likelihood estimate and return it with random effects estimates and/or standard errors.

Usage

```
runLaplace(
  laplace,
  pStart,
  originalScale = TRUE,
  randomEffectsStdError = TRUE,
  jointCovariance = FALSE
)

runAGHQ(
  AGHQ,
  pStart,
  originalScale = TRUE,
  randomEffectsStdError = TRUE,
  jointCovariance = FALSE
)
```

Arguments

laplace	A (compiled or uncompiled) nimble laplace approximation object returned from 'buildLaplace' or 'buildAGHQ'. These return the same type of approximation algorithm object. 'buildLaplace' is simply 'buildAGHQ' with 'nQuad=1'.
pStart	Initial values for parameters to begin optimization search for the maximum likelihood estimates. If omitted, the values currently in the (compiled or uncompiled) model object will be used.
originalScale	If TRUE, return all results on the original scale of the parameters and/or random effects as written in the model. Otherwise, return all results on potentially unconstrained transformed scales that are used in the actual computations. Transformed scales (parameterizations) are used if any parameter or random effect has contraint(s) on its support (range of allowed values). Default = TRUE.
randomEffectsStdError	If TRUE, include standard errors for the random effects estimates. Default = TRUE.
jointCovariance	If TRUE, return the full joint covariance matrix (inverse of the Hessian) of parameters and random effects. Default = FALSE.
AGHQ	Same as laplace.

Details

Adaptive Gauss-Hermite quadrature is a generalization of Laplace approximation. runLaplace simply calles runAGHQ and provides a convenient name.

These functions manage the steps of calling the 'findMLE' method to obtain the maximum likelihood estimate of the parameters and then the 'summaryLaplace' function to obtain standard errors, (optionally) random effects estimates (conditional modes), their standard errors, and the full parameter-random effects covariance matrix.

Note that for ‘nQuad > 1‘ (see buildAGHQ), i.e., AGHQ with higher order than Laplace approximation, maximum likelihood estimation is available only if all random effects integrations are univariate. With multivariate random effects integrations, one can use ‘nQuad > 1‘ only to calculate marginal log likelihoods at given parameter values. This is useful for checking the accuracy of the log likelihood at the MLE obtained for Laplace approximation (‘nQuad == 1‘). ‘nQuad‘ can be changed using the ‘updateSettings‘ method of the approximation object.

See summaryLaplace, which is called for the summary components.

Value

A list with elements MLE and summary.

MLE is the result of the findMLE method, which contains the parameter estimates and Hessian matrix. This is considered raw output, and one should normally use the contents of summary instead. (For example note that the Hessian matrix in MLE may not correspond to the same scale as the parameter estimates if a transformation was used to operate in an unconstrained parameter space.)

summary is the result of summaryLaplace (or equivalently summaryAGHQ), which contains parameter estimates and standard errors, and optionally other requested components. All results in this object will be on the same scale (parameterization), either original or transformed, as requested.

---

runNestedApprox *Run a nested approximation, returning a summary object with default inference*

Description

Uses a nested approximation (compiled or uncompiled) returned from buildNestedApprox) to do default inference and return a summary object that can be used for additional tailored inference. It estimates marginal distributions for parameters (quantiles and expectations), and can optionally return posterior samples for the latent nodes and parameters

Usage

```
runNestedApprox(
  approx,
  quantiles = c(0.025, 0.25, 0.5, 0.75, 0.975),
  originalScale = TRUE,
  improve1d = TRUE,
  nSamplesLatents = 0,
  nSamplesParams = 0
)
```

Arguments

approx	a compiled or uncompiled nestedApprox object created by buildNestedApprox.
quantiles	numeric vector of quantiles to estimate for each parameter. Default is c(0.025, 0.25, 0.5, 0.75, 0.975).

originalScale	logical; if TRUE, report results on the original scales of the parameters and latent nodes. Default is TRUE.
improve1d	logical; if TRUE and there is a single parameter, improve the estimate of the estimate of marginal distribution for the marginal by directly using the Laplace/AGHQ approximate marginal distribution rather than the asymmetric Gaussian approximation. Default is TRUE.
nSamplesLatents	number of samples of the latent nodes to draw. Default is 0.
nSamplesParams	number of samples of the parameter nodes to draw. Default is 0.

Details

This is the main user interface for running a nested approximation. It carries out default inference and then returns a summary object that can be used for further inference by calling methods on the summary object, as seen in the examples (or running the equivalent function calls with the first argument being the summary object).

Value

An object of class approxSummary containing initial results that can be used to carry out further inference.

Methods available for object of class approxSummary

Once the default inference has been run, inference can then be improved by calling different available methods within the returned object. Each method is explained in detail in their documentation, but the user may choose the following options:

- setParamGrid. Allows the user to change the parameter grid used in the nested approximation.
- improveParamMarginals. Improve univariate parameter marginals using grid-based quadrature.
- calcMarginalLogLikImproved. Calculate improved marginal log-likelihood using grid-based quadrature.
- sampleParams. Sample from the parameter posterior distribution.
- sampleLatents. Sample from the posterior distribution of the latent nodes.
- qmarginal. Compute quantiles for a parameter.
- dmarginal. Compute marginal density values for a parameter.
- rmarginal. Draw random samples from the marginal posterior of a parameter.
- emarginal. Compute the expectation of a function of a parameter under the marginal posterior distribution.
- plotMarginal. Plot the marginal posterior for a parameter.

Author(s)

Christopher Paciorek

Examples

```
data(penicillin, package="faraway")
code <- nimbleCode({
    for(i in 1:n) {
        mu[i] <- inprod(b[1:nTreat], x[i, 1:nTreat]) + re[blend[i]]
        y[i] ~ dnorm(mu[i], tau = Tau)
    }
    # Priors corresponding simply to INLA defaults and not being recommended.
    # Instead consider uniform or half-t distributions on the standard deviation scale
    # or penalized complexity priors.
    Tau ~ dgamma(1, 5e-05)
    Tau_re ~ dgamma(1, 5e-05)
    for( i in 1:nTreat ){ b[i] ~ dnorm(0, tau = 0.001) }
    for( i in 1:nBlend ){ re[i] ~ dnorm(0, tau = Tau_re) }
})
X <- model.matrix(~treat, data = penicillin)
data = list(y = penicillin$yield)
constants = list(nTreat = 4, nBlend = 5, n = nrow(penicillin),
                 x = X, blend = as.numeric(penicillin$blend))
inits <- list(Tau = 1, Tau_re = 1, b = c(mean(data$y), rep(0,3)), re = rep(0,5))

model <- nimbleModel(code, data = data, constants = constants,
                 inits = inits, buildDerivs = TRUE)
approx <- buildNestedApprox(model = model)


comp_model <- compileNimble(model)
comp_approx <- compileNimble(approx, project = model)
result <- runNestedApprox(comp_approx)
# Improve marginals for a parameter node using AGHQ.
result$improveParamMarginals(nodes = 'Tau_re', nMarginalGrid = 9)
# Specify other quantiles of interest.
result$qmarginal('Tau_re', quantiles = c(.05, .95))
# Compute other expectations of interest, here the mean on the standard deviation scale
result$emarginal('Tau_re', function(x) 1/sqrt(x))

# Sample from the approximate posterior for the latent nodes.
latent_sample <- result$sampleLatents(n = 1000)
# For joint inference on parameters, sample from the approximate posterior for the parameters.
param_sample <- result$sampleParams(n = 1000)
```

---

sampleLatents *Sample from the posterior distribution of the latent nodes*

Description

Draws samples from the posterior distribution of the latent nodes. Optionally includes parameter values corresponding to each sample.

Usage

```
sampleLatents(summary, n = 1000, includeParams = FALSE)
```

Arguments

summary	an approxSummary object, returned by runNestedApprox.
n	Number of samples to draw (default: 1000).
includeParams	logical; if TRUE, include parameter values corresponding to each sample. Default is FALSE.

Details

The sampling approach uses stratified sampling from a weighted mixture of multivariate normals, where the weights are based on the approximate marginal density at each grid point in the parameter grid. For each point, the multivariate normal is based on Laplace approximation, using the maximum for the mean and the inverse Hessian for the covariance matrix. This approach is not valid for sparse AGHQ due to negative quadrature weights. This can be updated by setParamGrid and any quadrature rule other than AGHQSPARSE.

The parameter values corresponding to the samples can be requested via includeParams.

Note that NIMBLE's nested approximation framework does not provide marginals for the latent nodes based on analytic approximation, so both joint and univariate inference on the latent nodes is from sampling.

See runNestedApprox for example usage.

Value

Matrix of latent samples.

---

sampleParams *Sample from the parameter posterior distribution*

Description

Draws samples from the parameter posterior using the asymmetric Gaussian approximation. Optionally uses a copula approach to match the univariate marginals to the currently-available marginal distributions (based on either the initial asymmetric Gaussian approximation or improved marginals from calling improveParamMarginals).

Usage

```
sampleParams(summary, n = 1000, matchMarginals = TRUE)
```

Arguments

summary	an approxSummary object, returned by runNestedApprox.
n	number of samples to draw. Default is 1000.
matchMarginals	logical; if TRUE (the default), match marginals using copula approach.

Details

Draws samples from the joint parameter posterior distribution (marginalized with respect to the latent nodes) using the asymmetric Gaussian approximation.

This is useful for joint inference on the parameters, including inference on functions of more than one parameter.

See `runNestedApprox` for example usage.

Value

Matrix of parameter samples.

`setParamGrid` *Set the parameter grid for the nested approximation*

Description

Allows the user to change the parameter grid used in the nested approximation.

Usage

```
setParamGrid(summary, quadRule = "NULL", nQuad = -1, prune = -1)
```

Arguments

`summary`	an approxSummary object, returned by `runNestedApprox`.
`quadRule`	quadrature rule to use for the parameter grid. Can be any of `"CCD"`, `"AGHQ"`, `"AGHQSPARSE"`, or `"USER"`, the latter to use the user-defined rule provided to `buildNestedApprox`.
`nQuad`	number of quadrature points (not used for `"CCD"`.
`prune`	pruning parameter for removing AGHQ points at low-density points.

Details

If the chosen quadrature rule is `"AGHQSPARSE"`, then `sampleLatents` will no longer work as it requires that all the quadrature weights are non-negative, which is no longer true for sparse AGHQ.

To use a user-defined quadrature rule, this needs to be passed to `buildNestedApprox`.

Value

None. Modifies the summary object in place.

summaryLaplace *Summarize results from Laplace or adaptive Gauss-Hermite quadrature approximation*

Description

Process the results of the ‘findMLE‘ method of a nimble Laplace or AGHQ approximation into a more useful format.

Usage

```
summaryLaplace(
  laplace,
  MLEoutput,
  originalScale = TRUE,
  randomEffectsStdError = TRUE,
  jointCovariance = FALSE
)

summaryAGHQ(
  AGHQ,
  MLEoutput,
  originalScale = TRUE,
  randomEffectsStdError = TRUE,
  jointCovariance = FALSE
)
```

Arguments

laplace	The Laplace approximation object, typically the compiled one. This would be the result of compiling an object returned from ‘buildLaplace‘.
MLEoutput	The maximum likelihood estimate using Laplace or AGHQ, returned from e.g. ‘approx$findMLE(...)‘, where approx is the algorithm object returned by ‘buildLaplace‘ or ‘buildAGHQ‘, or (more typically) the result of compiling that object with ‘compileNimble‘. See ‘help(buildLaplace)‘ for more information.
originalScale	Should results be returned using the original parameterization in the model code (TRUE) or the potentially transformed parameterization used internally by the Laplace approximation (FALSE). Transformations are used for any parameters and/or random effects that have constrained ranges of valid values, so that in the transformed parameter space there are no constraints. (default = TRUE)
randomEffectsStdError	If TRUE, calculate the standard error of the estimates of random effects values. (default = TRUE)
jointCovariance	If TRUE, calculate the joint covariance matrix of the parameters and random effects together. If FALSE, calculate the covariance matrix of the parameters. (default = FALSE)

AGHQ — Same as `laplace`. Note that ‘buildLaplace‘ and ‘buildAGHQ‘ create the same kind of algorithm object that can be used interchangeably. ‘buildLaplace‘ simply sets the number of quadrature points (‘nQuad‘) to 1 to achieve Laplace approximation as a special case of AGHQ.

Details

The numbers obtained by this function can be obtained more directly by ‘approx$summary(...)‘. The added benefit of ‘summaryLaplace‘ is to arrange the results into data frames (for parameters and random effects), with row names for the model nodes, and also adding row and column names to the covariance matrix.

Value

A list with data frames ‘params‘ and ‘randomEffects‘, each with columns for ‘estimate‘ and (possibly) ‘se‘ (standard error) and row names for model nodes, a matrix ‘vcov‘ with the covariance matrix with row and column names, and ‘originalScale‘ with the input value of ‘originalScale‘ so it is recorded for later use if wanted.

Index