Package 'bisque'

April 26, 2019

Type Package

Title Approximate Bayesian Inference via Sparse Grid Quadrature Evaluation (BISQuE) for Hierarchical Models

Version 1.0.1

Date 2019-04-25

Author Joshua Hewitt

Maintainer Joshua Hewitt <joshua.hewitt@colostate.edu>

Description
Implementation of the 'bisque' strategy for approximate Bayesian posterior inference. See Hewitt and Hoeting (2019) <arXiv:1904.07270> for complete details. 'bisque' combines conditioning with sparse grid quadrature rules to approximate marginal posterior quantities of hierarchical Bayesian models. The resulting approximations are computationally efficient for many hierarchical Bayesian models. The 'bisque' package allows approximate posterior inference for custom models; users only need to specify the conditional densities required for the approximation.

License GPL-3

RoxygenNote 6.1.1

Suggests testthat, fields

Depends R (>= 3.0.2)

Imports mvQuad, Rcpp, doRNG, foreach, itertools

LinkingTo Rcpp (>= 0.12.4), RcppArmadillo, RcppEigen (>= 0.3.3.3.1)

SystemRequirements A system with a recent-enough C++11 compiler (such as g++-4.8 or later).

NeedsCompilation yes

Encoding UTF-8

Repository CRAN

Date/Publication 2019-04-26 09:30:11 UTC

R topics documented:

`createLocScaleGrid` *Create a centered and scaled sparse integration grid*

Description

Enhances mvQuad::createNIGrid by shifting and scaling a sparse integration grid, and evaluating the weight function at each of the grid nodes.

Usage

```
createLocScaleGrid(mu = 0, prec = 1, level = 2, quadError = FALSE,
  prec.chol = chol(prec))
```

Arguments

`mu`	location at which grid should be centered
`prec`	"precision matrix" associated with the integration grid. When building a sparse integration grid for a density, `prec` is often the negative of the hessian at the mode.
`level`	accuracy level. This is typically number of grid points for the underlying 1D quadrature rule. [description from mvQuad::createNIGrid]
`quadError`	provide additional information about the grid points and integration weights for the quadrature rule with `level-1`. This information can facilitate approximating the quadrature error.
`prec.chol`	Upper-triangular Cholesky decomposition of precision matrix.

See Also

`mvQuad::createNIGrid`

Examples

```
g = createLocScaleGrid(mu = c(1,0), prec = diag(c(1,.5)), level = 2 )
```

`dmix` *Evaluate a mixture density*

Description

Evaluates mixture densities of the form

$$f(x) = \sum_{j=1}^{k} f(x|\theta^{(k)}) w_k$$

where the w_k are (possibly negative) weights that sum to 1 and $f(x|\theta^{(k)})$ are densities that are specified via parameters $\theta^{(k)}$, which are passed in the function argument `params`. A unique feature of this function is that it is able to evaluate mixture densities in which some of the mixture weights w_k are negative.

Usage

```
dmix(x, f, params, wts, log = FALSE, errorNodesWts = NULL, ...)
```

Arguments

`x`	Points at which the mixture should be evaluated. If the density is multivariate, then each row of `x` should contain one set of points at which the mixture should be evaluated.
`f`	Density used in the mixture. The function should be defined so it is can be called via `f(x, params, log, ...)`. The density f is evaluated at the points in `x` using one set of parameters `params`, i.e., for some specific $\theta^{(k)}$. if `log==TRUE`, then $ln(f)$ is returned. Additional parameters may be passed to f via $\ldots$.
`params`	Matrix in which each row contains parameters that define f. The number of rows in `params` should match the number of mixture components k.
`wts`	vector of weights for each mixture component
`log`	TRUE to return the log of the mixture density
`errorNodesWts`	list with elements `inds` and `weights` that point out which `params` get used to compute an approximation of the quadrature error.
`...`	additional arguments to be passed to `f`

Examples

```
# evaluate mixture density at these locations
x = seq(0, 1, length.out = 100)

# density will be a mixture of beta distributions
f = function(x, theta, log = FALSE) {
  dbeta(x, shape1 = theta[1], shape2 = theta[2], log = log)
}

# beta parameters are randomly assigned
params = matrix(exp(2*runif(10)), ncol=2)

# mixture components are equally weighted
wts = rep(1/nrow(params), nrow(params))

# evaluate mixture density
fmix = dmix(x = x, f = f, params = params, wts = wts)

# plot mixture density
plot(x, fmix, type='l', ylab = expression(f(x)),
     ylim = c(0, 4))

# plot component densities
for(i in 1:length(wts)){
  curve(f(x, params[i,]), col = 2, add = TRUE)
}
```

`emix` *Compute expectations via weighted mixtures*

Description

Approximates expectations of the form

$$E[h(\theta)] = \int h(\theta) f(\theta) d\theta$$

using a weighted mixture

$$E[h(\theta)] \approx \sum_{j=1}^{k} h(\theta^{(k)}) w_k$$

Usage

```
emix(h, params, wts, ncores = 1, errorNodesWts = NULL, ...)
```

Arguments

h	Function for which the expectation should be taken. The function should be defined so it is can be called via `f(params, ...)`. Additional parameters may be passed to h via `...`.
params	Matrix in which each row contains parameters at which h should be evaluated. The number of rows in `params` should match the number of mixture components k.
wts	vector of weights for each mixture component
ncores	number of cores over which to evaluate mixture. this function assumes a parallel backend is already registered.
errorNodesWts	list with elements `inds` and `weights` that point out which `params` get used to compute an approximation of the quadrature error.
...	additional arguments to be passed to `h`

Examples

```
# density will be a mixture of betas
params = matrix(exp(2*runif(10)), ncol=2)

# mixture components are equally weighted
wts = rep(1/nrow(params), nrow(params))

# compute mean of distribution by cycling over each mixture component
h = function(p) { p[1] / sum(p) }

# compute mixture mean
mean.mix = emix(h, params, wts)

# (comparison) Monte Carlo estimate of mixture mean
nsamples = 1e4
component = sample(x = 1:length(wts), size = nsamples, prob = wts,
                   replace = TRUE)
x = sapply(component, function(cmp) {
  rbeta(n = 1, shape1 = params[cmp, 1], shape2 = params[cmp, 2])
})
mean.mix.mc = mean(x)

# compare estimates
c(emix = mean.mix, MC = mean.mix.mc)
```

---

furseals *Data from a capture-recapture study of fur seal pups*

Description

These data are used in the book "Computational Statistics" by G.H. Givens and J.A. Hoeting (2013). They are discussed in Chapter 7, Examples 7.2,7.3,7.8, and Exercise 7.2.

Usage

```
furseals
```

Format

A data.frame with variables:

i The census attempt

c Number of pups captured in census attempt

m Number of newly captured pups

Details

As described by the authors:

Source: Richard Barker, University of Otago, New Zealand

Description: Data from a capture-recapture study conducted on the Otago Penninsula, South Island, New Zealand. Fur seal pups were marked and released during 7 census attempts in one season. The population is assumed closed. For each census attempt, the number of pups captured and the number of these captures corresponding to pups never previously caught are recorded.

Source

https://www.stat.colostate.edu/computationalstatistics/

https://www.stat.colostate.edu/computationalstatistics/datasets.zip

Examples

```
data("furseals")
str(furseals)
```

---

itx *Named inverse transformation functions*

Description

Evaluates the inverse of the named link function at the locations `x`.

Usage

```
itx(x, link, linkparams)
```

Arguments

x	Values at which to evaluate the inverse link function
link	Character vector specifying link function for which the inverse link function should be evaluated. Supports 'identity', 'log', and 'logit'.
linkparams	Optional list of additional parameters for link functions. For example, the logit function can be extended to allow mappings to any closed interval. There should be one list entry for each link function. Specify NA if defaults should be used.

Examples

```
bisque:::itx(0, 'logit', list(NA))
```

---

jac.exp *Jacobian for exponential transform*

Description

Let $X = exp(Y)$ be a transformation of a random variable Y. This function computes the jacobian $J(x)$ when using the density of Y to evaluate the density of X via

$$f(x) = f_y(ln(x))J(x)$$

where

$$J(x) = d/dxln(x).$$

Usage

```
jac.exp(x, log = TRUE)
```

Arguments

x	value at which to evaluate $J(x)$
log	TRUE to return $log(J(x))$

Examples

```
jac.exp(1)
```

jac.invlogit *Jacobian for logit transform*

Description

Let $X = logit^{-1}(Y)$ be a transformation of a random variable Y. This function computes the jacobian $J(x)$ when using the density of Y to evaluate the density of X via

$$f(x) = f_y(logit(x))J(x)$$

where

$$J(x) = d/dx logit(x).$$

Usage

```
jac.invlogit(x, log = TRUE)
```

Arguments

x	value at which to evaluate $J(x)$
log	TRUE to return $log(J(x))$

Examples

```
jac.invlogit(1)
```

jac.log *Jacobian for log transform*

Description

Let $X = log(Y)$ be a transformation of a random variable Y. This function computes the jacobian $J(x)$ when using the density of Y to evaluate the density of X via

$$f(x) = f_y(exp(x))J(x)$$

where

$$J(x) = d/dx exp(x).$$

Usage

```
jac.log(x, log = TRUE)
```

Arguments

x	value at which to evaluate $J(x)$
log	TRUE to return $log(J(x))$

Examples

```
jac.log(1)
```

---

jac.logit *Jacobian for logit transform*

Description

Let $X = logit(Y)$ be a transformation of a random variable Y that lies in the closed interval (L,U). This function computes the jacobian $J(x)$ when using the density of Y to evaluate the density of X via

$$f(x) = f_y(logit^{-1}(x) * (U - L) + L)J(x)$$

where

$$J(x) = (U - L)d/dx logit^{-1}(x).$$

Usage

```
jac.logit(x, log = TRUE, range = c(0, 1))
```

Arguments

x	value at which to evaluate $J(x)$
log	TRUE to return $log(J(x))$
range	vector specifying min and max range of the closed interval for the logit. While the logit is defined for real numbers in the unit interval, we extend it to real numbers in arbitrary closed intervals (L,U).

Examples

```
jac.logit(1)
```

kCompute *Use sparse grid quadrature techniques to integrate (unnormalized) densities*

Description

This function integrates (unnormalized) densities and may be used to compute integration constants for unnormalized densities, or to marginalize a joint density, for example.

Usage

```
kCompute(f, init, method = "BFGS", maxit = 10000, level = 2,
  log = FALSE, link = NULL, linkparams = NULL, quadError = FALSE,
  ...)
```

Arguments

f	(Unnormalized) density to integrate. the function f should include an argument log, which returns $log(f(x))$.
init	Initial guess for the density's mode
method	method to be used to search for the density's mode
maxit	maximum number of iterations optim should use in searching for the density's mode
level	accuracy level (typically number of grid points for the underlying 1D quadrature rule) [description from mvQuad::createNIGrid]
log	TRUE to return log of integration constant
link	character vector that specifies transformations used during optimization and integration of f(theta2 | X). while theta2 may be defined on arbitrary support, wtdMix performs optimization and integration of theta2 on an unconstrained support. the link vector describes the transformations that must be applied to each element of theta2. Jacobian functions for the transformations will automatically be added to the optimization and integration routines. currently supported link functions are 'log', 'logit', and 'identity'.
linkparams	Optional list of additional parameters for link functions. For example, the logit function can be extended to allow mappings to any closed interval. There should be one list entry for each link function. Specify NA if no additional arguments are passed.
quadError	TRUE if integration nodes and weight should be computed for the level-1 integration grid, so that quadrature approximation error can be estimated.
...	additional arguments to pass to f

Examples

```
kCompute(dgamma, init = 1, shape=2, link='log', level = 5)
```

logjac *Wrapper to abstractly evaluate log-Jacobian functions for transforms*

Description

Wrapper to abstractly evaluate log-Jacobian functions for transforms

Usage

```
logjac(x, link, linkparams)
```

Arguments

x	values at which to evaluate $J(x)$
link	Character vector specifying link function for which the inverse link function should be evaluated. Supports 'identity', 'log', and 'logit'.
linkparams	Optional list of additional parameters for link functions. For example, the logit function can be extended to allow mappings to any closed interval. There should be one list entry for each link function. Specify NA if defaults should be used.

See Also

jac.log, jac.logit

Examples

```
bisque:::logjac(1, 'logit', list(NA))
```

mergePars *Merge pre-computed components of f(theta1 | theta2, X)*

Description

For use in the parallel call in wtdMix()

Usage

```
mergePars(x, y)
```

Arguments

x	Output from one of the parallel calls in wtdMix()
y	Another output from one of the parallel calls in wtdMix()

sFit *Fit a spatially mean-zero spatial Gaussian process model*

Description

Uses a Gibbs sampler to estimate the parameters of a Matern covariance function used to model observations from a Gaussian process with mean 0.

Usage

```
sFit(x, coords, nSamples, thin = 1, rw.initsd = 0.1, inits = list(),
  C = 1, alpha = 0.44, priors = list(sigmasq = list(a = 2, b = 1),
  rho = list(L = 0, U = 1), nu = list(L = 0, U = 1)))
```

Arguments

x	Observation of a spatial Gaussian random field, passed as a vector
coords	Spatial coordinates of the observation
nSamples	(thinned) number of MCMC samples to generate
thin	thinning to be used within the returned MCMC samples
rw.initsd	initial standard devaition for random walk proposals. this parameter will be adaptively tuned during sampling
inits	list of initial parameters for the MCMC chain
C	scale factor used during tuning of the random walk proposal s.d.
alpha	target acceptance rate for which the random walk proposals should optimize
priors	parameters to specify the prior distributions for the model

Examples

```
library(fields)

simulate.field = function(n = 100, range = .3, smoothness = .5, phi = 1){
  # Simulates a mean-zero spatial field on the unit square
  #
  # Parameters:
  #  n - number of spatial locations
  #  range, smoothness, phi - parameters for Matern covariance function

  coords = matrix(runif(2*n), ncol=2)

  Sigma = Matern(d = as.matrix(dist(coords)),
                 range = range, smoothness = smoothness, phi = phi)

  list(coords = coords,
       params = list(n=n, range=range, smoothness=smoothness, phi=phi),
       x = t(chol(Sigma)) %*%  rnorm(n))
```

```
}

# simulate data
x = simulate.field()

# configure gibbs sampler
it = 100

# run sampler using default posteriors
post.samples = sFit(x = x$x, coords = x$coords, nSamples = it)

# build kriging grid
cseq = seq(0, 1, length.out = 10)
coords.krig = expand.grid(x = cseq, y = cseq)

# sample from posterior predictive distribution
burn = 75
samples.krig = sKrig(x$x, post.samples, coords.krig = coords.krig, burn = burn)
```

`sKrig` *Draw posterior predictive samples from a spatial Gaussian process model*

Description

Draw posterior predictive samples from a spatial Gaussian process model

Usage

```
sKrig(x, sFit, coords.krig, coords = sFit$coords, burn = 0,
  ncores = 1)
```

Arguments

`x`	Observation of a spatial Gaussian random field, passed as a vector
`sFit`	posterior samples of model parameters; output from bisque::sFit
`coords.krig`	Spatial coordinates at which the field should be interpolated
`coords`	Spatial coordinates at which observations are available
`burn`	number of posterior samples to discard from sFit before sampling
`ncores`	Kriging is done via composition sampling, which may be done in parallel. `ncores` specifies the number of cores over which sampling is done. If `ncores>1`, bisque::sKrig assumes that a parallel backend suitable for use with the foreach package is already registered.

Examples

```
library(fields)

simulate.field = function(n = 100, range = .3, smoothness = .5, phi = 1){
  # Simulates a mean-zero spatial field on the unit square
  #
  # Parameters:
  #  n - number of spatial locations
  #  range, smoothness, phi - parameters for Matern covariance function

  coords = matrix(runif(2*n), ncol=2)

  Sigma = Matern(d = as.matrix(dist(coords)),
                 range = range, smoothness = smoothness, phi = phi)

  list(coords = coords,
       params = list(n=n, range=range, smoothness=smoothness, phi=phi),
       x = t(chol(Sigma)) %*%  rnorm(n))
}

# simulate data
x = simulate.field()

# configure gibbs sampler
it = 100

# run sampler using default posteriors
post.samples = sFit(x = x$x, coords = x$coords, nSamples = it)

# build kriging grid
cseq = seq(0, 1, length.out = 10)
coords.krig = expand.grid(x = cseq, y = cseq)

# sample from posterior predictive distribution
burn = 75
samples.krig = sKrig(x$x, post.samples, coords.krig = coords.krig, burn = burn)
```

`tx` *Named transformation functions*

Description

Evaluates the named link function at the locations `x`.

Usage

```
tx(x, link, linkparams)
```

Arguments

x	Values at which to evaluate the link function
link	Character vector specifying link function to evaluate. Supports 'identity', 'log', and 'logit'.
linkparams	Optional list of additional parameters for link functions. For example, the logit function can be extended to allow mappings to any closed interval. There should be one list entry for each link function. Specify NA if defaults should be used.

Examples

```
bisque:::tx(0.5, 'logit', list(NA))
```

---

wBuild *Derive parameters for building integration grids*

Description

Note: w is defined on the transformed scale, but for convenience f is defined on the original scale.

Usage

```
wBuild(f, init, dim.theta2 = length(init), approx = "gaussian",
  link = rep("identity", length(init)), link.params = rep(list(NA),
  length(init)), optim.control = list(maxit = 5000, method = "BFGS"),
  ...)
```

Arguments

f	function used to derive the weight function w. f must be able to be called via f(par, log, ...)
init	initial guess for mode of f.
dim.theta2	wBuild assumes par is partitioned such that par=c(theta1,theta2). dim.theta2 specifies the size of the partition. The default is to assume that f is defined without a theta1 component.
approx	Style of approximation (i.e., w) to be created from mode of f.
	'gaussian' Gaussian approximation for theta2 at the mode of f. Assumes f is proportional to the marginal posterior density for theta2.
	'condgauss' Gaussian approximation for theta2 at the mode of f. The approximation is conditioned on the value of the mode for theta1. Assumes f is proportional to the joint posterior density for theta1,theta2.
	'condgauss-laplace' Gaussian approximation for theta2 at the mode of f. The approximation is conditioned on a separate laplace approximation of the marginal posterior mode for theta1. Assumes f is proportional to the joint posterior density for theta1,theta2.

'margauss' Gaussian approximation for theta2 at the mode of f. Assumes f is proportional to the joint posterior density for theta1,theta2., then uses the marginal mean and covariance from the posterior's gaussian approximation.

link — character vector that specifies transformations used during optimization and integration of $f(\theta_2|X)$. While θ_2 may be defined on arbitrary support, wtdMix performs optimization and integration of θ_2 on an unconstrained support. The link vector describes the transformations that must be applied to each element of θ_2. Jacobian functions for the transformations will automatically be added to the optimization and integration routines. Currently supported link functions are 'log', 'logit', and 'identity'.

link.params — Optional list of additional parameters for link functions. For example, the logit function can be extended to allow mappings to any closed interval. There should be one list entry for each link function. Specify NA if no additional arguments are passed.

optim.control — List of arguments to pass to stat::optim when used to find mode of f.

maxit Maximum number of iterations to run optim for.

method Optimization routine to use with optim.

... — additional arguments needed for function evaluation.

Examples

```
# Use BISQuE to approximate the marginal posterior distribution for unknown
# population f(N|c, r) for the fur seals capture-recapture data example in
# Givens and Hoeting (2013), example 7.10.

data('furseals')

# define theta transformation and jacobian
tx.theta = function(theta) {
  c(log(theta[1]/theta[2]), log(sum(theta[1:2])))
}
itx.theta = function(u) {
  c(exp(sum(u[1:2])), exp(u[2])) / (1 + exp(u[1]))
}
lJ.tx.theta = function(u) {
  log(exp(u[1] + 2*u[2]) + exp(2*sum(u[1:2]))) - 3 * log(1 + exp(u[1]))
}

# compute constants
r = sum(furseals$m)
nC = nrow(furseals)

# set basic initialization for parameters
init = list(U = c(-.7, 5.5))
init = c(init, list(
  alpha = rep(.5, nC),
  theta = itx.theta(init$U),
  N = r + 1
```

```
))


post.alpha_theta = function(theta2, log = TRUE, ...) {
  # Function proportional to f(alpha, U1, U2 | c, r)

  alpha = theta2[1:nC]
  u = theta2[-(1:nC)]
  theta = itx.theta(u)
  p = 1 - prod(1-alpha)

  res = - sum(theta)/1e3 - r * log(p) + lJ.tx.theta(u) -
    nC * lbeta(theta[1], theta[2])
  for(i in 1:nC) {
    res = res + (theta[1] + furseals$c[i] - 1)*log(alpha[i]) +
      (theta[2] + r - furseals$c[i] - 1)*log(1-alpha[i])
  }

  if(log) { res } else { exp(res) }
}

post.N.mixtures = function(N, params, log = TRUE, ...) {
  # The mixture component of the weighted mixtures for f(N | c, r)
  dnbinom(x = N-r, size = r, prob = params, log = log)
}

mixparams.N = function(theta2, ...) {
  # compute parameters for post.N.mixtures
  1 - prod(1 - theta2[1:nC])
}


w.N = wBuild(f = post.alpha_theta, init = c(init$alpha, init$U),
             approx = 'gauss', link = c(rep('logit', nC), rep('identity', 2)))

m.N = wMix(f1 = post.N.mixtures, f1.precompute = mixparams.N,
           f2 = post.alpha_theta, w = w.N)



# compute posterior mean
m.N$expectation$Eh.precompute(h = function(p) ((1-p)*r/p + r),
                                  quadError = TRUE)

# compute posterior density
post.N.dens = data.frame(N = r:105)
post.N.dens$d = m.N$f(post.N.dens$N)

# plot posterior density
plot(d~N, post.N.dens, ylab = expression(f(N~'|'~bold(c),r)))
```

wMix *Construct a weighted mixture object*

Description

For a Bayesian model

$$X \; f(X|\theta_1, \theta_2)$$
$$(\theta_1, \theta_2) \; f(\theta_1, \theta_2),$$

the marginal posterior $f(\theta_1|X)$ distribution can be approximated via weighted mixtures via

$$f(\theta_1|X) \approx \sum_{j=1}^{K} f(\theta_1|X, \theta_2) w_j$$

where w_j is based on $f(\theta_2^{(j)}|X)$ and weights $\tilde{w}_j$, where $\theta_2^{(j)}$ and $\tilde{w}_j$ are nodes and weights for a sparse-grid quadrature integration scheme. The quadrature rule is developed by finding the posterior mode of $f(\theta_2|X)$, after transforming θ_2 to an unconstrained support. For best results, θ_2 should be a continuous random variable, or be able to be approximated by one.

Usage

```
wMix(f1, f2, w, f1.precompute = function(x, ...) {     x },
  spec = "ff", level = 2, c.int = NULL, c.level = 2,
  c.init = NULL, c.link = rep("identity", length(c.init)),
  c.link.params = rep(list(NA), length(c.init)),
  c.optim.control = list(maxit = 5000, method = "BFGS"), ncores = 1,
  quadError = TRUE, ...)
```

Arguments

f1	evaluates $f(\theta_1|X, \theta_2)$. f1 must be able to be called via f1(theta1, params, log, ...).
	theta1 a matrix of parameters at which to evaluate $f(\theta_1|X, \theta_2)$. each row should be one set of values at which the density should be evaluated
	params a vector of parameters needed to evaluate $f(\theta_1|X, \theta_2)$. In most cases params will equal $theta_2$, but in some cases, $f(\theta_1|X, \theta_2)$ depends on functions of θ_2, which can be pre-evaluated as the weighted mixture approximation is constructed.
	log TRUE to return $ln(f(\theta_1|X, \theta_2))$
	... additional arguments needed for function evaluation
f2	evaluates $f(theta_2|X)$. f2 must be able to be called via f2(theta2, log, ...).
w	wBuild object created by wBuild function. w contains posterior mode of $f(\theta_2|X)$ and wrapper functions to generate quadrature grid.
f1.precompute	function that pre-computes parameters for evaluating $f(\theta_1|X, \theta_2)$. f1.precompute must be able to be called via f1.precompute(theta2, ...) and return the argument params for the function f1.

spec	Specification of whether f1 and f2 are known exactly, or need numerical approximation to determine integration constants. 'ff' if both functions are known, 'gg' if f1 is proportional to the full conditional distribution $f(\theta_1\|\theta_2, X)$, but needs the integration constant computed, and if the marginal posterior $f(theta_2\|X)$ is equal to f2 times the integration constant that needs to be numerically approximated.
level	accuracy level of the numerical approximation (typically number of grid points for the underlying 1D quadrature rule) [description from mvQuad::createNIGrid]
c.int	If spec=='gg', then c.int specifies the function that can be integrated in order to yield the missing integration constant.
c.level	accuracy level of the numerical approximation for c.int (typically number of grid points for the underlying 1D quadrature rule) [description from mvQuad::createNIGrid]
c.init	initial guess for mode of c.int.
c.link	character vector that specifies transformations used during optimization and integration of c.int. See corresponding documentation in wBuild function for more details.
c.link.params	Optional list of additional parameters for link functions used with c.int. See corresponding documentation in wBuild function for more details.
c.optim.control	Arguments used to find mode of c.int. See corresponding documentation in wBuild function for more details.
ncores	number of cores used to parallelize computation of parameters for $f(\theta_1\|\theta_2, X)$.
quadError	TRUE if integration nodes and weight should be computed for the level-1 integration grid, so that quadrature approximation error can be estimated.
...	Additional arguments to pass to f1, f1.precompute, f12, and f2.

Value

A list with class wMix, which contains the following items.

f Function for evaluating the posterior density $f(\theta_1|X)$. f is callable via f(theta1, log, ...).

mix A matrix containing the pre-computed parameters for evaluating the mixture components $f(\theta_1|\theta_2, X)$. Each row of the matrix contains parameters for one of the K mixture components.

wts Integration weights for each of the mixture components. Some of the weights may be negative.

expectation List containing additional tools for computing posterior expectations of $f(\theta_2|X)$. However, posterior expectations of $f(\theta_1|X)$ can also be computed when expectations of $f(\theta_1|\theta_2, X)$ are known. The elements of expectation are

- Eh Function to compute $E[h(\theta_2)|X]$. Eh is callable via Eh(h, ...), where h is a function callable via h(theta2, ...) and ... are additional arguments to the function. The function h is evaluated at the quadrature nodes $\theta_2^{(j)}$.
- Eh.precompute Exactly the same idea as Eh, but the function h is evalauted at the quadrature nodes after being passed through the function f1.precompute.
- grid The sparse-quadrature integration grid used. Helpful for seeing the quadrature nodes $\theta_2^{(j)}$.

wts The integration weights for approximating the expectation $E[h]$. Note that these integration weights may differ from the main integration weights for evaluating the posterior density $f(\theta_1|X)$.

Examples

```
# Use BISQuE to approximate the marginal posterior distribution for unknown
# population f(N|c, r) for the fur seals capture-recapture data example in
# Givens and Hoeting (2013), example 7.10.

data('furseals')

# define theta transformation and jacobian
tx.theta = function(theta) {
  c(log(theta[1]/theta[2]), log(sum(theta[1:2])))
}
itx.theta = function(u) {
  c(exp(sum(u[1:2])), exp(u[2])) / (1 + exp(u[1]))
}
lJ.tx.theta = function(u) {
  log(exp(u[1] + 2*u[2]) + exp(2*sum(u[1:2]))) - 3 * log(1 + exp(u[1]))
}

# compute constants
r = sum(furseals$m)
nC = nrow(furseals)

# set basic initialization for parameters
init = list(U = c(-.7, 5.5))
init = c(init, list(
  alpha = rep(.5, nC),
  theta = itx.theta(init$U),
  N = r + 1
))


post.alpha_theta = function(theta2, log = TRUE, ...) {
  # Function proportional to f(alpha, U1, U2 | c, r)

  alpha = theta2[1:nC]
  u = theta2[-(1:nC)]
  theta = itx.theta(u)
  p = 1 - prod(1-alpha)

  res = - sum(theta)/1e3 - r * log(p) + lJ.tx.theta(u) -
    nC * lbeta(theta[1], theta[2])
  for(i in 1:nC) {
    res = res + (theta[1] + furseals$c[i] - 1)*log(alpha[i]) +
      (theta[2] + r - furseals$c[i] - 1)*log(1-alpha[i])
  }

  if(log) { res } else { exp(res) }
}
```

```
post.N.mixtures = function(N, params, log = TRUE, ...) {
  # The mixture component of the weighted mixtures for f(N | c, r)
  dnbinom(x = N-r, size = r, prob = params, log = log)
}

mixparams.N = function(theta2, ...) {
  # compute parameters for post.N.mixtures
  1 - prod(1 - theta2[1:nC])
}


w.N = wBuild(f = post.alpha_theta, init = c(init$alpha, init$U),
             approx = 'gauss', link = c(rep('logit', nC), rep('identity', 2)))

m.N = wMix(f1 = post.N.mixtures, f1.precompute = mixparams.N,
           f2 = post.alpha_theta, w = w.N)



# compute posterior mean
m.N$expectation$Eh.precompute(h = function(p) ((1-p)*r/p + r),
                                    quadError = TRUE)

# compute posterior density
post.N.dens = data.frame(N = r:105)
post.N.dens$d = m.N$f(post.N.dens$N)

# plot posterior density
plot(d~N, post.N.dens, ylab = expression(f(N~'|'~bold(c),r)))
```

Index