Package ‘bigGP’

August 29, 2016

Version 0.1-6

Date 2015-07-07

Title Distributed Gaussian Process Calculations

Depends R (>= 3.0.0), Rmpi (>= 0.6-2), methods

Suggests rlecuyer, fields

LazyData Yes

Description Distributes Gaussian process calculations across nodes in a distributed memory setting, using Rmpi. The bigGP class provides high-level methods for maximum likelihood with normal data, prediction, calculation of uncertainty (i.e., posterior covariance calculations), and simulation of realizations. In addition, bigGP provides an API for basic matrix calculations with distributed covariance matrices, including Cholesky decomposition, back/forwardsolve, crossproduct, and matrix multiplication.

SystemRequirements OpenMPI or MPICH2

OS_type unix

License GPL (>= 2)

URL http://www.jstatsoft.org/v63/i10/

Collate 'auxil.R' 'bigGP.R' 'collectDistribute.R' 'distributedComputation.R' 'krigeProblem.R'

NeedsCompilation yes

Author Christopher Paciorek [aut, cre],
Benjamin Lipshitz [aut],
Prabhat [ctb],
Cari Kaufman [ctb],
Tina Zhuo [ctb],
Rollin Thomas [ctb]

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

Repository CRAN

Date/Publication 2015-07-08 13:49:56

R topics documented:

`alloc` *Create Object with its Own Memory*

Description

`alloc` is an internal auxiliary function that creates an object of the size of the input with the goal of allocating new memory for use in the C functions used by the package.

Usage

```
alloc(input, inputPos = '.GlobalEnv')
```

Arguments

input	an object name, given as a character string, giving the name of the object whose size is to be mimiced in creating the output, or the length of the output vector to be created.
inputPos	where to look for the input, given as a character string (unlike `get`). This can indicate an environment, a list, or a ReferenceClassobject.

Value

A new numeric vector of the appropriate size.

---

bigGP — *Package for Calculations with Big Gaussian Processes*

Description

`bigGP` is a collection of functions for doing distributed calculations in the context of various kinds of Gaussian process models, combined with a ReferenceClass, `krigeProblem`, for doing kriging calculations based on maximum likelihood estimation.

Details

Full details on doing distributed kriging can be found in the help page for `krigeProblem`. For general calculations with distributed vectors and matrices, including extending the package for additional use cases beyond standard kriging, one first needs to create the needed vectors and matrices in a distributed fashion on the slave processes. To do this, the indices associated with relevant vectors and matrices need to be found for each slave process; see `localGetVectorIndices`. Then these indices need to be used by user-created functions to create the pieces of the vectors and matrices residing on each slave process; see `localKrigeProblemConstructMean` and `localKrigeProblemConstructMean` for examples. Following this, one can use the various functions for distributed linear algebra.

The functions provided for distributed linear algebra are:

`remoteCalcChol`: calculates the Cholesky decomposition of a (numerically) positive definite matrix, $C = LL^\top$. Matrices that are not numerically positive. definite will cause failure as pivoting is not implemented.

`remoteForwardsolve`: does a forwardsolve using an already-calculated Cholesky factor into a vector or matrix, $L^{-1}Z$.

`remoteBacksolve`: does a backsolve using an already-calculated Cholesky factor into a vector or matrix, $L^{-\top}Z$.

`remoteMultChol`: multiplies and an already-calculated Cholesky factor by a vector or matrix, LZ.

remoteCrossProdMatVec: multiplies the transpose of a matrix by a vector, $X^\top z$.

remoteCrossProdMatSelf: does the crossproduct of a matrix, $X^\top X$.

remoteCrossProdMatSelfDiag: finds the diagonal of the crossproduct of a matrix, diag$(X^\top X)$.

remoteConstructRnormVector: generates a vector of random standard normal variables.

remoteConstructRnormMatrix: generates a matrix of random standard normal variables.

remoteCalc: does arbitrary calculations on one or two inputs.

Warnings

Note that the block replication factor, h, needs to be consistent in any given calculation. So if one is doing a forwardsolve, the replication factor used in distributing the original matrix (and therefore its Cholesky factor) should be the same as that used in distributing the vector being solved into (or the rows of the matrix being solved into).

Also note that when carrying out time-intensive calculations on the slave processes, the slaves will not be responsive to additional interaction, so commands such as `remoteLs` may appear to hang. This may occur because the slave process needs to finish a previous calculation before responding.

Note that distributed vectors and distributed one-column matrices are stored differently, with matrices stored with padded columns. When using `remoteForwardSolve`, `remoteBacksolve`, `remoteMultChol`, you should use `n2 = NULL` when the second argument is a vector and `n2 = 1` when the second column is a one-column matrix.

Note that triangular and symmetric matrices are stored as vectors, column-major order, of the lower triangular elements. To collect a distributed symmetric matrix on the master process, one uses `collectTriangularMatrix`. `collectTriangularMatrix` always fills the upper triangle as the transpose of the lower triangle.

Author(s)

Christopher Paciorek and Benjamin Lipshitz, in collaboration with Tina Zhuo, Cari Kaufman, Rollin Thomas, and Prabhat.

Maintainer: Christopher Paciorek <paciorek@alumni.cmu.edu>

References

Paciorek, C.J., B. Lipshitz, W. Zhuo, Prabhat, C.G. Kaufman, and R.C. Thomas. 2015. Parallelizing Gaussian Process Calculations in R. Journal of Statistical Software, 63(10), 1-23. www.jstatsoft.org/v63/i10.

Paciorek, C.J., B. Lipshitz, W. Zhuo, Prabhat, C.G. Kaufman, and R.C. Thomas. 2013. Parallelizing Gaussian Process Calculations in R. arXiv:1305.4886. http://arxiv.org/abs/1305.4886.

See Also

See `bigGP.init` for the necessary initialization steps, and `krigeProblem` for doing kriging based on maximum likelihood estimation.

Examples

```
# this is an example of using the API to do distributed linear algebra
#    for Gaussian process calculations; we'll demonstrate generating from
#    a Gaussian process with exponential covariance; note that this can
#    be done more easily through the krigeProblem ReferenceClass
## Not run:
bigGP.init(3)
params <- c(sigma2 = 1, rho = 0.25)
# for this example, we'll use a modest size problem, but to demo on a
#    cluster, increase m to a larger value
m <- 80
gd <- seq(0, 1, length = m)
locns = expand.grid(x = gd, y = gd)
# indices will be a two column matrix with the index of the first set of
#  locations in the first column and of the second set in the second column
covfunc <- function(params, locns, indices) {
dd <- sqrt( (locns$x[indices[,1]] - locns$x[indices[,2]])^2 +
(locns$y[indices[,1]] - locns$y[indices[,2]])^2 )
return(params["sigma2"] * exp(-dd / params["rho"]))
}
mpi.bcast.Robj2slave(params)
mpi.bcast.Robj2slave(covfunc)
mpi.bcast.Robj2slave(locns)
mpi.bcast.cmd(indices <- localGetTriangularMatrixIndices(nrow(locns)))
mpi.bcast.cmd(C <- covfunc(params, locns, indices))
remoteLs() # this may pause before reporting, as slaves are busy doing
#    computations above
remoteCalcChol('C', 'L', n = m^2)
remoteConstructRnormVector('z', n = m^2)
remoteMultChol('L', 'z', 'x', n1 = m^2)
x <- collectVector('x', n = m^2)
image(gd, gd, matrix(x, m))

## End(Not run)
```

`bigGP-meta` *Information about the number and identities of the processes*

Description

The `.bigGP` object (an environment) contains information about the processes involved in the distributed computation. `.bigGP.fill` is an internal auxiliary function that fills the `.bigGP` object with the values of P, D, I, and J.

Usage

```
.bigGP
.bigGP.fill(init = FALSE)
```

Arguments

`init`	logical, indicating whether to initialize values to their defaults for before the processes are set up

---

`bigGP.exit` *Exit bigGP Environment*

Description

`bigGP.exit` terminates the package's execution environment and detaches the package. After that, you can still work in R.

`bigGP.quit` terminates the package's execution environment and quits R.

Usage

```
bigGP.exit()
bigGP.quit(save = "no")
```

Arguments

`save`	the same argument as `quit`, but defaulting to "no".

Details

These functions should be used to safely leave the **"bigGP"** execution context, specifically MPI, when R is started via MPI such as by calling mpirun or analogous executables. They close the slave processes and then invoke either `mpi.exit` or `mpi.quit`.

If leaving R altogether, one simply uses `bigGP.quit`.

See Also

`mpi.exit` `mpi.quit`

`bigGP.init` *Initialize bigGP package*

Description

`bigGP.init` initializes the bigGP and must be called before using any bigGP functionality. It starts slave processes, if not already started, and sets up the necessary objects containing information for distributing calculations correctly. It also initializes the RNG on the slave processes.

Usage

```
bigGP.init(P = NULL, parallelRNGpkg = "rlecuyer", seed = 0)
```

Arguments

`P`	Number of slave processes. Should be equal to D(D+1)/2 for some integer D. If `NULL`, will be taken to be `mpi.comm.size()-1`, where the additional process is the master.
`parallelRNGpkg`	Package to be used for random number generation (RNG). At the moment this should be one of **relecuyer** or **rsprng**, and these packages must be installed.
`seed`	Seed to be used for initializing the parallel RNG.

Details

The initialization includes starting the slave processes, calculating the partition factor, D, and providing the slave processes with unique identifying information. This information is stored in the `.bigGP` object on each slave process.

Note that in general, the number of processes (number of slave processes, `P`, plus one for the master) should not exceed the number of physical cores on the machine(s) available.

`bigGP.init` also sets up random number generation on the slaves, using `parallelRNGpkg` when specified, and setting appropriate seeds on each slave process.

Examples

```
## Not run:
bigGP.init(3, seed = 1)

## End(Not run)
```

---

calcD *Calculate Partition Factor*

Description

`calcD` is an internal auxiliary function that calculates the partition factor, D, based on the number of slave processes, P.

Usage

```
calcD(P)
```

Arguments

`P`	a positive integer, the number of slave processes.

calcIJ *Calculate Slave Process Identifiers*

Description

calcIJ is an internal auxiliary function that calculates a unique pair of identifiers for each slave process, corresponding to the row and column of the block assigned to the slave process (things are more complicated when the block replication factor, h, is greater than one).

Usage

```
calcIJ(D)
```

Arguments

D	a positive integer, the partition factor.

collectDiagonal *Return the Diagonal of a Distributed Square Matrix to the Master Process*

Description

collectDiagonal retrieves the diagonal elements of a distributed square matrix from the slave processes in the proper order. Values can be copied from objects in environments, lists, and ReferenceClass objects as well as the global environment on the slave processes.

Usage

```
collectDiagonal(objName, objPos = '.GlobalEnv', n, h = 1)
```

Arguments

objName	an object name, given as a character string, giving the name of the matrix on the slave processes.
objPos	where to look for the matrix, given as a character string (unlike get). This can indicate an environment, a list, or a ReferenceClass object.
n	a positive integer, the number of rows (and columns) of the matrix.
h	a positive integer, the block replication factor, h, relevant for the matrix.

Value

collectDiagonal returns a vector of length n.

See Also

pull collectVector collectTriangularMatrix collectRectangularMatrix distributeVector

Examples

```
## Not run:
if(require(fields)) {
nProc <- 3
n <- nrow(SN2011fe_subset)
inputs <- c(as.list(SN2011fe_subset), as.list(SN2011fe_newdata_subset),
  nu =2)
# initialize the problem
prob <- krigeProblem$new("prob", h_n = 1, numProcesses = nProc, n = n,
  meanFunction = SN2011fe_meanfunc, covFunction = SN2011fe_covfunc,
  inputs = inputs, params = SN2011fe_mle$par,
  data = SN2011fe_subset$flux, packages = c("fields"))
# calculate log density, primarily so Cholesky gets calculated
prob$calcLogDens()
diagC <- collectDiagonal('C', "prob", n = n, h = 1)
diagL <- collectDiagonal('L', "prob", n = n, h = 1)
diagC[1:5]
diagL[1:5]
}

## End(Not run)
```

---

collectRectangularMatrix

Return a Distributed Rectangular Matrix to the Master Process

---

Description

collectRectangularMatrix retrieves a distributed rectangular matrix from the slave processes, reconstructing the blocks correctly on the master process. Objects can be copied from environments, lists, and ReferenceClass objects as well as the global environment on the slave processes. WARNING: do not use with a distributed symmetric square matrix; instead use collectTriangularMatrix.

Usage

```
collectRectangularMatrix(objName, objPos = '.GlobalEnv', n1, n2, h1 = 1, h2 = 1)
```

Arguments

objName	an object name, given as a character string, giving the name of the object on the slave processes.
objPos	where to look for the object, given as a character string (unlike get). This can indicate an environment, a list, or a ReferenceClass object.
n1	a positive integer, the number of rows of the matrix.

n2	a positive integer, the number of columns of the matrix.
h1	a positive integer, the block replication factor relevant for the rows of the matrix.
h2	a positive integer, the block replication factor relevant for the columns of the matrix.

Value

collectRectangularMatrix returns a matrix of dimension, $n1 \times n2$.

See Also

pull collectVector collectTriangularMatrix collectDiagonal distributeVector

Examples

```
## Not run:
if(require(fields)) {
nProc <- 3
n <- nrow(SN2011fe_subset)
m <- nrow(SN2011fe_newdata_subset)
inputs <- c(as.list(SN2011fe_subset), as.list(SN2011fe_newdata_subset),
  nu =2)
# initialize the problem
prob <- krigeProblem$new("prob", h_n = 1, h_m = 1, numProcesses =
  nProc, n = n, m = m,
  meanFunction = SN2011fe_meanfunc, predMeanFunction = SN2011fe_predmeanfunc,
  covFunction = SN2011fe_covfunc, crossCovFunction = SN2011fe_crosscovfunc,
predCovFunction = SN2011fe_predcovfunc, params = SN2011fe_mle$par,
  inputs = inputs, data = SN2011fe_subset$flux, packages = c("fields"))
# do predictions, primarily so cross-covariance gets calculated
pred <- prob$predict(ret = TRUE, verbose = TRUE)

crossC <- collectRectangularMatrix('crossC', "prob", n1 = n, n2 = m,
h1 = 1, h2 = 1)
crossC[1:5, 1:5]
}

## End(Not run)
```

---

collectTriangularMatrix

Return a Distributed Symmetric or Triangular Matrix to the Master Process

---

Description

collectTriangularMatrix retrieves a distributed symmetric or triangular matrix from the slave processes, reconstructing the blocks correctly on the master process. Objects can be copied from environments, lists, and ReferenceClass objects as well as the global environment on the slave processes.

Usage

```
collectTriangularMatrix(objName, objPos = '.GlobalEnv', n, h = 1)
```

Arguments

objName	an object name, given as a character string, giving the name of the object on the slave processes.
objPos	where to look for the object, given as a character string (unlike get). This can indicate an environment, a list, or a ReferenceClass object.
n	a positive integer, the number of rows (and columns) of the matrix.
h	a positive integer, the block replication factor, h, relevant for the matrix.

Value

collectTriangularMatrix returns a matrix of dimension, $n \times n$. Note that for lower triangular matrices, the upper triangle is non-zero and is filled with the transpose of the lower triangle, and vice versa for upper triangular matrices.

See Also

pull collectVector collectRectangularMatrix collectDiagonal distributeVector

Examples

```
## Not run:
if(require(fields)) {
nProc <- 3
n <- nrow(SN2011fe_subset)
inputs <- c(as.list(SN2011fe_subset), as.list(SN2011fe_newdata_subset),
  nu =2)
# initialize the problem
prob <- krigeProblem$new("prob", h_n = 1, numProcesses = nProc, n = n,
  meanFunction = SN2011fe_meanfunc, covFunction = SN2011fe_covfunc, inputs = inputs,
  params = SN2011fe_mle$par, data = SN2011fe_subset$flux, packages =
  c("fields"))
# calculate log density, primarily so Cholesky gets calculated
prob$calcLogDens()
C <- collectTriangularMatrix('C', "prob", n = n, h = 1)
L <- collectTriangularMatrix('L', "prob", n = n, h = 1)
C[1:5, 1:5]
L[1:5, 1:5]
}

## End(Not run)
```

collectVector — *Return a Distributed Vector to the Master Process*

Description

`collectVector` retrieves a distributed vector from the slave processes, reconstructing in the correct order on the master process. Objects can be copied from environments, lists, and ReferenceClass objects as well as the global environment on the slave processes.

Usage

```
collectVector(objName, objPos = '.GlobalEnv', n, h = 1)
```

Arguments

`objName`	an object name, given as a character string, giving the name of the object on the slave processes.
`objPos`	where to look for the object, given as a character string (unlike `get`). This can indicate an environment, a list, or a ReferenceClass object.
`n`	a positive integer, the length of the vector.
`h`	a positive integer, the block replication factor, h, relevant for the vector.

Value

`collectVector` returns a vector of length, n.

See Also

`pull` `collectTriangularMatrix` `collectRectangularMatrix` `collectDiagonal` `distributeVector`

Examples

```
## Not run:
bigGP.init(3)
n <- 3000
x <- rnorm(n)
distributeVector(x, 'tmp', n = n)
y <- collectVector('tmp', n = n)
identical(x, y)

## End(Not run)
```

`distributedKrigeProblem-class` *ReferenceClass for Distributed Components of the krigeProblem ReferenceClass*

Description

`distributedKrigeProblem` contains the distributed components of the core vectors and matrices of the krigeProblem class, as well as copies of the functions for calculating mean vectors and covariance matrices, parameter values, and information about the pieces of the distributed objects contained on a given slave process. The only method associated with the ReferenceClass is a constructor.

See Also

`krigeProblem`

`distributeVector` *Distribute a Vector to the Slave Processes*

Description

`distributeVector` distributes a vector to the slave processes, breaking into the appropriate pieces, in some cases with padded elements. Objects can be distributed to environments and ReferenceClass objects as well as the global environment on the slave processes.

Usage

```
distributeVector(obj, objName = deparse(substitute(obj)), objPos = '.GlobalEnv', n, h = 1)
```

Arguments

`obj`	object on master process to be copied, given either as the name of an object or as a character.
`objName`	an object name, given as a character string, giving the name to be used for the object on the slave processes. If not provided, will be the same as the name of `obj` in the calling environment.
`objPos`	where to do the assignment, given as a character string (unlike `assign`). This can indicate an environment or a ReferenceClass object.
`n`	a positive integer, the length of the vector.
`h`	a positive integer, the block replication factor, h, to be used when distributing the vector.

See Also

push collectVector collectTriangularMatrix collectRectangularMatrix collectDiagonal

Examples

```
## Not run:
bigGP.init(3)
n <- 3000
x <- rnorm(n)
distributeVector(x, 'tmp', n = n)
y <- collectVector('tmp', n = n)
identical(x, y)

## End(Not run)
```

---

getDistributedVectorLength

Find Length of Subset of Vector or Matrix Stored on Slave Process

---

Description

getDistributedVectorLength, getDistributedTriangularMatrixLength, and getDistributedRectangularLength are internal auxiliary functions that find the length of the vector needed to store the subset of a vector or matrix contained on a given slave process.

Usage

```
getDistributedVectorLength(n, h = 1)
getDistributedTriangularMatrixLength(n, h = 1)
getDistributedRectangularMatrixLength(n1, n2, h1 = 1, h2 = 1)
```

Arguments

n	length of vector.
h	replication factor.
n1	number of rows.
n2	number of columns.
h1	replication factor for the rows.
h2	replication factor for the columns.

krigeProblem-class *Class* "krigeProblem"

Description

The `krigeProblem` class provides functionality for kriging using distributed calculations, based on maximum likelihood estimation. The class includes methods for standard kriging calculations and metadata necessary for carrying out the methods in a distributed fashion.

To carry out kriging calculations, one must first initialize an object of the `krigeProblem` class. This is done using `krigeProblem$new` and help on initialization can be obtained via `krigeProblem$help('initialize')` (but noting that the call is `krigeProblem$new` not `krigeProblem$initialize`).

Note that in what follows I refer to observation and prediction 'locations'. This is natural for spatial problems, but for non-spatial problems, 'locations' is meant to refer to the points within the relevant domain at which observations are available and predictions wish to be made.

The user must provide functions that create the subsets of the mean vector(s) and the covariance matrix/matrices. Functions for the mean vector and covariance matrix for observation locations are required, while those for the mean vector for prediction locations, the cross-covariance matrix (where the first column is the index of the observation locations and the second of the prediction locations), and the prediction covariance matrix for prediction locations are required when doing prediction and posterior simulation. These functions should follow the form of `SN2011fe_meanfunc`, `SN2011fe_predmeanfunc`, `SN2011fe_covfunc`, `SN2011fe_predcovfunc`, and `SN2011fe_crosscovfunc`. Namely, they should take three arguments, the first a vector of all the parameters for the Gaussian process (both mean and covariance), the second an arbitrary list of inputs (in general this would include the observation and prediction locations), and the third being indices, which will be provided by the package and will differ between slave processes. For the mean functions, the indices will be a vector, indicating which of the vector elements are stored on a given process. For the covariance functions, the indices will be a two column matrix, with each row a pair of indices (row, column), indicating the elements of the matrix stored on a given process. Thus, the user-provided functions should use the second and third arguments to construct the elements of the vectors/matrices belonging on the slave process. Note that the elements of the matrices are stored as vectors (vectorizing matrices column-wise, as natural for column-major matrices). Users can simply have their functions operate on the rows of the index matrix without worrying about ordering. An optional fourth argument contains cached values that need not be computed at every call to the user-provided function. If the user wants to make use of caching of values to avoid expensive recomputation, the user function should mimic `SN2011fe_covfunc`. That is, when the user wishes to change the cached values (including on first use of the function), the function should return a two-element list, with the first element being the covariance matrix elements and the second containing whatever object is to be cached. This cached object will be provided to the function on subsequent calls as the fourth argument.

Note that one should have all necessary packages required for calculation of the mean vector(s) and covariance matrix/matrices installed on all machines used and the names of these packages should be passed as the `packages` argument to the `krigeProblem` initialization.

Help for the various methods of the class can be obtained with `krigeProblem$help('methodName')` and a list of fields and methods in the class with `krigeProblem$help()`.

In general, n (or n1 and n2) refer to the length or number of rows/columns of vectors and matrices and h (or h1 and h2) to the block replication factor for these vectors and matrices. More details on block replication factors can be found in the references in 'references'; these are set at reasonable values automatically, and for simplicity, one can set them at one, in which case the number of blocks into which the primary covariance matrix is split is P, the number of slave processes. Cross-covariance matrices returned to the user will have number of rows equal to the number of observation locations and number of columns to the number of prediction locations. Matrices of realizations will have each realized field as a single column.

Fields

localProblemName: Object of class "character" containing the name to be used for the object on the slave processes.

n: Object of class "numeric" containing the number of observation locations.

h_n: Object of class "numeric" containing the block replication factor for the observation locations, will be set to a reasonable value by default upon initialization of an object in the class.

h_m: Object of class "numeric" containing the block replication factor for the prediction locations, will be set to a reasonable value by default upon initialization of an object in the class.

meanFunction: Object of class "function" containing the function used to calculate values of the mean function at the observation locations. See above for detailed information on how this function should be written.

predMeanFunction: Object of class "function" containing the function used to calculate values of the mean function at the prediction locations. See above for detailed information on how this function should be written.

covFunction: Object of class "function" containing the function used to calculate values of the covariance function for pairs of observation locations. See above for detailed information on how this function should be written.

crossCovFunction: Object of class "function" containing the function used to calculate values of the covariance function for pairs of observation and prediction locations. See above for detailed information on how this function should be written.

predCovFunction: Object of class "function" containing the function used to calculate values of the covariance function for pairs of prediction locations. See above for detailed information on how this function should be written.

data: Object of class "ANY" containing the vector of data values at the observation locations. This will be numeric, but is specified as of class "ANY" so that can default to NULL.

params: Object of class "ANY" containing the vector of parameter values. This will be numeric, but is specified as of class "ANY" so that can default to NULL. This vector is what will be passed to the mean and covariance functions.

meanCurrent: Object of class "logical" indicating whether the current distributed mean vector (for the observation locations) on the slaves is current (i.e., whether it is based on the current value of params).

predMeanCurrent: Object of class "logical" indicating whether the current distributed mean vector (for the prediction locations) on the slaves is current (i.e., whether it is based on the current value of params).

postMeanCurrent: Object of class "logical" indicating whether the current distributed posterior mean vector (for the prediction locations) on the slaves is current (i.e., whether it is based on the current value of params).

covCurrent: Object of class "logical" indicating whether the current distributed covariance matrix (for the observation locations) on the slaves is current (i.e., whether it is based on the current value of params).

crossCovCurrent: Object of class "logical" indicating whether the current distributed cross-covariance matrix (between observation and prediction locations) on the slaves is current (i.e., whether it is based on the current value of params).

predCovCurrent: Object of class "logical" indicating whether the current distributed prediction covariance matrix on the slaves is current (i.e., whether it is based on the current value of params).

postCovCurrent: Object of class "logical" indicating whether the current distributed posterior covariance matrix on the slaves is current (i.e., whether it is based on the current value of params).

cholCurrent: Object of class "logical" indicating whether the current distributed Cholesky factor of the covariance matrix (for observation locations) on the slaves is current (i.e., whether it is based on the current value of params).

predCholCurrent: Object of class "logical" indicating whether the current distributed Cholesky factor of the covariance matrix (the prior covariance matrix for prediction locations) on the slaves is current (i.e., whether it is based on the current value of params). Note this is likely only relevant when generating realizations for prediction locations not conditional on the observations.

postCholCurrent: Object of class "logical" indicating whether the current distributed Cholesky factor of the posterior covariance matrix on the slaves is current (i.e., whether it is based on the current value of params).

Methods

new(localProblemName = NULL, numProcesses = NULL, h_n = NULL, h_m = NULL, n = length(data), m = NUL
Initializes new krigeProblem object, which is necessary for distributed kriging calculations.

calcH(n): Internal method that calculates a good choice of the block replication factor given n.

show(verbose = TRUE): Show (i.e., print) method.

initializeSlaveProblems(packages): Internal method that sets up the slave processes to carry out the krigeProblem distributed calculations.

setParams(params, verbose = TRUE): Sets (or changes) the value of the parameters.

remoteConstructMean(obs = TRUE, pred = !obs, verbose = FALSE): Meant for internal use; calculates the value of the specified mean vector (for observation and/or prediction locations) on the slave processes, using the appropriate user-provided function.

remoteConstructCov(obs = TRUE, pred = FALSE, cross = FALSE, verbose = FALSE): Meant for internal use; calculates the value of the specified covariance matrices on the slave processes, using the appropriate user-provided function.

calcLogDeterminant(): Calculates the log-determinant of the covariance matrix for the observation locations.

`calcLogDens(newParams = NULL, newData = NULL, negative = FALSE, verbose = TRUE)`: Calculates the log-density of the data given the parameters.

`optimizeLogDens(newParams = NULL, newData = NULL, method = "Nelder-Mead", verbose = FALSE, gr = NUL` Finds the maximum likelihood estimate of the parameters given the data, using `optim`.

`predict(ret = FALSE, verbose = FALSE)`: Calculates kriging predictions (i.e., the posterior mean for the prediction locations).

`calcPostCov(returnDiag = TRUE, verbose = FALSE)`: Calculates the prediction covariance (i.e., the posterior covariance matrix for the prediction locations), returning the diagonal (the variances) if requested.

`simulateRealizations(r = 1, h_r = NULL, obs = FALSE, pred = FALSE, post = TRUE, verbose = FALSE)`: Simulates realizations, which would generally be from the posterior distribution (i.e., conditional on the data), but could also be from the prior distribution (i.e., not conditional on the data) at either observation or predition locations.

Extends

All reference classes extend and inherit methods from "`envRefClass`".

Author(s)

Christopher Paciorek and Benjamin Lipshitz, in collaboration with Tina Zhuo, Cari Kaufman, Rollin Thomas, and Prabhat.

Maintainer: Christopher Paciorek <paciorek@alumni.cmu.edu>

References

Paciorek, C.J., B. Lipshitz, W. Zhuo, Prabhat, C.G. Kaufman, and R.C. Thomas. 2015. Parallelizing Gaussian Process Calculations in R. Journal of Statistical Software, 63(10), 1-23. http://www.jstatsoft.org/v63/i10.

Paciorek, C.J., B. Lipshitz, W. Zhuo, Prabhat, C.G. Kaufman, and R.C. Thomas. 2013. Parallelizing Gaussian Process Calculations in R. arXiv:1305.4886. http://arxiv.org/abs/1305.4886.

See Also

See `bigGP` for general information on the package and `bigGP.init` for the necessary initialization steps required before using the package, including the `krigeProblem` class.

Examples

```
## Not run:
doSmallExample <- TRUE

if(require(fields)) {

if(doSmallExample){
  SN2011fe <- SN2011fe_subset
  SN2011fe_newdata <- SN2011fe_newdata_subset
  SN2011fe_mle <- SN2011fe_mle_subset
```

```
  nProc <- 3
} else {
# users should select number of processors based on their system and the
# size of the full example
nProc <- 210
}


n <- nrow(SN2011fe)
m <- nrow(SN2011fe_newdata)
nu <- 2
inputs <- c(as.list(SN2011fe), as.list(SN2011fe_newdata), nu = nu)

prob <- krigeProblem$new("prob", numProcesses = nProc, n = n, m = m,
 predMeanFunction = SN2011fe_predmeanfunc, crossCovFunction = SN2011fe_crosscovfunc,
predCovFunction = SN2011fe_predcovfunc, meanFunction = SN2011fe_meanfunc,
covFunction = SN2011fe_covfunc,  inputs = inputs, params = SN2011fe_mle$par,
data = SN2011fe$flux, packages = c("fields"))

prob$calcLogDens()

prob$optimizeLogDens(method = "L-BFGS-B", verbose = TRUE,
lower = rep(.Machine$double.eps, length(SN2011fe_initialParams)),
control = list(parscale = SN2011fe_initialParams, maxit = 2))
# the full optimization can take some time; only two iterations are done
# are specified here; even this is not run as it takes 10s of seconds

prob$setParams(SN2011fe_mle$par)

pred <- prob$predict(ret = TRUE, se.fit = TRUE, verbose = TRUE)
realiz <- prob$simulateRealizations(r = 10, post = TRUE, verbose = TRUE)

show(prob)
}

## End(Not run)
```

`localAssign` *Assign a New Name to an Object on Slave Process*

Description

`localAssign` is an internal auxiliary function used to assign a new name to an object in an environment on a slave process. The function needs to be executed on the slave processes.

Usage

```
localAssign(nameToAssign, currentName, objPos = ".GlobalEnv")
```

Arguments

nameToAssign	a variable name, given as a character string, giving the new name for the object.
currentName	a variable name, given as a character string, giving the current name for the object.
objPos	where to do the assignment, given as a character string (unlike assign). This can indicate an environment or a ReferenceClass object.

Details

This function is primarily for internal use, but might be useful for developers extending the package for use cases other than the kriging use case contained in krigeProblem ReferenceClass.

Examples

```
## Not run:
bigGP.init(3)
mpi.bcast.cmd(e <- new.env())
mpi.bcast.cmd(a <- 7)
mpi.remote.exec(localAssign, "x", "a", objPos = "e")
mpi.remote.exec(e$x, ret = TRUE)

## End(Not run)
```

---

localCalc *Local Calculation Functions*

Description

These internal functions carry out the calculations of their respective remote counterpart functions, e.g., remoteCalc, on the slave process. The functions need to be executed on the slave processes.

---

localCollectVector *Local Distribution and Collection Functions*

Description

These internal functions carry out the tasks of their respective primary functions, e.g., collectVector, on the slave process. The functions need to be executed on the slave processes.

Usage

```
localCollectVector(objName, objPos, n, h)
localCollectVectorTest(objName, objPos, n, h)
localDistributeVector(objName, objPos, n, h)
localDistributeVectorTest(objName, objPos, n, h)
localPull(objName, objPos, tag = 1)
localCollectDiagonal(objName, objPos, n, h)
localCollectDiagonalTest(objName, objPos, n, h)
localCollectTriangularMatrix(objName, objPos, n, h)
localCollectTriangularMatrixTest(objName, objPos, n, h)
localCollectRectangularMatrix(objName, objPos, n1, n2, h1, h2)
localCollectRectangularMatrixTest(objName, objPos, n1, n2, h1, h2)
```

Arguments

`objName`	name of object as a character string.
`objPos`	where to look for the object, given as a character string (unlike `get`). This can indicate an environment, a list, or a ReferenceClass object.
`n`	length of vector.
`h`	replication factor.
`tag`	MPI tag.
`n1`	number of rows.
`n2`	number of columns.
`h1`	replication factor for the rows.
`h2`	replication factor for the columns.

`localGetVectorIndices` *Get Indices of Vector or Matrix Elements Stored on Slave Process*

Description

`localGetVectorIndices`, `localGetTriangularMatrixIndices`, and `localGetRectangularMatrixIndices` are internal auxiliary functions that determine the indices of the elements of a vector or matrix that are stored on a slave process. These are primarily meant for internal use, but can also be used in the process of creating distributed vectors and matrices on the slave processes. The functions need to be executed on the slave processes.

Usage

```
localGetVectorIndices(n, h = 1)
localGetTriangularMatrixIndices(n, h = 1)
localGetRectangularMatrixIndices(n1, n2, h1 = 1, h2 = 1)
```

Arguments

n	a positive integer, giving the length of the vector or number of rows and columns of the triangular/square matrix.
n1	a positive integer, giving the number of rows of the rectangular matrix.
n2	a positive integer, giving the number of columns of the rectangular matrix.
h	a positive integer, giving the block replication factor for the vector or triangular/square matrix.
h1	a positive integer, giving the block replication factor for the rows of the rectangular matrix.
h2	a positive integer, giving the block replication factor for the columns of the rectangular matrix.

Value

`localGetVectorIndices` returns the indices (as a one-column matrix) of the subset of a distributed vector that will be stored on the process on which the function is called. `localGetTriangularMatrixIndices`, and `localGetRectangularMatrixIndices` return a two-column matrix with the indices for the subset of the distributed matrix that will be stored (as a vector) on the process on which the function is called. I.e., the ith row of the matrix gives the (row, column) position in the full matrix for the ith element of the vector on the local process that contains a subset of that matrix.

Warning: in some cases there is a small amount of buffering involved in the distributed objects so that the blocks on each process are of the same size. In this case, the index of the first element will generally be added one or more times to the end of the indices assigned to the last process.

---

localKrigeProblemConstructMean

Calculate Mean Vector or Covariance Matrix on Slave Process

Description

`localKrigeProblemConstructMean` and `localKrigeProblemConstructCov` are internal wrapper functions for calculating a mean vector or covariance matrix on the slave processes. They are called by member functions of the `krigeProblem` ReferenceClass.

Usage

```
localKrigeProblemConstructMean(problemName, obs, pred)
localKrigeProblemConstructCov(problemName, obs, pred, cross)
```

Arguments

problemName	name of the problem as a character string.
obs	logical, whether to compute the mean or covariance for the observation locations.
pred	logical, whether to compute the mean or covariance for the prediction locations.
cross	logical, whether to compute the cross-covariance.

See Also

`krigeProblem`

`localRm` *Remove Objects on Slave Process*

Description

`localRm` is an internal auxiliary function used by `remoteRm` to remove objects on a slave process.

Usage

```
localRm(list)
```

Arguments

`list`	a character vector naming objects to be removed.

`pull` *Copy Object from Slave Processes to Master*

Description

Copies all objects with a given name from the slave processes to the master process, returning a list with one element per slave process. Objects can be copied from lists, environments, and ReferenceClass objects as well as the global environment on the slave processes.

Usage

```
pull(objName, objPos = ".GlobalEnv", tag = 1)
```

Arguments

`objName`	a variable name, given as a character string, giving the name of the object on the slave processes.
`objPos`	where to look for the object, given as a character string (unlike `get`). This can indicate an environment, a list, or a ReferenceClass object.
`tag`	non-negative integer, as in `mpi.send` and `mpi.recv`. Use `mpi.any.tag` for any tag flag.

Value

`pull` returns a list, with one element per slave process.

Warning

Vectors and matrices that are part of the distributed linear algebra computations are broken up in very specific ways on the slave processes and often include padded elements. In general one should not use `pull` for retrieving such objects from the slave processes. Rather, use `collectVector`, `CollectTriangularMatrix`, etc.

See Also

`push` `collectVector` `collectTriangularMatrix` `collectRectangularMatrix` `collectDiagonal`

Examples

```
## Not run:
bigGP.init(3)
a <- 3
push(a)
remoteLs()
pull('a')

## End(Not run)
```

`push` *Copy Object from Master to Slave Processes*

Description

Copies an objects from the master process to all slave processes. Objects can be copied to environments and ReferenceClass objects as well as the global environment on the slaves.

Usage

```
push(.tmp, objName = deparse(substitute(.tmp)), objPos = ".GlobalEnv")
```

Arguments

`.tmp`	object on master process to be copied, given either as the name of an object or as a character.
`objName`	the name to use for the object on the slave processes.
`objPos`	where to do the assignment, given as a character string (unlike `assign`). This can indicate an environment or a ReferenceClass object.

Warning

Vectors that are part of the distributed linear algebra computations are broken up in very specific ways on the slave processes and often include padded elements. In general one should not use `push` to distribute such objects as `push` would distribute the entire vector to each slave process. Rather, use `distributeVector`.

See Also

pull distributeVector

Examples

```
## Not run:
bigGP.init(3)
a <- 3
push(a)
remoteLs()

## End(Not run)
```

---

remoteCalc *Return a Distributed Vector to the Master Process*

Description

remoteCalc applies a function to either one or two input objects on the slave processes. Input objects can be obtained environments, lists, and ReferenceClass objects as well as the global environment on the slave processes. The output object can be assigned into a environment or a ReferenceClass objects as well as the global environment on the slave processes.

Usage

```
remoteCalc(input1Name, input2Name = NULL, FUN, outputName, input1Pos = '.GlobalEnv',
input2Pos = '.GlobalEnv', outputPos = '.GlobalEnv')
```

Arguments

input1Name	an object name, given as a character string, giving the name of the first input on the slave processes.
input2Name	an object name, given as a character string, giving the name of the first input on the slave processes. This is optional so that one can carry out a calculation on a single input.
FUN	the function to be applied, see 'details'. In the case of operators like +, the function name must be backquoted.
outputName	an object name, given as a character string, giving the name to be used for the result of the function call.
input1Pos	where to look for the first input, given as a character string (unlike get). This can indicate an environment, a list, or a ReferenceClass object.
input2Pos	where to look for the second input, given as a character string (unlike get). This can indicate an environment, a list, or a ReferenceClass object.
outputPos	where to do the assignment of the output, given as a character string (unlike assign). This can indicate an environment or a ReferenceClass object.

Details

FUN is found by a call to match.fun and typically is either a function or a symbol (e.g., a backquoted name) or a character string specifying a function to be searched for from the environment of the call to remoteCalc.

Examples

```
## Not run:
bigGP.init(3)
mpi.bcast.cmd(x <- 0:mpi.comm.rank())
remoteCalc('x', FUN = exp, outputName = 'exp.x')
remoteLs()
pull('exp.x')
remoteCalc('x', 'exp.x', FUN = `+`, outputName = 'silly')
pull('silly')

## End(Not run)
```

---

remoteCalcChol *Calculate Distributed Cholesky Decomposition*

Description

remoteCalcChol calculates a distributed Cholesky decomposition from a distributed positive definite matrix. The Cholesky factor and the original matrix can both be contained within environments and ReferenceClass objects as well as the global environment on the slave processes.

Usage

```
remoteCalcChol(matName, cholName, matPos = '.GlobalEnv', cholPos = '.GlobalEnv', n, h = 1)
```

Arguments

matName	name of the input (positive definite) matrix, given as a character string, giving the name of the object on the slave processes.
cholName	an name, given as a character string, giving the name to be used for the Cholesky factor matrix on the slave processes.
matPos	where to look for the input matrix, given as a character string (unlike get). This can indicate an environment, a list, or a ReferenceClass object.
cholPos	where to do the assignment of the Cholesky factor matrix, given as a character string (unlike assign). This can indicate an environment or a ReferenceClass object.
n	a positive integer, the number of rows and columns of the input matrix.
h	a positive integer, the block replication factor, h, relevant for the input matrix and used for the Cholesky factor as well.

Details

Computes the distributed Cholesky decomposition using a blocked algorithm similar to that in ScaLapack. When h is 1, the number of blocks, representing the lower triangle of the original matrix and of the Cholesky factor, is equal to the number of processes. For larger values of h, there are multiple blocks assigned to each process.

References

Paciorek, C.J., B. Lipshitz, W. Zhuo, Prabhat, C.G. Kaufman, and R.C. Thomas. 2015. Parallelizing Gaussian Process Calculations in R. Journal of Statistical Software, 63(10), 1-23. www.jstatsoft.org/v63/i10.

Paciorek, C.J., B. Lipshitz, W. Zhuo, Prabhat, C.G. Kaufman, and R.C. Thomas. 2013. Parallelizing Gaussian Process Calculations in R. arXiv:1305.4886. http://arxiv.org/abs/1305.4886.

See Also

bigGP

Examples

```
## Not run:
if(require(fields)) {
  SN2011fe <- SN2011fe_subset
  SN2011fe_newdata <- SN2011fe_newdata_subset
  SN2011fe_mle <- SN2011fe_mle_subset
  nProc <- 3
n <- nrow(SN2011fe)
m <- nrow(SN2011fe_newdata)
nu <- 2
inputs <- c(as.list(SN2011fe), as.list(SN2011fe_newdata), nu = nu)
prob <- krigeProblem$new("prob", numProcesses = nProc, n = n, m = m,
predMeanFunction = SN2011fe_predmeanfunc, crossCovFunction = SN2011fe_crosscovfunc,
predCovFunction = SN2011fe_predcovfunc, meanFunction = SN2011fe_meanfunc,
 covFunction = SN2011fe_covfunc,  inputs = inputs, params = SN2011fe_mle$par,
data = SN2011fe$flux, packages = c("fields"))
remoteCalcChol(matName = 'C', cholName = 'L', matPos = 'prob',
  cholPos = 'prob', n = n, h = prob$h_n)
L <- collectTriangularMatrix('L', objPos = 'prob', n = n, h = prob$h_n)
}

## End(Not run)
```

---

remoteConstructRnormVector *Create Distributed Vector or Matrix of Random Normals*

Description

`remoteConstructRnormVector` constructs a distributed vector of standard normal random variables, while `remoteConstructRnormMatrix` constructs a distributed matrix. The output object can both be contained within environments or ReferenceClass objects as well as the global environment on the slave processes.

Usage

```
remoteConstructRnormVector(objName, objPos = ".GlobalEnv", n, h = 1)
 remoteConstructRnormMatrix(objName, objPos = ".GlobalEnv", n1, n2, h1 = 1, h2 = 1)
```

Arguments

objName	the name to use for the vector or matrix, on the slave processes.
objPos	where to do the assignment of the output matrix or vector, given as a character string (unlike `assign`). This can indicate an environment or a ReferenceClass object.
n	a positive integer, the length of the vector
h	a positive integer, the block replication factor, h, relevant for the vector
n1	a positive integer, the number of rows of the matrix.
n2	a positive integer, the number of columns of the matrix.
h1	a positive integer, the block replication factor, h, relevant for the rows of the matrix.
h2	a positive integer, the block replication factor, h, relevant for the columns of the matrix.

Warning

Note that a vector and a one-column matrix are stored differently, with padded columns included for the matrix. For other distributed computation functions, providing the argument `n2 = NULL` indicates the input is a vector, while `n2 = 1` indicates a one-column matrix.

See Also

bigGP

Examples

```
## Not run:
if(require(fields)) {
  SN2011fe <- SN2011fe_subset
  SN2011fe_newdata <- SN2011fe_newdata_subset
  SN2011fe_mle <- SN2011fe_mle_subset
  nProc <- 3
n <- nrow(SN2011fe)
m <- nrow(SN2011fe_newdata)
nu <- 2
inputs <- c(as.list(SN2011fe), as.list(SN2011fe_newdata), nu = nu)
```

```
prob <- krigeProblem$new("prob", numProcesses = nProc, n = n, m = m,
predMeanFunction = SN2011fe_predmeanfunc, crossCovFunction = SN2011fe_crosscovfunc,
predCovFunction = SN2011fe_predcovfunc, meanFunction = SN2011fe_meanfunc,
covFunction = SN2011fe_covfunc,  inputs = inputs, params = SN2011fe_mle$par,
data = SN2011fe$flux, packages = c("fields"))
remoteCalcChol(matName = 'C', cholName = 'L', matPos = 'prob',
  cholPos = 'prob', n = n, h = prob$h_n)
remoteConstructRnormVector('z', n = n, h = prob$h_n)
remoteMultChol(cholName = 'L', inputName = 'z', outputName = 'result',
cholPos = 'prob', n1 = n, h1 = prob$h_n)
realiz <- collectVector('result', n = n, h = prob$h_n)

r = 10
remoteConstructRnormMatrix('z2', n1 = n, n2 = r, h1 = prob$h_n, h2 = 1)
remoteMultChol(cholName = 'L', inputName = 'z2', outputName = 'result2',
cholPos = 'prob', n1 = n, n2 = r, h1 = prob$h_n, h2 = 1)
realiz2 <- collectRectangularMatrix('result2', n1 = prob$n, n2 = r, h1
= prob$h_n, h2 = 1)
}

## End(Not run)
```

`remoteCrossProdMatSelf` *Distributed Crossproduct of a Rectangular Matrix with Itself*

Description

`remoteCrossProdMatSelf` multiplies the transpose of a distributed rectangular matrix by itself. `remoteCrossProdMatSelf` calculates only the diagonal of the crossproduct. The objects can both be contained within environments or ReferenceClass objects as well as the global environment on the slave processes.

Usage

```
remoteCrossProdMatSelf(inputName, outputName, inputPos = '.GlobalEnv',
outputPos = '.GlobalEnv', n1, n2, h1 = 1, h2 = 1)
remoteCrossProdMatSelfDiag(inputName, outputName, inputPos =
'.GlobalEnv', outputPos = '.GlobalEnv', n1, n2, h1 = 1, h2 = 1)
```

Arguments

`inputName`	name of the matrix, given as a character string, giving the name of the object on the slave processes.
`outputName`	the name to use for resulting matrix, on the slave processes.
`inputPos`	where to look for the matrix, given as a character string (unlike `get`). This can indicate an environment, a list, or a ReferenceClass object.

outputPos	where to do the assignment of the output matrix, given as a character string (unlike assign). This can indicate an environment or a ReferenceClass object.
n1	a positive integer, the number of rows of the matrix.
n2	a positive integer, the number of columns of the matrix.
h1	a positive integer, the block replication factor, h, relevant for the rows of the matrix.
h2	a positive integer, the block replication factor, h, relevant for the columns of the matrix.

Details

Computes the distributed product, $X^T X$ using a blocked algorithm, resulting in a distributed matrix.

References

Paciorek, C.J., B. Lipshitz, W. Zhuo, Prabhat, C.G. Kaufman, and R.C. Thomas. 2015. Parallelizing Gaussian Process Calculations in R. Journal of Statistical Software, 63(10), 1-23. www.jstatsoft.org/v63/i10.

Paciorek, C.J., B. Lipshitz, W. Zhuo, Prabhat, C.G. Kaufman, and R.C. Thomas. 2013. Parallelizing Gaussian Process Calculations in R. arXiv:1305.4886. http://arxiv.org/abs/1305.4886.

See Also

bigGP

Examples

```
## Not run:
if(require(fields)) {
  SN2011fe <- SN2011fe_subset
  SN2011fe_newdata <- SN2011fe_newdata_subset
  SN2011fe_mle <- SN2011fe_mle_subset
  nProc <- 3
n <- nrow(SN2011fe)
m <- nrow(SN2011fe_newdata)
nu <- 2
inputs <- c(as.list(SN2011fe), as.list(SN2011fe_newdata), nu = nu)
prob <- krigeProblem$new("prob", numProcesses = nProc, n = n, m = m,
predMeanFunction = SN2011fe_predmeanfunc, crossCovFunction =
SN2011fe_crosscovfunc,  predCovFunction = SN2011fe_predcovfunc,
meanFunction = SN2011fe_meanfunc, covFunction = SN2011fe_covfunc,
inputs = inputs, params = SN2011fe_mle$par, data = SN2011fe$flux,
packages = c("fields"))

remoteCalcChol(matName = "C", cholName = "L", matPos = "prob",
  cholPos = "prob", n = n, h = prob$h_n)
prob$remoteConstructCov(obs = FALSE, pred = FALSE, cross = TRUE, verbose = TRUE)
# we now have a rectangular cross-covariance matrix named 'crossC'
remoteForwardsolve(cholName = "L", inputName = "crossC", outputName = "tmp1",
```

```
cholPos = "prob", inputPos = "prob", n1 = n, n2 = m, h1 = prob$h_n, h2 = prob$h_m)

remoteCrossProdMatSelf(inputName = "tmp1", outputName = "result", n1 = n,
n2 = m, h1 = prob$h_n, h2 = prob$h_m)
result <- collectTriangularMatrix("result", n = m, h = prob$h_m)

remoteCrossProdMatSelfDiag(inputName = "tmp1", outputName = "resultDiag",
n1 = n, n2 = m, h1 = prob$h_n, h2 = prob$h_m)
resultDiag <- collectVector("resultDiag", n = m, h = prob$h_m)
}

## End(Not run)
```

---

remoteCrossProdMatVec *Distributed Crossproduct of a Rectangular Matrix and a Vector*

Description

`remoteCrossProdMatVec` multiplies the transpose of a distributed rectangular matrix by a distributed vector or matrix. The objects can both be contained within environments or ReferenceClass objects as well as the global environment on the slave processes.

Usage

```
remoteCrossProdMatVec(matName, inputName, outputName, matPos = '.GlobalEnv',
inputPos = '.GlobalEnv', outputPos = '.GlobalEnv', n1, n2, h1 = 1, h2 = 1)
```

Arguments

`matName`	name of the rectangular matrix, given as a character string, giving the name of the object on the slave processes.
`inputName`	name of the vector being multiplied by, given as a character string, giving the name of the object on the slave processes.
`outputName`	the name to use for resulting vector, on the slave processes.
`matPos`	where to look for the rectangular matrix, given as a character string (unlike `get`). This can indicate an environment, a list, or a ReferenceClass object.
`inputPos`	where to look for the input vector, given as a character string (unlike `get`). This can indicate an environment, a list, or a ReferenceClass object.
`outputPos`	where to do the assignment of the output vector, given as a character string (unlike `assign`). This can indicate an environment or a ReferenceClass object.
`n1`	a positive integer, the number of rows of the matrix.
`n2`	a positive integer, the number of columns of the matrix.
`h1`	a positive integer, the block replication factor, h, relevant for the rows of the matrix.
`h2`	a positive integer, the block replication factor, h, relevant for the columns of the matrix.

Details

Computes the distributed product using a blocked algorithm, resulting in a distributed vector.

References

Paciorek, C.J., B. Lipshitz, W. Zhuo, Prabhat, C.G. Kaufman, and R.C. Thomas. 2015. Parallelizing Gaussian Process Calculations in R. Journal of Statistical Software, 63(10), 1-23. www.jstatsoft.org/v63/i10.

Paciorek, C.J., B. Lipshitz, W. Zhuo, Prabhat, C.G. Kaufman, and R.C. Thomas. 2013. Parallelizing Gaussian Process Calculations in R. arXiv:1305.4886. http://arxiv.org/abs/1305.4886.

See Also

bigGP

Examples

```
## Not run:
if(require(fields)) {
  SN2011fe <- SN2011fe_subset
  SN2011fe_newdata <- SN2011fe_newdata_subset
  SN2011fe_mle <- SN2011fe_mle_subset
  nProc <- 3
n <- nrow(SN2011fe)
m <- nrow(SN2011fe_newdata)
nu <- 2
inputs <- c(as.list(SN2011fe), as.list(SN2011fe_newdata), nu = nu)
prob <- krigeProblem$new("prob", numProcesses = nProc, n = n, m = m,
predMeanFunction = SN2011fe_predmeanfunc, crossCovFunction =
SN2011fe_crosscovfunc,  predCovFunction = SN2011fe_predcovfunc,
meanFunction = SN2011fe_meanfunc, covFunction = SN2011fe_covfunc,
inputs = inputs, params = SN2011fe_mle$par, data = SN2011fe$flux,
packages = c("fields"))

remoteCalcChol(matName = "C", cholName = "L", matPos = "prob",
  cholPos = "prob", n = n, h = prob$h_n)
remoteCalc("data", "mean", `-`, "tmp1", input1Pos = "prob", input2Pos = "prob")
remoteForwardsolve(cholName = "L", inputName = "tmp1", outputName = "tmp2",
cholPos = "prob", n1 = n, h1 = prob$h_n)
remoteBacksolve(cholName = "L", inputName = "tmp2", outputName = "tmp3",
 cholPos = "prob", n1 = n, h1 = prob$h_n)
prob$remoteConstructCov(obs = FALSE, pred = FALSE, cross = TRUE, verbose = TRUE)
# we now have a rectangular cross-covariance matrix named 'crossC'
remoteCrossProdMatVec(matName = "crossC", inputName = "tmp3", outputName = "result",
matPos = "prob", n1 = n, n2 = m, h1 = prob$h_n, h2 = prob$h_m)

result <- collectVector("result", n = n, h = prob$h_n)
}

## End(Not run)
```

remoteForwardsolve *Solve a Distributed Triangular System*

Description

Solves a distributed system of linear equations where the coefficient matrix is lower triangular. `remoteBacksolve` solves $L^{\top}X = C$ for vector or matrix X, while `remoteForwardsolve` solves $LX = C$. Any of the matrices or vectors can be contained within environments and ReferenceClass objects as well as the global environment on the slave processes.

Usage

```
remoteBacksolve(cholName, inputName, outputName, cholPos = '.GlobalEnv',
 inputPos = '.GlobalEnv', outputPos = '.GlobalEnv', n1, n2 = NULL, h1 = 1,
h2 = NULL)
remoteForwardsolve(cholName, inputName, outputName, cholPos =
'.GlobalEnv', inputPos = '.GlobalEnv', outputPos = '.GlobalEnv', n1, n2
= NULL, h1 = 1, h2 = NULL)
```

Arguments

`cholName`	name of the input lower triangular matrix matrix (the matrix of coefficients), given as a character string, of the object on the slave processes.
`inputName`	name of the vector or matrix being solved into (the right-hand side(s) of the equations), given as a character string, of the object on the slave processes.
`outputName`	the name to use for the output object, the solution vector or matrix, on the slave processes.
`cholPos`	where to look for the lower triangular matrix, given as a character string (unlike `get`). This can indicate an environment, a list, or a ReferenceClass object.
`inputPos`	where to look for the input right-hand side matrix or vector, given as a character string (unlike `get`). This can indicate an environment, a list, or a ReferenceClass object.
`outputPos`	where to do the assignment of the output matrix or vector, given as a character string (unlike `assign`). This can indicate an environment or a ReferenceClass object.
`n1`	a positive integer, the number of rows and columns of the input matrix.
`n2`	a positive integer, the number of columns of the right-hand side values. When equal to one, indicates a single right-hand side vector.
`h1`	a positive integer, the block replication factor, h, relevant for the input matrix and used for the solution (either for a vector, or the rows of the solution for a matrix).
`h2`	a positive integer, the block replication factor, h, relevant for the columns of the solution when the right-hand side is a matrix.

Details

Computes the solution to a distributed set of linear equations, with either a single or multiple right-hand side(s) (i.e., solving into a vector or a matrix). Note that these functions work for any distributed lower triangular matrix, but `bigGP` currently only provides functionality for computing distributed Cholesky factors, hence the argument names `cholName` and `cholPos`.

When the right-hand side is vector that is stored as a vector, such as created by `distributeVector` or `remoteConstructRnormVector`, use `n2 = NULL`. When multiplying by a one-column matrix, use `n2 = 1`.

References

Paciorek, C.J., B. Lipshitz, W. Zhuo, Prabhat, C.G. Kaufman, and R.C. Thomas. 2015. Parallelizing Gaussian Process Calculations in R. Journal of Statistical Software, 63(10), 1-23. www.jstatsoft.org/v63/i10.

Paciorek, C.J., B. Lipshitz, W. Zhuo, Prabhat, C.G. Kaufman, and R.C. Thomas. 2013. Parallelizing Gaussian Process Calculations in R. arXiv:1305.4886. http://arxiv.org/abs/1305.4886.

See Also

bigGP

Examples

```
## Not run:
if(require(fields)) {
  SN2011fe <- SN2011fe_subset
  SN2011fe_newdata <- SN2011fe_newdata_subset
  SN2011fe_mle <- SN2011fe_mle_subset
  nProc <- 3
n <- nrow(SN2011fe)
m <- nrow(SN2011fe_newdata)
nu <- 2
inputs <- c(as.list(SN2011fe), as.list(SN2011fe_newdata), nu = nu)
prob <- krigeProblem$new("prob", numProcesses = nProc, n = n, m = m,
predMeanFunction = SN2011fe_predmeanfunc, crossCovFunction = SN2011fe_crosscovfunc,
predCovFunction = SN2011fe_predcovfunc, meanFunction = SN2011fe_meanfunc,
covFunction = SN2011fe_covfunc,  inputs = inputs, params = SN2011fe_mle$par,
data = SN2011fe$flux, packages = c("fields"))
remoteCalcChol(matName = "C", cholName = "L", matPos = "prob",
  cholPos = "prob", n = n, h = prob$h_n)
remoteForwardsolve(cholName = "L", inputName = "data", outputName =
"tmp", cholPos = "prob", inputPos = "prob", n1 = n, h1 = prob$h_n)
LinvY <- collectVector("tmp", n = n, h = prob$h_n)
remoteBacksolve(cholName = "L", inputName = "tmp", outputName =
"tmp2", cholPos = "prob", inputPos = "prob", n1 = n, h1 = prob$h_n)
CinvY <- collectVector("tmp2", n = n, h = prob$h_n)
}

## End(Not run)
```

remoteGetIndices — *Determine Indices of Vector or Matrix Elements Stored on all Processes*

Description

`remoteGetIndices` determines the indices of the subset of a matrix or vector that are stored on each process.

Usage

```
remoteGetIndices(type = "vector", objName, objPos = ".GlobalEnv", n1,
 n2 = NULL, h1 = 1, h2 = 1)
```

Arguments

`type`	a string, one of 'vector', 'symmetric', 'triangular', or 'rectangular' giving the type of object for which one wants the indices. Note that square and symmetric matrices are both stored as lower triangles, so these options both return the same result. For square, non-symmetric matrices, use 'rectangular'.
`objName`	the name to use for the object containing the indices on the slave processes.
`objPos`	where to do the assignment of the object, given as a character string (unlike `assign`). This can indicate an environment or a ReferenceClass object.
`n1`	a positive integer, giving the length of the vector, number of rows and columns of a symmetric or triangular matrix and number of rows of a rectangular matrix, including square, non-symmetric matrices.
`n2`	a positive integer, giving the number of columns of a a rectangular matrix.
`h1`	a positive integer, giving the block replication factor for a vector, a symmetric or triangular matrix, or the rows of a rectangular matrix.
`h2`	a positive integer, giving the block replication factor for the columns of the rectangular matrix.

Details

`remoteGetIndices` calculates the indices as described in `localGetVectorIndices`, `localGetTriangularMatrixIndices`, and `localGetRectangularMatrixIndices`, and writes them to an object named `objName`.

remoteLs *Remote List Objects*

Description

remoteLs returns the names of the objects in the global environment on each slave process, as a list of character vectors.

Usage

```
remoteLs(all.names = FALSE)
```

Arguments

all.names	a logical value. If 'TRUE', all object names are returned. If 'FALSE', names which begin with a '.' are omitted.

Value

A list, with each element a vector of character strings giving the names of the objects on a given slave process.

See Also

remoteRm

Examples

```
## Not run:
bigGP.init(3)
a <- 3
b <- 7
push(a); push(b)
remoteLs()
remoteRm(a)
remoteLs()

## End(Not run)
```

`remoteMultChol` *Distributed Multiplication of Lower Triangular Matrix and a Vector or Matrix*

Description

`remoteMultChol` multiplies a distributed lower triangular matrix by a distributed vector or matrix. The objects can both be contained within environments or ReferenceClass objects as well as the global environment on the slave processes.

Usage

```
remoteMultChol(cholName, inputName, outputName, cholPos = '.GlobalEnv',
inputPos = '.GlobalEnv', outputPos = '.GlobalEnv', n1, n2 = NULL, h1 = 1,
h2 = NULL)
```

Arguments

`cholName`	name of the lower triangular matrix, given as a character string, giving the name of the object on the slave processes.
`inputName`	name of the vector or matrix being multiplied by, given as a character string, giving the name of the object on the slave processes.
`outputName`	the name to use for resulting vector or matrix product, on the slave processes.
`cholPos`	where to look for the lower triangular matrix, given as a character string (unlike `get`). This can indicate an environment, a list, or a ReferenceClass object.
`inputPos`	where to look for the input matrix or vector, given as a character string (unlike `get`). This can indicate an environment, a list, or a ReferenceClass object.
`outputPos`	where to do the assignment of the output matrix or vector, given as a character string (unlike `assign`). This can indicate an environment or a ReferenceClass object.
`n1`	a positive integer, the number of rows and columns of the lower triangular matrix.
`n2`	a positive integer, the number of columns of the vector or matrix being multiplied by. When equal to one, indicates multiplication by a vector.
`h1`	a positive integer, the block replication factor, h, relevant for the input matrix and used for the solution (either for a vector, or the rows of the solution for a matrix).
`h2`	a positive integer, the block replication factor, h, relevant for the columns of the input and output matrices when the lower triangular matrix is multiplied by a matrix.

Details

Computes the distributed product using a blocked algorithm. Note that the function works for any distributed lower triangular matrix, but bigGP currently only provides functionality for computing distributed Cholesky factors, hence the argument names cholName and cholPos.

When multiplying by a vector that is stored as a vector, such as created by distributeVector or remoteConstructRnormVector, use n2 = NULL. When multiplying by a one-column matrix, use n2 = 1.

References

Paciorek, C.J., B. Lipshitz, W. Zhuo, Prabhat, C.G. Kaufman, and R.C. Thomas. 2015. Parallelizing Gaussian Process Calculations in R. Journal of Statistical Software, 63(10), 1-23. www.jstatsoft.org/v63/i10.

Paciorek, C.J., B. Lipshitz, W. Zhuo, Prabhat, C.G. Kaufman, and R.C. Thomas. 2013. Parallelizing Gaussian Process Calculations in R. arXiv:1305.4886. http://arxiv.org/abs/1305.4886.

See Also

bigGP

Examples

```
## Not run:
if(require(fields)) {
  SN2011fe <- SN2011fe_subset
  SN2011fe_newdata <- SN2011fe_newdata_subset
  SN2011fe_mle <- SN2011fe_mle_subset
  nProc <- 3
n <- nrow(SN2011fe)
m <- nrow(SN2011fe_newdata)
nu <- 2
inputs <- c(as.list(SN2011fe), as.list(SN2011fe_newdata), nu = nu)
prob <- krigeProblem$new("prob", numProcesses = nProc, n = n, m = m,
predMeanFunction = SN2011fe_predmeanfunc, crossCovFunction = SN2011fe_crosscovfunc,
predCovFunction = SN2011fe_predcovfunc, meanFunction = SN2011fe_meanfunc,
covFunction = SN2011fe_covfunc,  inputs = inputs, params = SN2011fe_mle$par,
data = SN2011fe$flux, packages = c("fields"))
remoteCalcChol(matName = 'C', cholName = 'L', matPos = 'prob',
  cholPos = 'prob', n = n, h = prob$h_n)
remoteConstructRnormVector('z', n = n, h = prob$h_n)
remoteMultChol(cholName = 'L', inputName = 'z', outputName = 'result',
cholPos = 'prob', n1 = n, h1 = prob$h_n)
realiz <- collectVector('result', n = n, h = prob$h_n)
}

## End(Not run)
```

remoteRm — *Remote Remove Objects*

Description

remoteRm is used to remove objects from the global environment on the slave processes.

Usage

```
remoteRm(..., list = character())
```

Arguments

...	the objects to be removed, as names (unquoted) or character strings (quoted).
list	a character vector naming objects to be removed

Details

This is a distributed version of rm. It removes the named objects from all of the slave processes. Unlike rm, remoteRm is currently not enabled to remove objects from other than the global environment. Note that unless options(warn = 2) is set on the slave processes, no warning is reported if one tries to remove objects that do not exist.

See Also

remoteLs

Examples

```
## Not run:
bigGP.init(3)
a <- 3
b <- 7
push(a); push(b)
remoteLs()
remoteRm(a)
remoteLs()

## End(Not run)
```

SN2011fe *SN2011fe Supernova Dataset*

Description

SN2011fe is a dataset of flux values and estimated standard errors, as a function of phase and wavelength, from the SN 2011fe supernova event. Data were collected over multiple nights (phases) and multiple wavelengths.

Format

The SN2011fe object is a data frame containing the following columns:

phase: time of measurement in days.

wavelength: wavelength of measurement in Å.

flux: flux measurement in erg s^{-1} cm^{-2} $Å^{-1}$.

fluxerror: estimated standard deviation of the error in measurement of the flux.

phaseindex: 1-based index value of the time of measurement [check this]

logwavelength: log of wavelength.

The SN2011fe_newdata object is a data frame of prediction points on a fine grid of phases and wavelengths. The columns correspond to the phase and wavelength columns in SN2011fe but the initial 'p' stands for 'prediction'.

The SN2011fe_mle object is the output from maximum likelihood fitting of the parameters of a statistical model for the dataset, with the par element containing the MLEs.

The objects labeled '_subset' are analogous objects for a small subset of the dataset feasible to be fit without parallel processing.

The SN2011fe_initialParams object is a set of starting values for the maximum likelihood fitting.

The functions SN2011fe_meanfunc, SN2011fe_predmeanfunc, SN2011fe_covfunc, SN2011fe_crosscovfunc, and SN2011fe_predcovfunc are functions for calculating the various mean vectors and covariance matrices used in the statistical analysis of the dataset. Users will need to create analogous functions for their own kriging problems, so these are provided in part as templates.

Warning

Note that the SN2011fe_newdata set of prediction points was chosen to ensure that the points were not so close together as to result in numerically non-positive definite covariance matrices when simulating posterior realizations.

Source

http://snfactory.lbl.gov/snf/data/SN2011fe.tar.gz

References

For more details on the dataset, see: R. Pereira, et al., 2013, "Spectrophotometric time series of SN 2011fe from the Nearby Supernova Factory," Astronomy and Astrophysics, accepted (arXiv:1302.1292v1), DOI: http://dx.doi.org/10.1051/0004-6361/201221008.

For more details on the statistical model used to fit the data, see:

Paciorek, C.J., B. Lipshitz, W. Zhuo, Prabhat, C.G. Kaufman, and R.C. Thomas. 2015. Parallelizing Gaussian Process Calculations in R. Journal of Statistical Software, 63(10), 1-23. www.jstatsoft.org/v63/i10.

or

Paciorek, C.J., B. Lipshitz, W. Zhuo, Prabhat, C.G. Kaufman, and R.C. Thomas. 2013. Parallelizing Gaussian Process Calculations in R. arXiv:1305.4886. http://arxiv.org/abs/1305.4886.

See Also

krigeProblem-class

Examples

```
## Not run:
doSmallExample <- TRUE

if(require(fields)) {
if(doSmallExample){
  SN2011fe <- SN2011fe_subset
  SN2011fe_newdata <- SN2011fe_newdata_subset
  SN2011fe_mle <- SN2011fe_mle_subset
  nProc <- 3
} else {
# users should select number of processors based on their system and the
# size of the full example
nProc <- 210
}

n <- nrow(SN2011fe)
m <- nrow(SN2011fe_newdata)
nu <- 2
inputs <- c(as.list(SN2011fe), as.list(SN2011fe_newdata), nu = nu)

prob <- krigeProblem$new("prob", numProcesses = nProc, n = n, m = m,
predMeanFunction = SN2011fe_predmeanfunc, crossCovFunction = SN2011fe_crosscovfunc,
predCovFunction = SN2011fe_predcovfunc, meanFunction =
SN2011fe_meanfunc, covFunction = SN2011fe_covfunc,  inputs = inputs,
params = SN2011fe_mle$par, data = SN2011fe$flux, packages = c("fields"))

prob$calcLogDens()
}

## End(Not run)
```

Index