Package: hitandrun (via r-universe)

October 28, 2026

Type Package

Title ``Hit and Run'' and ``Shake and Bake'' for Sampling Uniformly from Convex Shapes

Version 0.5-6

Date 2022-05-26

Description The ``Hit and Run'' Markov Chain Monte Carlo method for sampling uniformly from convex shapes defined by linear constraints, and the ``Shake and Bake'' method for sampling from the boundary of such shapes. Includes specialized functions for sampling normalized weights with arbitrary linear constraints. Tervonen, T., van Valkenhoef, G., Basturk, N., and Postmus, D. (2012) <doi:10.1016/j.ejor.2012.08.026>. van Valkenhoef, G., Tervonen, T., and Postmus, D. (2014) <doi:10.1016/j.ejor.2014.06.036>.

URL https://github.com/gertvv/hitandrun

License GPL-3

LazyLoad yes

Imports rcdd (>= 1.1), stats

Suggests testthat (>= 0.8)

NeedsCompilation yes

Author Gert van Valkenhoef [aut, cre, cph], Tommi Tervonen [aut]

Maintainer Gert van Valkenhoef <gert@gertvv.nl>

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

Date/Publication 2022-05-27 12:10:02 UTC

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

RemoteRef HEAD

RemoteSha f999355267f8e3dab25ff4617291cc61c5cbf87a

Contents

hitandrun-package . . . 2
bbReject . . . 4
createBoundBox . . . 5
createSeedPoint . . . 6
createTransform . . . 7
eliminateRedundant . . . 8
findExtremePoints . . . 9
findFace . . . 10
findInteriorPoint . . . 11
findVertices . . . 12
har . . . 13
harConstraints . . . 15
hitandrun . . . 16
hypersphere.sample . . . 19
sab . . . 19
shakeandbake . . . 21
simplex.createConstraints . . . 24
simplex.createTransform . . . 25
simplex.sample . . . 27
solution.basis . . . 28
transformConstraints . . . 29

Index **31**

`hitandrun-package` *"Hit and Run" sampling*

Description

This package provides a "Hit and Run" sampler that generates a Markov chain whose stable state converges on the uniform distribution over a convex polytope. The polytope is given by a set of inequality constraints in standard linear programming form ($Ax \leq b$) and optionally a set of equality constraints. In addition, there is a "Shake and Bake" sampler to generate points from the boundary of such a shape.

Utilities are provided for sampling from subsets of the unit simplex (i.e. random variates that can be interpreted as weights satisfying certain constraints) and for specifying common constraints.

Details

`hitandrun` and `shakeandbake` now provide the most general interface for sampling from spaces defined by arbitrary linear equality and inequality constraints. The functions described in the following provide lower level functionality on which it is built.

`har` is the core "Hit and Run" sampler, `sab` is the core "Shake and Bake" sampler, `bbReject` is the bounding box rejection sampler, and `simplex.sample` samples uniformly from the unit simplex.

See simplex.createTransform and simplex.createConstraints for sampling from subsets of the unit simplex. Utilities to specify common constraints are described in harConstraints.

When the sampling space is restricted by different linear equality constraints, use solution.basis, createTransform, and transformConstraints. This is a generalization of the methods for sampling from the simplex.

Note

"Hit and Run" is a Markov Chain Monte Carlo (MCMC) method, so generated samples form a correlated time series. To get a uniform sample, you need $O^*(n^3)$ samples, where n is the dimension of the sampling space.

Author(s)

Maintainer: Gert van Valkenhoef <gert@gertvv.nl>

References

Tervonen, T., van Valkenhoef, G., Basturk, N., and Postmus, D. (2012) "Hit-And-Run enables efficient weight generation for simulation-based multiple criteria decision analysis". *European Journal of Operational Research* 224(3) 552-559. doi: 10.1016/j.ejor.2012.08.026 van Valkenhoef, G., Tervonen, T., and Postmus, D. (2014) "Notes on 'Hit-And-Run enables efficient weight generation for simulation-based multiple criteria decision analysis'". *European Journal of Operational Research* (in press). doi: 10.1016/j.ejor.2014.06.036

See Also

hitandrun har

bbReject simplex.sample hypersphere.sample

solution.basis createTransform transformConstraints

simplex.createTransform simplex.createConstraints

harConstraints

createSeedPoint createBoundBox

Examples

```
# Example: sample weight vectors where w_1 >= w_2 and w_1 >= w_3
n <- 3 # length of weight vector
constr <- mergeConstraints(
  ordinalConstraint(n, 1, 2),
  ordinalConstraint(n, 1, 3))
transform <- simplex.createTransform(n)
constr <- simplex.createConstraints(transform, constr)
seedPoint <- createSeedPoint(constr, homogeneous=TRUE)
N <- 1000
w <- har(seedPoint, constr, N=N * (n-1)^3, thin=(n-1)^3,
  homogeneous=TRUE, transform=transform)$samples
stopifnot(all(w[,1] >= w[,2]) && all(w[,1] >= w[,3]))
```

bbReject *Bounding box rejection sampler*

Description

Generates uniform random variates over a convex polytope defined by a set of linear constraints by generating uniform variates over a bounding box and rejecting those outside the polytope.

Usage

```
bbReject(lb, ub, constr, N, homogeneous=FALSE, transform=NULL)
```

Arguments

lb	Lower bound for each dimension (not including homogeneous coordinate)
ub	Upper bound for each dimension (not including homogeneous coordinate)
constr	Constraint definition (see details)
N	Number of samples to generate
homogeneous	Whether constr and transform are given in homogeneous coordinate representation (see details)
transform	Transformation matrix to apply to the generated samples (optional)

Details

See har for a description of the constraint definition and the homogeneous coordinate representation.

Value

A list, containing:

samples	A matrix containing the generated samples as rows.
rejectionRate	The mean number of samples rejected for each accepted sample.

Author(s)

Gert van Valkenhoef

See Also

createBoundBox

harConstraints simplex.createTransform simplex.createConstraints

Examples

```
# constraints: x_1 >= 0, x_2 >= 0, x_1 + x_2 <= 1
A <- rbind(c(-1, 0), c(0, -1), c(1, 1))
b <- c(0, 0, 1)
d <- c("<=", "<=", "<=")
constr <- list(constr=A, rhs=b, dir=d)

# create a bounding box that contains the polytope
lb <- c(0, 0)
ub <- c(1, 1)

# sample 10,000 points
samples <- bbReject(lb, ub, constr, 1E4)$samples

# Check dimension of result
stopifnot(dim(samples) == c(1E4, 2))

# Check that x_i >= 0
stopifnot(samples >= 0)

# Check that x_1 + x_2 <= 1
stopifnot(samples[,1] + samples[,2] <= 1)

plot(samples)
```

---

createBoundBox *Calculate a bounding box*

Description

Calculate a bounding box around a polytope given by a set of linear constraints.

Usage

```
createBoundBox(constr, homogeneous=FALSE)
```

Arguments

constr	Constraint definition
homogeneous	Whether constr is given in homogeneous coordinate representation

Details

See har for a description of the constraint definition and the homogeneous coordinate representation.

This function uses findExtremePoints to find extreme points along each dimension.

Value

lb	Lower bound for each dimension (not including homogeneous coordinate).
ub	Upper bound for each dimension (not including homogeneous coordinate).

Author(s)

Gert van Valkenhoef

See Also

har

findExtremePoints

Examples

```
# constraints: x_1 >= 0, x_2 >= 0, x_1 + x_2 <= 1
A <- rbind(c(-1, 0), c(0, -1), c(1, 1))
b <- c(0, 0, 1)
d <- c("<=", "<=", "<=")
constr <- list(constr=A, rhs=b, dir=d)

bb <- createBoundBox(constr)
stopifnot(bb$lb == c(0.0, 0.0))
stopifnot(bb$ub == c(1.0, 1.0))
```

---

createSeedPoint *Generate a seed point*

Description

Generate a seed point inside a polytope given by a set of linear constraints.

Usage

```
createSeedPoint(constr, homogeneous=FALSE, randomize=FALSE, method="slacklp")
```

Arguments

constr	Constraint definition
homogeneous	Whether constr is given in homogeneous coordinate representation
randomize	If TRUE, randomize the starting point
method	How to obtain the starting point: "slacklp" for a linear program that maximizes the minimum slack, or "vertices" for a weighted average of the vertices of the polytope

Details

See har for a description of the constraint definition and the homogeneous coordinate representation.

- The "slacklp" method solves a linear program that maximizes the minimum slack on the inequality constraints. When randomized, the slack on each constraint is randomly rescaled before maximization.
- The "vertices" method enumerates all vertices of the polytope and then calculates the weighted arithmetic mean of this set of points. If 'randomize' is set, the weights are randomly generated, otherwise they are all equal and the generated point is the centroid of the polytope.

Value

A coordinate vector in the appropriate coordinate system.

Author(s)

Gert van Valkenhoef

See Also

har

findExtremePoints findVertices

Examples

```
# constraints: x_1 >= 0, x_2 >= 0, x_1 + x_2 <= 1
A <- rbind(c(-1, 0), c(0, -1), c(1, 1))
b <- c(0, 0, 1)
d <- c("<=", "<=", "<=")
constr <- list(constr=A, rhs=b, dir=d)

x0 <- createSeedPoint(constr)
stopifnot(x0 >= 0)
stopifnot(sum(x0) <= 1)
```

---

createTransform *Create transformation matrices*

Description

This function takes a basis, consisting of an $n \times m$ change of basis matrix and an n-vector representing the origin of the m-space, and generates a matrix to transform points in the m-space, given in homogeneous coordinates, to the n-space.

The inverse transform can also be generated, and conversion can be to homogeneous coordinates instead of Cartesian ones.

Usage

```
createTransform(basis, inverse=FALSE, keepHomogeneous=inverse)
```

Arguments

basis	Basis (and origin) for the m-space (see solution.basis)
inverse	TRUE to convert from n-space coordinates to m-space coordinates
keepHomogeneous	TRUE to convert to homogeneous coordinates rather than Cartesian

Details

Multiply a coordinate vector in homogeneous coordinates by pre-multiplying by the generated matrix (see examples).

Value

A transformation matrix.

Author(s)

Gert van Valkenhoef

See Also

solution.basis

---

eliminateRedundant *Eliminate redundant linear constraints*

Description

Given a set of linear constraints, gives a subset of these constraints that are non-redundant.

Usage

```
eliminateRedundant(constr)
```

Arguments

constr	Constraints

Details

If no constraints are redundant, returns the same set of constraints.

Value

A set of non-redundant constraints.

Author(s)

Gert van Valkenhoef, Tommi Tervonen

See Also

harConstraints

Examples

```
constr <- list(
  constr = rbind(
    c(-1  ,  0),
    c( 0  , -1),
    c( 1  ,  1),
    c( 0.5, -1)),
  dir = c('<=', '<=', '=', '<='),
  rhs = c(0, 0, 1, 0))

constr <- eliminateRedundant(constr)

stopifnot(nrow(constr$constr) == 3) # eliminates one constraint
```

---

findExtremePoints *Find extreme points*

Description

Find extreme points of a polytope given by a set of linear constraints along each dimension.

Usage

```
findExtremePoints(constr, homogeneous=FALSE)
```

Arguments

constr	Constraint definition
homogeneous	Whether constr is given in homogeneous coordinate representation

Details

See har for a description of the constraint definition and the homogeneous coordinate representation.

For n-dimensional coordinate vectors, solves 2n LPs to find the extreme points along each dimension.

Value

A matrix, in which each row is a coordinate vector in the appropriate coordinate system.

Author(s)

Gert van Valkenhoef

See Also

har

findInteriorPoint findVertices

lpcdd

Examples

```
# constraints: x_1 >= 0, x_2 >= 0, x_1 + x_2 <= 1
A <- rbind(c(-1, 0), c(0, -1), c(1, 1))
b <- c(0, 0, 1)
d <- c("<=", "<=", "<=")
constr <- list(constr=A, rhs=b, dir=d)

findExtremePoints(constr, homogeneous=FALSE)
```

---

findFace *Find the closest face (constraint) to an interior point of a polytope.*

Description

Find the closest face (constraint) to an interior point of a polytope defined by a set of linear constraints.

Usage

```
findFace(x, constr)
```

Arguments

x	An interior point
constr	Constraint definition

Details

See har for a description of the constraint definition.

Value

A face index.

Author(s)

Gert van Valkenhoef

See Also

har

Examples

```
# constraints: x_1 >= 0, x_2 >= 0, x_1 + x_2 <= 1
A <- rbind(c(-1, 0), c(0, -1), c(1, 1))
b <- c(0, 0, 1)
d <- c("<=", "<=", "<=")
constr <- list(constr=A, rhs=b, dir=d)

stopifnot(findFace(c(0.1, 0.2), constr) == 1)
stopifnot(findFace(c(0.2, 0.1), constr) == 2)
stopifnot(findFace(c(0.4, 0.4), constr) == 3)
```

---

findInteriorPoint *Find an interior point*

Description

Find an interior point of a polytope given by a set of linear constraints along each dimension.

Usage

```
findInteriorPoint(constr, homogeneous=FALSE, randomize=FALSE)
```

Arguments

constr	Constraint definition
homogeneous	Whether constr is given in homogeneous coordinate representation
randomize	Whether the point should be randomized

Details

See har for a description of the constraint definition and the homogeneous coordinate representation.

Solves a slack-maximizing LP to find an interior point of the polytope defined by the given constraints. The randomized version randomly scales the slack on each (non-redundant) constraint.

Value

A vector.

Author(s)

Gert van Valkenhoef

See Also

har

findExtremePoints findVertices

lpcdd

Examples

```
# constraints: x_1 >= 0, x_2 >= 0, x_1 + x_2 <= 1
A <- rbind(c(-1, 0), c(0, -1), c(1, 1))
b <- c(0, 0, 1)
d <- c("<=", "<=", "<=")
constr <- list(constr=A, rhs=b, dir=d)

findInteriorPoint(constr, homogeneous=FALSE)
```

---

findVertices *Find vertices of the polytope*

Description

Find the vertices of a polytope given by a set of linear constraints.

Usage

```
findVertices(constr, homogeneous=FALSE)
```

Arguments

constr	Constraint definition
homogeneous	Whether constr is given in homogeneous coordinate representation

Details

See har for a description of the constraint definition and the homogeneous coordinate representation.

Uses the Avis-Fukuda pivoting algorithm to enumerate the vertices of the polytope.

Value

A matrix, in which each row is a vertex of the polytope.

Author(s)

Gert van Valkenhoef

See Also

har

findExtremePoints findInteriorPoint

scdd

Examples

```
# constraints: x_1 >= 0, x_2 >= 0, x_1 + x_2 <= 1
A <- rbind(c(-1, 0), c(0, -1), c(1, 1))
b <- c(0, 0, 1)
d <- c("<=", "<=", "<=")
constr <- list(constr=A, rhs=b, dir=d)

findVertices(constr, homogeneous=FALSE)
```

---

har *"Hit and Run" sampler*

Description

The "Hit and Run" method generates a Markov Chain whose stable state converges on the uniform distribution over a convex polytope defined by a set of linear constraints.

Usage

```
har(x0, constr, N, thin=1, homogeneous=FALSE, transform=NULL)
```

Arguments

x0	Starting point (must be in the polytope)
constr	Constraint definition (see details)
N	Number of iterations to run
thin	Thinning factor (keep every 'thin'-th sample)
homogeneous	Whether x0, constr and transform are given in homogeneous coordinate representation (see details)
transform	Transformation matrix to apply to the generated samples (optional)

Details

The constraints, starting point and transformation matrix can be given in homogeneous coordinate representation (an extra component is added to each vector, equal to 1.0). This enables affine transformations (such as translation) to be applied to the coordinate vectors by the constraint and transformation matrices. Be aware that while non-affine (perspective) transformations are also possible, they will not in general preserve uniformity of the generated samples.

Constraints are given as a list(constr=A, rhs=b, dir=d), where d should contain only "<=". See hitandrun for a "Hit and Run" sampler that also supports equality constraints. The constraints define the polytope as usual for linear programming: $Ax \leq b$. In particular, it must be true that $Ax_0 \leq b$.

Value

A list, containing:

samples	A matrix containing the generated samples as rows.
xN	The last generated sample, untransformed. Can be used as the starting point for a continuation of the chain.

Note

"Hit and Run" is a Markov Chain Monte Carlo (MCMC) method, so generated samples form a correlated time series. To get a uniform sample, you need $O^*(n^3)$ samples, where n is the dimension of the sampling space.

Author(s)

Gert van Valkenhoef

References

Smith, R. L. (1984) "Efficient Monte Carlo Procedures for Generating Points Uniformly Distributed over Bounded Regions". *Operations Research* 32(6): 1296-1308. doi: 10.1287/opre.32.6.1296

See Also

harConstraints hitandrun

Examples

```
# constraints: x_1 >= 0, x_2 >= 0, x_1 + x_2 <= 1
A <- rbind(c(-1, 0), c(0, -1), c(1, 1))
b <- c(0, 0, 1)
d <- c("<=", "<=", "<=")
constr <- list(constr=A, rhs=b, dir=d)

# take a point x0 within the polytope
x0 <- c(0.25, 0.25)

# sample 10,000 points
samples <- har(x0, constr, 1E4)$samples

# Check dimension of result
stopifnot(dim(samples) == c(1E4, 2))

# Check that x_i >= 0
stopifnot(samples >= 0)

# Check that x_1 + x_2 <= 1
stopifnot(samples[,1] + samples[,2] <= 1)

plot(samples)
```

harConstraints *Constraint formulation utility functions*

Description

These utility functions generate linear constraints

Usage

```
simplexConstraints(n)
lowerBoundConstraint(n, i, x)
upperBoundConstraint(n, i, x)
lowerRatioConstraint(n, i, j, x)
upperRatioConstraint(n, i, j, x)
exactRatioConstraint(n, i, j, x)
ordinalConstraint(n, i, j)
mergeConstraints(...)
```

Arguments

n	Number of dimensions (vector components)
i	Index of first component
j	Index of second component
x	Scalar bound
...	Constraint definitions, or a single list of constraint definitions

Details

See har for a description of the constraint format.

simplexConstraints encodes the n-simplex: $\forall_k w_k \geq 0$ and $\sum_k w_k = 1$

lowerBoundConstraint encodes $w_i \geq x$

upperBoundConstraint encodes $w_i \leq x$

lowerRatioConstraint encodes $w_i/w_j \geq x$

upperRatioConstraint encodes $w_i/w_j \leq x$

exactRatioConstraint encodes $w_i/w_j = x$

ordinalConstraint encodes $w_i \geq w_j$

mergeConstraints merges the constraints it is given. Alternatively, the function takes a single list of constraint definitions which are to be merged.

Value

A constraint definition (concatenation of the given constraint definitions).

Author(s)

Gert van Valkenhoef

See Also

`eliminateRedundant` `hitandrun` `har`

Examples

```
# create an ordinal constraint
c1 <- ordinalConstraint(2, 1, 2)
stopifnot(c1$constr == c(-1, 1))
stopifnot(c1$rhs == c(0))
stopifnot(c1$dir == c("<="))

# create our own constraints
c2 <- list(constr=t(c(-1, 0)), rhs=c(0), dir=c("<="))
c3 <- list(constr=t(c(1, 1)), rhs=c(1), dir=c("<="))

# merge the constraints into a single definition
c <- mergeConstraints(c1, c2, c3)
stopifnot(c$constr == rbind(c(-1, 1), c(-1, 0), c(1, 1)))
stopifnot(c$rhs == c(0, 0, 1))
stopifnot(c$dir == c("<=", "<=", "<="))

# test the alternative (list) method
l <- mergeConstraints(list(c1, c2, c3))
stopifnot(c$constr == l$constr)
stopifnot(c$rhs == l$rhs)
stopifnot(c$dir == l$dir)

# test iteratively merging
l <- mergeConstraints(mergeConstraints(c1, c2), c3)
stopifnot(c$constr == l$constr)
stopifnot(c$rhs == l$rhs)
stopifnot(c$dir == l$dir)
```

`hitandrun` *"Hit and Run" sampler*

Description

The "Hit and Run" method generates a Markov Chain whose stable state converges on the uniform distribution over a convex polytope defined by a set of linear inequality constraints. `hitandrun` further uses the Moore-Penrose pseudo-inverse to eliminate an arbitrary set of linear equality constraints before applying the "Hit and Run" sampler.

`har.init` and `har.run` together provide a re-entrant version of `hitandrun` so that the Markov chain can be continued if convergence is not satisfactory.

Usage

```
hitandrun(constr, n.samples=1E4,
    thin.fn = function(n) { ceiling(log(n + 1)/4 * n^3) }, thin = NULL,
    x0.randomize=FALSE, x0.method="slacklp", x0 = NULL, eliminate = TRUE)

har.init(constr,
    thin.fn = function(n) { ceiling(log(n + 1)/4 * n^3) }, thin = NULL,
    x0.randomize=FALSE, x0.method="slacklp", x0 = NULL, eliminate = TRUE)

har.run(state, n.samples)
```

Arguments

constr	Linear constraints that define the sampling space (see details)
n.samples	The desired number of samples to return. The sampler is run for n.samples * thin iterations
thin.fn	Function that specifies a thinning factor depending on the dimension of the sampling space after equality constraints have been eliminated. Will only be invoked if thin is NULL
thin	The thinning factor
x0	Seed point for the Markov Chain. The seed point is specified in the original space, and transformed to the sampling space automatically.
x0.method	Method to generate the seed point if x0 is unspecified, see createSeedPoint
x0.randomize	Whether to generate a random seed point if x0 is unspecified
eliminate	Whether to eliminate redundant constraints before constructing the transformation to the sampling space and (optionally) calculating the seed point.
state	A state object, as generated by har.init (see value)

Details

The constraints are given as a list with the elements constr, dir and rhs. dir is a vector with values '=' or '<='. constr is a matrix and rhs a vector, which encode the standard linear programming constraint froms $Ax = b$ and $Ax \leq b$ (depending on dir). The lengths of rhs and dir must match the number of rows of constr.

hitandrun applies solution.basis to generate a basis of the (translated) solution space of the linear constraints (if any). An affine transformation is generated using createTransform and applied to the constraints. Then, a seed point satisfying the inequality constraints is generated using createSeedPoint. Finally, har is used to generate the samples.

Value

For hitandrun, a matrix containing the generated samples as rows.

For har.init, a state object, containing:

basis	The basis for the sampling space. See solution.basis.
transform	The sampling space transformation. See createTransform.

constr	The linear inequality constraints translated to the sampling space. See transformConstraints.
x0	The generated seed point. See createSeedPoint.
thin	The thinning factor to be used.

For har.run, a list containing:

samples	A matrix containing the generated samples as rows.
state	A state object that can be used to continue sampling from the Markov chain (i.e. x0 has been modified).

Note

"Hit and Run" is a Markov Chain Monte Carlo (MCMC) method, so generated samples form a correlated time series. To get a uniform sample, you need $O^*(n^3)$ samples, where n is the dimension of the sampling space.

Author(s)

Gert van Valkenhoef

See Also

harConstraints har

Examples

```
# Sample from the 3-simplex with the additional constraint that w_1/w_2 = 2
# Three inequality constraints, two equality constraints
constr <- mergeConstraints(simplexConstraints(3), exactRatioConstraint(3, 1, 2, 2))
samples <- hitandrun(constr, n.samples=1000)
stopifnot(dim(samples) == c(1000, 3))
stopifnot(all.equal(apply(samples, 1, sum), rep(1, 1000)))
stopifnot(all.equal(samples[,1]/samples[,2], rep(2, 1000)))

# Sample from the unit rectangle (no equality constraints)
constr <- list(
  constr = rbind(c(1,0), c(0,1), c(-1,0), c(0,-1)),
  dir=rep('<=', 4),
  rhs=c(1, 1, 0, 0))
state <- har.init(constr)
result <- har.run(state, n.samples=1000)
samples <- result$samples
stopifnot(all(samples >= 0 & samples <= 1))
# Continue sampling from the same chain:
result <- har.run(result$state, n.samples=1000)
samples <- rbind(samples, result$samples)
```

hypersphere.sample *Sample uniformly from an n-hypersphere*

Description

Generates uniform random variates over an n-hypersphere

Usage

```
hypersphere.sample(n, N)
```

Arguments

n	Dimension of the hypersphere
N	Number of samples

Value

A single n-dimensional sample from the hypersphere.

Author(s)

Tommi Tervonen <tommi@smaa.fi>

Examples

```
n <- 3 # Dimension
N <- 5 # Nr samples

sample <- hypersphere.sample(n, N)

# Check summing to unity
vec.norm <- function(x) { sum(x^2) }
stopifnot(all.equal(apply(sample, 1, vec.norm), rep(1, N)))
```

sab *"Shake and Bake" sampler*

Description

The "Shake and Bake" method generates a Markov Chain whose stable state converges on the uniform distribution over the boundary of a convex polytope defined by a set of linear constraints.

Usage

```
sab(x0, i0, constr, N, thin=1, homogeneous=FALSE, transform=NULL)
```

Arguments

x0	Starting point (must be in the polytope)
i0	Index of the closest face to the starting point
constr	Constraint definition (see details)
N	Number of iterations to run
thin	Thinning factor (keep every 'thin'-th sample)
homogeneous	Whether x0, constr and transform are given in homogeneous coordinate representation (see details)
transform	Transformation matrix to apply to the generated samples (optional)

Details

The constraints, starting point and transformation matrix can be given in homogeneous coordinate representation (an extra component is added to each vector, equal to 1.0). This enables affine transformations (such as translation) to be applied to the coordinate vectors by the constraint and transformation matrices. Be aware that while non-affine (perspective) transformations are also possible, they will not in general preserve uniformity of the generated samples.

Constraints are given as a list(constr=A, rhs=b, dir=d), where d should contain only "<=". See shakeandbake for a "Shake and Bake" sampler that also supports equality constraints. The constraints define the polytope as usual for linear programming: $Ax \leq b$. In particular, it must be true that $Ax_0 \leq b$. Points are generated from the boundary of the polytope (where equality holds for one of the constraints), using the "running" shake and bake sampler, which samples the direction vector so that every move point is accepted (Boender et al. 1991).

Value

A list, containing:

samples	A matrix containing the generated samples as rows.
faces	A vector containing the indices of the faces on which the samples lie.
xN	The last generated sample, untransformed. Can be used as the starting point for a continuation of the chain.
iN	Face on which the last generated sample lies.

Note

"Shake and Bake" is a Markov Chain Monte Carlo (MCMC) method, so generated samples form a correlated time series.

Author(s)

Gert van Valkenhoef

References

Boender, C. G. E., Caron, R. J., McDonald, J. F., Rinnooy Kan, A. H. G., Romeijn, H. E., Smith, R. L., Telgen, J., and Vorst, A. C. F. (1991) "Shake-and-Bake Algorithms for Generating Uniform Points on the Boundary of Bounded Polyhedra". *Operations Research* 39(6):945-954. doi: 10.1287/opre.39.6.945

See Also

harConstraints shakeandbake

Examples

```
# constraints: x_1 >= 0, x_2 >= 0, x_1 + x_2 <= 1
A <- rbind(c(-1, 0), c(0, -1), c(1, 1))
b <- c(0, 0, 1)
d <- c("<=", "<=", "<=")
constr <- list(constr=A, rhs=b, dir=d)

# take a point x0 within the polytope
x0 <- c(0.25, 0.25)

# sample 10,000 points
result <- sab(x0, 1, constr, 1E4)
samples <- result$samples

# Check dimension of result
stopifnot(dim(samples) == c(1E4, 2))

# Check that x_i >= 0
stopifnot(samples >= -1E-15)

# Check that x_1 + x_2 <= 1
stopifnot(samples[,1] + samples[,2] <= 1 + 1E-15)

# check that the results lie on the faces
faces <- result$faces
stopifnot(all.equal(samples[faces==1,1], rep(0, sum(faces==1))))
stopifnot(all.equal(samples[faces==2,2], rep(0, sum(faces==2))))
stopifnot(all.equal(samples[faces==3,1] + samples[faces==3,2], rep(1, sum(faces==3))))

plot(samples)
```

---

shakeandbake *"Shake and Bake" sampler*

Description

The "Shake and Bake" method generates a Markov Chain whose stable state converges on the uniform distribution over a the boundary of a convex polytope defined by a set of linear inequality constraints. shakeandbake further uses the Moore-Penrose pseudo-inverse to eliminate an arbitrary set of linear equality constraints before applying the "Shake and Bake" sampler.

sab.init and sab.run together provide a re-entrant version of shakeandbake so that the Markov chain can be continued if convergence is not satisfactory.

Usage

```
shakeandbake(constr, n.samples=1E4,
    thin.fn = function(n) { ceiling(log(n + 1)/4 * n^3) }, thin = NULL,
    x0.randomize=FALSE, x0.method="slacklp", x0 = NULL, eliminate = TRUE)

sab.init(constr,
    thin.fn = function(n) { ceiling(log(n + 1)/4 * n^3) }, thin = NULL,
    x0.randomize=FALSE, x0.method="slacklp", x0 = NULL, eliminate = TRUE)

sab.run(state, n.samples)
```

Arguments

constr	Linear constraints that define the sampling space (see details)
n.samples	The desired number of samples to return. The sampler is run for n.samples * thin iterations
thin.fn	Function that specifies a thinning factor depending on the dimension of the sampling space after equality constraints have been eliminated. Will only be invoked if thin is NULL
thin	The thinning factor
x0	Seed point for the Markov Chain. The seed point is specified in the original space, and transformed to the sampling space automatically.
x0.method	Method to generate the seed point if x0 is unspecified, see createSeedPoint
x0.randomize	Whether to generate a random seed point if x0 is unspecified
eliminate	Whether to eliminate redundant constraints before constructing the transformation to the sampling space and (optionally) calculating the seed point.
state	A state object, as generated by har.init (see value)

Details

The constraints are given as a list with the elements constr, dir and rhs. dir is a vector with values '=' or '<='. constr is a matrix and rhs a vector, which encode the standard linear programming constraint froms $Ax = b$ and $Ax \leq b$ (depending on dir). The lengths of rhs and dir must match the number of rows of constr.

shakeandbake applies solution.basis to generate a basis of the (translated) solution space of the linear constraints (if any). An affine transformation is generated using createTransform and

applied to the constraints. Then, a seed point satisfying the inequality constraints is generated using `createSeedPoint`. The closest face to this point is found using `findFace`. Finally, `sab` is used to generate the samples.

Value

For `shakeandbake`, a matrix containing the generated samples as rows.

For `sab.init`, a state object, containing:

`basis`	The basis for the sampling space. See `solution.basis`.
`transform`	The sampling space transformation. See `createTransform`.
`constr`	The linear inequality constraints translated to the sampling space. See `transformConstraints`.
`x0`	The generated seed point. See `createSeedPoint`.
`i0`	The index of the closest face. See `findFace`.
`thin`	The thinning factor to be used.

For `sab.run`, a list containing:

`samples`	A matrix containing the generated samples as rows.
`state`	A state object that can be used to continue sampling from the Markov chain (i.e. `x0` and `i0` have been modified).

Note

"Shake and Bake" is a Markov Chain Monte Carlo (MCMC) method, so generated samples form a correlated time series.

Author(s)

Gert van Valkenhoef

See Also

`harConstraints` `sab`

Examples

```
# Sample from the 3-simplex with the additional constraint that w_1/w_2 = 2
# Three inequality constraints, two equality constraints
constr <- mergeConstraints(simplexConstraints(3), exactRatioConstraint(3, 1, 2, 2))
samples <- shakeandbake(constr, n.samples=1000)
stopifnot(dim(samples) == c(1000, 3))
stopifnot(all.equal(apply(samples, 1, sum), rep(1, 1000)))

sel <- samples[,3] > 0.5 # detect which side we're on
stopifnot(all.equal(samples[sel,], matrix(rep(c(0,0,1), each=sum(sel)), ncol=3)))
stopifnot(all.equal(samples[!sel,], matrix(rep(c(2/3,1/3,0), each=sum(sel)), ncol=3)))

# Sample from the unit rectangle (no equality constraints)
constr <- list(
```

```
  constr = rbind(c(1,0), c(0,1), c(-1,0), c(0,-1)),
  dir=rep('<=', 4),
  rhs=c(1, 1, 0, 0))
state <- sab.init(constr)
result <- sab.run(state, n.samples=1000)
faces <- result$faces
samples <- result$samples
stopifnot(all(samples >= -1e-15 & samples <= 1 + 1e-15))

stopifnot(all.equal(samples[faces==1,1], rep(1, sum(faces==1))))
stopifnot(all.equal(samples[faces==2,2], rep(1, sum(faces==2))))
stopifnot(all.equal(samples[faces==3,1], rep(0, sum(faces==3))))
stopifnot(all.equal(samples[faces==4,2], rep(0, sum(faces==4))))

# Continue sampling from the same chain:
result <- sab.run(result$state, n.samples=1000)
samples <- rbind(samples, result$samples)
```

`simplex.createConstraints`

Create constraints that define the (n-1)-simplex

Description

This function takes a transformation matrix from the plane coincident with the (n-1) simplex and (optionally) additional constraints defined in n-dimensional space, and generates a set of constraints defining the simplex and (optionally) the additional constraints in the (n-1)-dimensional homogeneous coordinate system.

Usage

```
simplex.createConstraints(transform, userConstr=NULL)
```

Arguments

`transform`	Transformation matrix
`userConstr`	Additional constraints

Details

The transformation of the constraint matrix to (n-1)-dimensional homogeneous coordinates is a necessary preprocessing step for applying "Hit and Run" to subsets of the simplex defined by `userConstr`.

Value

A set of constraints in the (n-1)-dimensional homogeneous coordinate system.

Author(s)

Gert van Valkenhoef

See Also

simplex.createTransform har harConstraints

Examples

```
n <- 3
userConstr <- mergeConstraints(
 ordinalConstraint(3, 1, 2), ordinalConstraint(3, 2, 3))

transform <- simplex.createTransform(n)
constr <- simplex.createConstraints(transform, userConstr)
seedPoint <- createSeedPoint(constr, homogeneous=TRUE)

N <- 10000
samples <- har(seedPoint, constr, N, 1, homogeneous=TRUE, transform=transform)$samples

# Check dimension
stopifnot(dim(samples) == c(N, n))

# Check that w_i >= w_i+1
stopifnot(sapply(1:(n-1), function(i) {
 all(samples[,i]>=samples[,i+1])
}))

# Check that w_i >= 0
stopifnot(samples >= 0)

# Check that sum_i w_i = 1
E <- 1E-12
stopifnot(apply(samples, 1, sum) > 1 - E)
stopifnot(apply(samples, 1, sum) < 1 + E)
```

---

simplex.createTransform

Transform points on an (n-1)-simplex to n-dimensional space

Description

This function generates a matrix to transform points in an (n-1) dimensional homogeneous coordinate representation of the (n-1) simplex to n-dimensional Cartesian coordinates.

The inverse transform can also be generated, and conversion can be to homogeneous coordinates instead of Cartesion ones.

Usage

```
simplex.createTransform(n, inverse=FALSE, keepHomogeneous=inverse)
```

Arguments

n	Dimension of the space
inverse	TRUE to convert from n-space coordinates to (n-1)-simplex coordinates
keepHomogeneous	TRUE to convert to homogeneous coordinates rather than Cartesian

Details

Multiply a coordinate vector in homogeneous coordinates by pre-multiplying by the generated matrix (see examples).

Value

A transformation matrix.

Author(s)

Gert van Valkenhoef

See Also

simplex.createConstraints har

Examples

```
E <- 1E-12 # Allowed numerical error

# The origin in (n-1)-dimensional space should be the centroid of the simplex
# when transformed to n-dimensional space
transform <- simplex.createTransform(3)
x <- transform %*% c(0, 0, 1)
x
stopifnot(abs(x - c(1/3, 1/3, 1/3)) < E)

# The same should hold for the inverse transformation
invTransform <- simplex.createTransform(3, inverse=TRUE)
y <- invTransform %*% c(1/3, 1/3, 1/3, 1)
y
stopifnot(abs(y - c(0, 0, 1)) < E)

# Of course, an arbitrary weight vector should transform back to itself
transform <- simplex.createTransform(3, keepHomogeneous=TRUE)
x <- c(0.2, 0.5, 0.3, 1.0)
y <- transform %*% invTransform %*% x
y
stopifnot(abs(y - x) < E)
```

```
# And we can apply the tranform to a matrix:
a <- cbind(x, x, x)
b <- transform %*% invTransform %*% a
b
stopifnot(abs(b - a) < E)
```

`simplex.sample` *Sample uniformly from a simplex*

Description

Generates uniform random variates over the (n-1)-simplex in n-dimensional space.

Usage

```
simplex.sample(n, N, sort=FALSE)
```

Arguments

`n`	Dimension of the space
`N`	Number of samples to generate
`sort`	Whether to sort the components in descending order

Details

The samples will be uniform over the (n-1)-simplex.

Value

`samples`	A matrix containing the generated samples as rows.

Author(s)

Gert van Valkenhoef

Examples

```
n <- 3
N <- 10000
samples <- simplex.sample(n, N)$samples

# Check dimension
stopifnot(dim(samples) == c(N, n))

# Check that w_i >= 0
stopifnot(samples >= 0)

# Check that sum_i w_i = 1
E <- 1E-12
```

```
stopifnot(apply(samples, 1, sum) > 1 - E)
stopifnot(apply(samples, 1, sum) < 1 + E)


## Now with descending order
samples <- simplex.sample(n, N, sort=TRUE)$samples

# Check dimension
stopifnot(dim(samples) == c(N, n))

# Check that w_i >= 0
stopifnot(samples >= 0)

# Check that sum_i w_i = 1
E <- 1E-12
stopifnot(apply(samples, 1, sum) > 1 - E)
stopifnot(apply(samples, 1, sum) < 1 + E)

# Check w_i >= w_{i+1}
stopifnot(samples[,1] >= samples[,2])
stopifnot(samples[,2] >= samples[,3])
```

`solution.basis` *Calculate the basis for the solution space of a system of linear equations*

Description

Given a set of linear equality constraints, determine a translation and a basis for its solution space.

Usage

```
solution.basis(constr)
```

Arguments

`constr`	Linear equality constraints

Details

For a system of linear equations, $Ax = b$, the solution space is given by

$$x = A^{\dagger}b + (I - A^{\dagger}A)y$$

where $A^{\dagger}$ is the Moore-Penrose pseudoinverse of A. The QR decomposition of $I - A^{\dagger}A$ enables us to determine the dimension of the solution space and derive a basis for that space.

Value

A list, consisting of

`translate`	A point in the solution space
`basis`	A basis rooted in that point

Author(s)

Gert van Valkenhoef

See Also

`createTransform`

Examples

```
# A 3-dimensional original space
n <- 3

# x_1 + x_2 + x_3 = 1
eq.constr <- list(constr = t(rep(1, n)), dir = '=', rhs = 1)
basis <- solution.basis(eq.constr)
stopifnot(ncol(basis$basis) == 2) # Dimension reduced to 2
y <- rbind(rnorm(100, 0, 100), rnorm(100, 0, 100))
x <- basis$basis %*% y + basis$translate
stopifnot(all.equal(apply(x, 2, sum), rep(1, 100)))

# 2 x_2 = x_1; 2 x_3 = x_2
eq.constr <- mergeConstraints(
  eq.constr,
  list(constr = c(-1, 2, 0), dir = '=', rhs = 0),
  list(constr = c(0, -1, 2), dir = '=', rhs = 0))
basis <- solution.basis(eq.constr)
stopifnot(ncol(basis$basis) == 0) # Dimension reduced to 0
stopifnot(all.equal(basis$translate, c(4/7, 2/7, 1/7)))
```

`transformConstraints` *Apply a transformation to a set of linear constraints.*

Description

Given a set of linear constraints and a transformation matrix, return the constraints in the transformed space.

Usage

```
transformConstraints(transform, constr)
```

Arguments

transform	Transformation matrix
constr	Constraints

Details

Transforming the constraint matrix is a necessary preprocessing step for applying "Hit and Run" to subsets of a space defined by linear equality constraints. See solution.basis and createTransform for building the transformation matrix.

Value

A set of constraints in the new basis.

Author(s)

Gert van Valkenhoef

See Also

solution.basis createTransform har

Examples

```
# Sample from the space where 2*x_1 = x_2 + x_3 and
# 0 <= x_1, x_2, x_3 <= 5
n <- 3

eq.constr <- list(
  constr = matrix(c(2, -1, -1), nrow=1, ncol=n),
  dir = '=',
  rhs = 0)

ineq.constr <- list(
  constr = rbind(-diag(n), diag(n)),
  dir = rep('<=', n * 2),
  rhs = c(rep(0, n), rep(5, n)))

basis <- solution.basis(eq.constr)
transform <- createTransform(basis)
constr <- transformConstraints(transform, ineq.constr)
x0 <- createSeedPoint(constr, homogeneous=TRUE)
x <- har(x0, constr, 500, transform=transform, homogeneous=TRUE)$samples

stopifnot(all.equal(2 * x[,1], x[,2] + x[,3]))
stopifnot(all(x >= 0))
stopifnot(all(x <= 5))
```

Index

∗ **bounding box**
bbReject, 4
createBoundBox, 5
∗ **constraint**
eliminateRedundant, 8
harConstraints, 15
simplex.createConstraints, 24
transformConstraints, 29
∗ **hit-and-run**
har, 13
harConstraints, 15
hitandrun, 16
hitandrun-package, 2
∗ **hypersphere**
hypersphere.sample, 19
∗ **seed point**
createSeedPoint, 6
findExtremePoints, 9
findInteriorPoint, 11
findVertices, 12
∗ **shake-and-bake**
sab, 19
shakeandbake, 21
∗ **simplex**
simplex.createConstraints, 24
simplex.createTransform, 25
simplex.sample, 27
∗ **transform**
createTransform, 7
simplex.createTransform, 25
solution.basis, 28
transformConstraints, 29
∗ **uniform sampling**
bbReject, 4
har, 13
hitandrun, 16
hypersphere.sample, 19
sab, 19
shakeandbake, 21
simplex.sample, 27

bbReject, *2, 3*, 4

createBoundBox, *3, 4*, 5
createSeedPoint, *3*, 6, *17, 18*, *22, 23*
createTransform, *3*, 7, *17*, *22, 23*, *29, 30*

eliminateRedundant, 8, *16*
exactRatioConstraint *(*harConstraints*)*, 15

findExtremePoints, *5–7*, 9, *12, 13*
findFace, 10, *23*
findInteriorPoint, *10*, 11, *13*
findVertices, *7*, *10*, 12, *12*

har, *2–7*, *9–12*, 13, *13*, *15–18*, *25, 26*, *30*
har.init *(*hitandrun*)*, 16
har.run *(*hitandrun*)*, 16
harConstraints, *3, 4*, *9*, *14*, 15, *18*, *21*, *23*, *25*
hitandrun, *2, 3*, *13, 14*, 16, *16*
hitandrun-package, 2
hypersphere.sample, *3*, 19

lowerBoundConstraint *(*harConstraints*)*, 15
lowerRatioConstraint *(*harConstraints*)*, 15
lpcdd, *10*, *12*

mergeConstraints *(*harConstraints*)*, 15

ordinalConstraint *(*harConstraints*)*, 15

sab, *2*, 19, *23*
sab.init *(*shakeandbake*)*, 21
sab.run *(*shakeandbake*)*, 21
scdd, *13*
shakeandbake, *2*, *20*, 21, *21*
simplex.createConstraints, *3, 4*, 24, *26*

simplex.createTransform, *3, 4*, 25, *25*
simplex.sample, *2, 3*, 27
simplexConstraints *(*harConstraints*)*, 15
solution.basis, *3*, *8*, *17*, *22, 23*, 28, *30*

transformConstraints, *3*, *18*, *23*, 29

upperBoundConstraint *(*harConstraints*)*, 15
upperRatioConstraint *(*harConstraints*)*, 15