Package 'walkr'

June 29, 2019

Version 0.4.0

Date 2019-06-25

Type Package

Title Random Walks in the Intersection of Hyperplanes and the N-Simplex

Maintainer Andy Yao <andy.yao17@gmail.com>

Depends R (>= 3.1)

Suggests testthat, knitr, grid, gridExtra, rmarkdown

Imports hitandrun, limSolve, MASS, Rcpp (>= 0.11.6), shinystan

Description Consider the intersection of two spaces: the complete solution space to Ax = b and the N-simplex. The intersection of these two spaces is a non-negative convex polytope. The package walkr samples from this intersection using two Monte-Carlo Markov Chain (MCMC) methods: hit-and-run and Dikin walk. walkr also provide tools to examine sample quality. The package implements Dikin walk specified in Sachdeva (2016) <doi:10.1016/j.orl.2016.07.005>.

License GPL-3

URL https://github.com/andyyao95/walkr

BugReports https://github.com/andyyao95/walkr/issues

LinkingTo Rcpp, RcppEigen

NeedsCompilation yes

VignetteBuilder knitr

RoxygenNote 6.1.1

Author Andy Yao [aut, cre],
David Kane [aut],
Jeffrey Enos [aut]

Repository CRAN

Date/Publication 2019-06-29 05:10:03 UTC

R topics documented:

`calc_rhat` *R hat*

Description

This function calculates the rhat of each parameter given the list of chains that walkr produces. Since this is just an internal function, I'll document it more later.

Usage

```
calc_rhat(x)
```

Arguments

`x`	is the list of chains

Value

a vector of rhats

Examples

```
x <- matrix(seq(1,12,1), ncol = 3, nrow = 4)
calc_rhat(list(x,x,x))
```

complete_solution *Complete Solution*

Description

Given matrix equation $Aw = b$, find the basis representation of the complete solution (an affine transformation), returning the homogeneous and particular solution which describe the polytope in alpha-space

Usage

```
complete_solution(A, b)
```

Arguments

A	is the lhs of the matrix equation $Ax = b$
b	is the rhs of the matrix equation $Ax = b$

Value

a list object, with the first element($particular) as the particular solution and the the second element as a matrix with its columns containing the basis of the null space($homogeneous)

Examples

```
A <- matrix(1, ncol = 3)
b <- 0.5
complete_solution(A, b)
```

dikin_walk *Dikin Walk*

Description

This function implements the Dikin Walk using the Hessian of the Log barrier function. Note that a r of 1 guarantees that the ellipsoid generated won't leave our polytope K (see Theorems online)

Usage

```
dikin_walk(A, b, x0 = list(), points, r = 1, thin = 1, burn = 0,
  chains = 1)
```

Arguments

A	is the lhs of Ax <= b
b	is the rhs of Ax <= b
x0	is the starting point (a list of points)
points	is the number of points we want to sample
r	is the radius of the ellipsoid (1 by default)
thin	every thin-th point is stored
burn	the first burn points are deleted
chains	is the number of chains we run

Value

a list of chains of the sampled points, each chain being a matrix object with each column as a point

Examples

```
A <- rbind(c(1, 0), c(0, 1))
b <- c(1, 1)
sampled_points <- dikin_walk(A = A, b = b, points = 10, x0 = list(c(0.5,0.5)))

## Not run:
## note that this Ax <= b is different from Ax=b that the
## user specifies for walkr (see transformation section in vignette)
dikin_walk(A = A, b = b, x0, points = 100,
           r = 1thin = 1, burn = 0, chains = 1)

## End(Not run)
```

---

explore_walkr *Explore Walkr*

Description

This function takes in a list of chains and diagnoses them using the shinyStan interface. The app contains the confidence interval of each dimension's coordiates across the whole sets of points, Gelman-Rubin statistics, trace-plots, and other diagnostic tools for examining convergence.

Usage

```
explore_walkr(x)
```

Arguments

x	is the set of points sampled, with its columns as the sampled points. If multiple chains are present, then the columns should be ordered such that each chain follow each other.

Value

a shiny interface that display the diagnostics of the MCMC random walk

Examples

```
explore_walkr("list")

## Not run:
## x is the set of points sampled, this uses shinystan to visualize results!
explore_walkr(x)

## End(Not run)
```

`hit_and_run` *Hit and Run*

Description

This function provides a wrapper for the har function of the hit-and-run package

Usage

```
hit_and_run(A, b, x0, points, thin = 1, burn = 0, chains = 1)
```

Arguments

`A`	is the lhs of Ax <= b
`b`	is the rhs of Ax <= b
`x0`	is the starting point (a list of points)
`points`	is the number of points we want to sample
`thin`	every thin-th point is stored
`burn`	the first burn points are deleted
`chains`	is the number of chains we run

Value

a list of chains of the sampled points, each chain being a matrix object with each column as a point

Examples

```
A <- rbind(c(-1, 0), c(0, -1), c(1, 1))
b <- c(0, 0, 1)
hit_and_run(A = A, b = b, x0 = list(c(0.1,0.1)), points = 5)
```

rcppeigen_fcrossprod *Fast Matrix Cross-Product*

Description

Computes using RcppEigen the product of t(A) and B

Usage

```
rcppeigen_fcrossprod(A, B)
```

Arguments

A	is the first parameter in t(A) times B
B	is the second parameter in t(A) times B

Value

matrix cross-product t(A) times B

Examples

```
## Not run:
rcppeigen_fcrossprod(A, B)

## End(Not run)
```

rcppeigen_fdet *Fast Matrix Determinant*

Description

Computes using RcppEigen the determinant of A

Usage

```
rcppeigen_fdet(A)
```

Arguments

A	is the matrix whose determinant calculated

Value

determinant of A

Examples

```
## Not run:
rcppeigen_fdet(A)

## End(Not run)
```

`rcppeigen_fprod` *Fast Matrix Product*

Description

Computes using RcppEigen the product of A and B

Usage

```
rcppeigen_fprod(A, B)
```

Arguments

`A`	is the first parameter in A times B
`B`	is the second parameter in A times B

Value

matrix product A times B

Examples

```
## Not run:
rcppeigen_fprod(A, B)

## End(Not run)
```

`rcppeigen_fsolve` *Fast Matrix Inverse*

Description

Computes using RcppEigen the inverse of A

Usage

```
rcppeigen_fsolve(A)
```

Arguments

A	is the matrix being inverted

Value

inverse of A

Examples

```
## Not run:
rcppeigen_fsolve(A)

## End(Not run)
```

---

rcppeigen_ftcrossprod *Fast Matrix T-Cross-Product*

Description

Computes using RcppEigen the product of A and t(B)

Usage

```
rcppeigen_ftcrossprod(A, B)
```

Arguments

A	is the first parameter in A times t(B)
B	is the second parameter in A times t(B)

Value

matrix tcross-product A times t(B)

Examples

```
## Not run:
rcppeigen_ftcrossprod(A, B)

## End(Not run)
```

rcppeigen_ftrans *Fast Matrix Transpose*

Description

Computes using RcppEigen transposed of A

Usage

```
rcppeigen_ftrans(A)
```

Arguments

A	is the matrix being transposed

Value

transpose of A

Examples

```
## Not run:
rcppeigen_ftrans(A)

## End(Not run)
```

start_point *start point*

Description

Given $Ax \leq b$, which defines a convex polytope, this function picks n random starting "center" points using linear programming.

Usage

```
start_point(A, b, n = 1, average = 10)
```

Arguments

A	is the lhs of $Ax \leq b$
b	is the rhs of $Ax \leq b$
n	is the number of points we want to return
average	is the number of boundary points we want to take the average of

Value

a matrix, with each column as a point

Examples

```
A <- rbind(c(-1, 0), c(0, -1), c(1, 1))
b <- c(0, 0, 1)
start_point(A = A, b = b, n = 1, average = 10)
```

`walkr` *The walkr package.*

Description

Package	walkr
Type:	Package
Version:	0.3.1
Date:	2015-07-14
License:	GPL-3

Given $Ax = b$, `walkr` samples points from the intersection of $Ax = b$ with the n-simplex ($\sum x = 1$, $x_i \geq 0$). The Ax = b must be underdetermined, otherwise there is an unique solution and there will be no sampling.

Usage

```
walkr(A, b, points, method = "dikin", thin = 1, burn = 0.5,
  chains = 1, ret.format = "matrix")
```

Arguments

`A`	is the lhs of the matrix equation A
`b`	is the rhs of the matrix equation b
`points`	is the number of points we want to sample
`method`	is the MCMC sampling method. Please enter "hit-and-run", "dikin", or " optimized-dikin"
`thin`	every thin-th point is stored
`burn`	the first burn points are deleted
`chains`	is the number of chains we run
`ret.format`	is the format in which walkr returns the answer. Please enter "list" (of chains) or "matrix".

Details

The `walkr` package samples points using MCMC random walks from the intersection of the N-Simplex with M hyperplanes. Mathematically speaking, the sampling space is all vectors x that satisfy $Ax = b$, $\sum x = 1$, and $x_i \geq 0$. The sampling algorithms implemented are hit-and-run and Dikin Walk. `walkr` also provides tools to examine and visualize the convergence properties of the random walks.

The main function of the package is `walkr`. The user specifies A and b in $Ax = b$, and the `walkr` function samples points from the complete solution to $Ax = b$ intersected with the N-simplex. The user can choose either `"dikin"` or `"hit-and-run"` as the sampling method, and the function also provides other MCMC parameters such as thinning and burning.

Before the sampling, walkr internally performs the affine transformation which takes the complete solution of $Ax = b$ and that intersected with the unit simplex into a space parametrized by coefficients, which we call the "alpha-space". The specific set of procedures taken is written in detail in the vignette. Essentially, the space is transformed, the sampling takes place in the transformed space, and in the end `walkr` transforms back into the original coordinate system and returns the result. This transformation is affine, so the uniformity and mixing properties of the MCMC algorithms are not affected. The current MCMC sampling methods supported are "hit-and-run", "dikin" and "optimized-dikin" (a Rcpp boosted version for speed).

1) Hit-and-run is computationally less expensive and also guarantees uniformity asympotically with complexity of O(n^3) points with respect to dimension n. However, in real practice, as dimensions ramp up, the mixing of hit-and-run is poor compared to Dikin. Thus, a lot of thinning would be needed as dimension ramps up.

2) Dikin Walk is a nearly uniform method known for its very strong mixing properties. However, each Dikin step is much more computationally expensive than hit-and-run, so it takes more time to sample every point. Thus, the "dikin" method uses RcppEigen to speed up the core computationally expensive operations in the algorithm.

Value

Either a list of chains (with each chain as a matrix of points) or a matrix containing all the points. Each column is a point sampled.

Examples

```
## 4D constraint
A <- matrix(c(2,0,1,3), ncol = 4)
b <- 0.5
sampled_points <- walkr(A = A, b = b, points = 100, method = "dikin")
```

Index