Package ‘sumR’

May 9, 2026

Type Package

Title Approximate Summation of Series

Version 0.4.16

Date 2025-11-27

Maintainer Guido A. Moreira <guidoalber@gmail.com>

Description Application of theoretical results which ensure that the summation of an infinite discrete series is within an arbitrary margin of error of its true value. The C code under the hood is shared through header files to allow users to sum their own low level functions as well. Based on the paper by Braden (1992) <doi:10.2307/2324995>.

License GPL (>= 3)

Encoding UTF-8

RoxygenNote 7.3.2

Imports matrixStats

NeedsCompilation yes

Author Guido A. Moreira [cre, aut] (ORCID: <https://orcid.org/0000-0001-7557-0874>), Luiz Max Carvalho [ctb] (ORCID: <https://orcid.org/0000-0001-5736-5578>)

Repository CRAN

Date/Publication 2025-11-28 13:10:11 UTC

Contents

finiteSum 2
infiniteSum 3
infiniteSum_batches 6
precompiled 8
summed-objects 10

Index **12**

`finiteSum` *Calculates the sum of a discrete series with a single maximum for a pre-set number of evaluations*

Description

A discrete finite series is calculated exactly with no need for approximations. This can also be used for infinite series approximation with a pre-determined number of iterations, but this has no guarantee of quality of approximation. Result is returned in the log scale.

Usage

```
finiteSum(logFunction, parameters = numeric(), n, n0 = 0)
```

Arguments

`logFunction`	The function that returns the series value a_n in the log scale. Can either be an `R` function or a string indicating one of the precompiled functions. See `precompiled()` for a list of available functions. If defined in `R`, the function's definition must have two arguments. The first argument must be the integer argument equivalent to n in a_n and the second must be a vector of numeric parameters.
`parameters`	A numeric vector with parameters used in logFunction. Vectorized summation over various parameter values sets is not implemented. Use `apply()` or their variants to achieve this.
`n`	A single integer positive number indicating the number of iterations to perform in the function.
`n0`	The sum will be performed for the series starting at this value.

Value

A `summed-objects()` object. Note that the sum is returned in the `log` scale.

See Also

`precompiled()` provides a list with precompiled functions that can be used for the summation.

Examples

```
# Sum values from 5 to 100
finiteSum(function(x, p) log(x), numeric(), 100 - 5, 5)
```

`infiniteSum` *Approximates the sum of a positive discrete infinite series with a single maximum*

Description

For series that pass the ratio test, the approximation is analytically guaranteed to have an error that is smaller than epsilon. This can occasionally not happen due to floating point arithmetic. Result is returned in the log scale.

Usage

```
infiniteSum(
  logFunction,
  parameters = numeric(),
  logL = NULL,
  alternate = FALSE,
  epsilon = 1e-15,
  maxIter = 1e+05,
  n0 = 0,
  forceAlgorithm = 0
)
```

Arguments

`logFunction`	The function that returns the series absolute value $\lvert a_n \rvert$ in the log scale. If it is an alternating series, this is defined in argument `alternate`. Can either be an `R` function or a string indicating one of the precompiled functions. See `precompiled()` for a list of available functions. If defined in `R`, the function's definition must have two arguments. The first argument must be the integer argument equivalent to n in a_n and the second must be a vector of numeric parameters.
`parameters`	A numeric vector with parameters used in `logFunction`. Vectorized summation over various parameter values sets is not implemented. Use `apply()` or their variants to achieve this.
`logL`	The log of the limit value of a_{n+1}/a_n which must be smaller than 1, or smaller than 0 in the log scale. Ignored if the series is alternating, defined with argument `alternate`. If left as `NULL` and `logFunction` is defined in `R`, the `batches` algorithm with default settings is used. See 'details'.
`alternate`	Either -1, 0 or 1. If 0 (or `FALSE`), the series is not alternating and positive. Otherwise, the series is alternating where the first element's sign is either 1 or -1, as entered in this parameter. If not 0, arguments `logL` and `forceAlgorithm` are ignored.
`epsilon`	The desired error margin for the approximation. See 'details'.

`maxIter`	The maximum number of iterations for the approximation. In most cases, this number will not be reached unless it is very small. A value too high is not recommended as an array of this size is reserved in memory during the algorithm.
`n0`	The sum will be approximated for the series starting at this value for the first parameter of `logFunction`.
`forceAlgorithm`	A value to control which summation algorithm to use. Ignored if the series is alternating, defined with argument `alternate`. See 'details'.

Details

The approximated sum is based on some theoretical results which, analytically, guarantee that the approximation will be within `epsilon` distance to the true value. It is possible that the numerical result fails to fall in this distance due to floating point arithmetic. The `C` code under the hood is being continuously reviewed to minimize this problem. They seem to occur more often when the series decays very fast to zero or when the total is a large number.

For these theoretical results to work, the series must pass the ratio test, which means that the ratio a_{n+1}/a_n must converge to a number $L < 1$ when n goes to infinity. The log of L should be provided to the function for a better approximation. This is not necessary in case a precompiled function is used. In this case the value of L is coded into the package.

Another requirement in the current installment of this function is that the series must have only a single maximum. This is the case for most discrete probability distributions and marginalization problems. This limitation will be addressed in the future.

There are currently two implemented algorithms that perform the calculations. The first, called Sum-To-Threshold, sums the series values until the series values are smaller than `epsilon`. This is the fastest algorithm, but it is only guaranteed to provide an approximation within the desired error margin when $L < 0.5$.

The second algorithm, called Error bounding pairs is based on a more general result which works for any $0 \leq L < 1$. This algorithm sums the series until

$$\left| \frac{a_{n+1}}{1-L} - \frac{a_{n+1}a_n}{a_n - a_{n+1}} \right| < 2\epsilon.$$

Then the approximation is the added values of the sum plus

$$0.5(\frac{a_{n+1}}{1-L} + \frac{a_{n+1}a_n}{a_n - a_{n+1}}).$$

The Error bounding pairs method usually requires less function evaluations than the Sum-To-Threshold one, however the convergence checking is more demanding, which means that it is typically slower, albeit slightly. If $L = 0$, the convergence checking can be reduced and the Error bounding pairs becomes almost as fast as the Sum-To-Threshold method.

The third algorithm is called batches method and is used when L is left at `NULL`. Its use requires some fine tuning, so there is a standalone function for it called `infiniteSum_batches()`. Its use and functionality can be seen in its own documentation. When called as a result of this function, default settings are used.

The `forceAlgorithm` parameter can be used to control which algorithm to use. When it is 0, the program automatically selects the Sum-to-threshold when $L < 0.5$ and the Error bounding pairs

when $0.5 \leq L < 1$. Method batches is selected when L is left NULL. If forceAlgorithm is set to 1, the Sum-To-Threshold algorithm is forced. If it is 2, then the Error bounding pairs is forced. A small note, the Error bounding pairs algorithm can go up to maxIter + 1 function evaluations. This is due to its convergence checking dependence on a_{n+1}. Finally, if the parameter is set as 3, the batches algorithm is used with default settings.

If the series is alternating, the Sum-To-Threshold convergence condition on the series absolute value guarantees the result, regardless of the ratio limit.

The function to be summed can be an R function or a string naming the precompiled function in the package. The list of precompiled functions can be found in precompiled(), and more functions will be added in time. As is intuitive, using a precompiled function is much faster than using an R function. In fact, it has been observed to be dozens times faster.

The advanced user can program their own precompiled functions and use the package's summation algorithms by linking the appropriate header file. See the GitHub readme for the a quick tutorial.

Value

A summed-objects() object. Note that the sum is returned in the log scale.

See Also

precompiled() provides a list with precompiled functions that can be used for the summation. infiniteSum_batches() is an alternate method which does not require knowledge of the logL argument.

Examples

```
## Define some function that is known to pass the ratio test.
param <- 0.1
funfun <- function(k, p) return(k * log1p(-p[1]))
result <- infiniteSum(funfun, parameters = param, logL = log1p(-param))

## This series is easy to verify analytically
TrueSum <- -log(param)
TrueSum - result$sum
# Since exp(logL) = 0.9, the Error bounding pairs
# algorithm is used. Notice that it only required
# 2 function evaluations for the approximation, that is
result$n

## A common problem is finding the normalizing constant for the
## Conway-Maxwell-Poisson distribution. It has already been included
## in the precompiled list of functions.
comp_params <- c(lambda = 5, nu = 3)
result <- infiniteSum("COMP", comp_params)
result
```

`infiniteSum_batches` *Approximates the sum of a positive discrete infinite series with a single maximum using the batches algorithm*

Description

A simple method to perform the summation. It adds the values in batches and stops when the accumulated batch is smaller than the desired threshold. There is an implementation purely in R and one in C. The one in R is usually slightly faster due to vectorized computing. Result is returned in the log scale.

Usage

```
infiniteSum_batches(
  logFunction,
  parameters = numeric(),
  batch_size = 40,
  epsilon = 1e-15,
  maxIter = 1e+05,
  n0 = 0
)

infiniteSum_batches_C(
  logFunction,
  parameters = numeric(),
  batch_size = 40,
  epsilon = 1e-15,
  maxIter = 1e+05,
  n0 = 0
)
```

Arguments

`logFunction`	The function that returns the series value a_n in the log scale. Can either be an R function or a string indicating one of the pre-coded functions. See `precompiled()` for a list of available functions. If defined in R, the function's definition must have two arguments. The first argument must be the integer argument equivalent to n in a_n and the second must be a vector of numeric parameters.
`parameters`	A numeric vector with parameters used in logFunction. Vectorized summation over various parameter values sets is not implemented. Use `apply()` or their variants to achieve this.
`batch_size`	The batch size at which point convergence checking is performed. The algorithm perform at least twice this number of function evaluations. See 'details'.
`epsilon`	The desired error margin for the approximation. See 'details'.
`maxIter`	The maximum number of iterations for the approximation. In most cases, this number will not be reached unless it is very small.
`n0`	The sum will be approximated for the series starting at this value.

Details

The series a_n must pass the ratio convergence test, meaning that the ratio a_{n+1}/a_n must converge to a number $L < 1$ when n goes to infinity.

The batches algorithm consists of evaluating the function a fixed number of times for two checkpoints. If the difference between the sum at these checkpoints is smaller than `epsilon`, the code stops and the later checkpoint sum is returned. Else, continue summing until the next checkpoint. All checkpoints are `batch_size` long.

This function's efficiency is reliant on the choice of `batch_size`. If it is set too large, the algorithm overshoots the necessary number of function evaluations too much. If it is set too small, the algorithm will need to process too many partial summations which slows it down. However, if they are well calibrated for the series, they can potentially be very efficient.

Since the batch sizes are known before the calculations are made, function evaluations can be vectorized. This is why there are two functions available. `infiniteSum_batches` does the calculations at the `R` level, while `infiniteSum_batches_C` interfaces the low level `C` code. However, the `C` code does not use vectorization since it isn't available on long double precision type, and therefore the `R` level function should be faster in most cases.

Another difference is that the low level code uses double precision for the calculations. This means that it is less prone to rounding errors. But this also means that the two functions can sometimes require a different number of iterations and function evaluations to reach the stop criteria. This is shown in the examples.

Another requirement in the current installment of this function is that the series must have only a single maximum. This is the case for most discrete probability distributions and marginalization problems. This limitation will be addressed in the future.

Value

A `summed-objects()` object. Note that the sum is returned in the `log` scale.

See Also

`precompiled()` provides a list with precompiled functions that can be used for the summation. `infiniteSum()` is a more efficient algorithm.

Examples

```
## Define some function that is known to pass the ratio test.
param = 0.1
funfun <- function(k, p) return(k * log1p(-p[1]))
result <- infiniteSum_batches(funfun, parameters = param)

## This series is easy to verify analytically
TrueSum = -log(param)
TrueSum - result$sum
# Notice that it required 400 function evaluations for the approximation.
result$n

# If we use the C function, it reaches a lower error, but requires more
# iterations
```

```
result_C <- infiniteSum_batches_C(funfun, parameters = param)
TrueSum - result_C$sum
result_C$n

## A common problem is finding the normalizing constant for the
## Conway-Maxwell-Poisson distribution. It has already been included
## in the precompiled list of functions.
comp_params = c(lambda = 5, nu = 3)
result <- infiniteSum_batches("COMP", comp_params)
# With a specifically chosen argument value, the summation can be done with
# fewer iterations. But it is usually hard to know the ideal choice for
# applications beforehand
result$n
infiniteSum_batches("COMP", comp_params, batch_size = 11)$n
# A small batch_size ensures a small number of iterations, but slows the
# method due to multiple checking.
infiniteSum_batches("COMP", comp_params, batch_size = 2)$n
```

`precompiled` *List of precompiled functions in the sumR package*

Description

More functions are periodically added to this list for convenience and speed. These functions are all evaluated in the log scale and pass the ratio test, that is, the limit of a_{n+1}/a_n as n goes to infinity is a value $0 \leq L < 1$. The value of L is indicated in each entry. It is calculated automatically when the precompiled functions are used in the summation.

Conway-Maxwell-Poisson normalizing constant

This series is the kernel of the Conway-Maxwell-Poisson distribution, which generalizes the Poisson and Geometric distributions. Its form is

$$a_n = \frac{\lambda^n}{(n!)^\nu},$$

$$L = 0, log(L) = -\infty,$$

for $\lambda > 0$ and $\nu > 0$.

When $\nu = 1$, this series reduces to the Poisson distribution kernel and the sum (in the log scale) is known to be λ. When $\nu = 0$ and $0 < \lambda < 1$, the series reduces to the Geometric distribution kernel with parameter $1 - \lambda$. The series is known to sum to 1. Finally, as ν goes to ∞ the distribution approaches a Bernoulli distribution with parameter $\lambda/(1 - \lambda)$.

Another known result is when $\nu = 2$, in which case the sum is the modified Bessel function of the first kind of order 0 evaluated at $2\sqrt{\lambda}$.

- String to access the precompiled function: `"COMP"`.
- parameter vector: `c(lambda, nu)`.

Double Poisson normalizing constant

This series is the kernel of the double Poisson distribution, which is a special case of the double exponential family, which extends it. Its form is

zwXYBvkBdj7cKoNB4JNit2jw7atSlIxA

$$L = 0, log(L) = -\infty,$$

for $\lambda > 0$ and $\phi > 0$.

When $\phi = 1$, this series reduces to the Poisson distribution kernel and the sum (in the log scale) is known to be 0.

- String to acccess the precompiled function: `"double_poisson"`.
- parameter vector: `c(mu, phi)`

Modified Bessel function of the first kind

This is the series form solution for the function. There are more time efficient methods for its evaluation however they don't guarantee good approximations with large parameters. Its form is

LPkucubB065u8qnbMphsm3DX68fYfIE1-1

$$L = 0, log(L) = -\infty,$$

for $x > 0$ and α any real value.

The modified Bessel function of the second kind can be obtained with

$$K_\alpha(x) = \frac{\pi}{2} \frac{I_{-\alpha}(x) - I_\alpha(x)}{\sin \quad \alpha\pi},$$

where I represents the modified function of the first kind and K of the second kind. It is worth remembering the `infiniteSum()` function returns the sum in the log scale, which must be adjusted for the formula above.

- String to access the precompiled function: `"bessel_I"`
- parameter vector: `c(x, alpha)`

Modified Bessel function of the first kind with log argument

This is the same function as the one above, except that parameter x is given in the log scale. This is provided for numerical stability. For the cases where `x` is not very large, sums using this function and the above should return the same sum.

- String to access the precompiled function: `"bessel_I_logX"`
- parameter vector: `c(logx, alpha)`

Note

In some cases, the sum of the series is known in closed form for some values of the parameters. The package function does not check for these cases and just performs the approximation. If the exact value is desired by the user when it is known, they must take responsibility for checking and providing these values.

Another important thing to note is that the precompiled functions perform all calculations with twice the numerical precision than R. Therefore, in some cases, there might be very small differences in the sum when comparing the results of the function using the precompiled function and the same function defined at the R level.

See Also

`infiniteSum()`, `finiteSum()` and `infiniteSum_batches()`

---

summed-objects *S3 Class for objects containing iterated summations*

Description

Contains the summations in the log scale. The value can either be an approximation to an infinite series or a finite sum.

Usage

```
## S3 method for class 'summed'
print(x, ...)

## S3 method for class 'summed'
as.double(x, ...)
```

Arguments

`x`	The `summed` object.
`...`	Currently unused.

Value

For `print`: The invisible object.

For `as.numeric`/`as.double`: The approximated sum.

Elements in the list

`sum` The resulting sum in the log scale.

`n` The performed number of iterations. This value represents the number of series elements evaluations performed during the summation.

`method` The method used for the summation.

`maxReached` `TRUE` or `FALSE`. Indicates whether the maximum iterations was reached.

See Also

`infiniteSum()`, `infiniteSum_batches()` and `finiteSum()` for available methods.

Index

apply(), *2, 3*, *6*
as.double.summed *(summed-objects)*, 10

finiteSum, 2
finiteSum(), *10, 11*

infiniteSum, 3
infiniteSum(), *7*, *9–11*
infiniteSum_batches, 6
infiniteSum_batches(), *4, 5*, *10, 11*
infiniteSum_batches_C
(infiniteSum_batches), 6

precompiled, 8
precompiled(), *2, 3*, *5–7*
print.summed *(summed-objects)*, 10

summed-objects, 10