Package: RMSNumpress (via r-universe)

September 7, 2024

Type Package

Title 'Rcpp' Bindings to Native C++ Implementation of MS Numpress

Version 1.0.1

Date 2021-02-04

Description 'Rcpp' bindings to the native C++ implementation of MS Numpress, that provides two compression schemes for numeric data from mass spectrometers. The library provides implementations of 3 different algorithms, 1 designed to compress first order smooth data like retention time or M/Z arrays, and 2 for compressing non smooth data with lower requirements on precision like ion count arrays. Refer to the publication (Teleman et al., (2014) <doi:10.1074/mcp.O114.037879>) for more details.

License BSD_3_clause + file LICENSE

Imports Rcpp (>= 1.0.3)

LinkingTo Rcpp

Suggests testthat

RoxygenNote 7.0.2

NeedsCompilation yes

Author Justin Sing [cre, aut], Johan Teleman [aut]

Maintainer Justin Sing <justincsing@gmail.com>

Date/Publication 2021-02-04 17:20:09 UTC

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

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

RemoteRef HEAD

RemoteSha a8c2264f99326a2c0ff785fd501634b135b26b27

Contents

`RMSNumpress-package` *Rcpp bindings to native C++ implementation of MS Numpress*

Description

MS Numpress

==========

Implementations of two compression schemes for numeric data from mass spectrometers.

The library provides implementations of 3 different algorithms, 1 designed to compress first order smooth data like retention time or M/Z arrays, and 2 for compressing non smooth data with lower requirements on precision like ion count arrays.

Numpress Pic

==========

MS Numpress positive integer compression

Intended for ion count data, this compression simply rounds values to the nearest integer, and stores these integers in a truncated form which is effective for values relatively close to zero.

Numpress Slof

==========

MS Numpress short logged float compression

Also targeting ion count data, this compression takes the natural logarithm of values, multiplies by a scaling factor and rounds to the nearest integer. For typical ion count dynamic range these values fits into two byte integers, so only the two least significant bytes of the integer are stored.

The scaling factor can be chosen manually, but the library also contains a function for retrieving the optimal Slof scaling factor for a given data array. Since the scaling factor is variable, it is stored as a regular double precision float first in the encoding, and automatically parsed during decoding.

Numpress Lin

==========

MS Numpress linear prediction compression

This compression uses a fixed point representation, achieve by multiplication by a scaling factor and rounding to the nearest integer. To exploit the assumed linearity of the data, linear prediction is then used in the following way.

The first two values are stored without compression as 4 byte integers. For each following value a linear prediction is made from the two previous values:

Xpred = (X(n) - X(n-1)) + X(n)

Xres = Xpred - X(n+1)

The residual Xres is then stored, using the same truncated integer representation as in Numpress Pic.

The scaling factor can be chosen manually, but the library also contains a function for retrieving the optimal Lin scaling factor for a given data array. Since the scaling factor is variable, it is stored as a regular double precision float first in the encoding, and automatically parsed during decoding.

Truncated integer representation

==========

This encoding works on a 4 byte integer, by truncating initial zeros or ones. If the initial (most significant) half byte is 0x0 or 0xf, the number of such halfbytes starting from the most significant is stored in a halfbyte. This initial count is then followed by the rest of the ints halfbytes, in little-endian order. A count halfbyte c of

0 <= c <= 8 is interpreted as an initial c 0x0 halfbytes

9 <= c <= 15 is interpreted as an initial (c-8) 0xf halfbytes

Examples:

int c rest

0 => 0x8

-1 => 0xf 0xf

23 => 0x6 0x7 0x1

Author(s)

Maintainer: Justin Sing <justincsing@gmail.com>

References

See: https://github.com/ms-numpress/ms-numpress

See Also

encodeLinear, decodeLinear, encodeSlof, decodeSlof, encodePic, decodePic, optimalLinearFixedPoint, optimalSlofFixedPoint, optimalLinearFixedPointMass,

Examples

```
## Not run:
  # Encode Numpress Linear
  ## Retention time array
  rt_array <- c(4313.0, 4316.4, 4319.8, 4323.2, 4326.6, 4330.1)
```

```
 ## encode retention time array
 rt_encoded <- encodeLinear(rt_array, 500)
 #> [1] 40 7f 40 00 00 00 00 00 d4 e7 20 00 78 ee 20 00 88 86 23

 # Decode Numpress Linear
 ## Retention time data that is encoded with encodeLinear and is zlib compressed
 ### NOTE: For the sake of this example, I have broken the raw vector into several parts
 ###       to avoid Rd line widths (>100 characters) issues with CRAN build checks.
rt_raw1 <- c("78", "9c", "73", "50", "61", "00", "83", "aa", "15", "0c", "0c", "73", "80")
rt_raw2 <- c("b8", "a3", "5d", "fe", "47", "07", "84", "28", "fc", "8f", "c4", "40", "e5")
rt_raw3 <- c("61", "51", "84", "a9", "85", "08", "e1", "06", "00", "06", "be", "41", "cf")
## Add all character representation of raw data back together and convert back to hex raw vector
 rt_blob <- as.raw(as.hexmode(c(rt_raw1, rt_raw2, rt_raw3 )))
 ## Decompress blob
 rt_blob_uncompressed <- as.raw(Rcompression::uncompress( rt_blob, asText = FALSE ))
 ## Decode to rentention time double values
 rt_array <- decodeLinear(rt_blob_uncompressed)

## End(Not run)
```

`decodeLinear` *decodeLinear*

Description

Decodes data encoded by encodeLinear.

Usage

```
decodeLinear(data)
```

Arguments

`data`	pointer to array of bytes to be decoded (need memorycont. repr.)

Details

result vector guaranteed to be shorter or equal to (|data| - 8) * 2

Note that this method may throw a const char* if it deems the input data to be corrupt, i.e. that the last encoded int does not use the last byte in the data. In addition the last encoded int need to use either the last halfbyte, or the second last followed by a 0x0 halfbyte.

Value

the number of decoded doubles, or -1 if dataSize < 4 or 4 < dataSize < 8

See Also

[encodeLinear]

Examples

```
## Not run:
## Retention time data that is encoded with encodeLinear and is zlib compressed
### NOTE: For the sake of this example, I have broken the raw vector into several parts
###       to avoid Rd line widths (>100 characters) issues with CRAN build checks.
rt_raw1 <- c("78", "9c", "73", "50", "61", "00", "83", "aa", "15", "0c", "0c", "73", "80")
rt_raw2 <- c("b8", "a3", "5d", "fe", "47", "07", "84", "28", "fc", "8f", "c4", "40", "e5")
rt_raw3 <- c("61", "51", "84", "a9", "85", "08", "e1", "06", "00", "06", "be", "41", "cf")
## Add all character representation of raw data back together and convert back to hex raw vector
rt_blob <- as.raw(as.hexmode(c(rt_raw1, rt_raw2, rt_raw3 )))
## Decompress blob
rt_blob_uncompressed <- as.raw(Rcompression::uncompress( rt_blob, asText = FALSE ))
## Decode to rentention time double values
rt_array <- decodeLinear(rt_blob_uncompressed)

## End(Not run)
```

`decodePic` *decodePic*

Description

Decodes data encoded by encodePic

result vector guaranteed to be shorter of equal to |data| * 2

Usage

```
decodePic(data)
```

Arguments

`data`	pointer to array of bytes to be decoded (need memorycont. repr.)

Details

Note that this method may throw a const char* if it deems the input data to be corrupt, i.e. that the last encoded int does not use the last byte in the data. In addition the last encoded int need to use either the last halfbyte, or the second last followed by a 0x0 halfbyte.

Value

the number of decoded doubles

See Also

[encodePic]

decodeSlof *decodeSlof*

Description

Decodes data encoded by encodeSlof

The return will include exactly (|data| - 8) / 2 doubles.

Usage

```
decodeSlof(data)
```

Arguments

data	pointer to array of bytes to be decoded (need memorycont. repr.)

Details

Note that this method may throw a const char* if it deems the input data to be corrupt.

Value

the number of decoded doubles

See Also

[encodeSlof]

Examples

```
## Not run:
## Intensity array to encode
### NOTE: For the sake of this example, I have broken the intensity vector into several parts
###       to avoid Rd line widths (>100 characters) issues with CRAN build checks.
int_array1 <- c(0.71773432,  0.43443741,  1.71883610, 0.13220307,  0.90664242)
int_array2 <- c(0.00000000, 0.00000000,  0.64213755,  0.43443741, 0.47221479)
## Comcatenate into one intensity array
int_array <- c(int_array1, int_array2)
## Encode intensity array using encodeSlof
int_encode <- encodeSlof( int_array, 16 )

## End(Not run)
```

encodeLinear *encodeLinear*

Description

Encodes the doubles in data by first using a
- lossy conversion to a 4 byte 5 decimal fixed point representation
- storing the residuals from a linear prediction after first two values
- encoding by encodeInt (see above)

The resulting binary is maximally 8 + dataSize * 5 bytes, but much less if the data is reasonably smooth on the first order.

This encoding is suitable for typical m/z or retention time binary arrays. On a test set, the encoding was empirically show to be accurate to at least 0.002 ppm.

Usage

```
encodeLinear(data, fixedPoint)
```

Arguments

`data`	pointer to array of double to be encoded (need memorycont. repr.)
`fixedPoint`	the scaling factor used for getting the fixed point repr. This is stored in the binary and automatically extracted on decoding (see optimalLinearFixedPoint or optimalLinearFixedPointMass)

Value

the number of encoded bytes

See Also

[decodeLinear]

Examples

```
## Not run:
## Retention time array
rt_array <- c(4313.0, 4316.4, 4319.8, 4323.2, 4326.6, 4330.1)
## encode retention time array
rt_encoded <- encodeLinear(rt_array, 500)
#> [1] 40 7f 40 00 00 00 00 00 d4 e7 20 00 78 ee 20 00 88 86 23

## End(Not run)
```

`encodePic` *encodePic*

Description

Encodes ion counts by simply rounding to the nearest 4 byte integer, and compressing each integer with encodeInt.

Usage

```
encodePic(data)
```

Arguments

`data`	pointer to array of double to be encoded (need memorycont. repr.)

Details

The handleable range is therefore 0 -> 4294967294. The resulting binary is maximally dataSize * 5 bytes, but much less if the data is close to 0 on average.

Value

the number of encoded bytes

See Also

[decodePic]

`encodeSlof` *encodeSlof*

Description

Encodes ion counts by taking the natural logarithm, and storing a fixed point representation of this. This is calculated as

unsigned short fp = log(d + 1) * fixedPoint + 0.5

Usage

```
encodeSlof(data, fixedPoint)
```

Arguments

`data`	pointer to array of double to be encoded (need memorycont. repr.)
`fixedPoint`	fixed point to use for encoding (see optimalSlofFixedPoint)

Details

the result vector is exactly |data| * 2 + 8 bytes long

Value

the number of encoded bytes

See Also

[decodeSlof]

`optimalLinearFixedPoint` *optimalLinearFixedPoint*

Description

Compute the maximal linear fixed point that prevents integer overflow.

Usage

```
optimalLinearFixedPoint(data)
```

Arguments

`data`	pointer to array of double to be encoded (need memorycont. repr.)

Value

the linear fixed point safe to use

`optimalLinearFixedPointMass` *optimalLinearFixedPointMass*

Description

Compute the optimal linear fixed point with a desired m/z accuracy.

Usage

```
optimalLinearFixedPointMass(data, mass_acc)
```

Arguments

`data`	pointer to array of double to be encoded (need memorycont. repr.)
`mass_acc`	desired m/z accuracy in Th

Value

the linear fixed point that satisfies the accuracy requirement (or -1 in case of failure).

Note

If the desired accuracy cannot be reached without overflowing 64 bit integers, then a negative value is returned. You need to check for this and in that case abandon numpress or use optimalLinearFixedPoint which returns the largest safe value.

`optimalSlofFixedPoint` *optimalSlofFixedPoint*

Description

Compute the maximal natural logarithm fixed point that prevents integer overflow.

Usage

```
optimalSlofFixedPoint(data)
```

Arguments

`data`	pointer to array of double to be encoded (need memorycont. repr.)

Value

the slof fixed point safe to use

Index