# Package 'rabi'

January 24, 2019

**Type** Package

**Title** Generate Codes to Uniquely and Robustly Identify Individuals for Animal Behavior Studies

**Version** 1.0.0

**Description** Facilitates the design and generation of optimal color (or symbol) codes that can be used to mark and identify individual animals. These codes are made such that the IDs are robust to partial erasure: even if sections of the code are lost, the entire identity of the animal can be reconstructed. Thus, animal subjects are not confused and no ambiguity is introduced.

**License** GPL-3

**LazyData** TRUE

**VignetteBuilder** knitr

**RoxygenNote** 6.1.1

**Depends** R (>= 3.2.5)

**Imports** numbers, polynom, shiny, stringdist, stats, utils

**Suggests** knitr, rmarkdown

**NeedsCompilation** no

**Author** Andrew Taylor Burchill [aut, cre], Theodore P. Pavlic [ctb]

**Maintainer** Andrew Taylor Burchill <andrew.burchill@asu.edu>

**Repository** CRAN

**Date/Publication** 2019-01-24 06:10:03 UTC

## R topics documented:

---

`brute_IDs` *Brute force color coding scheme generator*

---

## Description

Generates "color" coding schemes used to mark and identify individual animals. The codes are robust to an arbitrary number of partial erasures. This method uses a sloppy, very slow, stochastic brute force method.

## Usage

```
brute_IDs(total.length, redundancy, alphabet, num.tries = 10,
  available.colors = NULL)
```

## Arguments

| | |
|---|---|
| `total.length` | the number of unique positions to be marked on the animal. (This can be thought of as the total number of positions on which color bands or paint marks will be applied.) |
| `redundancy` | the number of erasures that can occur without disrupting surety of unique identification. This value determines how robust the scheme is to erasures. |
| `alphabet` | an integer representing the 'alphabet size.' This is the number of unique markings (think different paint colors, symbols, or varieties of bands) at your disposal.Note: unlike the Reed-Solomon inspired function, `rs_IDs`, this function can take non-prime values. |
| `num.tries` | the number of iterations that will be run before choosing the best option. Increasing this number increases the running time. |
| `available.colors` | an optional list of strings that contains the names of the unique markings which compose the given 'alphabet' (e.g. "blue", "red", "yellow", etc.). If left blank, the mapping can be done at any later time using `codes_to_colors`. Additionally, the length of this list must match the 'alphabet size' given above. |

## Details

This function generates the list of all possible unique ID codes given the 'alphabet size' (`alphabet`) and the number of unique positions available for marking (`total.length`). The list of combinations is then iteratively pruned down until the required robustness has been reached; the distance between any two ID codes must greater than the value specified in `redundancy`.

However, the iterative pruning is done randomly, so it is likely that resulting list of codes does not contain the maximum possible number of robust codes. Thus, the process is repeated multiple times (`num.tries`) and the list that contains the largest number of robust codes is kept and returned.

## Value

a list of unique ID codes that fit the provided parameters.

If an appropriate argument for `available.colors` is provided, each code will be a sequence of strings, otherwise, each code will be a sequence of numeric values.

## Note

the `rs_IDs` function always generates the maximum number of unique codes per scheme. However, `rs_IDs` suffers from certain limitations that `brute_IDs` does not: it requires `alphabet` to be a prime number, `total.length` to be less than or equal to `alphabet`, etc.

## Author(s)

Andrew Burchill, <andrew.burchill@asu.edu>

## See Also

`rs_IDs`, `tweaked_IDs`, `simple_IDs`. Also see the vignette `loosebirdtag` for demonstrations and additional uses.

## Examples

```
total.length <- 6  #we have six positions to mark,
redundancy <- 2    #we want surety even with two erasures,
alphabet <- 4      #and we currently have five types of paint in stock

 #This gives a warning because rs_IDs() only accepts
 #a pretty constrained set of parameters
codes <- rs_IDs(total.length, redundancy, alphabet)
length(codes)

 #However, we can do it with brute_IDs() to get more unique IDs
codes <- brute_IDs(total.length, redundancy, alphabet, num.tries = 1)
length(codes)

 #Let's make those into human-readable color sequences
color.names <- c("blue","red","green","pink","yellow-with-a-stripe")
color.codes <- codes_to_colors(codes, color.names)
```

---

# codes_to_colors *Converting numeric ID codes to listed color name codes*

---

## Description

This is a helper function that transforms a list numeric ID sequences into a list of human-readable sequences. Sequences of 1s, 2s, and 3s can become sequences of "red"s, "blue"s, and "yellow"s, etc.

## Usage

```
codes_to_colors(codes, available.colors = NULL)
```

## Arguments

| | |
|---|---|
| codes | a list or matrix of numeric ID sequences generated by rs_IDs, brute_IDs, etc. |
| available.colors | a list of strings that contains the names of the unique markings which compose the given 'alphabet' (e.g. "blue", "red", "yellow", etc.). The length of this list must match the 'alphabet size' used to generate the input codes. |

## Value

a list of unique, named codes that fit the provided parameters.

## Author(s)

Andrew Burchill, <andrew.burchill@asu.edu>

## Examples

```
total.length <- 3  #we have three positions to mark,
redundancy <- 1    #we want codes robust to a single erasure,
alphabet <- 3      #and we currently have three types of paint in stock

 #Create a list of codes
codes <- rs_IDs(total.length, redundancy, alphabet)

 #Let's make those into human-readable color sequences
color.names <- c("blue","red","pink-striped-orange")
codes_to_colors(codes, color.names)

 #We can also skip the whole function and plug the names straight into the code generator
rs_IDs(total.length, redundancy, alphabet, available.colors = color.names)
```

---

# exampleGUI *A Shiny-based GUI for the rs_IDs function*

---

## Description

Launches a (possibly buggy) Shiny app that acts as a graphical user interface for the rs_IDs function. It's a bit hacky, so its performance is not guaranteed.

## Usage

```
exampleGUI()
```

### Author(s)

Andrew Burchill, <andrew.burchill@asu.edu>

### See Also

rs_IDs and the vignette loosebirdtag.

### Examples

```
## Not run:
exampleGUI()  #yeah, just run it.

## End(Not run)
```

---

how_many *Assistance with choosing ID scheme parameters*

---

### Description

Displays the maximum number of unique and robust IDs possible given various combinations of parameters used in the rabi package. Several tables, centered around the supplied inputs or the default values, are printed to help the user choose which set of physical parameters would be most useful in their study. This is based on the equation:

$$max\#ofIDs = alphabet^{total\_length - redundancy}$$

### Usage

```
how_many(total.length = 5, redundancy = 2, alphabet = 6)
```

### Arguments

| | |
|---|---|
| total.length | the desired number (or estimation) of unique positions to be marked on the animal. (This can be thought of as the total number of positions on which color bands or paint marks will be applied.) |
| redundancy | the desired number (or estimation) of erasures that can occur without disrupting surety of unique identification. This value determines how robust the scheme is to erasures. |
| alphabet | an integer representing the desired (or estimated) 'alphabet size.' This is the number of unique markings (think different paint colors, symbols, or varieties of bands) at your disposal. |

## Note

The rs_IDs function generates codes that have the maximum number of unique IDs; these are the theoretical values listed in the tables. However, rs_IDs has several restrictions on the parameter combinations it can accept. Asterisks ('*') are used in the table to indicate which values are a result of such illegal combinations. Other functions such as brute_IDs or simple_IDs can be used generate schemes from those particular parameter combinations, but they may fail to achieve the theoretical maximums listed in the table.

## Author(s)

Andrew Burchill, <andrew.burchill@asu.edu>

## See Also

how_robust.

## Examples

```
 #Let's generate some tables to see the number of unique IDs we could get given:
total.length <- 4  #we have ~4 positions to mark,
redundancy <- 1    #we're interested in being robust to a single erasure,
alphabet <- 5      #and we currently have 5 types of color bands in stock

how_many(total.length, redundancy, alphabet)
```

---

# how_robust *Quick method to see how robust a list of ID codes is to erasures*

---

## Description

Given a list (or matrix) of generated numeric ID codes, this function does a crosswise comparison. It compares the 'Hamming distance' between every pair of given ID sequences, then returns a contingency table with the frequency of Hamming distances found. These Hamming distances represent how robust the coding scheme is to erasure errors. If a particular robustness to erasure is desired, there should be no distances equal to or lower than that robustness.

## Usage

```
how_robust(codes)
```

## Arguments

| | |
|---|---|
| codes | a list of numeric ID sequences generated by rs_IDs, brute_IDs, or tweaked_IDs. This can be either in matrix or list form. |

## Value

a named, flattened list that contains a contingency table with the frequency of crosswise Hamming distances

## Author(s)

Andrew Burchill, <andrew.burchill@asu.edu>

## References

For information on Hamming distances.

For information on erasure coding.

## See Also

how_many.

## Examples

```
 #Let's generate some unique IDs given:
total.length <- 4  #we have four positions to mark,
redundancy <- 2    #we're interested in being robust to two erasures,
alphabet <- 5      #and we currently have five types of color bands in stock

codes <- rs_IDs(total.length, redundancy, alphabet)
 #Given that we specified a robustness of 2,
 #there should be no counts of "dist.2" or lower
how_robust(codes)
```

---

rabi *rabi: a package for generating* **R***obust* **A***nimal* **B***ehavior* **I***Ds*

---

## Description

This package facilitates the design and generation of color (or symbol) codes that can be used to mark and identify individual animals. These codes can be selected such that the IDs are robust to partial erasure: even if parts of the code are lost, the entire identity of the animal can be reconstructed Thus, animal subjects are not confused and no ambiguity is introduced.

## Details

Rigorous study of animal behavior, is often dependent on the researcher's ability to track and maintain the unique identity of individual animals or groups. Most individual animals are not reliably recognized on the intra-specific level. Thus, many methods for applying unique visual markings to animals have been developed and used. Many commonly used methods share a common element: a sequence of colors (or symbols, though for brevity and clarity we will refer to them as just 'colors').

Such color coding methods allow observers to conduct studies from a distance, even through binoculars. Color marking remains simple, cheap, and invaluable for fieldwork situations and human-based tracking. However, many external markers suffer from a lack of permanence: leg bands can be torn off by parrots, spots of dyed fur are often rubbed off, tags on turtle shells can be abraded, etc. The partial loss or visual obstruction of markings can disrupt identification by rendering two or more individuals virtually indistinguishable. Fortunately, careful selection of color coding schemes can affect how robust identification is to partial erasure. Even despite the lack of a rigorous method for generating sequences, researchers often use personal heuristics to select codes that they think have a lower chance of potentially being confused. However much better this may be than blind selection, they are far from optimal.

Drawing from tools found in the engineering field of signal processing , we describe a flexible methodology to create personalized color coding identification schemes that are robust to partial loss or obstruction: even if part of the code is missing, the entire unique sequence of colors can be reconstructed.

### Getting Started and Vignettes

See README for a quick dive into the package.

See the vignette for demonstrations and additional uses.

Run `exampleGUI` to use a Shiny-based GUI for creating ID schemes.

### Author(s)

Andrew Burchill: <andrew.burchill@asu.edu>

---

## rs_IDs *Polynomial color coding scheme generator*

---

### Description

Creates color (or symbol) coding schemes used to mark and identify individual animals using polynomial oversampling based on Reed-Solomon error-correction codes. The codes are robust to an arbitrary number of color-slot erasures.

### Usage

```
rs_IDs(total.length, redundancy, alphabet, available.colors = NULL)
```

### Arguments

| | |
|---|---|
| `total.length` | the number of unique positions to be marked on the animal. (This can be thought of as the total number of positions on which color bands or paint marks will be applied.) Note: Reed-Solomon coding requires the total length of the ID to be less than or equal to the value of `alphabet`. |
| `redundancy` | the number of erasures that can occur without disrupting surety of unique identification. This value determines how robust the scheme is to erasures. |

`alphabet` an integer representing the 'alphabet size.' This is the number of unique markings (think different paint colors, symbols, or varieties of bands) at your disposal. Note: Reed-Solomon coding requires this value to be a prime number. If a non-prime is entered, the function will automatically adjust it to the nearest previous prime.

`available.colors` an optional list of strings that contains the names of the unique markings which compose the given 'alphabet' (e.g. "blue", "red", "yellow", etc.). If left blank, the mapping can be done at any later time using `codes_to_colors`. Additionally, the length of this list must match the 'alphabet size' given above.

## Value

a list containing the maximum possible number of unique ID codes that fit the provided parameters.

## Author(s)

Andrew Burchill, <andrew.burchill@asu.edu>

## References

For information on Reed-Solomon error correction. For information on polynomial oversampling.

## See Also

`brute_IDs`, `tweaked_IDs`, `simple_IDs`. See the vignette `loosebirdtag` for demonstrations and additional uses. Run `exampleGUI` for a more user-friendly Shiny GUI version of the function.

If an appropriate argument for `available.colors` is provided, each code will be a sequence of strings, otherwise, each code will be a sequence of numeric values.

## Examples

```
total.length <- 6  #we have six positions to mark,
redundancy <- 2    #we want surety even with two erasures,
alphabet <- 5      #and we currently have five types of paint in stock

 #This gives a warning because rs_IDs() doesn't
 #allow 'total.length' to be larger than 'alphabet'
codes <- rs_IDs(total.length, redundancy, alphabet)
length(codes)

 #Now the output should be the same as above, but no warning is issued.
codes <- rs_IDs(total.length = 5, redundancy, alphabet)
length(codes)

 #Let's make those into human-readable color sequences
color.names <- c("blue","red","pink-striped-orange", "yellow", "green")
codes_to_colors(codes, color.names)
```

`simple_IDs` *Simple color coding scheme generator*

## Description

Creates a simple color (or symbol) coding scheme used to mark and identify individual animals. The sum of each IDs numeric sequence is a multiple of the number of colors used in the scheme. Even if one marking is removed, the entire ID code can be reconstructed.

## Usage

```
simple_IDs(total.length, alphabet, available.colors = NULL)
```

## Arguments

| | |
|---|---|
| `total.length` | the number of unique positions to be marked on the animal. (This can be thought of as the total number of positions on which color bands or paint marks will be applied.) |
| `alphabet` | an integer representing the 'alphabet size.' This is the number of unique markings (think different paint colors, symbols, or varieties of bands) at your disposal. |
| `available.colors` | an optional list of strings that contains the names of the unique markings which compose the given 'alphabet' (e.g. "blue", "red", "yellow", etc.). If left blank, the mapping can be done at any later time using `codes_to_colors`. Additionally, the length of this list must match the 'alphabet size' given above. |

## Value

a list containing the maximum possible number of unique ID codes that fit the provided parameters.

If an appropriate argument for `available.colors` is provided, each code will be a sequence of strings, otherwise, each code will be a sequence of numeric values.

## Author(s)

Andrew Burchill, <andrew.burchill@asu.edu>

## See Also

`rs_IDs`, `brute_IDs`.

## Examples

```
total.length <- 4  #we have four positions to mark
alphabet <- 5      #and we currently have five types of paint in stock

 #Generate codes where the sum of the sequence is a multiple of five
simple_IDs(total.length, alphabet)

 #Let's make those into human-readable color sequences
color.names <- c("blue","red","green","pink","cyan")
simple_IDs(total.length, alphabet, available.colors = color.names)
```

---

`tweaked_IDs` *Tweakable brute force color coding scheme generator*

---

## Description

Generates "color" coding schemes used to mark and identify individual animals, given a list of numeric sequences. The codes are robust to an arbitrary number of partial code erasures. This method uses a sloppy, slow, stochastic brute force method.

## Usage

```
tweaked_IDs(combos, redundancy, num.tries = 10,
  available.colors = NULL)
```

## Arguments

| | |
|---|---|
| `combos` | a list of numeric sequences or a matrix where each row is a unique sequence. The length of the sequences or the width matrix corresponds to the `total.length` variable seen in `rs_IDs`. The numeric elements should ideally be between zero and one less than the alphabet size (`0:(alphabet - 1)`) |
| `redundancy` | the number of erasures that can occur without disrupting surety of unique identification. This value determines how robust the scheme is to erasures. |
| `num.tries` | the number of iterations that will be run before choosing the best option. Increasing this number increases the running time. |
| `available.colors` | an optional list of strings that contains the names of the unique markings which compose the given 'alphabet' (e.g. "blue", "red", "yellow", etc.). If left blank, the mapping can be done at any later time using `codes_to_colors`. Additionally, the length of this list must match the 'alphabet size' given above. |

## Details

tweaked_IDs runs pretty much the same as brute_IDs. However, unlike brute_IDs, tweaked_IDs must be first given a list or matrix of acceptable ID sequences. Instead of randomly pruning down a list of ALL possible ID sequences, we can specify our constraints first and then generate the final ID scheme. This allows the user, in the face of some constraints, to potentially generate more unique IDs that otherwise available.

However, the iterative pruning is done randomly, so it is likely that resulting list of codes does not contain the maximum possible number of robust codes. Thus, the process is repeated multiple times (num.tries) and the list that contains the largest number of robust codes is kept and returned.

## Value

a list of unique ID codes that fit the provided parameters.

If an appropriate argument for available.colors is provided, each code will be a sequence of strings, otherwise, each code will be a sequence of numeric values.

## Note

This function is aimed at more advanced users. We would suggest using other functions to generate ID lists unless you are familiar with how the rabi package works.

## Author(s)

Andrew Burchill, <andrew.burchill@asu.edu>

## See Also

brute_IDs. Also see the vignette loosebirdtag for demonstrations and additional uses.

## Examples

```
alphabet <- 8      # the number of colors or symbols we have
total.length <- 5  # the number of positions we want mark
redundancy <- 2    # how many marks we can lose but still ID perfectly

  #Create a function for determining odd or even
 odd <- function(x){ x %% 2 == 1 }

  #Create a matrix of all possible sequences
perms <- rep(list(seq_len(alphabet)),total.length)
combos <- as.matrix(expand.grid(perms)) - 1
  #Only keep sequences that fit our constraints.
  #We want the first position to only be odd numbers
  #and the second position to only be even.
combos <- combos[which(odd(combos[,1]) & !odd(combos[,2])), ]
## Not run:
codes <- tweaked_IDs(combos, redundancy, num.tries = 1)


print(paste0("The 'tweaked' list contains ", length(codes), " unique IDs."))
```

```
## End(Not run)
```

# Index