Package ‘FunWithNumbers’

May 7, 2026

Type Package

Title Fun with Fractions and Number Sequences

Version 1.2

Date 2024-04-06

Description A collection of toys to do things like generate Collatz and other interesting sequences, calculate a fraction which is a close approximation to some value (e.g., 22/7 or 355/113 for pi), and so on.

License LGPL-3

LazyData FALSE

Imports Rmpfr, gmp, bigBits, methods

NeedsCompilation no

Author Carl Witthoft [aut, cre]

Maintainer Carl Witthoft <cellocgw@gmail.com>

Repository CRAN

Date/Publication 2024-04-07 02:32:57 UTC

Contents

`FunWithNumbers-package` *Fun with Fractions and Number Sequences*

Description

A collection of toys to do things like generate Collatz and other interesting sequences, calculate a fraction which is a close approximation to some value (e.g., 22/7 or 355/113 for pi), and so on.

Details

The DESCRIPTION file:

Package:	FunWithNumbers
Type:	Package
Title:	Fun with Fractions and Number Sequences
Version:	1.2
Date:	2024-04-06
Authors@R:	c(person(given = "Carl", family = "Witthoft", role = c("aut","cre"), email= "cellocgw@gmail.com"))
Description:	A collection of toys to do things like generate Collatz and other interesting sequences, calculate a fraction whic
License:	LGPL-3
LazyData:	FALSE
Imports:	Rmpfr, gmp, bigBits, methods
Author:	Carl Witthoft [aut, cre]
Maintainer:	Carl Witthoft <cellocgw@gmail.com>

Author(s)

NA

Maintainer: NA

`aliquot` *Generate the Aliquot Sequence.*

Description

Each term in the aliquot sequence is generated by summing all proper divisors of the previous term. The value "1" is included in this collection of divisors. In number theory, aliquot is closely related to terms such as "sociable" and "amicable" numbers

Usage

```
aliquot(x, maxiter = 100)
```

Arguments

x	An integer , bigz integer, or character string representing an integer to start the desired sequence
maxiter	Set a limit on the number of terms to calculate. See Details for reasons why to do so.

Details

While many aliquot sequences terminate in the values c(prime_number, 1, 0), many numbers drop into a short loop or a repeating value (perfect numbers do this). If the sequence repeats or terminates, the sequence is returned. If maxiter is reached, the output converged is set to FALSE and the sequence so far is returned.

Value

A list containing theseq, the sequence generated converged, a logical value indicating whether convergence (including cyclic) was reached. thecycle, the subset of theseq which is cyclic

Author(s)

Carl Witthoft, <carl@witthoft.com>

Examples

```
aliquot(20)
# $theseq
# Big Integer ('bigz') object of length 7:
# [1] 20 22 14 10 8  7  1
# $converged
# [1] TRUE
# $thecycle
# Big Integer ('bigz') :
# [1] 1
# (aliquot(1264460))
# $theseq
# Big Integer ('bigz') object of length 5:
# [1] 1264460 1547860 1727636 1305184 1264460
# $converged
# [1] TRUE
# $thecycle
# Big Integer ('bigz') object of length 5:
# [1] 1264460 1547860 1727636 1305184 1264460
```

`benprob` *Generate random numbers based on the Benford distribution*

Description

This function produces numbers whose distribution is based on Benford's Law of the occurrence of the values 1 through 9 in the first digit of numbers.

Usage

```
benprob(numsamp = 100, numbase = 10)
```

Arguments

`numsamp`	How many values to generate.
`numbase`	Specify the base system (binary, octal, decimal, or whatever is desired) in which to apply the Benford distribution. The default is "10," i.e. decimal.

Details

"Benford's Law," https://en.wikipedia.org/wiki/Benford%27s_law can be used to assess the "true" randomness of demographic data. Probably its most well-known use has been to detect fraudulent patterns in voting and investment returns claimed by various fund operators. The probability function is `prob(d) = log(d+1) - log(d)`, where `d` can take on the values 1:(log_base_in_use -1) .The data generated with this function can be used to calculate various statistics such as variance, skew, etc., which can then be compared with the real-world sample set being analyzed.

Value

A vector of random values.

Author(s)

Carl Witthoft, <carl@witthoft.com>

References

https://en.wikipedia.org/wiki/Benford%27s_law https://projecteuclid.org/euclid.ss/1177009869/

Examples

```
samps <- benprob(1000)
 sd(samps)
 hist(samps)
```

bestFrac *Generate a fraction close to the input value.*

Description

Inspired by the well-known approximations to pi, i.e. 22/7 and 355/113, this function allows the user to find the best-match fraction for any number, within the specified maximum magnitude of the numerator and denominator

Usage

```
bestFrac(x, intrange)
```

Arguments

x	A character string representing a number to be "converted" to a fraction of nearly equal value.
intrange	If a single value, the function tests all combinations of numerator and denominator between one and intrange . If two values, the 'testing range' is intrange[1]:intrange[2]. Otherwise, whatever vector of values is supplied will be used.

Details

For irrationals and the like, the simplest way to generate the input parameter string x is to use sprintf with as many digits to the right of the decimal point as desired. The returned values are in reduced form, i.e. the numerator and denominator are relatively prime.

Value

bestmatch	The numerator and denominator of the best-matching fraction
goodmatch	An N-by-2 array of the progressively better matches found (numerators and denominators in the columns)
matcherr	A vector of the differences between the 'matcherr' fractions and the input value. This is limited in precision to the machine limit for doubles (floats).

Author(s)

Carl Witthoft, <carl@witthoft.com>

Examples

```
gpi <- sprintf("%1.30f", pi)
bestFrac(gpi, 100)
# $bestmatch
# [1] 22  7
# $goodmatch
#          [,1] [,2]
```

```
# goodmatch    0    0
#              1    1
#              2    1
#              3    1
#             13    4
#             16    5
#             19    6
#             22    7
# $matcherr
# [1] 1.000000e+02 6.816901e-01 3.633802e-01 4.507034e-02 3.450713e-02
#     1.859164e-02 7.981306e-03 4.024994e-04
bestFrac(gpi, 100:400)
# $bestmatch
# [1] 355 113
# $goodmatch
#           [,1] [,2]
# goodmatch    0    0
#            100   31
#            100   32
#            101   32
#            104   33
#            107   34
#            110   35  # notice this is 22/7
#            179   57
#            201   64
#            223   71
#            245   78
#            267   85
#            289   92
#            311   99
#            333  106
#            355  113
# $matcherr
#  [1] 1.000000e+02 2.680608e-02 5.281606e-03 4.665578e-03 3.158429e-03
#      1.739936e-03 4.024994e-04
#  [8] 3.952697e-04 3.080137e-04 2.379631e-04 1.804857e-04 1.324752e-04
#      9.177057e-05 5.682219e-05
# [15] 2.648963e-05 8.491368e-08)
```

---

bpp — *Function which calculates pi, or other irrationals, using the Bailey-BorweinPlouffe formula ~~*

Description

THe BPP algorithm consists of a double summation over specified fractions. Rather than go into the gory details here, please refer to the link in the References section.

Usage

```
bpp(k,pdat = c(1,16,8,4,0,0,-2,-1,-1,0,0), init = 0, chunk = 1e4,...)
```

Arguments

k
: The number of terms in the series to calculate. Note that zero is a valid entry. If a single value, the terms `0:k` are used. If two values are provided (see information for the input parameter `init`), then the terms `k{1}:k[2]` are run.

pdat
: The parameter `P` which is used to define the coefficients used in all fractions in each term of the series. In brief, pdat contains the following BPP parameters: `pdat(s,b,m,A)` where `A`comprises all elements of the vector `pdat` after the first three. There are strict rules about the length of `A`; see the Details section. The default value will calculate pi.

init
: If there's a previous value calculated with `bpp` for a certain value of `k`, this term allows the user to continue the calculation. Assign the previous output's `bppgmp` value to `init` . Note that one must set up the input `k` to start at one more than the previous run's maximum "k" value.

chunk
: There is a call to `sum` in the main loop of this function. Use `chunk` to specify how many terms to pass to the sum call at a time, thus reducing the peak memory requirements of this function. The more RAM available on your machine, the larger this number can be. Set to a value greater than the argument `k` to run a single "chunk," which is the fastest approach if sufficient memory is available.

...
: Optional arguments to pass to `.bigq2mpfr` .

Details

The BPP algorithm calculates the sum(K=0,k, 1/(b^K) * FracSum) , where FracSum is defined by the sum(M=1,m, A[M]/(m*K + M)^s) . This means that the number of elements of `A` must equal `m`. Zero values are legal and are used to reject fractions not wanted in the inner sum. The default values for `pdat` correspond to the coefficients used to generate pi (the sum to infinity is mathematically equal to pi). Other values have been found to calculate a few other irrationals but there is as yet no known procedure to generate the `pdat` set for any given number.

Value

A list containing `bppgmp` , the `gmp` fraction calculated; `bppval`, the `mpfr` decimal representation of said fraction; and `kvals`, echoing the input `k`.

Author(s)

Carl Witthoft, `<carl@witthoft.com>`

References

https://en.wikipedia.org/wiki/Bailey-Borwein-Plouffe_formula and references cited there.

Examples

```
# Compare the decimal outputs to the first 130 digits of pi, which are:
#  [1] 3 . 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6 4
#  [26] 3 3 8 3 2 7 9 5 0 2 8 8 4 1 9 7 1 6 9 3 9 9 3 7 5
#  [51] 1 0 5 8 2 0 9 7 4 9 4 4 5 9 2 3 0 7 8 1 6 4 0 6 2
```

```
#  [76] 8 6 2 0 8 9 9 8 6 2 8 0 3 4 8 2 5 3 4 2 1 1 7 0 6
# [101] 7 9 8 2 1 4 8 0 8 6 5 1 3 2 8 2 3 0 6 6 4 7 0 9 3
# [126] 8 4 4 6 0

# Lots of precision, but most of the digits are inaccurate.
(bpp(5))

# extend the series.
(bpp(20))
```

`collatz` *Test the Collatz Conjecture. ~~*

Description

This function calculates the Collatz (aka Hailstone) sequence based on the selected starting integer.

Usage

```
collatz(x, div=2, mul=3, add= 1, maxKnown=1, maxiter = 1000)
```

Arguments

`x`	The integer, or `bigz` integer to start with.
`div`	The integer to divide by. Default is `2` per the original Collatz formula.
`mul`	The integer to multiply by. Default is `3` per the original Collatz formula.
`add`	The integer to add after multiply. Default is `1` per the original Collatz formula.
`maxKnown`	An integer to use as a "shortcut" if you know that said value converges. This allows the user to avoid repeating previous calculations. Default value is `1`, i.e. no previous knowledge of converging numbers.
`maxiter`	A "safety switch" to avoid possible lengthy runtimes (when starting with very very large numbers), terminating the function prior to convergence.

Details

The Collatz sequence follows simple rules: If the current number is even, divide it by two; else if it is odd, multiply it by three and add one. Convergence occurs in < 200 cycles for initial values < 10 million or so. Note: a serious Collatz generator would memoize previous successful sequences, thus greatly reducing the calculation time required to test new numbers. This function is provided "for amusement only."

Value

A vector of `bigz` integers representing the sequence, either to convergence or as limited by `maxiter`

Author(s)

Carl Witthoft, <carl@witthoft.com>

Examples

```
(collatz(20))
# 20 10 5  16 8  4  2
(collatz(234568))
# [1] 234568 117284 58642  29321  87964  43982  21991  65974  32987  98962
#  49481  148444 74222  37111
# [15] 111334 55667 167002 83501 250504 125252 62626 31313 93940 46970 23485
# 70456 35228 17614
# [29] 8807  26422 13211 39634 19817 59452 29726 14863 44590 22295 66886
# 33443 100330 50165
# [43] 150496 75248 37624 18812 9406  4703  14110 7055  21166 10583
#31750 15875 47626 23813
# [57] 71440 35720 17860 8930  4465  13396 6698  3349  10048 5024
#  2512  1256  628  314
# [71] 157  472  236  118  59   178  89   268  134  67   202  101  304  152
# [85] 76   38   19   58   29   88   44   22   11   34   17   52   26   13
# [99] 40   20   10   5   16   8   4   2
```

---

dontWorry — *Test for Happy Numbers ~~*

Description

This function calculates the sequence sum((digits_of_x)^p) to determine whether the value x is a "happy" number in the specified base.

Usage

```
dontWorry(x, pwr = 2, base = 10, maxiter = 100)
```

Arguments

x	The integer, bigz or character string representing an integer .
pwr	The exponent defining the power to which each digit is raised.
base	The base of the input x . Bases 2 through 36 are supported.
maxiter	A "safety switch" to avoid possible lengthy runtimes (when starting with very very large numbers), terminating the function prior to convergence.

Details

The dontWorry sequence follows simple rules: Take each digit in the current base and raise it to the specified power. Take the sum as the next input. If the sequence converges to 1, the number is happy. If the sequence ends in a cycle, the number is not happy.

Value

A list containing: isHappy: a logical value indicating whether or not x is happy given the specified base and power. theseq: the sequence of values found cyclic: TRUE if a cycle is found. (it should always be the case that cyclic = !isHappy)

Author(s)

Carl Witthoft, <carl@witthoft.com>

References

https://en.wikipedia.org/wiki/Happy_number

Examples

```
(dontWorry(20))
# $isHappy
#[1] FALSE
#$theseq
#[1] "20"  "4"   "16"  "37"  "58"  "89"  "145" "42"  "20"
#$cyclic
#[1] TRUE
(dontWorry('2254', base = 6))
# $isHappy
# [1] TRUE
#
# $theseq
# [1] "2254" "121"  "10"   "1"
#
# $cyclic
# [1] FALSE
```

---

factorion — *Find Factorion Numbers. ~~*

Description

This function calculates the factorion value for the specified input integer. A number is a 'factorion' if it equals the sum of the factorials of its digits in a given base.

Usage

```
factorion(x, base = 10 , findcycle = FALSE, maxiter = 100)
```

Arguments

`x`	The integer, or `bigz` integer, or character string representing an integer.
`base`	The the base of the input integer `x`. Bases 2 through 36 are supported. Default is 10.
`findcycle`	Logical value indicating whether or not to test for cyclic values. See Details for more information.
`maxiter`	A "safety switch" to avoid possible infinite loops, terminating the function prior to convergence. Only applicable when `findcycle` is TRUE.

Details

Most numbers are neither factorion nor part of a factorion cycle. A cycle occurs if the sequence `factorion(factorion(...factorion(x)))` repeats.

Value

A list containing: `x` echoing back the input argument. `isfactorion` value 1 if `x` is a factorion and 0 otherwise. `theseq` the sequence of values as each output is fed back to the input.

Author(s)

Carl Witthoft, `<carl@witthoft.com>`

References

https://en.wikipedia.org/wiki/Factorion https://oeis.org/A014080

Examples

```
(factorion(3,base=8,findcycle= TRUE))
# 3   6   1320   12; isit = 0
(factorion(41, base = 6))
# isit = 1
```

---

harshad   *Find Harshad Numbers ~~*

Description

This function tests whether the input is a Harshad number in the specified base. If the input divided by the sum of the digits (in the base) is an integer, then it is a Harshad number.

Usage

```
harshad(x, base = 10)
```

Arguments

x	The integer, bigz or character string representing the integer to test.
base	The base of the input x . Bases 2 through 36 are supported.

Details

The harshad sequence follows simple rules: Take the digits of a number in the specified base and sum them. Divide the input value by that sum. If the result is an integer, i.e. the sum divides the input, it's a Harshad number.

Value

A single item equal to one if the input is Harshad and equal to zero if not.

Author(s)

Carl Witthoft, <carl@witthoft.com>

References

https://en.wikipedia.org/wiki/Harshad_number

Examples

```
harshad(20)
# [1] 1
# 2 + 0 = 2, divides  20
harshad(37,base = 8)
#[1] 0
# 3 + 7 = 12 base8 , does not divide 37 base8 (in base 10, 10 does not divide 31)
```

---

juggatz *Function which calculates the "Juggler" sequence ~~*

Description

The "Juggler" sequence is similar to the Collatz sequence, but generates exponential changes rather than multiplicative changes to calculate each term. See Details for the algorithm.

Usage

```
juggatz(x, maxiter = 1000, prec = 100)
```

Arguments

x	The numeric, mpfr, or bigz integer to start with.
maxiter	A "safety switch" to avoid possible lengthy runtimes (when starting with very very large numbers), terminating the function prior to convergence.
prec	This specifies the number of binary digits of precision to use when the function converts numeric input x to a mpfr object.

Details

The Juggler algorithm uses the following rules: x[j+1] = floor(if even, x[j]^0.5; if odd x[j]^1.5). Since the mpfr-class objects represent approximations to the various powers and roots calculated, juggatz dynamically adjusts the number of bits of precision for the next value in the sequence. This ensures that the correct decision as to even or odd is made at each step.

Value

A vector of mpfr integers representing the sequence, either to convergence or as limited by maxiter

Author(s)

Carl Witthoft, <carl@witthoft.com>

Examples

```
 (juggatz(10))
# 8 'mpfr' numbers of precision  10 .. 100  bits
# [1] 10  3  5 11 36  6  2  1
(juggatz(37))
# 18 'mpfr' numbers of precision 10 .. 1000 bits
# [1]  37  225  3375  196069 86818724  9317
# [7]  899319 852846071 24906114455136 4990602  2233  105519
# [13] 34276462  5854  76  8  2  1
```

---

kaprekar — *Calculate the Kaprekar Sequence. ~~*

Description

This function calculates the Kaprekar sequence based on the selected starting integer.

Usage

```
kaprekar(x, base = 10, addZeros = FALSE, maxiter = 1000)
```

Arguments

- `x` — The integer, `bigz`, or character string representing the integer to start with.
- `base` — The base of `x`. Bases 2 through 36 are supported.
- `addZeros` — If a calculated term has fewer digits than the original `x`, `addZeros`, when TRUE, prepends zeros to maintain the number of digits.
- `maxiter` — A "safety switch" to avoid possible lengthy runtimes (when starting with very very large numbers), terminating the function prior to convergence.

Details

The kaprekar sequence follows simple rules: In the given base, sort the digits in ascending order. Reverse that order, and calculate the absolute value of the difference. Feed that result back to the algorithm. In some cases, the sequence will converge to a value which produces itself. In others, the sequence may fall into a repeating cycle.

Value

A list, containing: theseq: a vector of `bigz` integers representing the sequence, either to convergence or as limited by `maxiter` converged: a logical value indicating whether the sequence reached a value or a cycle. This is useful primarily if `maxiter` is reached and it's not immediately clear whether convergence occurred.

Author(s)

Carl Witthoft, <carl@witthoft.com>

References

https://en.wikipedia.org/wiki/Kaprekar's_routine

Examples

```
(kaprekar(2031))
# $theseq
# Big Integer ('bigz') object of length 5:
# [1] 2031 3087 8352 6174 6174
# $converged
# [1] TRUE

(kaprekar('0099'))
# $theseq
# Big Integer ('bigz') object of length 3:
# [1] 99 0  0
# $converged
# [1] TRUE

 (kaprekar('0099', addZeros = TRUE) )
# $theseq
# Big Integer ('bigz') object of length 6:
# [1] 99   9801 9621 8352 6174 6174
```

```
# $converged
# [1] TRUE
```

---

morris *Generate the Morris sequence*

Description

The Morris sequence, aka "Look-Say," is an old puzzler sequence.

Usage

```
morris(x, reps)
```

Arguments

x	Either a starting value from 1 to 9, or a numeric vector containing a Morris sequence previously generated.
reps	Specifies the number of new Morris sequences to generate, starting with the input x

Details

The Morris sequence is built by taking the verbal description of a number sequence and converting every number or named numeral to a number in order. Typically, starting with the integer 1, the spoken description is "One 1," so the next sequence is c(1,1). Read that out loud as "Two ones", so the next sequence is c(2,1) and so on.

Value

A list variable containing all the sequences generated as numeric vectors. ...

Author(s)

Carl Witthoft, <carl@witthoft.com>

pdi *Calculate Perfect Digital Invariant ~~*

Description

This function calculates PDI, defined as the sum of a number's digits raised to a given power.

Usage

```
pdi(x, pwr = 2, base = 10)
```

Arguments

x	The bigz integer to start with.
pwr	The power to which digits are raised. Default is 2.
base	The base of the source integer. Note that calculations within this function are performed in base 10. Default is 10.

Details

This is a helper function for dontWorry. Use at your own risk.

Value

The PDI of the input value and base, as a bigz integer.

Author(s)

Carl Witthoft, <carl@witthoft.com>

preciseNumbersAsChar *High-precision values for some common constants, in character strings.*

Description

These are provided for use when playing around with some of the functions in this package, e.g., bestFrac or cfrac

Details

These represent, in order, "e" (natural log base), the golden ratio (1+sqrt(5))/2 aka "phi", "pi", and the square root of 2 as generated via rmpfr with 10 000 binary bits of precision. There are many websites which can provide upwards of a million decimal digits for these constants for those who are interested.

Author(s)

Carl Witthoft, <carl@witthoft.com>

`sptable` *Calculate the number of unique values in the cross-table of sums and products for the input set of numbers*

Description

This function tests the proposition that the sum of all unique values in the cross-table of sums and products for a set of N input values is "close" to N^2 .

Usage

```
sptable(x)
```

Arguments

`x`	A vector of integer values.

Value

.

`uniqsum`	vector of the unique values of the outer sum `outer(x,x,'+')`

.

`uniqprod`	vector of the unique values of the outer product `outer(x,x)`

.

`spratio`	The ratio `uniqsum/uniquprod`
`exponentOfN`	The (numeric) solution to N^(exponentOfN) = uniqsum+uniqprod. If Erdos is right, this will always be "close" to 2.

Author(s)

Carl Witthoft, <carl@witthoft.com>

References

This conjecture is discussed in https://www.quantamagazine.org/the-sum-product-problem-shows-how-addition-

Examples

```
(sptable(1:10))
# $uniqsum
# [1] 19
# $uniqprod
# [1] 42
# $spratio
# [1] 0.452381
# $exponentOfN
# [1] 1.78533
set.seed(42)
sptable(sample(1:100,20,rep=FALSE))
# $uniqsum
# [1] 123
# $uniqprod
# [1] 202
# $spratio
# [1] 0.6089109
# $exponentOfN
# [1] 1.930688
```

---

vaneck *Generate a sequence 'invented' by Jan Ritsema Van Eck*

Description

This function generates an interesting (to the author, at least) sequence listed as number A181391 in the https://oeis.org/. See Details for a full description.

Usage

```
vaneck(howlong = 100, ve = NULL, ...)
```

Arguments

howlong	How many terms to generate.
ve	Optional argument. Enter a previously generated ("VanEck") sequence here as a numeric vector, or a single integer to use as an initiator.
...	reserved for possible future use.

Details

The rule here is that you start with 0, and whenever you get to a number you have not seen before, the following term is a 0. But if the number k has appeared previously in the sequence, then you count the number of terms since the last appearance of k, and that number is the following term. In more detail:

Term 1: The first term is 0 by definition. Term 2: Since we havent seen 0 before, the second term is 0. Term 3: Since we have seen a 0 before, one step back, the third term is 1 Term 4: Since we havent seen a 1 before, the fourth term is 0 Term 5: Since we have seen a 0 before, two steps back, the fifth term is 2. And so on. As of this release of this R-package, how fast max(sequence) grows, and whether every number eventually appears, are open questions. The latest investigations and theorems related to this sequence can be found at https://oeis.org/A181391/

Value

ve	The vector (ve for "VanEck") of the sequence values calculated
uniqs	a vector of the unique values in ve

Author(s)

Carl Witthoft, <carl@witthoft.com>

References

https://oeis.org/A181391/

Examples

```
(vaneck(20))
# $ve
#  [1] 0 0 1 0 2 0 2 2 1 6 0 5 0 2 6 5 4 0 5 3 0
# $uniqs
# [1] 0 1 2 6 5 4 3
```

Index