PANGRANDOMONIUM : Rhizome Function Anagram Random Generator

Raze Buckbriar

2026.07.24 | octade.net | sci.crypt | archive.org/details/@buckbriar

[doi : 10.5281/zenodo.21551292]

Herein are described principles and a scheme of random number generation styled as, *pangrandomonium,* or '*pangrand*'. Pangrand is a anagrammatic random number generator that uses rhizome functions as chaotic indexing and shuffling instructions to reorder and sum perfect pangrams. Brief descriptions are given for the principles of pangram summing and rhizome indexing functions, Source code of the random number generator is included herewith.

Pangrams

A pangram is a sentence or phrase that contains each letter of an alphabet or character set at least once. A perfect pangram is an anagram of the alphabet or character set with exactly one occurrence of each character.

By combining pangrams with a rhizome function to shuffle them we can produce a stream of random outputs using seed data as index swapping instructions.

Rhizome Functions

Function iteration is described mathematically as:

$f^n(x)$ ::: where $f^5(x)$ would be expanded to ::: f(f(f(f(x)))) ,

which in pseudocode or Pascal language is ::: f[f[f[f[f[x]]]]] .

This is analogous to the computer science method of chained indirect addressing, a technique sometimes used in short chains for forward and backward memory lookups and addressing data trees or tables.

Rhizome function is a descriptive term of art for a chaotic and deep-walking form of chained addressing or iterated functional addressing or mapping. The rhizome method is similar to function iteration in a directed graph over a series of successor function values. A rhizome function also iterates over multiple graphs, indexing from one graph to another in a chain.

In a rhizome function a series of randomized walk procedures are chained through a series of array indexes, with or without a parallel value transform function. Multiple array indices are referenced by using the first index as a pointer to the next index, then using the value in the second index as a pointer to the third index, and so on. Chains are of arbitrary length suitable to a particular purpose and can be as long as the number of graph indexes. These index traversals may be done on a single array or across a set of arrays. The indexes may be used at

their face value or randomly altered in each step. Pictured below is a simple anatomy of a rhizome function:

```
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7        [ array index ]
---------------------------
7 | 5 | 3 | 4 | 6 | 0 | 2 | 1        [ index value ]


            3407                     [ instruction seed ]

3 → 4 ; 4 → 6 ; 6 → 2 ; 2 → 3    (clone pointer 3 is 2 ; clone index 3 is 6)
4 → 6 ; 6 → 2 ; 2 → 3 ; 3 → 4    (clone pointer 4 is 3 ; clone index 4 is 2)
0 → 7 ; 7 → 1 ; 1 → 5 ; 5 → 0    (clone pointer 0 is 5 ; clone index 0 is 1)
7 → 1 ; 1 → 5 ; 5 → 0 ; 0 → 7    (clone pointer 7 is 0 ; clone index 7 is 5)
```

The final walk index that points to the beginning entry index is the 'pointer node'. The pointer node is the chain link reached one step before a walk terminates its cycle at the root node's entry value, which is the same as the first pointer in the indexing chain. The permutation cycle node is the clone node. The node pointing to the cycle exit is the clone pointer node. The walks can be incremental and mixed with transformations, preventing or lengthening traversal to the end of a permutation cycle.

I chose the name, 'rhizome function' as a term of art because each array index is like a node in the spreading tendril of a botanical rhizome or fungal hyphae of rhizomorphs. The longer the chain is, the more likely the chain will cross back over one if its prior index values or its initial entry value (reproduce). This point is called the clone node. The walks through a set or graph are picturesque of a rhizome spreading beneath the soil to send up new shoots from nodes.

Rhizome Hardness Problems

In walking a small perfect pangram graph or array set we can calculate the identity pointer or clone pointer node without much difficulty. In walking a large graph or set the difficulty can increase by use of duplicate values, missing values, and transformation rules, or by using non-pangram value sets. Difficulty can be increased even more by using a transform function that changes or sums the values of indexes at each link in the chain of graph hopping steps. Such transforms may be crafted to increase the hardness more than linearly. Combining multiple sets and worming through them can add even more computational difficulty.

One usable problem is calculating the computational difficulty for a single clone node value in a pangram set or a non-pangram set. The difficulty could be greater for non-pangram sets as perfect pangram sets always have a clone node exit when walked without any transformation rules.

A similar problem is to determine, given a certain index transformation rule, whether or not a random graph has a rhizome clone node at all for a particular starting index. With a perfect pangram such a node always exists. With other kinds of sets and transforms this can be formed into a computationally hard problem.

Another problem is determining which subset combination of graphs within a set can be combined to find a node via walking the graphs.

Another problem is to determine the average length of chains for every clone exit value in a set.

Another problem is combining a subset of multiple sets to determine which combination will produce a predetermined walk length for an exit or clone node. Similarly the problem can be modified to determine which combination will reach a determined value from a determined chain length.

Another property of the rhizome method is that a state set of values can be repositioned without changing the actual values—by using rhizome walks to derive instructions for value pair swaps. This gives a combinatorial problem that increases the difficulty of correlating the hidden state from outputs generated by the state. Each additional walk in the chain increases the mathematical difficulty of the problem.

Searching a rhizome black box to find bias or predictability in the positioning of values is another problem to explore. Proving whether or not such an oracle could exist is an analogue to this problem.

A guesser trying to reconstruct a hidden rhizome state oracle would input known values to the black box and try correlating the outputs with clone nodes or other values to statistically reconstruct the internal state.

Random Generator Methods

Pangrand shuffles multiple randomized pangram permutations of a character set. Random seed data is used as index instructions to shuffle the arrays of pangram characters. After shuffling the multiple arrays are compressed by adding together to a single character array. This sum is the random output.

This pangrand method is simple to demonstrate and grasp. Consider first a sequence set as the initial pangram alphabet:

```
perfect pangram of base 8 mod 8 : 0 1 2 3 4 5 6 7
```

Use a seed of integers as instructions to shuffle two or more pangram sets. Take two shuffled sets of this pangram and add the columns MOD 8 to form a new set:

```
   7  0  4  5  3  1  6  2        [ array x ]
 +  5  7  0  3  4  6  2  1        [ array y ]
---------------------------
 =  4  7  4  0  7  7  0  3        [ random z ]
```

Many combined orderings of the pangrams will sum to the same output. More often than not the sum will not be a pangram. In large, randomly-shuffled pair sets the sums will in all probability never form a pangram in the lifespan of any shuffling machine.

For ease of coding the pangram sets can be of identical size and values, as long as each set repeats no character or value within itself. The modulus divisor must be within the range of 1 greater than the highest value in the set and 2 greater than the lowest value in the set.

Identical lengths are not required in pangram pairs or groups. If the pangrams are of different sizes, the smallest shuffled pangram must receive additions from the larger pangram(s) to form the output sum and the larger pangrams must contain all the values present in the smallest. Also the differing values must be very close to the lowest and highest values of the smallest pangram set, ideally no greater than double the greatest value of the smaller set. Too much biased range in the difference of sets could introduce biases that diminish the randomness of the output. In such unequal sets the modulus divisor must always be in the range of one greater than the top value of the smallest set and 2 greater than its lowest value.

An example of unequal pangrams (different lengths) would be one set with the values from 0 to 255 and another set with the values from 0 to 127. After shuffling both pangrams, the smaller pangram may be aligned randomly with the larger pangram anywhere between the first and last characters. Then additions of half the characters of the larger set would be done to transform the values of the smaller set into the output set. Each sum would be divided MOD 128 for the output. The smaller set could be aligned multiple times to receive additions as in this figure of base 16 hex sums:

```
6 | 1 | F | E | 8 | 7 | 5 | A | 4 | 0 | 3 | B | 2 | C | 9 | D

                    4 | 0 | 7 | 2 | 6 | 5 | 1 | 3
                    -----------------------------
                    B   5   1   6   6   8   C   5

6 | 1 | F | E | 8 | 7 | 5 | A | 4 | 0 | 3 | B | 2 | C | 9 | D

            4 | 0 | 7 | 2 | 6 | 5 | 1 | 3
            -----------------------------
            2   8   E   7   0   9   1   6
```

For simplicity the enclosed pangrand RNG software does not employ this method. Instead it uses rhizomes to randomly shift additions from one column to another. The principle is similar and randomizing the index pairings better mixes the output entropy.

Any number of sets greater than two may be summed to produce output. This method lends facility to sums of a two-dimensional array or matrix pair. A pair of two dimensional matrices may be aligned perpendicular to each other. One matrix would be considered as rows, and the other matrix as columns, and then the two matrices could be combined with overlay coordinate summation.

There are countless ways to tailor methods of combining shuffled pangram sets. I leave the remainder of such to the reader to imagine.

The distribution quality of pangrand randomness is a function of the weight of the values and their orderings. The tailored use of anagramming obscures input biases very shortly. Since

every value in the range of output values is present exactly twice it should be hard to analyze outputs to discern or predict prior or forward bias.

Below is purely procedural source code written in the FreePascal programming language. The code is structured to facilitate visual reference for study. Explanations and instructions are commented copiously in the source code. A plain text Pascal source file is embedded in the PDF document. Additional resource files are also embedded in the PDF file. These files can be extracted using the menu of a PDF reader or command-line tools.

Errata

```
PANGRANDOMONIUM is a chaotic portmanteau neologism combining:

    pangram   |   mania      |   pandemonium
    random    |   dominion   |   domino
    grand     |   minion     |   anagram

yielding a fund of portmanteaus:

    grandom      |   randomania       |   randominion
    pangrandom   |   pangrandomania   |   pangrandomonium
    pangrand     |   anagrandom       |   randomino
```

Source Code

The user may save the PDF or postscript file as plain text to extract the source code. Or the user may export the embedded source file contained within the PDF document. Do not copy and paste the following code, as formatting errors may prevent compilation.

Linux command line tools may also be used to extract the source file from the PDF document. The application 'pdftk' using the 'unpack_files' switch should suffice. Here is an example of using 'pdftk' to extract all files from the PDF file:

```
$> pdftk {filename}.pdf unpack_files output {destination/folder/path}
```

The source code provided below is designed to be read as a reference. The code is copiously commented to further illustrate the methods of mixing with rhizome functions and randomizing inputs to a stream of unbiased random data.

```
{ ////////////////////////////////////////////////////////////////////////// }

program pangrandomonium_5050 ;

{ +++ Copyright (c) 2026, Raze (OCTADE). +++ https://octade.net +++ }

{ Copious comments in the code describe the operations below. }

{ %%% Test vectors are neither provided nor possible.
      This demo version prohibits reproducible output. %%% }

{ Output: stream of random byte characters. }
{ Arch: Unix-like, 64-bit. }
{ Compiling : 'fpc -O4 [filename]'. }
{ Installing : Copy to $PATH directory. }
{ Usage : Cat or pipe output to target. }

{$INFO >    (PG5050) PANGRANDEMONIUM RNG DEMO   +----------------+   }
{$INFO >                                        |     OCTADE     |   }
{$INFO >     This software is only a demo.      +----------------+   }
{$INFO >     It is for research use only.       |    sci.crypt   |   }
{$INFO >     No warranty. No guarantee.         +----------------+   }
{$INFO >     Use entirely at your own risk.     |   octade.net   |   }
{$INFO >                                        +----------------+   }

{ PANGRANDOMONIUM is a chaotic portmanteau neologism combining:

    pangram   |   mania      |   pandemonium
    random    |   dominion   |   domino
    grand     |   minion     |   anagram

yielding a fund of portmanteaus:

    grandom      |   randomania       |   randominion
    pangrandom   |   pangrandomania   |   pangrandomonium
    pangrand     |   anagrandom       |   randomino

_ yeet _ ! }

{ ////////////////////////////////////////////////////////////////////////// }

INTRODUCTION

  This random number generator mixes unpredictable timestamp entropy into a
  pool by using the timestamp bytes as generators for shuffling three arrays
  of pangram bytes. Each pangram contains the bytes 0 to 255 shuffled randomly.
  After much shuffling two arrays are added together for output. One array is
  used as the seed for the next output.

  512 clock bytes are converted into 256 bytes of output in each round. The
  generator values are derived from an accumulator pair of 128 bits total.

  This method is useful because the shuffled bytes never change value. They just
  change their positions in the arrays. Using the shuffled data as instructions
  for transforming the input data guarantees a very good distribution of
  random instruction values. Then we use the transformed input as instructions
  to repeat the process on the shuffled data arrays. Moreover, if we add the
  bytes across the two pangram alphabets, the output can't be correlated to
  the secret values. A huge number of random array states yield same output of
  one block yet will not match on the following block of output.

  Since output values are dependent on pangram positions, and changing two value
  positions affects all forward transformations, it is mathematically hard for
  an attacker to manipulate internal state to a desired ordering. One change is
  likely to form a algebraic road block to further calculated changes of the
  pangram orderings.

  This demo code is for compiling on Linux OS. It might not compile on
  Windows, Mac, or other operating systems.

The main idea is simple:

    0. Begin with two pangram arrays, each containing one occurrence of each byte from 0 to
255.
    2. No duplicate bytes should exist in a single array.
    3. Shuffle each array independently using accumulator inputs as indexes.
    4. Shuffle each array rhizomally, using samples from one array to shuffle the other.
    5. Add the values from each index of the pangram pair to a third array.
    6. The third array is the random output.

NOTES

0. PANGRANDOMONIUM (pangrand for short) is a anagramming random generator.
1. Pangrand consists of a entropy mixer that shuffles generator values.
```

2. Pangrams can be used for cryptology (hashes, generators, mixers, combiners).
3. Pangramming and rhizoming are useful for de-biasing a input stream.
4. Combinatorial scattering via rhizomes provides strong decorrelation.
5. Pangram fields can produce a uniform random distribution of random values.
6. The pangramming mixer inhibits leakage of timestamp information.
7. Instead of fast key erasure, pangrandom does constant key overwrite erasure.
8. Constant key overwrite is the importation of 64 bits of entropy for each round.
9. Constant key overwrite ensures more entropy enters than leaves the state.
10. Therefore pangrand is constantly compressive.

NOVELTIES

0. Herein is introduced the concept of the 'rhizome function'.
1. A rhizome function is an indexing system or vector that traverses through multiple sets using each set index as an index to the next set or index in the chain. It is called rhizome being reminiscent of hidden nodes randomly spreading out hidden in the soil.

ATTACKS

To guess the output stream an attacker must:

0. Guess the first 1.5 megabytes of combined RDRAND and RDSEED data.
1. Guess the 5050 bits of internal shuffle state.
2. Guess the sequences of timestamps (calculate the clock jitter.)
3. Guess the seed array state.
4. Do some spooky action at a distance (perhaps on a shared host).

HOW PANGRAND WORKS (GENERALIZED SIMPLISTIC REDUCTION)

These instructions present a bare minimum, simplified version. See the source code below for simple examples of how to add bells and whistles. Better or faster generators can be built on these principles. The demo tries to keep things simple for educational purposes while still providing good randomness.

Implementers might keep in mind that a very simple version can still produce good random data ; and much complexity could introduce disastrous, hidden bugs.

0. Set 2 arrays each containing a pangram of all the unique values 0 to 255.
1. The starting order of the values can be considered a secret key or seed.
2. Set a 3rd array of 256 null bytes.
3. Set two quad words (128 bits) as accumulators via rotations, addition, and cubing.
4. Initialize the generator with a large number of loops > 16 times output block and hidden pangram size.
5. After initializing with random seed data continue inputs with 64-bit timestamps (or other seed source).
6. Whiten each timestamp by adding a large prime number of size double word or triple word.
7. Whiten each timestamp by cubing the result of the prime addition.
8. Whiten each timestamp by taking a modulus of a smaller prime (2 to 3 bytes), adding the modulus to the timestamp.
9. Rotate and add the cubed and whitened timestamp to the other quad word accumulator.
10. Set the secondary quad word accumulator as a sequence of 8 bytes.
11. Each byte is an array index for the pangram arrays.
12. Using these index bytes, instruct rhizome functions to swap bytes around in the pangram arrays.
13. After many iterations of time stamps and pangram swaps add the pangram arrays together to a new array.
14. Output the characters of the summed array as random data.
15. Null the output array for the next round, or set it to equal another secret pangram array.

// }

```
{$asmmode intel}

CONST
    forever : string = '4eva' ;

VAR
    _COUT_ : file of char ; { a CHAR is a burnt byte }
    _BOUT_ : Integer ;

    w, x, y, z : array [0..255] of byte ; { pangram shufflers + combiner }

    _yeet_ : array [0..255] of char absolute z ;

    WQ, RQ, TSQ : QWORD ; { big words for random indexing }

    BQ : array [0..7] of byte absolute WQ ; { address WQ as array }

    { address BQ array bytes as single variables }
    g0 : byte absolute BQ[0] ;
    g1 : byte absolute BQ[1] ;
    g2 : byte absolute BQ[2] ;
    g3 : byte absolute BQ[3] ;
    g4 : byte absolute BQ[4] ;
```

```
    g5 : byte absolute BQ[5] ;
    g6 : byte absolute BQ[6] ;
    g7 : byte absolute BQ[7] ;

    i : dword ; { LOopER }
    b : byte absolute i ; { reduce looper to byte (array length) }
    s : byte ; { SwApPeR }


PROCEDURE QSEED;
    var
        seed_iter : byte ;
begin;
    seed_iter := 0 ;
    while seed_iter < 1 do begin ;
    asm
        rdseed RAX
        mov RQ, RAX
    end;
    if RQ > 0 then inc(seed_iter, 1) ;
end ; { seed_iter }
end; { QSEED }


PROCEDURE QRAND;
begin;
    asm
        rdrand RAX
        add qword [WQ], RAX
    end;
end;


PROCEDURE QTIME ;
begin;
{ The VPS/host should have access to RDTSCP else (((crash))). }
{ If the host blocks RDTSC access re-factor the code to accept another seed stream source or
entropy broker. }
    asm
        rdtsc
        shl   rdx,   32
        or    rax,   rdx
        add   qword  [TSQ],  RAX
    end;
    TSQ += 9999991111111 ;
    TSQ := TSQ * TSQ * TSQ ; {  * & replenish ... cubic chaos reigns }
    TSQ += (TSQ mod 199999991) ; { addressed to the nines }
    WQ += RolQword(TSQ, (x[g0])) + RorQword(TSQ, (y[g1])) ;
end;

PROCEDURE PANGRANDOMONIUM ;
{ transform 512 bytes of clock values into 256 bytes of output via 5050 bits (631 bytes)
combinatorial sieve }
{ By selecting generator bytes from low order slots there are fewer nulls. }
{ The combination of RDRAND + RDTSC values guarantees starting with 5050 bits of true entropy.
}
{ Delete the QRAND instruction and increase loops to run on a host lacking RDRAND access. }
begin;

    { These rhizome functions worm through multiple arrays. }
    { This makes it harder for an attack to craft inputs to manipulate state. }
    { Also, each pangram array having 256 unique values makes attack harder. }
    { An attacker can try only to change the positions of values, not the values themselves. }

    { shuffle combo seed for next loop }
    s := w[x[y[g6]]] ; w[x[y[g6]]] := w[x[y[g7]]] ; w[x[y[g7]]] := s ;

    { storming swapperoni - 5050 combinatorial bits (w, x, y) }
    s := x[y[w[g0]]] ; x[y[w[g0]]] := x[y[w[g1]]] ; x[y[w[g1]]] := s ;
    s := y[w[x[g2]]] ; y[w[x[g2]]] := y[w[x[g3]]] ; y[w[x[g3]]] := s ;

    { shuffle the additive output array }
    s := z[w[x[y[g4]]]] ; z[w[x[y[g4]]]] := z[w[x[y[g5]]]] ; z[w[x[y[g5]]]] := s ;

    { sum the random pangram bytes to the already random output array }
    { pangram values are never exposed - obfuscating internal state }
    { many combos of 3 bytes are equally likely to produce the same output }

    { compress 3 hidden bytes to one clandestine character }
    z[b+000] += byte ( x[y[g0]] + y[x[g1]] ) ;
    z[b+064] += byte ( x[y[g2]] + y[x[g3]] ) ;
    z[b+128] += byte ( x[y[g4]] + y[x[g5]] ) ;
    z[b+192] += byte ( x[y[g6]] + y[x[g7]] ) ;
```

```
    { mix in 16 more random bytes - little rhizomes getting their tendrils in the nodes and
crannies }
    z[g0] += byte ( x[y[g0]] + y[x[g7]] ) ;
    z[g1] += byte ( x[y[g1]] + y[x[g6]] ) ;
    z[g2] += byte ( x[y[g2]] + y[x[g5]] ) ;
    z[g3] += byte ( x[y[g3]] + y[x[g4]] ) ;
    z[g4] += byte ( x[y[g4]] + y[x[g3]] ) ;
    z[g5] += byte ( x[y[g5]] + y[x[g2]] ) ;
    z[g6] += byte ( x[y[g6]] + y[x[g1]] ) ;
    z[g7] += byte ( x[y[g7]] + y[x[g0]] ) ;

    { reflect back x, y to the accumulator }
    g0 += x[g7] ;
    g1 += y[g6] ;
    g2 += x[g5] ;
    g3 += y[g4] ;
    g4 += x[g3] ;
    g5 += y[g2] ;
    g6 += x[g1] ;
    g7 += y[g0] ;

end; { PANGRANDOMONIUM }


BEGIN

{ initialize arrays with a clean slate }
fillchar (w, SizeOf(w), #0) ;
fillchar (x, SizeOf(x), #0) ;
fillchar (y, SizeOf(y), #0) ;
fillchar (z, SizeOf(z), #0) ;

{ start the accumulators with some N.U.M.S. (Nothing Up My Sleeve Numbers) }
WQ  := QWORD ($C0CAC01AC1A551C  + $C0A1B1ACCAD111AC + $60BB1E600DC0FFEE) ;
TSQ := QWORD ($5C00B1ED00B1ED00 + $CAD111ACE5CA1ADE + $D16E571B1ED16175) ;
RQ := 0 ;

{ build the byte pangram alphabets with unique modular orderings }
for i := 0 to 255 do begin ;
    { although some entropy shuffles in to _z_ don't count it }
    { x,y state values remain hidden behind a maze in a storm }
    w[i] := byte (i*119) ;
    x[i] := byte (i*191) ;
    y[i] := byte (i*911) ;
    z[i] := byte (i*919) ;
end ; { i }

{ initial mix to stack up entropy }
{ for safety margin assume floor of 0.1 bits of entropy per clock read }
for i := 0 to 65535 do begin ;
    QRAND;
    QSEED; WQ += RQ ;
    QSEED; TSQ += RQ ;
    QTIME ;
    PANGRANDOMONIUM ;
end ; { i }

{ open standard output for writing }

ASSIGN (_COUT_, '') ;
REWRITE (_COUT_, SizeOf(char));

while forever = '4eva' do begin ; { stamina activated }

QRAND ; { Trickle in 64 bits of TRNG stream }

for i := 0 to 63 do begin ;
    QTIME ;
    PANGRANDOMONIUM ;
end ; { i }

{ little words ; big mouth ; _yeet_! into the void }
BLOCKWRITE (_COUT_, _yeet_, SizeOf(_yeet_), _BOUT_) ;

z := w ; { drop z and replace with seed for next loop }

end ; { forever }

CLOSE (_COUT_) ; { 4eva out of reach }

END.
```