Content

1. dm_random_test to test non-randomness of data matrices for contingency tables.

2. $\boxed{\text{Essence}}$

Contents:

dm_random_test	function to test non-randomness of data matrices for contingency tables
contab	function that generate a contingency table from a two column data matrix
dm_random_test_DEMO	function to demonstrate dm_random_test
Demonstration	some application of dm_random_test_DEMO

Precondition: Set of G_$\star$-functions must be available.

Date: 21/7/1993

P.Naeve, B. Strohmeier, H.P. Wolf

3. $\boxed{\text{Statistical Problem}}$

Question

Are there any hints of non-randomness in the sample?

Explanation and Refinement

Many statistical procedures assume that the data are collected at random. Such a sample is called a simple random sample. If the assumption of randomness does not hold it is very difficult to interpret the computed results. Therefore, it is necessary to check the input data for randomness. Unfortunately the difficulty of this check increases with the number of dimensions of the problem (columns of the data matrix).

A data matrix that leads to a two-way contingency table cannot easily checked by eye. So a procedure has to be searched for.

4. Solution

Statistical solution—idea

Some types of non-randomness can be discovered by computing characteristics of subsets of the data matrix. We propose to construct a statistic T summarizing the differences between subsets and the whole data set by a χ^2-approach. To decide whether the empirical value of this statistic is extrem an approximation of it's distribution must be computed.

Computer science solution—informal algorithm

Four essential steps are easily fixed by some top level annoucements.

<*test non-randomness of a data matrix for a contingency table*>=
<*construct the contingency table from the input data matrix*>
<*compute the statistic* T>
<*find an approximate distribution of* T>
<*show the result*>

Remark

If you exchange the word contingency table by a *linguistic* variable you see that the four steps listed above characterize a very general procedure which is often useful in statistics.

More precise statistical solution—using terms of computer science

χ^2-approaches belong to the standard toolkit of a statistician and are widely used. So we propose the following algorithm:

1. Construct a contingency table with relative frequencies p_{ij} for cell (i, j).
2. Divide the whole data matrix with n entries in n_b batches of size $n_k = [n/n_b]$.
3. Construct for each batch it's contingency table with absolute frequencies $n_{k.ij}$ and compute the χ^2-statistic T_k.
$$T_k = \sum_i \sum_j \frac{(n_{k.ij} - n_k p_{ij})^2}{n_k p_{ij}}$$
4. Aggregate the T_k.
$$T = \sum T_k$$
5. Find an approximation of the distribution of T by simulation.
This can be performed by generating many permutations of the entries of the data matrix followed by the computation of associated T values. (For T is not a sum of independent random variables you cannot conclude that T has a χ^2-distribution.)
6. Exhibit the result.

Computer science solution—using WEB*-style*

5. $\boxed{\text{Syntax of dm\_random\_test}}$

The *APL2*-function dm_random_test should do the job.

∇ result←nbatches_nruns dm_random_test dm 5

See also section 7.

dm_random_test appears in sections 2, 7, 42, 45, 49, and 53.

6. $\boxed{\text{Data structures of arguments and result}}$

nbatches_nruns	The algorithm needs fixed values for the number of batches and permutations. These numbers are given by the left argument nbatches_nruns. If no left argument is found default values will be chosen: nbatches_nruns←(⌈n⋆.5),100 where n is the number of rows of the data matrix. If nbatches_nruns has only one element this will be interpreted as the number of batches; then the number of runs will be defined by the default value 100.
dm	The right argument dm is the data matrix. Rows of dm represent sample elements. It is assumed that dm consists of two columns; this condition should fall in later versions of the function.
result	result shows the relative position of the input data matrix in the set of the simulated situations according to the statistic T. A high value of result is an indication for non-randomness.

7. Definition of dm_random_test

Now we have to link the idea outlined above with the function header. We provide a place for initializations—for example to rearrange the input and define default values.

```
∇+ result←nbatches_nruns dm_random_test dm   5
⟨ set default values, initialize dm_random_test   30 ⟩
⟨ test non-randomness of a data matrix for a contingency table   8 ⟩
```

8. Next we rewrite the four steps of the informal algorithm in a more technical way.

```
⟨ test non-randomness of a data matrix for a contingency table   8 ⟩ ≡
⟨ construct the contingency table from the input data matrix   28 ⟩
⟨ compute the statistic T   32 ⟩
⟨ find an approximate distribution of T   34 ⟩
⟨ show the result   41 ⟩
```

This code is used in section 7.

9. What has to be done.

The announcements of the top level descriptions has to be replaced by elaborated ideas that are more detailed and at the end can directly be translated into pure *APL*2-code.

10. The function contab.

Before discussing further refinements we want to concentrate on a special problem that has to be solved in any case: How can we get the contingency table from the data matrix?

Asking Berni he replied: "This problem has been solved by the function contab. Just a moment please! Here we are!"

```
Oo←a contab o;Oy;Ox;x;y;ox;oy;o0x;o0y;O0x;O0y;oO;oo;OO;Oo_o
Oy←⍋y←o[;2] ⋄ y←y[Oy] ⋄ Ox←⍋x←o[Oy;1] ⋄ x←x[Ox]
ox←(o0x←1,(1↓x)≠¯1↓x)/x ⋄ oy←(o0y←1,(1↓y)≠¯1↓y)/y
O0x←+\o0x ⋄ O0y←(+\o0y)[Ox] ⋄ Oo←((⍴ox),⍴oy)↑0
oO←O0y+(O0x−1)×1↓⍴Oo ⋄ oo←oO⍳oO ⋄ OO←((1↓oo),1+⍴oo)−oo
Oo_o←,Oo ⋄ Oo_o[oO]←OO ⋄ Oo←(⍴Oo)⍴Oo_o ⋄ →(0=⎕NC 'a')/0
Oo←Oo ox oy (+/Oo) (+⌿Oo) (+/,Oo)
```

"No, no, that's not an answer—it seems to be a complicated *APL*2-o-oo-oO-Oo-OO-O-puzzle! I am sure you didn't develop it in this way. Please send us an understandable solution!"

"Okay, you are right. I will mail you the original WEB-paper."

The WEB-paper of contab is reprinted as the second major section of this paper.

11. The contab-function.

12. Computational Statistics Problem

A function is to be written which computes the contingency table from a two column data matrix of discrete random variables organized as column vectors.

13. Data Structures of Arguments and Result

The name of this dyadic function is contab and it takes as its right argument the two column numeric matrix m. Then the explicit result res is a simple matrix which contains the frequencies of all xy-combinations. The x- and y-values should be sorted in ascending order over the rows and columns of res respectively. If a left argument is given contab should add the margins to the table. Then the explicit result is a *nested vector* consisting of the raw contingency table as in the monadic case, the x-values, the y-values, the sum over the columns, the sum over the rows, and the number of entries.

14. Syntax of contab

The function header looks like this:

```
∇ res←mar contab m   14
```

See also section 16.

contab appears in sections 2, 10, 13, 16, 24, 28, 32, and 38.

15. Solution

Of course, there are several methods to solve this task but this time let's imagine that our *APL*2-function must be as fast as possible. So it's wise to avoid the obvious double loop. But how? What has to be done?

Well, certainly we have to compute the different x- und y-values. Then we can construct a prototype of the result. Furthermore, we must count the number of appearances of each xy-combination in our data matrix and we must assign this number at the right position[1]. We must not forget that the x- and y-values should be sorted in ascending order, so we will have to do some *gradeups.*

16. Definition of contab

We are ready to do the first refinements of contab.

```
∇+ res←mar contab m   14
⟨ Sort m in ascending order!   17 ⟩
⟨ Remove duplicated elements and compute the "ranks" of x and y!   19 ⟩
⟨ Build a prototype of the result!   21 ⟩
⟨ Count the number of appearances of each xy-pair!   22 ⟩
⟨ Do the final assignment!   24 ⟩
```

[1] This is a known problem called "scattered assignment".

⟨ Add the margins if a left argument is given! 26 ⟩

17. We will certainly need the x- and y-values as vectors, so instead of sorting the matrix we will carefully sort the column vectors of m without losing the information about the sample pairs. That means, me must remember the positions of the elements of the x- and the y-vectors.

These positions will be saved in the local variables orig_x and orig_y while the x- and y-values will be saved in the local variables x and y, respectively.

⟨ Sort m in ascending order! 17 ⟩ ≡

```
orig_y←⍋y←m[;2]
y←y[orig_y]
orig_x←⍋x←m[orig_y;1] ⍝ Do not forget the pairs!
x←x[orig_x]
```

This code is used in section 16.

18. ⟨ Local Variables of contab(14) 18 ⟩ ≡

```
orig_x,orig_y,x,y
```

See also sections 20, 23, and 25.

19. Now we are ready to count the number of different x- and y-values and their "ranks". Ranks in this sense are somewhat different from the usual definition in statistics. For example, the vector 4 5.2 5.2 7 7 8 has in our sense the ranks 1 2 2 3 3 4 while the statistics definition of ranks would give 1 2.5 2.5 4.5 4.5 6.

We will start with the removing of duplicated elements in x and y. Since the vectors x and y are already sorted, we could use the idiom (v≠1⌽v)/v if they are not all equal but for efficiency reason we prefer to do the compression "manually" through (1,(1↓v)≠¯1↓v)/v.

The computation of the ranks is a simple task: +\v≠1⌽v will do the job. Therefore, we will save the *notequal one rotate* part of our *remove duplicated statement* in the local variables neq_x and neq_y.

We save the elements of x and y in elem_x and elem_y, the ranks of x and y are saved in rank_x and rank_y.

⟨ Remove duplicated elements and compute the "ranks" of x and y! 19 ⟩ ≡

```
elem_x←(neq_x←1,(1↓x)≠¯1↓x)/x
elem_y←(neq_y←1,(1↓y)≠¯1↓y)/y
rank_x←+\neq_x
rank_y←(+\neq_y)[orig_x] ⍝ Do not forget the pairs!
```

This code is used in section 16.

20. ⟨ Local Variables of contab(14) 18 ⟩ +≡

```
elem_x,elem_y,rank_x,rank_y,neq_x,neq_y
```

21. We know how many columns and rows our result must have: (⍴elem_x),⍴elem_y. We build a matrix with that shape and fill it up with zeros.

⟨ Build a prototype of the result! 21 ⟩ ≡

```
res←((⍴elem_x),⍴elem_y)↑0
```

This code is used in section 16.

22. Now for the tricky part. This is the *scattered assignment.* It is needed so frequently that the idea behind it is worth *taking a closer look.*

We will use a *decode*-idea to transform matrix indices into a vector. Then we are able to assign values to a ravelled matrix through *indexed assignment* without loop constructions. For example, if we want to assign the elements of 5 6 7 to the matrix M←2 4↑0 at the index positions [1;1], [1;3], and [2;2] we would be tempted to write

```
(,M)[1+2 4⊥¯1+1 1 2,[.1]1 3 2]←5 6 7.
```

Unfortunately, this gives a SYNTAX ERROR. Therefore, we must do something similar to Shape←ρM⋄M←,M⋄M[1+2 4⊥¯1+1 1 2,[.1]1 3 2]←5 6 7⋄M←ShapeρM.

But since the left argument of ⊥ is the shape of res — a two element vector — we can do the "decoding" which is in this case one vector multiplication and one vector addition more efficiently "by hand":

In place of 1+(ρres)⊥¯1+rank_x,[.1]rank_y we shorter write rank_y+(rank_x−1)×1↓ρres.

Funny, first we found a wounderful *decode* trick, then we refused to use ⊥.

We store the vector of index positions in the local variable idx.

We notice that idx is sorted in ascending order because we used the shape of res as the (virtual) left argument of our (virtual) decode statement.

Now we have to count how often each element of idx appears in idx. Therefore, we first use *dyadic iota* to find idx in itself. These positions are stored in found. Then we easily find the number of occurrences of each element in found by subtracting found from (1↓found),1+ρfound and store this result in occ_num.

⟨ Count the number of appearances of each *xy*-pair! 22 ⟩ ≡

```
idx←rank_y+(rank_x−1)×1↓ρres
found←idxιidx
occ_num←((1↓found),1+ρfound)−found
```

This code is used in section 16.

23. ⟨ Local Variables of contab(14) 18 ⟩ + ≡

```
idx,found,occ_num
```

24. We can create a vector res_v by *ravelling* res and assign the number of occurrences occ_num at the index positions given by idx. At last we only have to *reshape* res, divide by the number of pairs, and contab is complete.

⟨ Do the final assignment! 24 ⟩ ≡

```
res_v←,res
res_v[idx]←occ_num
res←(ρres)ρres_v
```

This code is used in section 16.

25. ⟨ Local Variables of contab(14) 18 ⟩ + ≡

```
res_v
```

26. In some applications one needs the margins of the table. If there is a left argument we want to be given the margins.

⟨ Add the margins if a left argument is given! 26 ⟩ ≡

```
→(0=⎕NC 'mar')/0
res←res elem_x elem_y(+/res)(+⌿res)(+/,res)
```

This code is used in section 16.

27. From announcements to code.

Now it is necessary to elaborate the announcements stated above.

28. Construction of the contingency table.

This isn't any problem now. The names of the variables are chosen as usual. n_{ij}, $n_{.j}$, n_k, T_k, etc. are translated to variables n_ij, n__j, n_k, T_k, etc.

⟨ construct the contingency table from the input data matrix 28 ⟩ ≡

```
(n_ij mar_left mar_top n_i_ n__j n)←'with margins'contab dm
p_ij←n_ij÷n
```

This code is used in section 8.

29. Global variables could interfer with other things in the workspace, so we define all variables used as local.

⟨ Local Variables of dm_random_test(5) 29 ⟩ ≡

```
n_ij;mar_left;mar_top;n_i_;n__j;n;p_ij
```

See also sections 31, 33, 36, and 39.

30. The left arguments are split to make them more handy. Default values for the optional left argument are defined as described above.

⟨ set default values, initialize dm_random_test 30 ⟩ ≡

```
→0×⍳2≠⍴⍴dm ⋄ n←⊖⍴⍴dm
⍎(2≠⎕NC 'nbatches_nruns')/'nbatches_nruns←(⌈n⋆.5),100'
nbatches_nruns←2↑nbatches_nruns,100
nk←⌈n÷n_batches←nbatches_nruns[1]
n_runs←nbatches_nruns[2]
```

This code is used in section 7.

31.

⟨ Local Variables of dm_random_test(5) 29 ⟩ + ≡

```
n_batches;n_runs;nk
```

32. Having initialized the loop over the batches the computation of the statistic T_dm is straightforward. Technical Remark: Terms of the sum with denominator that is equal to 0 have to be deleted because of 1=0÷0. pos saves the position of the element of dm treated at last.

⟨ compute the statistic T 32 ⟩ ≡

```
 T_dm←pos←0
loopdm:
 (nk_ij mark_left mark_top nk_i_ nk__j nk)←'with margin'contab dm[pos+⍳nk;]
 expected←nk×p_ij[mar_left⍳mark_left;mar_top⍳mark_top]
 Tk←+/(0≠,expected)/Tk←,(Tk×Tk←nk_ij−expected)÷expected
 T_dm←T_dm+Tk
 →loopdm×⍳n≥nk+pos←pos+nk
```

This code is used in section 8.

33.

⟨ Local Variables of dm_random_test(5) 29 ⟩ + ≡

```
pos;T_dm;nk_ij;mark_left;mark_top;nk_i_;nk__j;nk;expected;Tk
```

34. The simulation runs iteratively. This is performed by a simple loop construction. Within the loop a new permutation of the data matrix is found for which the statistic T_run is computed as shown above. It is possible to save some lines of code but for clarities sake we spend an additional section for the computation of T_run.

⟨ find an approximate distribution of T 34 ⟩ ≡

```
⟨ initialize loop over simulation runs 35 ⟩
loopsim:
⟨ find a new permutation of dm 37 ⟩
⟨ calculate, save the T-statistic T_run of the simulation run 38 ⟩
⟨ save result, increment the run_count and jump back to loopsim 40 ⟩
```

This code is used in section 8.

35. The variable T collects the results of the runs.

⟨ initialize loop over simulation runs 35 ⟩ ≡

```
run_count←1
T←⍬
```

This code is used in section 34.

36.

⟨ Local Variables of dm_random_test(5) 29 ⟩ + ≡

```
T;run_count
```

37. No comment.

⟨ find a new permutation of dm 37 ⟩ ≡

```
dm←dm[n?n;]
```

This code is used in section 34.

38. This code is almost a copy of the one described in the section about the computation of T_dm.

⟨ calculate, save the T-statistic T_run of the simulation run 38 ⟩ ≡

```
T_run←pos←0
loopinner:
(nk_ij mark_left mark_top nk_i_ nk__j nk)←'with margin'contab dm[pos+⍳nk;]
expected←nk×p_ij[mar_left⍳mark_left;mar_top⍳mark_top]
Tk←+/(0≠,expected)/Tk←,(Tk×Tk←nk_ij−expected)÷expected
T_run←T_run+Tk
→loopinner×⍳n≥nk+pos←pos+nk
```

This code is used in section 34.

39.

⟨ Local Variables of dm_random_test(5) 29 ⟩ + ≡

```
T_run;Tk
```

40. The calculated T-values will be collected in the variable T.

⟨ save result, increment the run_count and jump back to loopsim 40 ⟩ ≡

```
T←T,T_run
→loopsim×ιn_runs≥run_count←run_count+1
```

This code is used in section 34.

41. Now we can summarize our findings. To do this we will construct a nice plot. The empirical distribution function of the T-values will be shown with additional lines which indicate the position corresponding to the input data matrix. For the plot some graphics functions beginning with G_⋆ and it's subroutines must be available. The result result will be the p-value of the input data matrix.

⟨ show the result 41 ⟩ ≡

```
result←(+/T_dm≥T←T[⍋T])÷n_runs
→0×ι3≠⎕NC 'G_begin'
G_begin
13 G_points T,[1.1](ιn_runs)÷n_runs
G_vlines T_dm
G_hlines result
G_end
```

This code is used in section 8.

42. Demonstration.

We need another small function to test whether the function dm_random_test really discovers non-randomness. Let us call it:

dm_random_test_DEMO

43. Idea

This function performs an experiment to demonstrate the sensitivity of the algorithm according to non-random input data. For this we will mix up random data with non-random data. Since the effect of different proportions of non-randomness is our main interest the proportions will be the right argument of the function. For each proportion a plot will be constructed.

44. Syntax of dm_random_test_DEMO

∇ dm_random_test_DEMO prop_of_random 44

See also section 46.

dm_random_test_DEMO appears in sections 2, 42, 46, and 56.

45. Data structures of arguments

prop_of_random is the vector of proportions of randomness of the input data used in dm_random_test calls.

Further definitons of the experiment are fixed by adequate local variables.

46. Definition of dm_random_test_DEMO

∇+ dm_random_test_DEMO prop_of_random 44
⟨ initialize dm_random_test_DEMO 47 ⟩
⟨ perform the experiment and show the result 49 ⟩

47. As default value we consider 3×3 contingency tables with 100 entries. Other realisations of the experiment are constructed by changing number_of_margin_cells or number_of_entries.

dmrd is a random data matrix with number_of_entries rows whose values are taken from ⍳number_of_margin_cells.

perm is a permutation of the row indices. It is used to choose rows that will be rearranged to realize non-randomness.

⟨ initialize dm_random_test_DEMO 47 ⟩ ≡

```
number_of_margin_cells←3
number_of_entries←100
dmrd←?(number_of_entries,2)⍴number_of_margin_cells
perm←number_of_entries?number_of_entries
```

This code is used in section 46.

48.

⟨ Local Variables of dm_random_test_DEMO(44) 48 ⟩ ≡

```
prop_of_random;number_of_entries;number_of_margin_cells;dmrd;perm
```

See also sections 52 and 54.

49. Using a loop we will construct a picture consisting of several plots drawn by dm_random_test. Some functions of the graphics group will be needed. counter is the loop variable counting these plots. dm is a local data matrix.

⟨ perform the experiment and show the result 49 ⟩ ≡

⟨ initialize graphics device and counter 51 ⟩

loop:

⟨ construct data matrix dm for dm_random_test call 53 ⟩

⟨ goto loop if another plot is to be made 55 ⟩

⟨ close graphics device 50 ⟩

This code is used in section 46.

50. The last top level description needs not to be commented.

⟨ close graphics device 50 ⟩ ≡

```
G_end
```

This code is used in section 49.

51. To get a nice output the screen is divided into viewports. n_viewport is a two element vector with the meaning *maximal number of viewports in vertical direction/maximal number of viewports in horizontal direction.*

⟨ initialize graphics device and counter 51 ⟩ ≡

```
n_viewport←⌊(ρprop_of_random)⋆.5
n_viewport←n_viewport,⌈(ρprop_of_random)÷n_viewport
counter←1
G_begin
```

This code is used in section 49.

52.

⟨ Local Variables of dm_random_test_DEMO(44) 48 ⟩ +≡

```
n_viewport;counter
```

53. Steps to realize non-randomness: 1. Select rows of dmrd defined by the first elements of perm. (The selected rows are kept by row_marks.) 2. Sort these rows according to their columns.

⟨ construct data matrix dm for dm_random_test call 53 ⟩ ≡

```
dm←dmrd
row_marks←(−⌊/prop_of_random[counter]×number_of_entries)↓perm
row_marks←row_marks[⍋row_marks]
dmh←dm[row_marks;]
dmh←dmh[⍋dmh[;2];]
```

```
dmh←dmh[⍋dmh[;1];]
dm[row_marks;]←dmh
n_viewport G_setviewport 1+n_viewport⊤counter−1
G_label 'prop_of_random ',(⍕⌊/prop_of_random[counter]×100),'%'
0 0⍴dm_random_test dm
```

This code is used in section 49.

54.

⟨ Local Variables of dm_random_test_DEMO(44) 48 ⟩ +≡

```
dmh;row_marks;dm
```

55. There ⍴prop_of_random plots to be made.

⟨ goto loop if another plot is to be made 55 ⟩ ≡

```
→loop×⍳(⍴prop_of_random)≥counter←counter+1
```

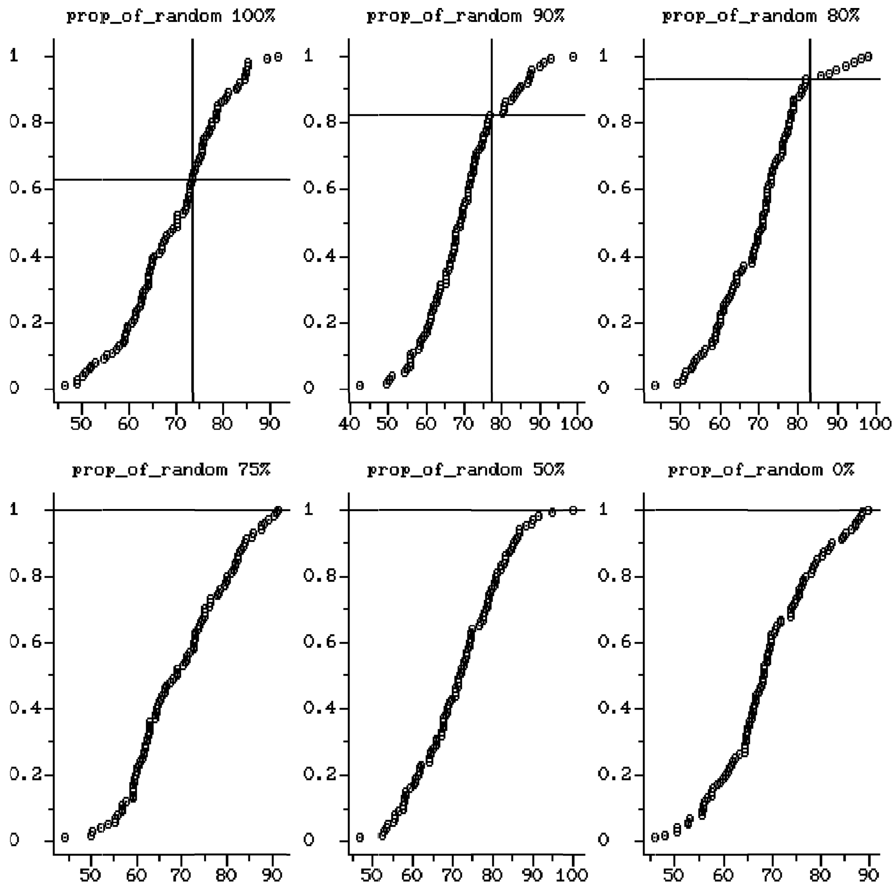
This code is used in section 49.

56. Samples.

A demonstration of dm_random_test_DEMO is required. Let us activate some statements:

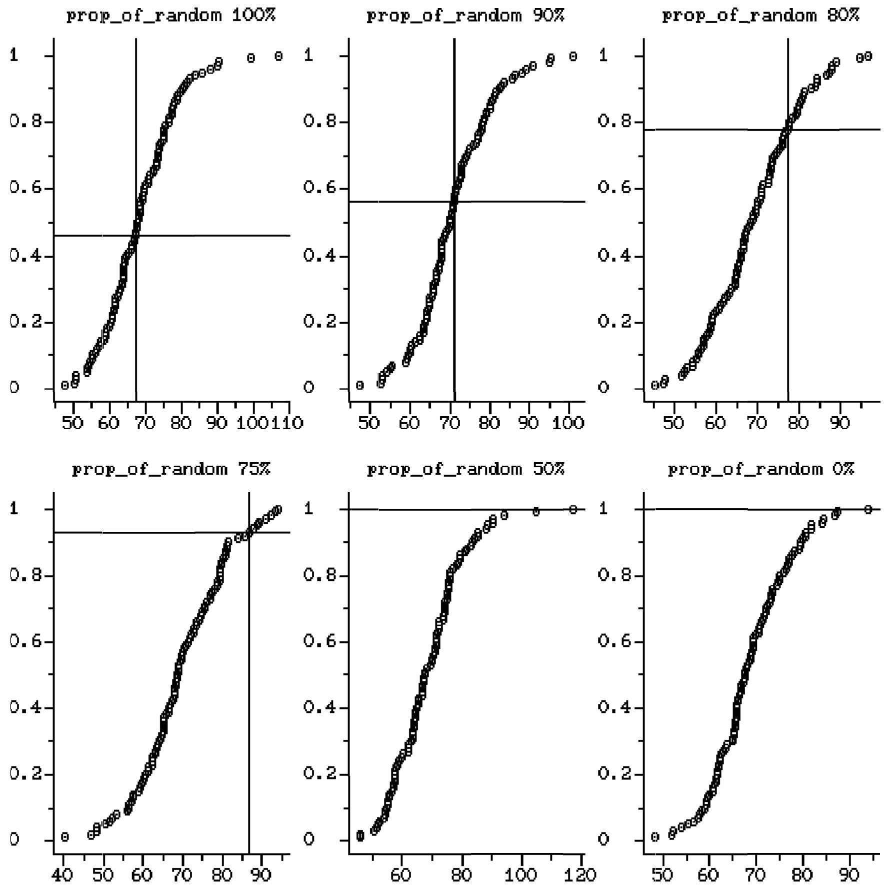
⎕RL ← 928500874 ⋄ dm_random_test_DEMO 1 0.9 0.8 0.75 0.5 0

The following picture will appear:



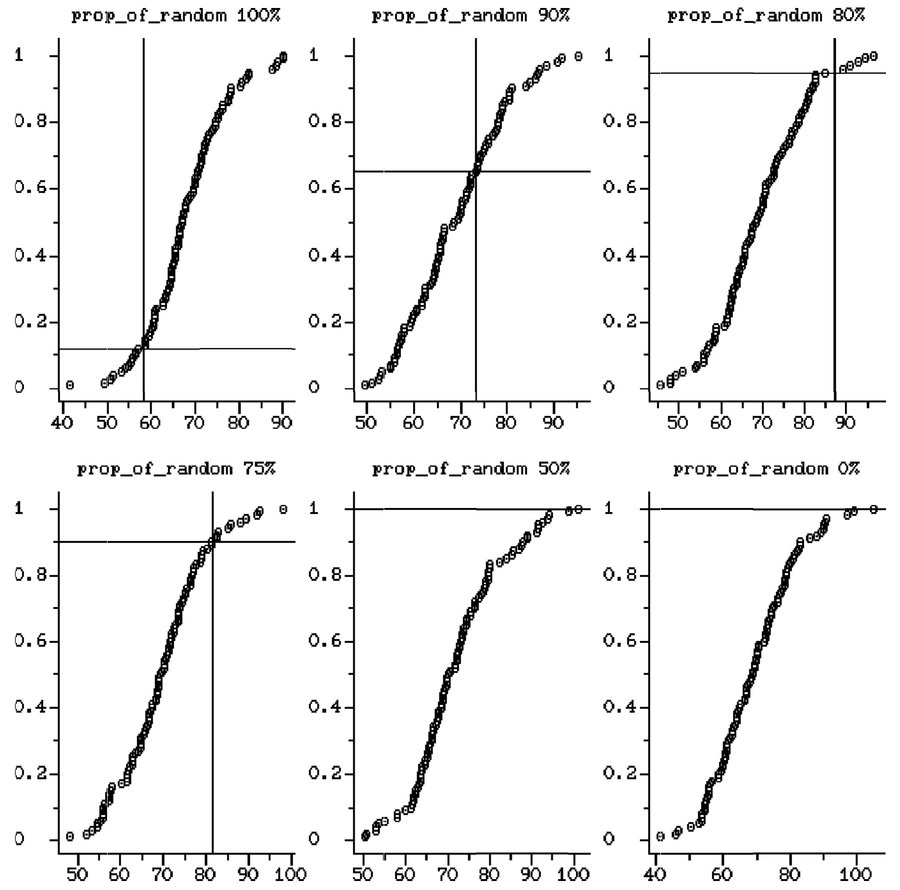
⎕RL ← 354467441 ⋄ dm_random_test_DEMO 1 0.9 0.8 0.75 0.5 0

The following picture will appear.



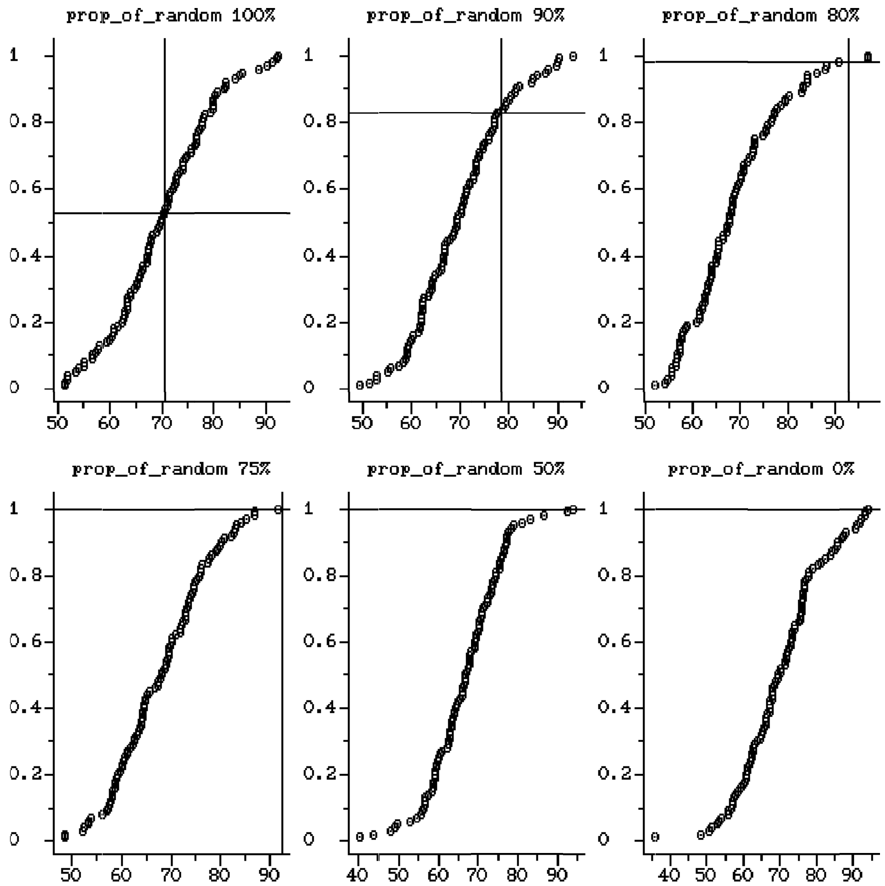
```
⎕RL ← 1102062042 ⋄ dm_random_test_DEMO 1 0.9 0.8 0.75 0.5 0
```

The following picture will appear.



⎕RL ← 1237320656 ⋄ dm_random_test_DEMO 1 0.9 0.8 0.75 0.5 0

The following picture will appear.



57. Index.