Contents

Version history

Version	This document	RaBiGeTe
050331	• Added the AMLS test; • changed the DFT test; • added the parameters *KSmin*, *ADmin* and *SLmax* for the partial overall values while *RaBiGeTe* is running; • specified that the *pval_digits* affects only the p-value formatting; • few minor changes.	• Added an improved version of the AMLS test; • improved the DFT test with a 2-D version; • added the partial KS, AD and SL calculation while *RaBiGeTe* is running; • fixed a rounding error bug for the optimized *n* in the DFT test; • the user can break the program by pressing the escape key; • changed "Test #xx" in "Sequence #xx".
050221	• Added the coupon collector's test, the NOTM test and the permutation test; • added the parameter *IdlePriority*; • revised §6.1 "Discrete Fourier Transform test"; • revised §5 "How to test the number generators"; • changed the minimum *n* for NOTM; • changed the file names in §3; • specified the syntax for *InFile* and *OutFile* parameters; • deleted the AMLS paragraph.	• DFT function replaced with FFTW, added the parameter *DFToptimize*; • fixed a minor bug in short block test for big bit distances; • fixed a bug when parsing NOTM parameters with negative steps; • added another check in the coupon collector's test; • minimum group size for the permutation test increased to 2 numbers; • added the parameter *IdlePriority*; • added a warning message for p-value = -1 when *showpvals* = 0; • allowed sequence length = 0 when an input file is specified; • few minor improvements; • the parsing routine has been moved in a separate module; • added "DHcheck" in the command line to test the DH tests; • an "usage" message is showed when no parameters are passed.
040920	• /	• Added the coupon collector's test; • improved NOTM test: block size form 2 to 31 bits, templates created on the fly (with no memory allocation), changed the parameters to be more stringent;
040828	• /	• Added the permutation test; • now also the sorted p-values are printed only if valid ($\neq$-1); • changed the order of the sorted p-values by SL;
040826	• Short blocks test changed with its generalized variant; • the step for the test parameters can be negative for increments of 2^{step}; • added the parameters: *TestName_pv* and *pval_digits*; • added suffixes for multiples of 1000 for the sequence length; • changed the parameter *filename* with *InFile*; • changed the name of the gorilla test, now it is SOB; • added the description of the SOB test; • added a brief general explanation of the Maurer's test; • changed the version number convention (yymmdd).	• AMLS test definitely withdrawn; • short blocks test replaced with its generalized variant; • the step for the test parameters can be negative for increments of 2^{step}; • added the parameters: *TestName_pv* and *pval_digits*; • added suffixes for multiples of 1000 for the sequence length; • changed the parameter *filename* with *InFile*; • fixed a minor bug in the NOTM test; • optimized for the speed: two-bit test (~61%), runs test (~27%) and cusum; • changed the name of the gorilla test, now it is SOB; • the sequence length is now unsigned; • some minor improvements; • changed the version number convention (yymmdd).
2.01 *2004-05-29*	• /	• AMLS test disabled; • fixed a minor bug in Bday test.
2.0 *2004-05-24*	• Added the windowed autocorrelation test; • added the wildcards for the bits file name; • added the test name meaning; • added the bits required for each test; • added sorting parameters by KS, AD, SL and the OutFile parameter; • added paragraph "Before starting" changing the program checking; • changed paragraph "Basic concepts"; • changed the range for NOTM and added the number of the templates; • improved the program test procedure; • specified that the test parameters are positive and that for *n* and *DFT max* the number can be floating point; • added the meaning of the return values when the program ends; • deleted "the user can change" in all the parameter section; • changed the font size; • changed the paragraph numbering.	• Added the windowed autocorrelation test; • added the ability to use wildcards for the bits file (filename parameter); • added results with sorted p-values; • added the range check for rank 6x8 and NOTM; • added the OutFile parameter for the output file name (was "_RaBiGeTe.txt"); • *n* and *DTF max* can be floating point numbers; • added the bits file name in the output file; • improved the file parameter parsing routine; • fixed memory leakage in Short Blk; • before the program ends, it shows "Results saved in: [OutFile]"; • changed the format for the time (was [h:mm:ss.d] now is [hh:mm] or [mm:ss], but "Running time" is always [h:mm:ss.d]); • when n < 1024 the floating point n is no longer displayed in the header (was "n= 459 bits (459.000 b)" now is "n= 459 bits").
1.1 *2004-04-07*	• Added the range for NOTM and Long blocks; • changed the max bit distance for autocorrelation (was max $\leq$ n/2); • added the test file "bits.bin".	• Fixed the first template in NOTM for m=2 (was 00 instead of 01); • fixed test enable/disable; • fixed the seconds $\geq$ 60 in "End date" (now "Date end", e.g.: 20:12:71) and changed the format for the time (was [sec.d] now is [h:mm:ss.d]); • added some parameter checking; • the program no longer checks the parameters of a disabled test; • saved the version number in "RaBiGeTe.txt".
1.0 *2004-04-03*	First release.	First release.

1 What my program does

With *RaBiGeTe* the user can test a random or pseudo-random bit or number generator to see whether it has the characteristics expected in a true random generator.

RaBiGeTe runs under Windows (in the DOS prompt).
It includes the following tests:

1. AMLS (an improved version of my old AMLS);
2. *DFT* (the new 2-D version of my very sensitive test);
3. *Short blocks generalized variant* (my test. A generalized version of the poker and monobit tests);
4. *Sparse Occupancy Bitstream (derived from the gorilla test of the Diehard suite);*
5. *Windowed autocorrelation;*
6. *Permutation;*
7. *Coupon collector;*

8. *Maurer* (improved version of the Maurer's universal statistical test);
9. *Autocorrelation;*

10. *Serial two-bit* (FIPS140);
11. *Blocks'n'gaps* (FIPS140 Runs);

12. *Long blocks* (NIST BlockFrequency);
13. *Runs* (NIST);
14. *Cumulative sum* (NIST);
15. *Serial* (NIST);
16. *Non overlapping template matchings* (NIST), my improved version;

17. *Rank for 6x8 matrices* (Diehard);
18. *Rank for 31x31 matrices* (Diehard);
19. *Rank for 32x32 matrices* (Diehard);
20. *GCD* (Diehard);
21. *Tough birthday spacings* (Diehard).

RaBiGeTe is highly configurable along with the test parameters.

RaBiGeTe needs one or more binary file (supplied by the user) containing the bits to test. It is possible to use the wildcards to test several files.
The user enables the tests he likes and he changes the test parameters using a configuration file (I included "para.txt").
In a DOS prompt the user writes: "rabigete para.txt" (without quotation marks).
The program will save the p-values and the overall results in the file specified by the parameter `OutFile` (the default name is "_RaBiGeTe.txt").

2 Basic concepts

To test a generator we check whether it has a certain characteristic that a true random generator would have. Each test calculates how much the inspected characteristic differs from the expected value and that difference is given as a number from 0 to 1 (called p-value).
If a random generator were to produce a sequence of 100 consecutive zero bits we would be highly suspicious that it was a bad generator. However, although it is extremely unlikely, even a perfect random generator might do this so we can never be certain as a result of a test that a generator is bad. Equally, even with a very good test of randomness, there is always a small chance that a bad generator will pass the test so we can never be certain that a generator is good. Nevertheless, if we run a given test a number of times and a generator systematically fails in a significant proportion of these tests, we can be highly confident that it is poor.
We should hence run each test against a consistent number of sequences (usually 50 or 100 sequences) and look for anomalies both in the individual tests and in the statistics of the tests as a whole.
If the sequences are random (or pseudo-random) we should obtain random (or pseudo-random) p-values from each test (this means that the p-values must be uniformly distributed).

Once the program terminated, we have, for example, 100 p-values for each single test (100 p-values for the DFT test, 100 p-values for the runs test and so forth).

Now we need to see the extent to which the p-values for each test are uniformly distributed. A good choice is the Kolmogorov-Smirnov test. Calculating the K-S test over the p-values from each test, we obtain a significance level α. We could say:

- if $\alpha > 0.01$ the generator is good (for the tests we have used);
- if $\alpha < 0.001$ the generator is certainly bad;
- if $0.001 < \alpha < 0.01$ we could run the test again until we don't get one of the above conditions.

This is not a rule, but just a practical suggestion.
Obviously, testing 1000 sequences instead of 50 or 100, we will obtain a stronger "declaration" (remember that no test can declare good a generator).

In this example I have tested 10 2-Mbits sequences using just few tests in the suite (I used just 10 sequences because this is an example; in the real testing we should use at least 50 sequences).

Sequence number

Systematic failure

Test name	1	2	3	4	5	6	7	8	9	10	K-S
DFT 1	0.0141	0.0209	0.0337	0.1303	0.0463	0.0017	0.0270	0.0078	0.0041	0.0036	0.0000
Maurer 6	0.5702	0.7679	0.8160	0.1792	0.1703	0.0120	0.0999	0.6421	0.2826	0.9058	0.6565
Maurer 8	0.9417	0.1641	0.4641	0.0431	0.6369	0.2907	0.9825	0.0114	0.4793	0.6053	0.9314
Long blk 1024	0.3450	0.3272	0.5627	0.1804	0.6267	0.8304	0.5270	0.8505	0.9457	0.2087	0.8663
Long blk 1536	0.3617	0.6120	0.4422	0.6805	0.2379	0.8573	0.5459	0.8026	0.7765	0.3032	0.5619
Long blk 2048	0.5122	0.7242	0.6342	0.3290	0.3104	0.8429	0.6126	0.6660	0.6770	0.2197	0.6630
Auto 4	0.6506	0.4942	0.9263	0.6655	0.8187	0.1956	0.5675	0.1651	0.5741	0.4304	0.6035
Auto 12	0.4769	0.4191	0.5778	0.7518	0.8543	0.1923	0.5334	0.6318	0.8219	0.4011	0.2713

P-values

Figure 1: In this example (obtained from a real generator) it is showed how to test a generator. Calculating the Kolmogorov-Smirnov test over the 10 p-values (one for each test), we see a systematic failure with the DFT test.

It is clearly showed how the generator fails systematically the DFT test. In this case it is possible to say that the process which generates the sequences (the generator) has too much periodic patterns and that its output is not random. Testing just one sequence, we can clearly see that most of the times the generator seems good.

3 Before starting

It is a good idea to test the program to see whether it runs as expected. To do this, I included the files *Test_files.zip* and *Test_vector.zip.* Decompress these file in the *RaBiGeTe* directory and do the following two tests.

3.1 First test

This test is intended to see if the load routine works.
1. In a DOS window write *rabigete paraTST* and press the *enter* key to run the program;
2. after short time you should see:

```
            !! LOADING ERROR !!
   File "bits3"
   Only 36864 bits have been read instead of 512000
```

3. press any key to continue;
4. the program shows the results and save them in the file *_RaBiGeTe.txt*.

The p-values in *_RaBiGeTe.txt* must be identical to those in *_RaBiGeTe_TST.txt* included in *Test_files.zip.* Notice the line "`!! WARNING !!  Only 3 sequence(s) have been tested.`". This is because in the file *paraTST* there is "`NT 4`" which means that the program should test 4 sequences, but the files

bits1, *bits2* and *bits3* are too short and only three sequences can be tested (see §4.1 for the parameter description).

3.2 Second test

This test is intended to see if the Diehard tests work.
In a DOS window write *rabigete DHcheck*.
When the program ends, compare the file *_RaBiGeTe.txt* with the file *_RaBiGeTe_DH.txt;* the p-values must be identical.

Once the two tests are done, the archives *Test_files.zip*, *Test_vector.zip* and its decompressed files are no longer needed.

4 How *RaBiGeTe* works

The user should save in one or more files the bits (or numbers) to test. If there are several files to test (not necessarily of the same size), the user must use the wildcard like '*' or '?'. For example, having *bits01*, *bits02*, …, *bits99* in the "*C:\random bits*" directory, the `filename` parameter should be "*C:\random bits\bits??*" or "*C:\random bits\bits**".
The program will load the files in alphabetical order (the file *bits10* comes before of the file *bits9*, so it should be used: *bits09*, *bits10*).
If the files are smaller than the expected size, *RaBiGeTe* will show an error message and it will show the overall results using the p-values calculated before the error.

4.1 Parameter description

To configure *RaBiGeTe* the user can change the configuration file (para.txt). It has several sections (enclosed in the square brackets) and its contents are case insensitive except for the unit of measure used with some parameter.
Unless otherwise stated in the description, the parameters are treated as integer numbers. For the floating point parameters only the decimal point "." is allowed (the decimal comma "," is not allowed).
The user can change the numbers (the test parameters) but not the parameter names.
To facilitate the comprehension of the names, the user can puts a remark preceded by the character #. The program will discard anything follows the # character.

The following is the list of the sections and the related parameters allowed (with an example of declaration and some remark).

Section: `[General]`

Parameter: `n 1 Mb  # Sequence length`
Description: sequences length (*floating point*). The user can writes just a number (like 1000000) to specify the length in bits, or he can use a unit of measure along with its suffixes and the b (bits) or B (bytes) unit too. The suffix can be: k (x10^3), m (x10^6), g (x10^9), K (x1024), M (x1024^2) and G (x1024^3).
For example, if *NT*= 50 and *n*= 1 Mb, the user should supply a 50-Mbits file (52,428,800 bits). Writing "n .5 Mb" is the same as "n 512 Kb".
Range: $20 \le n \le 2^{32}-1$.

Parameter: `NT 100  # Number of sequences to test`
Description: all the tests enabled by the user will be executed *NT* times (will be tested *NT* *n*-bit sequences). I suggest to use $NT \ge 50$.
Range: $1 \le NT \le 2^{31}-1$.

Parameter: `auto_n 1`
Description: if =1, *n* will be programmatically changed to run all the tests enabled by the user. For example, if the user enables the rank 6x8 test and *n* is 1 Mbits, the program will change *n* to 19,200,000 bits.

Parameter: `OverallTestNumber 0`
Description: if =1, *RaBiGeTe* will show the overall results relative to the test number (see §4.3). I suggest to leave 0 in this parameter (test skipped) because the p-values examined by

this test are heterogeneous; you can get a bad overall result even with good sequences (I included this test just because someone likes it).

Parameter: `SortKS 1, SortAD 0, SortSL 0`
Description: if =1, the overall test-name values will be sorted by K-S, A-D or SL (see §4.3). It is possible to set to 1 all the three parameters, in this case *RaBiGeTe* will show the overall result sorted by K-S, then by A-D and then by SL.

Parameter: `OutFile "_RaBiGeTe.txt"  # Full path of the output file`
Description: this is the name of the output file (the one with the results). The path must be always enclosed in quotation marks.

Parameter: `InFile "bits.bin"  # Full path of the input file`
Description: this is the name of the input file (the one with the sequences to test). The path must be always enclosed in quotation marks.

Parameter: `ShowPvals 1`
Description: if =1, *RaBiGeTe* will show the p-values during the test execution, while if =0, the p-values won't be showed on the screen, but in both cases they will be saved in the file `OutFile`.

Parameter: `TestName_pv 1`
Description: if =0, *RaBiGeTe* will write the `OutFile` using the format: *p-value + [TAB] + test_name*, while if ≠0 the format *test_name + [TAB] + p-value* will be used.

Parameter: `pval_digits 5`
Description: it is the number of decimal digits used to write the p-values. This parameter does not affect the calculations, it is used only for the p-value formatting.

Parameter: `IdlePriority 1`
Description: if ≠0, *RaBiGeTe* will run in idle priority.

Parameter: `KSmin .01, ADmin .01, SLmax 90`
Description: threshold for bad values (*floating point*). To obtain an useful answer, it is needed to test at least 50, 100 sequences (as suggested in §2) and this can take very long time. With a bad generator, many tests need just 5, 10 sequences to show the weakness. For this reason, when these parameters are enabled, *RaBiGeTe* calculates the "overall" values to see whether they are smaller than *KSmin* or *ADmin* or bigger than *SLmax*; in this case, *RaBiGeTe* will show a line with the calculated values (this line is not saved in the result file). If the values are too bad, the user can break the program (by pressing the "Escape" key) without wasting the time to complete the test of all the *NT* sequences.
Range: $0 < KSmin, ADmin < 1$;
$0 < SLmax < 100$;
if $KSmin, ADmin \leq 0$ or $SLmax \geq 100$ the parameter is disabled.

Section: [Enable]

In this section the user can enable or disable the tests and he can change their execution order. Don't change the names!
Writing:

```
1 DFT
1 Short blocks
0 Autocorrelation # This can be omitted being 0
1 Serial
1 Long blocks          # NIST's BlockFrequency
```

RaBiGeTe will run DFT, Short blocks, Serial and Long blocks. But if the user likes to run the Long blocks immediately after the Short blocks, he just need to write:

```
1 DFT
```

```
1 Short blocks
1 Long blocks           # NIST's BlockFrequency
1 Serial
```

Section: test parameters

Here the user can change the test parameters. Don't change the names!
The program will check all the parameters of the selected tests and it will show an error message if some of them is not valid.
These sections can have from 1 to 4 parameters separated by anything which is not a number or the minus sign ('-'). These expressions: "1,2,3", " 1 2 3", "1[tab]2, abc de3" are all legal and they will return 1, 2 and 3.
When you see `min max step`, it means that the test parameter will vary from `min` to `max` with step `step`. For example, writing `4 12 4`, the test parameter (which usually is the block size) will be: 4, 8 and 12 (so that test will output 3 p-values).
If not otherwise stated, `step` can be also negative. In this case the parameter will be multiplied by 2^{-step}. For example, if `min=2`, `max=19` and `step=-1`, then the parameter will be: 2, 4, 8 and 16. Obviously, `min` can be also an odd number. For the example above, if `min=3`, the parameter will be: 3, 6 and 12.

Section: `[DFT]`

```
max UM
optimize UM
min max step      # ROWS
m0  m1            # Mode 0, mode 1
```

only the first `max` bits will be tested by the DFT test (UM is the unit of measure, see the "n" parameter in the "General" section). This is useful because when `n` is 50, 100 Mbits this test can take very long time (and much memory). Usually it doesn't need too much bits to show weakness in a generator.
If the sequence length is greater than `optimize` it will be optimized (truncated) for the DFT test and only for this test. Third row: rows used and step for the row. Fourth row: mode. (please, see §6.1.2).
Range: max, optimize < 2^{32};
Rows > 0;
m_0, m_1 enabled/disabled (if not equal to zero the mode is enabled).

Section: `[Maurer]`

```
min max step N
```

block size (in bits) and step for the block size. This test needs $N \cdot 2^{max} \cdot max$ bits (see §6.2 for a detailed description).
Range: $2 \le min, max \le 16$; $N \ge 40$ (I suggest to use $N \ge 1000$ because the test will be much more sensitive).

Section: `[Short blocks]`

```
min max step # BLOCK SIZE [bits]
min max step # BITS DISTANCE [bits]
```

first row: block size (in bits) and step for the block size; second row: bit distance and step for the bit distance (see §6.5 for a detailed description).
RaBiGeTe allocates $2^{block\_size} \cdot 4$ bytes (2^{block_size} unsigned longs).
Important remark: for block size equal to 1 see §6.5.1.
Range: block size < 32
n / block size $\ge 5 \cdot 2^{max}$
distance $\le$ n / block size.

Section: `[Long blocks]`

```
min max step
```

block size (in bits) and the step for the block size.
Range: $2 \le min, max \le n$.

Section: `[Autocorrelation]`

```
min max step
```

distance (in bits) from the bits to test and step for the distance.

Range: $1 \le min, max \le \lfloor n/4 \rfloor$.

Section: [Windowed autocorrelation]
min max step # DISTANCE (in blocks)
min max step # BLOCK SIZE (in bits)
first row: distance (in blocks) from the blocks to test and step for the distance; second row: block size (in bits).
Range: distance: min ≥ 1 (see §6.3 for a detailed description)
block size: min ≥ 1.

Section: [Serial]
min max step
block size (in bits) and step for the block size. Setting the block size equal to 1 yields the Short blocks test with the block size and distance equal to 1.
Important remark: for block size equal to 1 see §6.5.1.
Range: max ≤ $\lfloor \log_2 n \rfloor - 2$.

Section: [Non overlapping template matchings]
min max step templates
block size (in bits), step for the block size and number of the templates to test.
The maximum number of the templates *N* for each block size *m* is showed in the table. If the user chooses max = 21, this test would generate 562152 p-values, but setting (for example) templates = 20, only 20 templates will be used because the test will use 1 template every $\lfloor 562152 / 20 \rfloor$.
Range: $2 \le min, max \le 31$ and $n > 8 \cdot (2^{max-1} + max)$

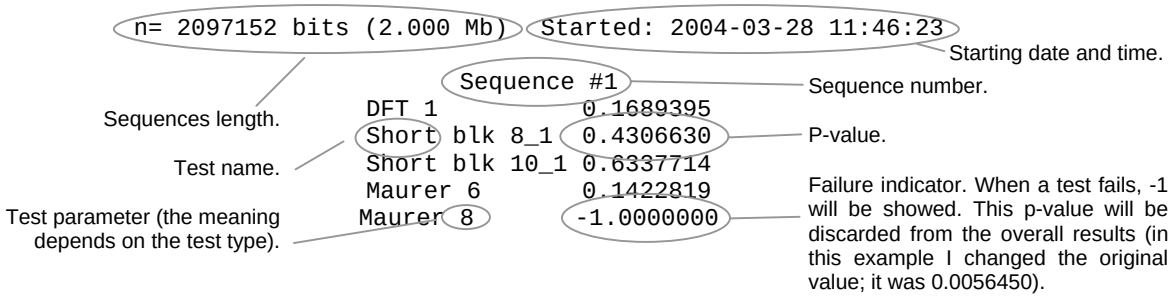
m	N	m	N	m	N
2	2	12	1116	22	1123736
3	4	13	2232	23	2247472
4	6	14	4424	24	4493828
5	12	15	8848	25	8987656
6	20	16	17622	26	17973080
7	40	17	35244	27	35946160
8	74	18	70340	28	71887896
9	148	19	140680	29	143775792
10	284	20	281076	30	287542736
11	568	21	562152	31	575085472

Section: [Rank 6x8]
min max step
the test forms 100,000 6 x 8 binary matrix taking a byte starting from the specified position in the range min ÷ max incremented by step.
Range: $0 \le min, max \le 24$, step ≥ 1.

Section: [AMLS]
min max step
block size (in kbits) and step for the block size. The block size can be only of the form 2^x, this implies that the step can be only negative (please, see the section "test parameters").
Range: the current version of *RaBiGeTe* implements $11 \le min, max \le 25$, step ≤ -1.

4.2 Running the program

Writing in a DOS prompt "rabigiete para.txt" (without quotation marks) the program loads the configuration file "para.txt" (the user can have several configuration files) and it checks all the configuration parameters. All the invalid parameters will be showed.
Setting ShowPvals 1, the program will show all the p-values while it runs. Setting that parameter to 0, it will be showed only the elapsed time, the estimated left time and the estimated end date.
The results are saved in the file specified with the parameter OutFile (the default is "_RaBiGeTe.txt") along with all the p-values obtained from each test. Warning: any existing report will be overwritten without prompt.
This is a typical output (when ShowPvals=1):



In the following table it is showed the meaning of the number displayed to the right of the test name and the required sequence length for each test:

Test name	Displayed name	Name meaning	Bits required
AMLS	AMLS s_n	s: block size [kbits]; n: number of block tested	$5 \cdot 1024 \cdot s$
Autocorrelation	Auto d	d: distance [bits]	$4 \cdot n$
Blocks'n'gaps	Blk'n'gap	/	/
Coupon collector	Coupon s_L	s: block size [bits]; L: "reliability level" (see §6.7.1)	/
Cumulative sum	CuSum m	M: mode (1= forward, 2= reverse)	/
DFT	DFT r_m	r: row; m: mode	$1000 \cdot r$ (see §6.1)
GCD	GCD stp / dst	/	625,000 k
Long blocks	Long blk m	m: block size [bits]	/
Maurer	Maurer L	L: block size [bits]	$N \cdot 2^L \cdot L$ (see §6.2.1)
Non overlapping template	NOTM m_h	m: block size [bits]; h: *m*-bit template in hexadecimal format	$(5 \cdot 2^m + m - 1) \cdot 3$
Permutation	Permutation s_t	s: block size [bits]; t: group size [s-bit numbers]	$5 \cdot t! \cdot t$ (1)
Rank for 31x31 matrices	Rank 31x31	/	38,750 k
Rank for 32x32 matrices	Rank 31x31	/	40,000 k
Rank for 6x8 matrices	Rank 6x8 b	b: starting bit	18,750 k
Runs	Runs	/	/
Serial	Serial m_x	m: block size [bits]; x: p-value # (1 or 2)	2^{2+m}
Serial two-bit	Two-bit	/	/
Short blocks	Short blk s_d	s: block size [bits]; d: distance [bits]	Maximum of: $5 \cdot 2^s \cdot s$ and $d \cdot s$
SOB	SOB	/	67,108,889 (2^{26}+25)
Tough birthday spacings	Bday	/	62,5 M
Windowed autocorrelation	AutoW m_d	m: block size [bits]; d: distance [blocks]	Maximum of: $m \cdot d$ and $5 / C(m, m/2) + d + 1$ (2)

(1): this lower bound may not be sufficient because the duplicated numbers inside a group are discarded.
(2): C(m, m/2) is the binomial coefficient $m! / (m/2)! / (m - m/2)!$.

4.3 Final reports

Once the program terminated, it will show three reports:

- ***Sequence-number*** overall result (from sequence #1 to sequence #NT);
- ***Test-name*** overall result (for DFT test, autocorrelation test, serial test, ...);
- ***All the p-values*** overall result.

As said in §2, only the second report should be considered. I included also the other reports because many people like them and they hope to get some useful information. But few trials should be enough to convince them that the first and the third reports are useless (and the first is really useless).

This is a typical report:

```
                Sequence #1
                    …
               Sequence #10
DFT                0.8901134
AutoW 32_1         0.5799365
Maurer 6           0.3831562
Maurer 8           0.4501849
Short blk 8_1      0.6138030
Short blk 10_1     0.7039236
Short blk 12_1     0.2913844
Long blk 1024      0.6668296
Long blk 1536      0.3188598
Long blk 2048      0.1175366
Auto 4             0.2465715
Auto 12            0.8585987
Runs               0.9430916
Serial 8_1         0.6573969
Serial 8_2         0.6312824
Serial 10_1        0.7854912
Serial 10_2        0.8785840
```

I suggest to skip this report (setting OverallTestNumber = 0) because it is done over heterogeneous values. We can see a failure even with a good generator.

SEQUENCE#	KS	AD	SL
1	0.033642	0.005895	8.678
2	0.151472	0.322141	4.270
3	0.683455	0.513697	2.649
4	0.497912	0.674255	2.077
5	0.562399	0.709130	1.401
6	0.057111	0.009835	13.527
7	0.368998	0.203082	3.239
8	0.643998	0.503060	1.790
9	0.031508	0.046019	9.678
10	0.304464	0.305524	4.099
Overall	0.220205	0.047295	5.1408

Overall KS, AD and SL for these p-values (sequence #10).

Number of p-values generated by each test. The values 8 and 9 indicate that some sequence failed big the test.

KS test of these p-values

AD test of these p-values

mean of these square errors

pvals	TEST name	KS	AD	SL
10	DFT 1	0.599571	0.642290	1.979
10	AutoW 32_1	0.173275	0.060226	9.529
10	Maurer 6	0.340868	0.419206	5.801
10	Maurer 8	0.978825	0.841863	1.033
9	Short blk 8_1	0.754328	0.789165	2.632
10	Short blk 10_1	0.530824	0.742531	3.349
8	Short blk 12_1	0.296080	0.468922	5.224
10	Long blk 1024	0.103148	0.084000	10.552
10	Long blk 1536	0.539214	0.437632	4.856
10	Long blk 2048	0.614148	0.714881	3.334
10	Auto 4	0.568389	0.832119	1.909
10	Auto 12	0.883638	0.852188	2.655
10	Runs	0.557762	0.467999	2.884
10	Serial 8_1	0.120280	0.090325	9.961
10	Serial 8_2	0.204781	0.249436	5.158
10	Serial 10_1	0.579496	0.427589	2.428
10	Serial 10_2	0.077037	0.196975	8.699
	Overall	0.401049	0.845519	4.8225

Overall KS, AD and SL for all the 10 DFT p-values (those obtained from sequence #1 to sequence #10).

Weakness are showed by a low KS or AD p-value (for example < 0.001) and by a high SL value (for example > 40).

These numbers are the most useful because with them it is possible to see a systematic failure of the generator.

Overall KS, AD and SL for these p-values.

```
All the 170 p-values   KS         AD         SL
Overall                0.008231   0.010875   1.397
```

We get 17 p-values for 10 times (from sequence #1 to sequence #10), so we have 170 (heterogeneous) p-values. For that reason this report is not very useful.

To check the distribution of the p-values I use three methods: Kolmogorov-Smirnov test, Anderson-Darling test (which is a weighted variant of the former) and my Straight Line test.

While the first two tests are well known, my SL test (although very simple) needs some explanation.

4.3.1 The Straight Line test

Like the other two tests (K-S and A-D), the SL test is used to see how much the distribution of the p-values differs from the uniform distribution.
Once the p-values have been sorted, we need to calculate these two errors:

$$\mathcal{E}_{\max} = \sum_{i=1}^{n} \left(\frac{i-1}{n-1} \right)^2 \tag{1}$$

$$\mathcal{E} = \sum_{i=1}^{n} \left(\frac{i-1}{n-1} - p_i \right)^2 \tag{2}$$

where n is the number of p-values and p_i is the i^{th} p-value.
With (1) we calculate the maximum square error for n p-values and with (2) the square error for the set of p-values under test.

To get a more readable and useful result we calculate:

$$\mathcal{E}_{\%} = 100 \frac{\mathcal{E}}{\mathcal{E}_{\max}} \tag{3}$$

which is the relative error expressed in percentage.

This is a graphical example in which the points are the p-values (50, in this case) and the bold line is the ideal position of the p-values; it starts from (1;0) and ends at (n;1).

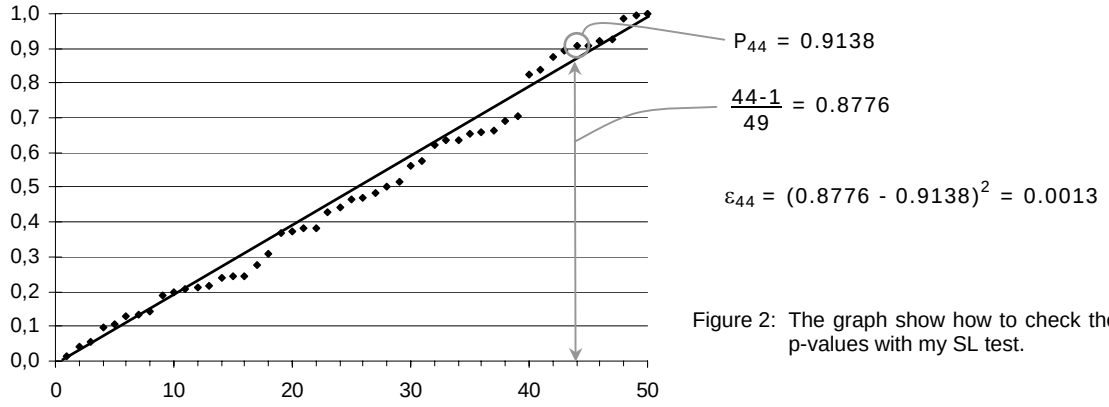


Figure 2: The graph show how to check the distribution of the p-values with my SL test.

5 How to test the number generators

Most of the tests in *RaBiGeTe* are intended for bit generators, but it is very easy to test also the structure of a *b*-bit number generator.

With a number generator the sequence can be generated taking only the i^{th} bit in the *b*-bit numbers. This way it is possible to see whether the generator has some non-random bit.
Suppose we suspect that the generator has bad low order bits (like in the LCG). The sequence to test will be generated using, for example, only the least significant bit of each *b*-bit number generated.

In the module "RBG.cpp" of *RaBiGeTe*, there are several examples which show how to take only the i^{th} bit. For example, to test the 3rd bit (starting from the lsb) of the *lfsr113* 32-bit number generator we just need the line:

```
for(int k=0; k<n; k++) set_bit( k, ( lfsr113() >> 2 )&1, bits)
```

where n is the sequence length.

That line of code is also useful to highlight the relationship between the SOB test (a bit generator test) and the gorilla test (a 32-bit generator test) of Diehard (from which the SOB test derives): that line in *RaBiGeTe* is exactly the same of k=29 in the gorilla test (please, see the DH source code).

6 Test description

6.1 Discrete Fourier Transform test

This test is very sensitive because it has showed the ability to see weakness even with few kbits. This feature makes it very well suited for testing true-RNG's (which often are very slow).

The *N*-bit sequence $x_0 \div x_{N-1}$ is arranged in an *nx* rows by *ny* columns array and it is transformed using a 2-D discrete Fourier transform. The user specifies the parameter *nx* and *RaBiGeTe* calculates $ny = \lfloor N/nx \rfloor$. If *nx* = 1 the test becomes the "usual" 1-D DFT test (like the old DFT test).
RaBiGeTe allows the user to store the bits in two ways: the mode 0 stores the bits sequentially (in the row order) and the mode 1 stores the bits in column order:

ny →

nx ↓ \	0	1	2	3	4
0	0	1	2	3	4
	5	6	7	8	9
	10	11	12	13	14
	15	16	17	18	19

Mode 0 with *nx* = 4: the bits are stored in the input vector in their "natural" order.

Mode 1 with *nx* = 4: the bits are stored in the input vector following a column order.

ny →

nx ↓ \	0	1	2	3	4
0	0	4	8	12	16
	1	5	9	13	17
	2	6	10	14	18
	3	7	11	15	19

Once the input bits have been transformed, we get $nx \cdot (\lfloor ny/2 \rfloor + 1)$ complex components from $X(r, 0) \div X(r, \lfloor ny/2 \rfloor)$ for $0 \le r < nx$:

ny/2+1 →

nx ↓ \	0	1	2
0	0	1	2
	3	4	5
	6	7	8
	9	10	11

The output consist of these imaginary components.

The imaginary components are approximately normal distributed with mean 0 and variance *N*/8, so that the statistic:

$$Z_i = \frac{\mathrm{Im}[Xi]}{\sqrt{N/8}}$$

(Im[*X*] means the imaginary part of *X*) has an approximated standard normal distribution. The real part is discarded.
For each sequence RaBiGeTe calculates the K-S test over all the $nx \cdot (\lfloor ny/2 \rfloor + 1)$ *z*'s, this p-value is showed as "DFT r_m" (where *r* is the row number and *m* is the mode), but the result for *nx* = 1 is simply displayed as "DFT 1" because in this case mode 0 and mode 1 give the same result.

6.1.1 Proof

The mean of each bit is ½ and the variance is ¼.
Calculate the imaginary part using:

$$S_i = \sum_{k=0}^{N-1} - x_i \bullet \sin(2\pi ki / N) \qquad (1)$$

its mean is:

$$\langle s_i \rangle = \frac{1}{2} \sum_{k=0}^{N-1} \sin(2\pi ki / N)$$

which is 0.
For the variance we have:

$$Var[s_i] = \frac{1}{4} \sum_{k=0}^{N-1} \sin^2(2\pi ki / N)$$

which is $N/8$.
The sum (1) is over N random variables from –1 ÷ +1; this is a sufficient condition for which the central limit theorem holds for s_i, this means that the distribution is well approximated by a normal distribution as N gets bigger. Also few hundreds of bits have a very good normal distribution, but, to make the test sensitive, I suggest to use at least 1000 or 2000 bits.

6.1.2 RaBiGeTe implementation

RaBiGeTe includes the fast and reliable FFTW library to calculate the DFT of the input bits.
Here I'll give a brief description of the library. For a detailed description see the FFTW home page http://www.fftw.org/ or the paper "FFTW: An adaptive software architecture for the FFT" by Matteo Frigo and Steven G. Johnson.

FFTW used in *RaBiGeTe* is a DLL which includes the support for float, double and long double types. Changing the #if directive in the module tDFT.cpp, it is possible to choose the float type or the double type (the long double type is not supported by *RaBiGeTe*).
The default type in *RaBiGeTe* is the float type which takes about $^2/_3$ of the running time needed for the double type and the error is usually around 10^{-4}, 10^{-3}.

FFTW is very fast when the sequence length N is of the form $2^a \cdot 3^b \cdot 5^c \cdot 7^d \cdot 11^e \cdot 13^f$, where $e + f$ is either 0 or 1 and the other exponents are arbitrary. Other N's are computed using a slower algorithm which nevertheless retains $O(N \log N)$ performance even for prime N's (see "fftw3.pdf" on page 24).
To take full advantage of that, I included the parameter `DFToptimize`; when N is greater than `DFToptimize`, *RaBiGeTe* calculates the biggest $N' \le N$ of the form $2^a \cdot 3^b \cdot 5^c \cdot 7^d \cdot 11^e \cdot 13^f$ with $e + f$ equal to either 0 or 1 and arbitrary a, b, c and d.
For example, if N = 3524034 bits and `DFToptimize` = 3000000, then *RaBiGeTe* will use $N' = 2^9 \cdot 3^0 \cdot 5^4 \cdot 7^0 \cdot 11^1 \cdot 13^0$ = 3520000 bits so that the sequence will be only 4034 bits shorter (about 0,1% shorter). With the same N, but with `DFToptimize` = 4000000, *RaBiGeTe* will use $N' = N$ (and this will lead to a slower running speed).

FFTW has another interesting feature which is useful to speed up the start-up phase: the wisdom. It is an internal state that FFTW uses to find the best algorithm to calculate the DFT.
RaBiGeTe saves the wisdom so that it can be reloaded prior to executing the DFT test (saving most of the start-up phase time). The wisdom file name will be "DFTwisdom_f" (for float type) or "DFTwisdom_d" (for double type).
If a problem is encountered during the saving or loading of the wisdom, *RaBiGeTe* will show a warning message; but this kind of problem will not affect the results (the running time will be a bit longer).

6.2 Maurer's universal statistical test

This test was proposed by Ueli M. Maurer. It is specified by three positive integer-valued parameters: L, Q and K. The sequence to test is partitioned into adjacent non-overlapping L-bit blocks. The sequence length is $N = (Q + K) \cdot L$ bits, where Q is the number of the L-bit blocks for the initialization step and K is the number of the L-bit blocks to test.

6.2.1 *RaBiGeTe* implementation

The N-bit sequence is partitioned into adjacent non-overlapping L-bit blocks with $2 \le L \le 16$, so we can think the sequence as formed by $n = \lfloor N/L \rfloor$ L-bit numbers, from $x_0 \div x_{n-1}$.
Starting from $i = 0$, store in a table T the index i (the position) of the last occurrence of x_i. When all the possible 2^L L-bit numbers occurred, set $Q = i$ (the index of the last number occurred and stored in the table) and $K = n - Q$. Calculate the test statistic

$$f_{Tu} = \frac{1}{K} \sum_{i=Q+1}^{n} \log_2 A_i$$

where $A_i = i - T(x_i)$ (the number of positions since the last occurrence of the number x_i).
Take the tabulated values for $E[f_{Tu}(L,K)]$ and $Var[f_{Tu}(L,K)]$, then the statistic

$$z = \frac{f_{Tu} - E[f_{Tu}(L,K)]}{\sqrt{Var[f_{Tu}(L,K)]}}$$

approximately follows a standard normal distribution. The resulting p-value is obtained by doing a two-sided test.

6.2.2 Test improvements

Usually the algorithm explained in the literature is not very optimized, so I slightly changed it in my implementation.

First improvement

The precise calculation of the mean and the variance (*E[fTu(L,K)]* and *Var[fTu(L,K)]*) requires a considerable computing effort.
Many implementations use a simplified (and fast) formula for the mean and the variance, but in this case the error in the reported p-value can be too big.

In my program I use tabulated values for the mean and the variance along with a formula of the kind $Var[fTu(L,K)] = f(L,K)$ which for $K \geq 33 \cdot 2^L$ has an excellent approximation.

Second improvement

Usually in the literature it is written that the *N*-bit sequence should be partitioned into two segments: $Q \geq 10 \cdot 2^L$ *L*-bit blocks and $K \geq 1000 \cdot 2^L$ *L*-bit blocks, but this seems not efficient for two reasons:

- sometimes *Q* is too small and the initialization phase leaves empty some table position. Putting *n* in those positions reduces the sensitiveness of the test;
- sometimes to fill the table are needed less of *Q* *L*-bit blocks and the initialization phase could terminate leaving some block for the second phase (the important one).

For those two reasons I changed the "classic" algorithm as explained above: *Q* is not a fixed number; the initialization phase terminate as soon as the table is totally filled, then the program checks whether $K \geq 33 \cdot 2^L$, if the condition is not satisfied the test returns with an error message.
K can be changed using the *N* parameter in the section "`Maurer`" (see §4.1). It sets the numbers of *L*-bit block used by this test (it is the above *n*). The test will need $N \cdot 2^{Lmax} \cdot L_{max}$ bits.

Suppose that $L = 12$ and $N = 1000$; we need $n = 49152000$ bits (4096000 12-bit blocks).
Running the test, suppose the initialization phase takes $Q = 34816$ ($8.5 \cdot 2^L$, instead of $10 \cdot 2^L$), so $K = 4061184$ ($991.5 \cdot 2^L$) and we have $(10 - 8.5) \cdot 2^L = 6144$ more 12-bit blocks to test.

6.3 Windowed autocorrelation test

The test is intended to show how much the bits in a sequence are correlated.
The *n*-bit sequence is partitioned into $N = \lfloor n/m \rfloor$ non-overlapping *m*-bit blocks with $m \geq 1$, any leftover bit is discarded.
All the blocks B_i and B_{i+d} ($0 \leq i < N\text{-}d$) are processed in pairs to see whether the bits in the same position *pos* ($0 \leq pos < m$) are different. Let *c* be the number of the different bits inside each block, it can range from 0 (all the bits are equal) to *m* (all the bits are different). The *c*'s follow a binomial distribution:

$$P_p(c \mid m) = \binom{m}{c} p^c (1-p)^{m-c}$$

where $P_p(c|m)$ is the probability to see *c* different bits in a *m*-bit block and *p* is the probability to see 0 or 1 in the sequence. For a random sequence $p = ½$ and the formula becomes:

$$p_{0.5}(c \mid m) = \frac{m!}{c!(m-c)!} 2^{-m} \qquad (1)$$

The *c*'s obtained from the test are compared with the expected values using the chi-square test with *m*-1 degrees of freedom:

$$\chi^2 = \sum_{i=0}^{m} \frac{[c_i - p_{0.5}(i \mid m)]^2}{p_{0.5}(i \mid m)}$$

The reported p-value is the probability $P(X > \chi^2)$ where *X* is a random variable.

6.3.1 *RaBiGeTe* implementation

With this test can be useful to use small *m* (e.g. 8, 16 bits), but also big *m* (e.g. 2000, 4000 bits), so the binomial coefficient (*m! / c! / (m-c)!*) cannot be calculated in a straightforward way because even using the *long double* C++ type, the maximum block size would be 1754 ($1755! = 3.4736 \cdot 10^{4933}$, while the maximum value a *long double* can handle is about $1.1 \cdot 10^{4932}$).

Knowing that $m! = 2 \cdot 3 \cdot \ldots \cdot (m-1) \cdot m$, we can write $\ln m! = \ln 2 + \ln 3 + \ldots + \ln (m-1) + \ln m$.
Knowing that $\ln (a/b) = \ln a - \ln b$, we can write $\ln [m! / c! / (m-c)!] = \ln m! - \ln c! - \ln (m-c)!$.
Calculating the logarithm of the binomial coefficient instead of the binomial coefficient, we can use *m* well beyond any useful limit.

In the equation 1 in §6.3, there is the coefficient 2^{-m}. Also in this case we can use the logarithm: $\ln 2^{-m} = -m \cdot \ln 2$.

With all these substitutions, the equation 1 becomes:

$$\ln[p_{0.5}(c \mid m)] = \sum_{i=2}^{m} \ln i - \sum_{i=2}^{c} \ln i - \sum_{i=2}^{m-c} \ln i - m \ln 2$$

With the exponentiation of both members we get $P_{0.5}(c|m)$.

6.4 SOB test

This acronym means “Sparse Occupancy Bitstream”.
It works like the gorilla test in the Diehard suite, the only difference is that my SOB test is intended for random bit generators, while the gorilla test is intended for random 32-bit number generators. See §5 to know how to test a particular bit of a *b*-bit number generator.

The $(2^{26}+25)$-bits sequence is divided into 26-bit blocks and it is scanned to count how many blocks do not appear. That count should be approximately normal distributed with mean 24687971 and standard deviation 4170, so that the statistic:

$$z = \frac{w - 24687971}{4170}$$

has an approximated standard normal distribution. The reported p-value is the probability $P(X < z)$ where X is a random variable.

6.5 Short blocks test

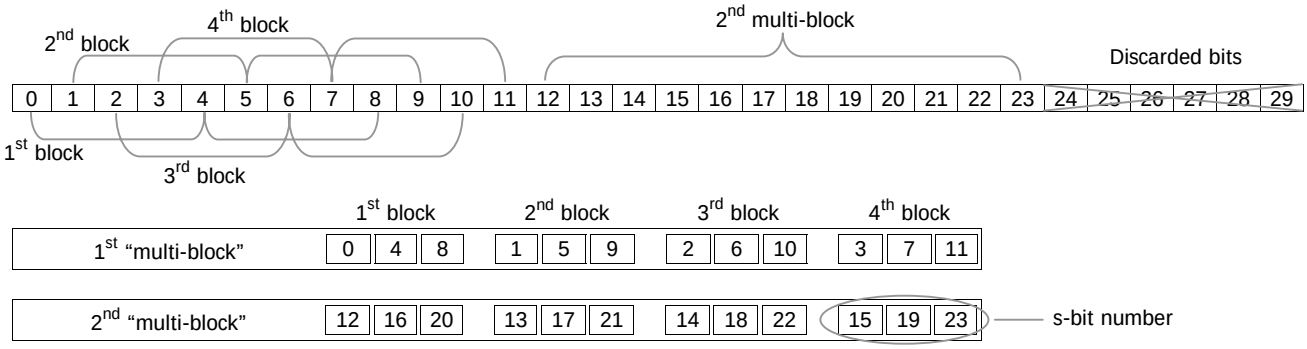
This test can be said a generalized version of the FIPS140 poker test because with my version the user can change the block size and the distance at which the bits are taken.
Setting both distance and block size equal to 1 yields the NIST and FIPS140 frequency (monobit) test and the NIST serial test with block size equal to 1 (only the p-value #1). Setting the distance equal to 1 and the block size equal to 4 yields the FIPS140 poker test.

Let *s* be the block size and *d* the bit distance with $s, d \geq 1$. The *n*-bit sequence is partitioned into *s*-bit disjoint blocks, any leftover bit is discarded. Each block is formed by taking *s* bits at a distance *d*, so the first block is formed by taking the bits 0, *d*, 2*d*, 3*d*, …, $(s-1) \cdot d$; the second block is formed by taking the bits 1, $d+1$, $2d+1$, $3d+1$, …, $(s-1) \cdot d+1$ and so on.
The *s* bits are converted in a number which can range form 0 to 2^s-1, so we have 2^s bins. In a random or pseudo-random sequence it is expected to see $n / 2^s / s$ numbers in each bin.
The *c* numbers contained in each bin are compared with the expected values using the chi-square test with 2^s-1 degrees of freedom:

$$\chi^2 = \sum_{i=0}^{2^s-1} \frac{\left(c_i - \frac{n}{2^s \cdot s}\right)^2}{\frac{n}{2^s \cdot s}}$$

The reported p-value is the probability $P(X > \chi^2)$ where X is a random variable.

Suppose we have a 30-bit sequence (from bit 0 to bit 29), block size s=3 and bit distance d=4. The first block is formed taking the bits 0, 4 and 8; the second block is formed taking the bits 1, 5 and 9, …, the fourth block is formed taking the bits 3, 7 and 11 as showed in the following scheme:



6.5.1 *RaBiGeTe* implementation

When the block size is equal to 1 (and only in this case), different distances will produce the same p-values.
This is true, for example, for distances of the form 2^x for any integer $x \geq 0$. Let p(s,d) the p-value obtained for block size s and distance d; we have for <u>any</u> sequence: p(1,1) = p(1,2) = p(1,4) = … = p(1,8192) and so forth. Moreover we have: p(1,6) = p(1,12), p(1,1536) = p(1,3072) and many other equalities.
But those equalities are unwanted because the overall results would have distorted by many equal p-values. So *RaBiGeTe* discards all the duplicated p-values. Nevertheless, it is still possible to see equal p-values if the parameter `pval_digits` is small enough.

This test with s=1 and d=1 gives the same p-value of the serial test with the block size equal to 1. For this reason, when this test is enabled with *min_blk_size* = 1 and *min_bit_distance* = 1, the serial test with block size = 1 will be skipped.

6.6 Permutation test

The n-bit sequence is partitioned into $N = \lfloor n/s \rfloor$ non-overlapping s-bit blocks with $2 \leq s \leq 12$, any leftover bit is discarded. The s-bit blocks are converted in N numbers from 0 to 2^s-1.
Then t numbers are grouped; if there is duplicate numbers inside the t-number group, the group is discarded.
When a valid group is found, it is scanned to calculate an index which characterizes the ordering of the t numbers inside the group. Let u_i the i^{th} s-bit number with $1 \leq i \leq t$, then an index f is calculated for $u_0 < u_1 < \ldots < u_t$, $u_1 < u_0 < \ldots u_t$, …, $u_t < \ldots < u_1 < u_0$, in other words we have different f's for different orderings.
All the f's are binned in F_i and compared with the expected counts using a chi-square test with $t!$-1 degrees of freedom:

$$\chi^2 = \sum_{i=0}^{t!-1} \frac{\left(F_i - N_{valid} \cdot p_i\right)^2}{N_{valid} \cdot p_i}$$

where N_{valid} is the number of the valid groups found (the ones with no duplicate numbers) and p_i = 1 / t!.

The reported p-value is the probability $P(X > \chi^2)$ where X is a random variable.

6.7 Coupon collector's test

The n-bit sequence is partitioned into $\lfloor n/s \rfloor$ non-overlapping s-bit blocks with $s \geq 2$, any leftover bit is discarded. The s-bit blocks are converted in numbers from 0 to 2^s-1
Let r be the consecutive s-bit numbers required to get a set of all the numbers from 0 to d-1 (where d=2^s). The sequence is scanned to count such r's and these counts are stored in C_r.
Obviously there is a lower bound for r ($r \geq d$, i.e. we need at least d numbers to see all the numbers from 0 to 2^s-1), but there is no upper bound, so when $r \geq t$ (an "arbitrarily" chosen constant) we increment the count for t (not for r).

When all the bits are scanned we have $t-d$ bins from C_d to C_{t-1} (for $d \le r < t$) and the bin C_t (for $r \ge t$).

Let N be the number of complete sets of coupons and P_i the probability for Ci, then the C's are compared with the expected values using the chi-square test with $t-d+1$degrees of freedom:

$$\chi^2 = \sum_{i=d}^{t} \frac{(C_i - Np_i)^2}{Np_i}$$

The probability to see r counts is $P_r = d! / d^r \cdot S(r-1,d-1)$ for $d \le r < t$ and $P_t = 1 - d! / d^{t-1} \cdot S(t-1,d)$ where $S(n,m)$ is the Stirling number of the second kind.

The reported p-value is the probability $P(X > \chi^2)$ where X is a random variable.

6.7.1 *RaBiGeTe* implementation

The calculation of P_i involves very big numbers and a multi-precision library is needed, so I have stored in a file ("coupon_prob.txt") the P's which *RaBiGeTe* needs.
The default file contains the P's for block size s from 2 to 9 bits and the constant t has been chosen so that $P_t < 0.03$, this way the counts C_i have a sufficient degree of significance because they are not all accumulated in C_t (with $P_t = 0.03$, C_t should contain only about 3% of the coupons).
Moreover, the smallest probability saved in the file is $P_{rmin} \ge 5 \cdot 10^{-8}$ this means that the expected count $N \cdot P_i$ is at least 5 when the bin contains 10^8 numbers (with such a small probability the test is very sensitive). The remaining probabilities ($P_r < 5 \cdot 10^{-8}$ from d to r_{min}) are summed up in a single bin.

When the expected number of coupons in a bin is less than 5, *RaBiGeTe* collects the counts C_i until the expected number is at least 5. But when many bins are collected the test become unreliable (it fails even with good generators).
To avoid that, *RaBiGeTe* calculates a "reliability level": $L = (N - C_t) / (t - r_{min})$, where $N - C_t$ is the number of complete sets of coupons excluding the cumulative bin (C_t) and $t - r_{min}$ is the number of available bins, this means that L is approximately the number of coupons expected in each bin.
The test becomes unreliable for $L < 3.6$, so that RaBiGeTe discards such tests.

6.8 Non overlapping template matchings

This is a NIST's test and it is explained in the *NIST Special Publication 800-22* (available as a pdf file: "SP800-22.pdf").
The original NIST's version seems too "weak" because almost any generator pass this test (even the very bad ones), so I changed the code to programmatically calculate the test parameters.

To understand how I have improved this test, I'll give a brief description; for a complete description see the NIST's paper.

The n-bit sequence is partitioned into N independent blocks (the NIST's version uses N=8), so each independent block has $M = \lfloor n/N \rfloor$ bits. The N blocks are scanned to count how many aperiodic patterns there are in the sequence. Once the N blocks have been scanned, we have N counts (or bins) which are compared with the expected count $E[w] = (M-m+1)/2^m$ using the chi-square test (m is the block size).

The weakness in the NIST's version is the small N; so *RaBiGeTe* calculates N for any sequence length and block size. It first tries to find N for which $E[w]$=100 (this means 100 counts in each bin) with $3 \le N \le 1000$. If N lies outside that range, the program sets N=3 or N=1000 and then it calculates $E[w]$. If $E[w]$<5 the test is skipped.
This way, this test is much more sensitive and it is able to detect the most common weak generators.

6.9 AMLS test

The test is based on the Michael Mitzenmacher's "Advanced Multi-Level Strategy" in which the bits are generated by applying Von Neumann's rule to the sequences in some fixed order (the AMLS is a very efficient unbiaser).

This test is purely empirical; this means that I found the expected mean and variance of the used parameter by extensive simulations (I tested more than 9 Tbits obtained from strong cryptographic primitives like RC6, AES, Serpent and SHA-1).

Although this test might seem insensitive with usual PRNG, I wanted to include it in the suite because it seems sensitive when a good random or pseudo-random source is mixed with a biased hardware source (I said "hardware" to mean the common bias found in that kind of sources). In my experiments many tests have not been able to discriminate bad sequences with that kind of generators, but the AMLS test has shown the bias.

As the name says, the unbiaser has several levels (from 0 to L_{max}) in which the bits are stored; the one used in *RaBiGeTe* has $11 \le L_{max} \le 25$. When the test starts all the levels are empty.
The n-bit sequence is partitioned into N independent blocks, so that each block has $M = \lfloor n/N \rfloor$ bits. The size of the blocks used in *RaBiGeTe* is of the form $M = 2^L$, this means that $2 \le M \le 32768$ kbits or $2^{11} \le M \le 2^{25}$ bits.

The M-bit blocks are elaborated by an algorithm which iteratively applies the Von Neumann's method to generate a new bit and to store it in the appropriate level (the position depends on the status of the levels).
With a biased generator many bits will be discarded, while with a good generator only few bits will be discarded.
As soon as the higher level is full, the test stops; at this time exactly 2^{Lmax} bits have been processed by the unbiaser and C bits have been emitted.

For any given L_{max} (or for any given input block size M) the C's are approximately normal distributed with mean $E[C]$ and variance $Var[C]$, so that the statistic:

$$\chi^2_{obs} = \sum_{i=1}^{N} \frac{(C_i - E[C])^2}{Var[C]}$$

has an approximated χ^2 distribution with N degrees of freedom. The reported p-value is the probability $P(X > \chi^2)$ where X is a random variable.

6.9.1 *RaBiGeTe* implementation

Let μ the true mean of C and σ^2 the true variance; extensive simulations showed that the test becomes unreliable when $Var[C]$ is outside the interval $\pm 1.28\ \sigma^2$ for any block size.
For $E[C]$ the "relative tolerance" is much smaller and it is different for different block size. For example, for M = 2 kbits the test is unreliable for $0.997\mu \le E[C] \le 1.003\mu$ (or $\pm 0.003\mu$); for M = 64 kbits the interval is $\pm 0.0003\mu$ and for M = 8192 kbits the interval is $\pm 1.5 \cdot 10^{-5}\ \mu$.
To avoid that $E[C]$ and $Var[C]$ fall outside those intervals, the confidence intervals have been calculated at a confidence level $1-\alpha$:

$$E[C] - t_{(\alpha/2;N-1)}\sqrt{Var[C]/N} \le \mu \le E[C] + t_{(\alpha/2;N-1)}\sqrt{Var[C]/N} \quad @\ 100\cdot(1-\alpha)\%$$

$$(N-1)Var[C]/\chi^2_{(\alpha/2;N-1)} \le \sigma^2 \le (N-1)Var[C]/\chi^2_{(1-\alpha/2;N-1)}$$

The following table shows the confidence interval equal to 99.9% and the samples tested:

Block size	Mean				Variance				Samples tested	
kbits	min	E[C]	max	ε_1 (1)	min	Var[C]	Max	ε_2 (2)	Blocks	Gbits
2	1820.6773	1820.6821	1820.6868	5.24e-6	98.61	98.673	98.736	0.0013	52617712	100.4
4	3725.9313	3725.9403	3725.9493	4.83e-6	159.8	159.954	160.11	0.0019	23920000	91.2
8	7590.6487	7590.6663	7590.6839	4.64e-6	257.6	257.976	258.35	0.0029	10092000	77.0
16	15408.137	15408.167	15408.197	3.93e-6	414.45	415.276	416.1	0.0040	5500000	83.9
32	31185.879	31185.925	31185.972	2.95e-6	666.73	668.317	669.91	0.0048	3824000	116.7
64	62972.627	62972.676	62972.726	1.58e-6	1072.6	1074.79	1077	0.0041	5255200	320.8
128	126920.64	126920.71	126920.78	1.11e-6	1728.9	1732.77	1736.7	0.0045	4218800	515.0
256	255422.87	255422.96	255423.04	6.85e-7	2787.2	2793.34	2799.5	0.0044	4427900	1081.0
512	513408.55	513408.74	513408.93	7.49e-7	4494.8	4512.02	4529.3	0.0076	1480500	722.9
1024	1030967.8	1030968.3	1030968.8	1.01e-6	7255.6	7314.56	7374.1	0.0162	330000	322.3
2048	2068656.3	2068657.4	2068658.5	1.05e-6	11601	11757.3	11917	0.0269	119780	233.9
4096	4148191	4148192.5	4148194	7.13e-7	18732	19002	19277	0.0287	105216	411.0
8192	8313992.2	8313993.9	8313995.5	3.97e-7	30458	30841.8	31232	0.0251	137530	1074.5
16384	16656478	16656480	16656483	2.56e-7	49429	50062.8	50708	0.0255	132930	2077.0
32768	33359070	33359073	33359077	1.95e-7	79561	80781.4	82028	0.0305	92892	2902.9

(1) $\varepsilon_1 = (E[C]_{max} - E[C]_{min}) / E[C]$.

(2) $\varepsilon_2 = (Var[C]_{max} - Var[C]_{min}) / Var[C]$.

As we can see, ε_1 and ε_2 are much smaller than the requested limits, this ensure a good sensitivity.
Another way to always ensure a good level of sensitivity is that at least 5 *C*'s must be obtained from the input sequence; for example, if one wants to use 2048-kbit blocks, then the sequence length must be at least $2048 \cdot 5 = 10$ Mbits.
This test is usually more sensitive with big block size; 32768-kbit blocks are usually the most sensitive, but the speed is much slower compared to the one obtained with 2-kbit blocks.