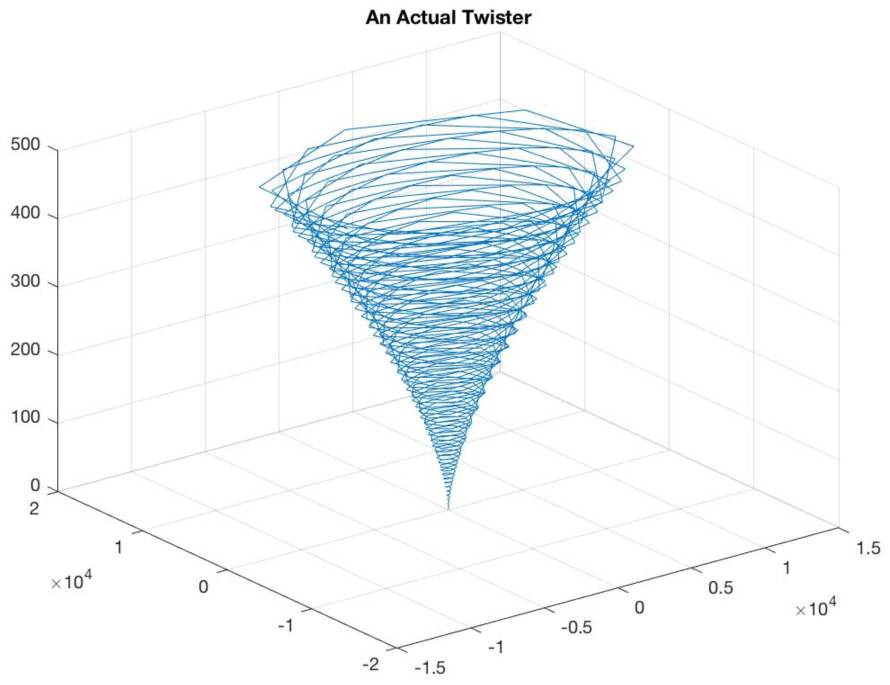
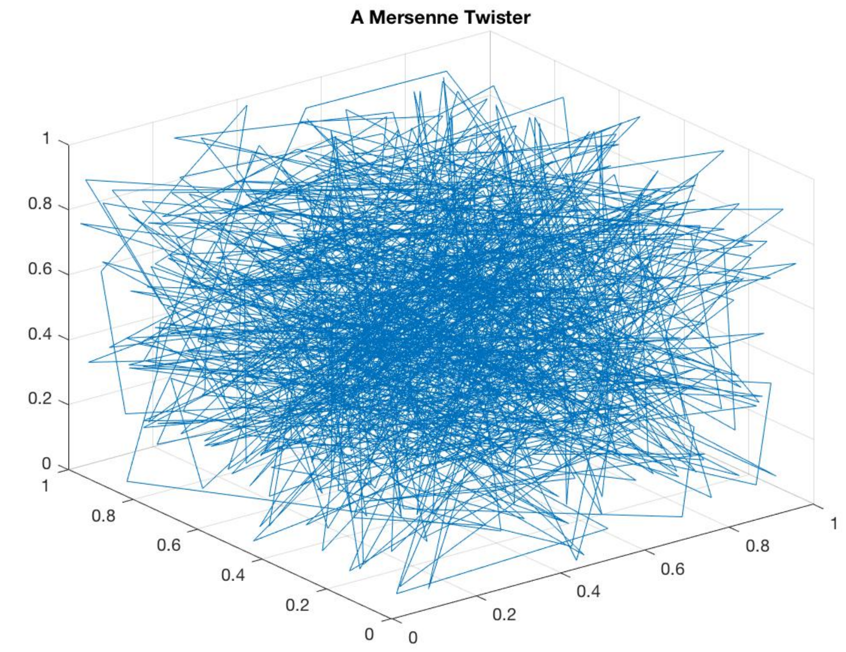
The Mersenne Twister Algorithm

Ammar Elsheshtawy





Ammar Elsheshtawy
Research Project
Math 320
15 December 2016

The Mersenne Twister Algorithm

The "random" numbers generated in software packages such as MATLAB are not truly random – they are "pseudorandom." That is, they are from sequences of numbers with properties similar to those of truly unpredictable sequences, but they do not exactly match these properties. They are sufficiently unpredictable to give the appearance of randomness but they do not meet the formal criteria for this property. Such pseudorandom sequences are generated using algorithms which take in a set number of inputs and output a pseudorandom sequence. The outputs of these algorithms are deterministic – given the same input (the "seed") they will always produce the same output. There are a wide variety of random number generator (RNG) algorithms whose simulations of true randomness are deficient in different ways. This paper discusses the Mersenne Twister (MT) algorithm which is one of the most widely used[1] and is the default option in MATLAB.

The MT was created by Makato Matsumoto and Tajuki Nishimura and the original paper describing the algorithm was published in 1998.[2] The preferred MT specification as originally described was notable for the large period of its pseudorandom sequences ($2^{19937} - 1$), having a high-dimensional "equidistribution," and being computationally efficient with a small working memory. Various adaptations and specifications of the MT have been made since the original published paper. This paper focuses on the MT19937 specification as described in the original published paper, although with a revised "initialization procedure" that was proposed in 2002.

The rest of the paper is organized as follows: an explanation of the MT algorithm (including selection of parameters), a custom MATLAB implementation of MT, a discussion of how the MT's "randomness" properties are verified, a discussion of the pros and cons of the MT, and a conclusion.

[1] Owen 2013, 4.
[2] Matsumoto and Nishimura 1998, 1.

1 – Explanation of Algorithm

The MT algorithm is described in detail in the original paper,[3] from which we base our explanation here. We first define the parameters that are used in the algorithm. These parameters can be divided into three classes:

I - Period parameters:

- $w - integer\ (machine\ word\ size)$
- $n - integer\ (degree\ of\ recursion\ and\ initialization\ array\ length)$
- $m - integer\ such\ that\ 1 \leq m < n\ (middle\ term\ used\ in\ the\ recursion)$
- $r - integer\ such\ that\ 1 \leq r \leq w\ (for\ truncation)$
- $A - constant\ w{\times}w\ \ matrix\ with\ entries\ in\ \mathbb{F}_2$
- $\boldsymbol{a} - vector\ of\ size\ w\ (last\ row\ of\ A)$

II - Tempering parameters:

- $u - integer$
- $s - integer$
- $t - integer$
- $l - integer$
- $\boldsymbol{b} - vector\ of\ size\ w\ (bitmask)$
- $\boldsymbol{c} - vector\ of\ size\ w\ (bitmask)$

III - Initialization parameter:

- $f - integer\ (32\ bit)$

The purpose of each parameter will be clearer after reading the explanation of the algorithmic process which follows.

The MT algorithm is designed to generate a sequence of "word vectors" – w-dimensional row vectors that are elements over the field $\mathbb{F}_2 = \{0,1\}$. Word vectors are given a bold typeface throughout this paper (e.g. $\boldsymbol{x}$, $\boldsymbol{a}$). The elements of these word vectors correspond to bits of binary integers in base-2. The terms of the MT output sequence are designed to be uniform pseudorandom integers in the interval $[0, 2^w - 1]$. Each term of the sequence can then be divided by $2^w - 1$ so that they are uniform pseudorandom real numbers in the interval $[0, 1]$ (i.e. from the $U(0,1)$ distribution).

The MT algorithm itself is based on the following linear recurrence:

$$\boldsymbol{x}_{\boldsymbol{k+n}} := \boldsymbol{x}_{\boldsymbol{k+m}} \oplus \left(\boldsymbol{x}_{\boldsymbol{k}}^{\boldsymbol{u}} \middle| \boldsymbol{x}_{\boldsymbol{k+1}}^{\boldsymbol{l}}\right) A, \quad (k = 0,1, \dots).$$

Where $\boldsymbol{x}_{\boldsymbol{k}}^{\boldsymbol{u}}$ is the upper $w - r$ bits of $\boldsymbol{x}_{\boldsymbol{k}}$ and $\boldsymbol{x}_{\boldsymbol{k+1}}^{\boldsymbol{l}}$ is the lower r bits of $\boldsymbol{x}_{\boldsymbol{k+1}}$. The operation $\oplus$ refers to bitwise addition modulo two, while the operation | refers to concatenating the word vector to the left of the symbol with the word vector to the right (in that order) which yields a

[3] Matsumoto and Nishimura 1998, 1.

word vector of dimension $w - r + r = w$. This concatenation can be computed using the bitwise OR operation.

The form of the matrix A is chosen so that the multiplication in the recurrence can be computed quickly. The form is:

$$A = \begin{pmatrix} 0 & I_{w-1} \\ a_{w-1} & (a_{w-2}, \dots, a_0) \end{pmatrix}$$

where I_{w-1} is the $w - 1$ identity matrix and $a_{w-1}, a_{w-2}, \dots, a_0$ are elements of a word vector $\boldsymbol{a}$.

With this form $\boldsymbol{x}A$ can be computed as follows using only one or two bit operations:

$$\boldsymbol{x}A = \begin{cases} shiftright(\boldsymbol{x}), & if\ x_0 = 0 \\ shiftright(\boldsymbol{x}) \oplus \boldsymbol{a}, & if\ x_0 = 1 \end{cases}$$

where the word vector $\boldsymbol{x} = (x_{w-1}, x_{w-2}, \dots, x_0)$.

Next is the "tempering" step. This step is done so that MT sequences achieve a higher value for a criterion that is seen as one measure of the strength of a pseudorandom sequence – "k-distribution to v-bit accuracy" – since the "raw" sequence obtained directly from the recurrence above has a "poor" value for this criterion.[4] To temper the sequence, each $\boldsymbol{x_{k+n}}$ generated by the recurrence is multiplied by a $w \times w$ invertible matrix T (called a "tempering matrix"). The multiplication $\boldsymbol{x}T \coloneqq \boldsymbol{z}$ is achieved through the following computations carried out in sequential order:

$$\begin{aligned} &\boldsymbol{y} \coloneqq \boldsymbol{x} \oplus (\boldsymbol{x} >> u) \\ &\boldsymbol{y} \coloneqq \boldsymbol{y} \oplus ((\boldsymbol{y} << s)\ AND\ \boldsymbol{b}) \\ &\boldsymbol{y} \coloneqq \boldsymbol{y} \oplus ((\boldsymbol{y} << t)\ AND\ \boldsymbol{c}) \\ &\boldsymbol{z} \coloneqq \boldsymbol{y} \oplus (\boldsymbol{y} >> l) \end{aligned}$$

where u, s, t, and l are fixed tempering parameters and $\boldsymbol{b}$ and $\boldsymbol{c}$ are fixed word vectors. The operators $>>$ and $<<$ are the bitwise right and left shifts respectively (the number of shifts is the integer on the right side of the operator).

To implement the algorithm as described above, a sequence of n word vectors $\{\boldsymbol{x_0}, \boldsymbol{x_1}, \dots, \boldsymbol{x_{n-1}}\}$ is needed so that the recurrence can be performed to generate $\boldsymbol{x_n}$ onwards. These word vectors are generated using the following recurrence relation which only requires a single w-bit integer "seed" $\boldsymbol{x_0}$ in order to be executed:

$$\boldsymbol{x_i} \coloneqq \left(f \times \left(\boldsymbol{x_{i-1}} \oplus \left(\boldsymbol{x_{i-1}} >> (w-2)\right)\right) + i\right) AND\ (2^w - 1), \quad (i = 1,2, \dots, n-1).$$

[4] Matsumoto and Nishimura 1998, 10.

where f is a fixed w-bit integer. The purpose of the AND operation in the recurrence is to select the last w bits of the expression to its left. This step ensures that each $\boldsymbol{x_i}$ is a w-dimensional word vector. The multiplier parameter f is taken to be 1812433253.[5]

We now demonstrate how the algorithm works using the parameters for the 32-bit version of the MT (specification MT19937, parameter table given later in this document) and $\boldsymbol{x_0} = \mathbf{777}$:

Step 1: Initialization

$$\boldsymbol{i} = \mathbf{1} \implies$$
$$\boldsymbol{x_1} \coloneqq \big(1812433253 \times \big(\boldsymbol{x_0} \oplus (\boldsymbol{x_0} >> 30)\big) + 1\big) AND\ (2^{32} - 1).$$

1. Choose seed $\boldsymbol{x_0} = \mathbf{777} = \underbrace{\mathbf{0 \dots 0}}_{\boldsymbol{22\ times}}\ \mathbf{1100001001}$.
2. $\boldsymbol{x_0} >> 30 = \underbrace{\mathbf{0 \dots 0}}_{\boldsymbol{32\ times}}$.
3. $\boldsymbol{x_0} \oplus 0 = \underbrace{\mathbf{0 \dots 0}}_{\boldsymbol{22\ times}}\ \mathbf{1100001001} \oplus \underbrace{\mathbf{0 \dots 0}}_{\boldsymbol{32\ times}} = \boldsymbol{x_0} = \mathbf{777}$.
4. $\mathbf{777 + 1 = 778}$.
5. $\Rightarrow \boldsymbol{x_1} = (1812433253 \times 777) AND\ (2^{32} - 1) = \underline{\mathbf{3,806,331,789}}$.
6. Repeat up to $i = 623$.

Step 2: MT Recurrence

$$\boldsymbol{k} = \mathbf{0} \implies$$
$$\boldsymbol{x_{624}} \coloneqq \boldsymbol{x_{397}} \oplus \left(\boldsymbol{x_0^u} \middle| \boldsymbol{x_1^l}\right) \begin{pmatrix} 0 & I_{31} \\ a_{31} & (a_{30}, \dots, a_0) \end{pmatrix}.$$

1. $\boldsymbol{x_0} = \mathbf{777} = \underbrace{\mathbf{0 \dots 0}}_{\boldsymbol{22\ times}}\ \mathbf{1100001001} \implies \boldsymbol{x_0^u} = \mathbf{0}$.
2. $\boldsymbol{x_1} = \mathbf{3,806,331,790} = \mathbf{1}1100010111000000000001110001110 \implies \boldsymbol{x_1^l} =$ 1100010111000000000001110001110.

[5] From http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/CODES/mt19937ar.c

3. $(\boldsymbol{x_0^u}|\boldsymbol{x_1^l}) = \mathbf{0}1100010111000000000001110001110 = \mathbf{1,658,848,142}$.
4. Recall: $\boldsymbol{x}A = \begin{cases} shiftright(\boldsymbol{x}), & if\ x_0 = 0 \\ shiftright(\boldsymbol{x}) \oplus \boldsymbol{a}, & if\ x_0 = 1 \end{cases}$
5. We have $(\boldsymbol{x_0^u}|\boldsymbol{x_1^l})_0 = 0$.
6. $\Rightarrow (\boldsymbol{x_0^u}|\boldsymbol{x_1^l})A = (1{,}658{,}848{,}142)A = \mathbf{0}0110001011100000000000111000111 =$ $\mathbf{829,424,071}$.
7. $\boldsymbol{x_{397}} = 4{,}139{,}538{,}926 = 11110110101111000101100111101110$ (from initialization).
8. $\Rightarrow \boldsymbol{x_{624}} = \boldsymbol{x_{397}} \oplus (\boldsymbol{x_0^u}|\boldsymbol{x_1^l})A = 11110110101111000101100111101110 \oplus$ $00110001011100000000000111000111 = \underline{\mathbf{3,352,057,897}}$.

Step 3: Tempering

Letting $\boldsymbol{x_{624}}T \coloneqq \boldsymbol{z_{624}}$ and given $\boldsymbol{x_{624}} = 3{,}352{,}057{,}897$:

1. $\boldsymbol{y_{624}} \coloneqq \boldsymbol{x_{624}} \oplus (\boldsymbol{x_{624}} >> 11) = \boldsymbol{x_{624}} \oplus (1{,}636{,}747) = 3{,}352{,}600{,}994$.
2. $\boldsymbol{y_{624}} \coloneqq \boldsymbol{y_{624}} \oplus ((\boldsymbol{y_{624}} << 7)\ AND\ 2636928640) = \boldsymbol{y_{624}} \oplus (2{,}281{,}721{,}856) =$ $1{,}339{,}355{,}554$.
3. $\boldsymbol{y_{624}} \coloneqq \boldsymbol{y_{624}} \oplus ((\boldsymbol{y_{624}} << 15)\ AND\ 4022730752) = \boldsymbol{y_{624}} \oplus (1{,}757{,}413{,}376) =$ $655{,}684{,}002$.
4. $\boldsymbol{z_{624}} \coloneqq \boldsymbol{y_{624}} \oplus (\boldsymbol{y_{624}} >> 43) = \boldsymbol{y_{624}} \oplus (0) = 655{,}684{,}002$.
5. $\Rightarrow \boldsymbol{R_{624}} = 655{,}684{,}002 \div (2^{32} - 1) = \underline{\mathbf{0.152663}\ldots} \in \mathbb{R}$.

Given the algorithm as described above, the question then becomes how to choose the parameters optimally so that the sequences generated by it have the most desirable qualities possible. We now comment on how the period and tempering parameters are chosen in turn:

I – Period Parameters

The period parameters are chosen such that $2^{nw-r} - 1$ is a Mersenne prime. Mersenne primes are prime numbers of the form $2^p - 1$, where p is called a "Mersenne-exponent."

The justification for this choice comes from a "general theorem of linear recurrence"[6] which states that for a linear recurring sequence produced by a linear transformation B (the main MT recurrence as described above) with characteristic polynomial $\varphi_B(t)$, "the sequence attains the maximal period $2^p - 1 = 2^{nw-r} - 1$, if and only if $\varphi_B(t)$ is primitive." A primitive polynomial is a "polynomial that generates all elements of an extension field from a base field"[7] and is irreducible – that is, it "cannot be factored into nontrivial polynomials over the same field."[8] In our case, the base field is the two-element field $\mathbb{F}_2$. For pseudo-RNG, another desirable property of the characteristic polynomial is having many terms – a property which the MT creators claim is correlated with good practical results.[9] In the case of the MT, despite having many terms (135) [10], its characteristic polynomial can still be feasibly tested for primitivity. Testing $\varphi_B(t)$ for primitivity (the MT creators use and explain an "The Inverse-decimation method" algorithm[11]) is greatly simplified when $2^p - 1$ is prime since the factors of this number do not have to be computed. Hence, choosing p to be a Mersenne-exponent allows a large period for the MT to be attained and verified relatively easily.

Fixing $w = 32$ at the machine word length for the 32-bit version of the MT (specification MT19937), we are left with choosing the parameters n and r such that $32n - r = p$ where p is a large Mersenne-exponent (so that the MT period will also be large). For a fixed p, several different choices of the parameters n and r could satisfy this equation. Hence, choosing them is not entirely trivial. The parameter n "is the order of equidistribution,"[12] and a larger equidistribution generally means a better pseudorandom generator, so it would be desirable to make this parameter as large as possible. However, n is also the number of machine words the algorithm must store so a larger n means that executing the algorithm requires more memory. Hence, there is a trade-off between making the properties of the MT as desirable as possible and adhering to practical constraints. MT19937 uses $n = 624$ indicating that this choice results in a good balance between the two competing objectives. Given p and n, r can be found using the equation above.

As in the selection of n, the selection of p is subject to a trade-off. A larger p means a larger period for the MT sequences, which is a desirable property, but it also means that the inverse-decimation method for the primitivity test to verify the period takes longer to compute (the paper mentions that this could be "several years"). The creators of the MT were able to run the test for $p = 19937$ in just two weeks. This value corresponds to a period of $2^{19937} - 1$, which is massive and hence sufficient for a wide variety of practical purposes, so $p = 19937$ is a suitable choice. Solving $32n - r = p$ with this value of p and $n = 624$ gives $r = 31$.

[6] Matsumoto and Nishimura 1998, 9.
[7] From MathWorld - A Wolfram Web Resource.
[8] From MathWorld - A Wolfram Web Resource.
[9] Matsumoto and Nishimura 1998, 3.
[10] Matsumoto and Nishimura 1998, 8.
[11] Matsumoto and Nishimura 1998, 13.
[12] Matsumoto and Nishimura 1998, 9.

II – Tempering Parameters

The techniques used to select the tempering parameters are rather complex and not quite as precise as those used to select the period parameters. The tempering parameters are chosen to achieve as good of a "k-distribution property" as feasible.[13]

The "k-distribution test" is designed to assess an RNG's high-dimensional uniformity properties. Ideally, pseudo-RNG algorithms should generate sequences that are not only uniform in the interval $[0,1]$ but also uniform in the k-dimensional unit cube $[0,1]^k$ when consecutive k-tuples of the sequence are placed in this cube.[14] Pseudo-RNGs have sometimes been found to lack uniformity for values of k as low as 2 or 3 despite apparent uniformity for $k = 1$.[15] Hence, considering high-dimensional uniformity is an important step to be taken when evaluating a pseudo-RNG.

The "k-distribution" can be interpreted as follows[16]: given an entire period of output, consisting of P points, take the consecutive k-tuples starting at each point and put them in the k-dimensional unit cube with coordinates $(x_i, x_{i+1}, \dots, x_{i+k-1}), (i = 0,1, \dots, P-1)$ – where the addition in the subscripts is done modulo P so that the tuple values loop once $i > P - k$. Then, divide each $[0,1]$ axis into 2^v equal chunks – so that the chunks separate coordinates that have bits which differ past the v^{th} position (in other words: "we are only considering the most significant v bits of each coordinate"). We have now divided the unit cube into 2^{kv} smaller cubes. We say that the sequence is k-distributed to v-bit accuracy if each of these cubes "contains the same number of points (except for the cube at the origin, which contains one less)." The "k-distribution test" involves finding the largest values of k and v such that this property holds. A larger $k(v)$ signifies a higher-dimensional equidistribution up to v bits of accuracy so, other things equal, it is desirable for these values to be as large as possible.

The MT creators find $k(v)$ values for the MT using the "lattice" method (explained in Matsumoto and Nishimura 1998, pg. 15-17). Theory suggests that an upper bound for the MT k-distribution $k(v) \leq [\frac{nw-r}{v}]$, although the MT creators were not able to find parameters which achieved this upper bound. Their search for tempering parameters (process described in Matsumoto and Kurita 1994) yielded values which give a good k-distribution (623-distributed to 32-bit accuracy), but not the maximum theoretically possible. These values are provided in the table below.

[13] Matsumoto and Nishimura 1998, 17.
[14] Matsumoto and Nishimura 1998, 2.
[15] http://statweb.stanford.edu/~owen/mc/Ch-unifrng.pdf
[16] Matsumoto and Nishimura (1998), pg. 3

Table 1: MT19937 Parameter Values

Period Parameter	Value
w	32
n	624
m	397
r	31
w	32
$\boldsymbol{a}$	2567483615
Tempering Parameter	Value
u	11
s	7
t	15
l	43
$\boldsymbol{b}$	2636928640
$\boldsymbol{c}$	4022730752
Initialization Parameter	Value
f	1812433253

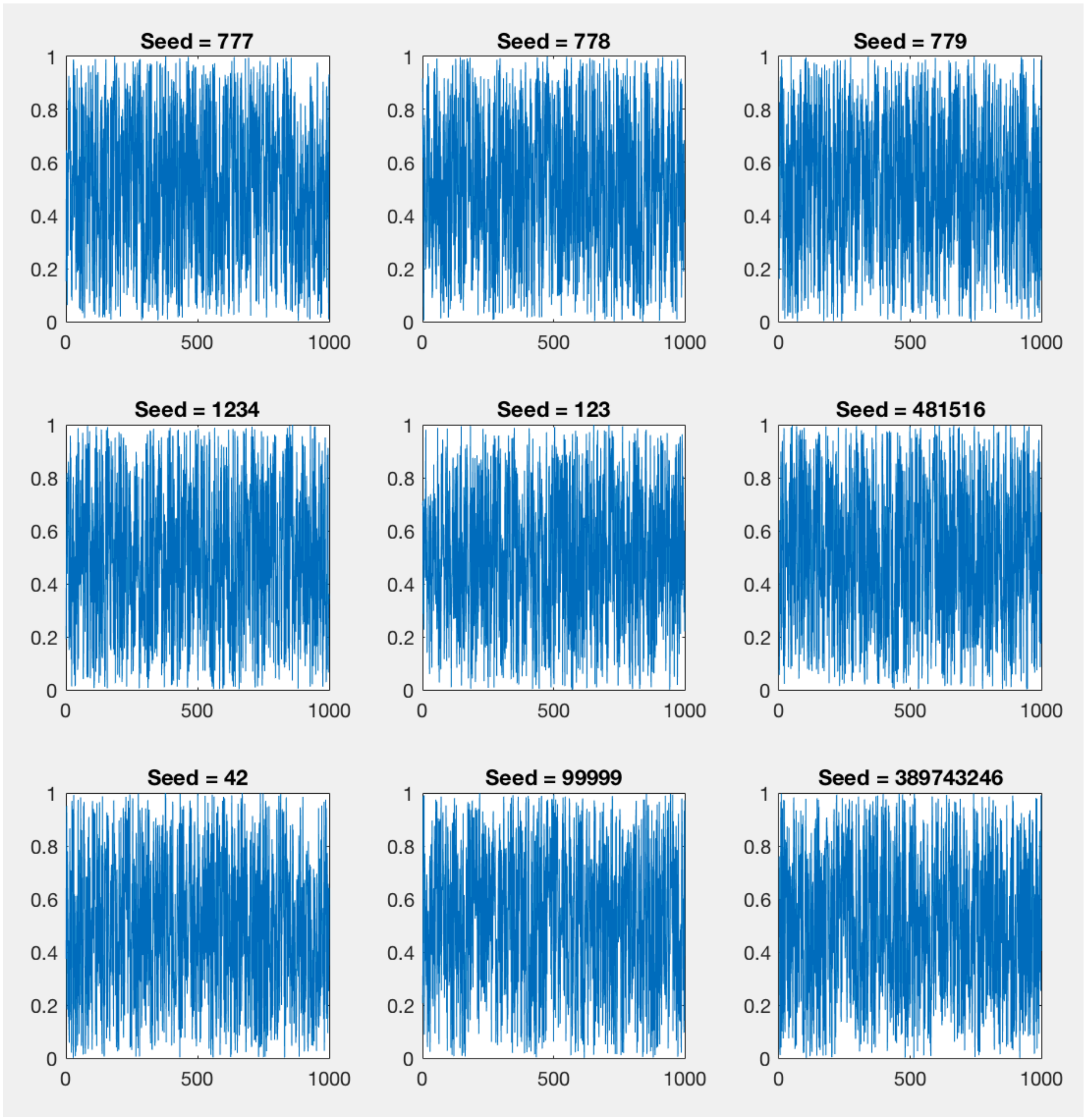
2 – MATLAB Implementation

What follows is a custom MATLAB implementation of the MT algorithm. Note that the following implementation is intended to demonstrate the algorithmic process and the properties of the MT output sequences but not to be as efficient as possible. The complete MT algorithm only require 624 machine words of working memory in order to operate (older terms of the sequence are written over when possible) while, for simplicity, no significant effort to limit working memory was made when creating the implementation below. The process for determining each pseudorandom number is as specified for the MT algorithm, so the output sequences should have identical properties, although the storing of these numbers is done slightly differently.

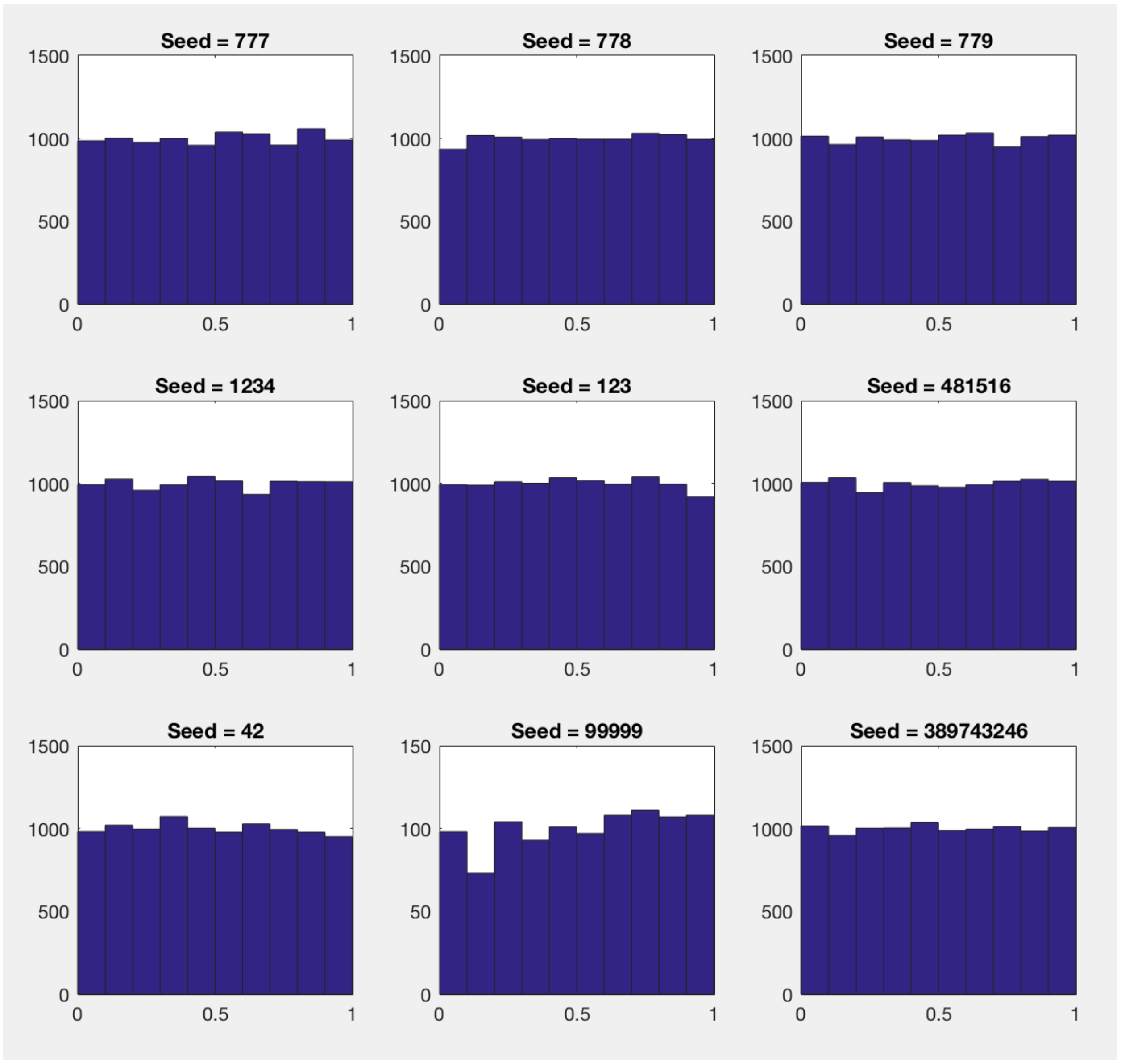
The function takes in an input seed (used to start the initialization recurrence) as well as the desired number of pseudorandom terms. The first part of the code defines the constants for MT19937. Then, a check is done to ensure that the input seed is a 32-bit integer. If this check is passed, the initialization recursion as defined above is executed and the 623 outputs are stored along with the input seed in a "state vector." Then, the actual MT algorithm begins using this state vector. The truncation and concatenation steps are carried out first, followed by multiplication by A, and then followed by bitwise addition. The word vector produced is then multiplied by the tempering matrix T through a series of bitwise operations as described above. Finally, this term is added to the end of the state vector and the recursion loops until the desired number of pseudorandom terms are computed. The terms produced from the MT recursion are then extracted from the state vector and transformed to real numbers in the interval [0,1] before being output.

```
function seq = myMT(seed, length)
% seq = myMT(seed, length) outputs a uniformly distributed pseudo-random
% sequence of real numbers in the interval [0,1] with the nunmber of terms
% specified through the input length. The seed input is used to start the
% initialzation recurrence. The output of this recurrence is then fed into
% the Mersenne twister recurrence from which the myMT output numbers are
% generated.
% input:
%   seed = seed value for initialization algorithm (32 bit unsigned integer)
%   length = number of terms desired for output sequence (positive integer)
% output:
%   seq = vector of pseudorandom real numbers in the interval [0,1]

% hard coded constants for MT19937
w = 32;   % machine word size
n = 624;  % degree of recursion and initialization array length
r = 31;   % for truncation step
m = 397;  % middle term used in the recursion

a = uint32(hex2dec('9908B0DF')); % last row of matrix A

u = 11;                  % tempering constant
s = 7;                   % tempering constant
b = uint32(2636928640); % tempering bitmask
t = 15;                  % tempering constant
c = uint32(4022730752); % tempering bitmask
l = 43;                  % tempering constant

f = uint64(1812433253); % initialization constant

% ensures seed is a 32 bit integer
if ((seed <= 0) | (seed >= (2^w - 1)) | ((floor(seed) - seed) ~= 0) |
((floor(seed) - seed) ~= 0))
    error('Seed must be a 32 bit integer')
end


% begin initialization
mask = uint64((2^w) - 1); % defining truncation mask
state = [uint64(seed)];   % putting chosen seed in state vector

% executing initialization recursion
for i = 1:(n-1)
    x_i = (f * (bitxor(state(i), (bitshift(state(i), -(w - 2), 'uint64')))))
+ i;
    x_i = bitand(x_i, mask); % lowest w bits
    state = [state x_i];
end

state = uint32(state); % converting to 32 bit integers for next step
% end initialization
% start MT algorithm
% defining truncation masks
mask_u = uint32((2^(w) - 1) - (2^(r) - 1));
mask_l = uint32(2^(r) - 1);

% loop to generate specified number of terms
for i = 1:length
    % truncation and concatenation step
    y = bitor((bitand(state(i), mask_u)),(bitand(state(i+1),mask_l)));

    % executing recursion
    % multiplying by A
    yA = bitshift(y, -1); % yA if lowest bit of y is 0
    if (mod(y,2) ~= 0)    % yA if lowest bit of y is 1
        yA = bitxor(yA, a);
    end

    x_km = state(m + i); % middle term
    x_i = bitxor(x_km, yA);  % bitwise addition

    % tempering steps
    y = x_i;
    y = bitxor(y, bitshift(y, -u));
    y = bitxor(y, bitand(bitshift(y, s),b));
    y = bitxor(y, bitand(bitshift(y, t),c));
    y = bitxor(y, bitshift(y, -l));

    state = [state y]; % adding term to state vector
end


output = state(625:(624 + length)); % removing initialization terms

seq = double(output) / ((2^w) - 1); % transforming output to [0,1]
end
```

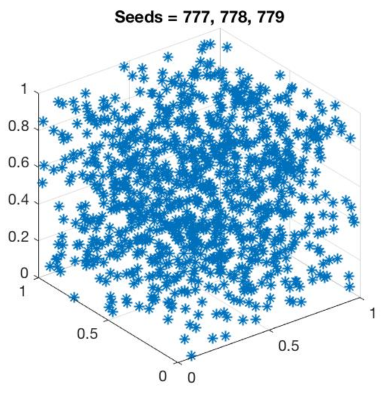
The following images show plots of the output of the above code for various seed values. There do not appear to be any obvious patterns in the sequences, and the sequences look quite different even for similar seed values:

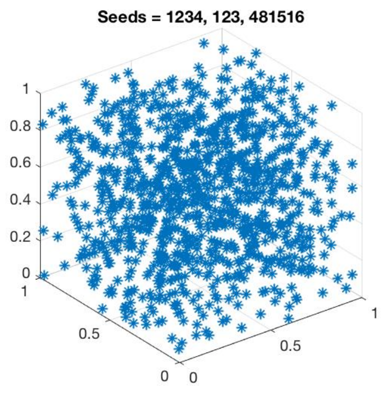


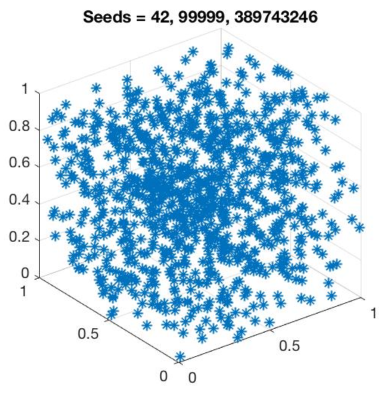
The following images show histograms of the output for the same seed values as above, although this time with 10,000 terms per sequence. The distributions appear quite close to that of a $U(0,1)$. Note that since the sequences are finite and well below the MT period length, it is expected that their distributions will not exactly match that of a $U(0,1)$. It is unsurprising that individual pseudo-RNG subsequences exhibit imperfections, some notable, as exemplified by the histogram below for a seed value of 99999 which has a notable dip in the frequency of observations for the second interval from the left:

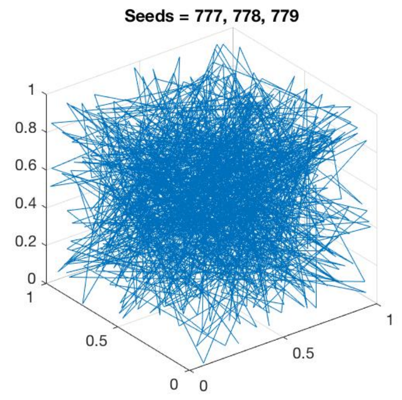


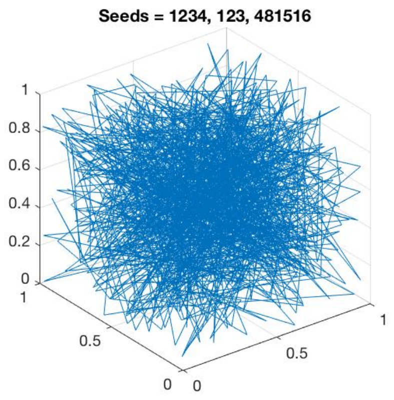
The following plots of three MT sequences against each other illustrate that the MT outputs for different seed values show little obvious correlation, and also show how values are dispersed within a low dimensional unit cube. The first three just show the points, while the second 3 join them together:

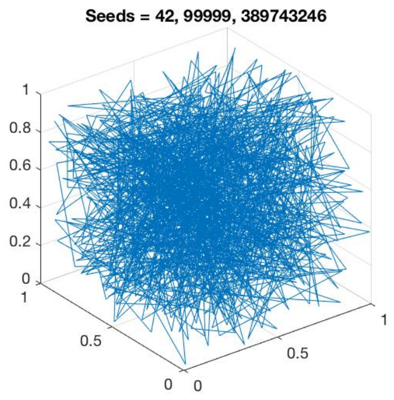






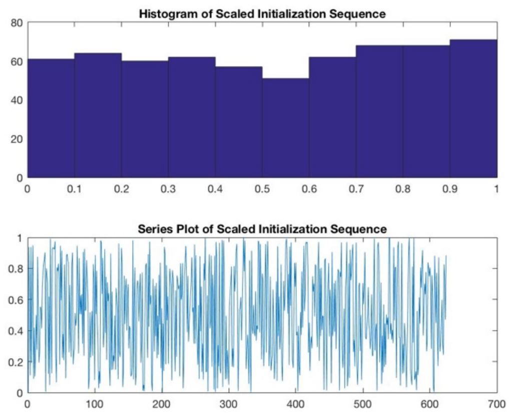




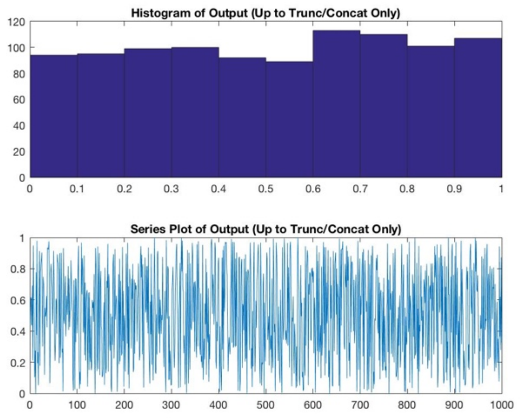


The following images show what the MT output looks like if the code is only executed up to certain stage for a seed value of 481516 and a desired sequence length of 1000:

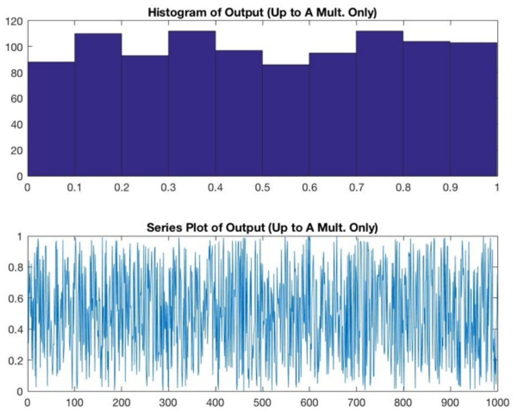
Stage 1 – Initialization Sequence (624 terms)



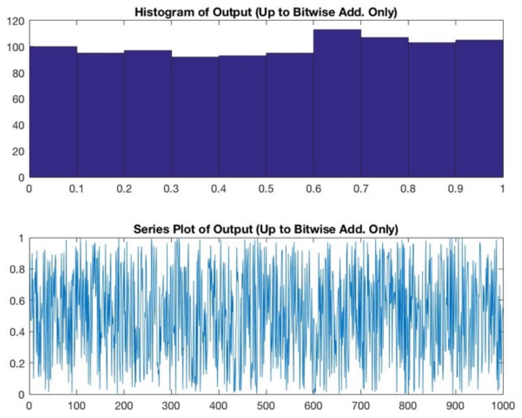
Stage 2 –Trunc., Conc. (1000 terms)



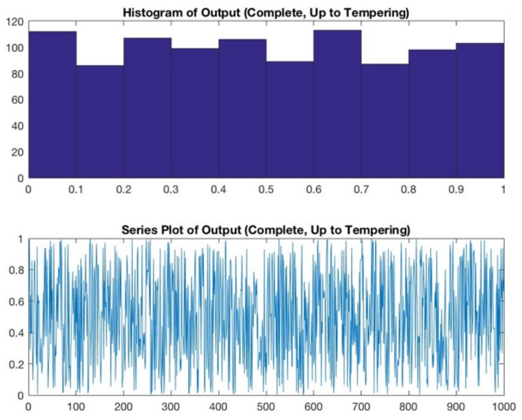
Stage 3 – Multiplying by A (1000 terms)



Stage 4 – Bitwise Add. (1000 terms)



Stage 5 – Tempering (1000 terms)



Just from looking at what the sequence distribution and plot look like at each stage, it is difficult to appreciate what the purpose of each step is. All of the series plots reflect what random, white noise should look like and the sequence terms at each stage appear to be fairly uniformly distributed in the interval a [0,1] as intended. These descriptions even apply to the initialization sequence – the sequence before any of the MT-specific steps are executed.

These similarities highlight just how subtle the differences between RNGs are. While it may not be visually apparent what exactly the value of each step is in making the output sequence “more random,” each step helps to ensure that the sequence satisfies some statistical and mathematical criteria that have been shown, through theory and practice, to be good measures of randomness.

3 – Evaluating the Randomness of MT Sequences

Intuitively, a sequence of random numbers is unpredictable and has no recognizable patterns. More formally, a random number (from a uniform distribution, as MT outputs simulate) is one that is drawn from a set of possible values, each of which is equally probable. In a sequence of random numbers, each random number drawn must also be statistically independent of the other.[17] [18] Of course, due to the large number of possible initialization arrays and massive length of each resulting MT sequence, it is infeasible to verify these intuitions for the entirety of every possible MT sequence.

As described above, the MT sequences are designed to have theoretical properties that are considered to be good measures of randomness, including a large k-distribution (623-distribution to 32-bit accuracy) and a characteristic polynomial of linear transformation with many terms. While the large k-distribution indicates a high dimensional equidistribution when considering an entire period of output, it does not necessarily indicate that subsets of these sequences will also have good properties.[19]

Most applications of the MT will not use its full period and, due to the nature of randomness, it is still possible to observe subsets of numbers that the average person would not suspect to be random or that would fail basic statistical checks for randomness.[20] While particular subsets of MT sequences may appear non-random, we would like to verify that the majority of subsets do in fact appear sufficiently random so that we can have confidence in using the MT for practical applications that require reasonable approximations of true randomness. Verifying this is difficult, however, due to the infinite number of potential patterns that one could look for in the sequences. Yet, many of these patterns are of little consequence for most practical applications so the solution lies in selecting tests that examine patterns which could pose problematic for the application in question.

[17] From https://www.random.org/randomness/
[18] From MathWorld--A Wolfram Web Resource.
[19] Owen 2013, 13.
[20] Owen 2013, 15.

The issue is summarized well in a 2007 paper by Pierre L'Ecuyer and Richard Simard:

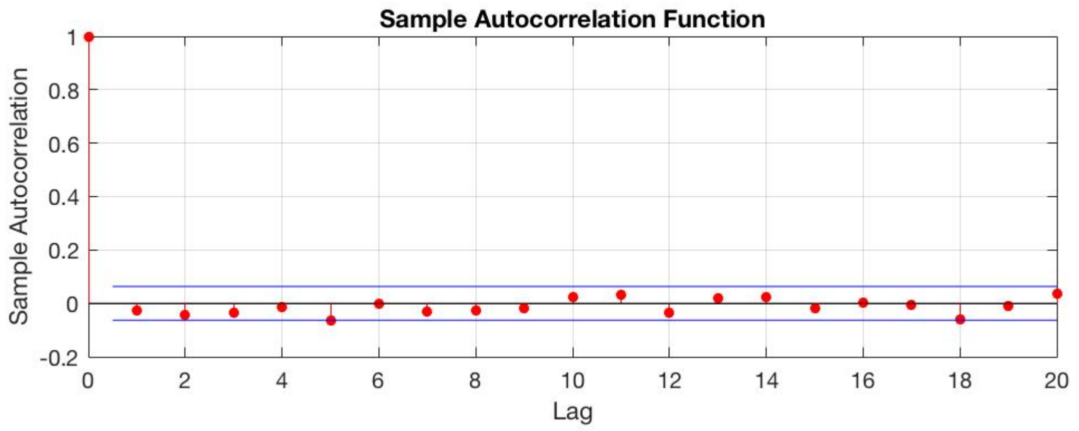
"No universal test or battery of tests can guarantee, when passed, that a given generator is fully reliable for all kinds of simulations. But even if statistical tests can never prove that an RNG is foolproof, they can certainly improve our confidence in it. One can rightly argue that no RNG can pass every conceivable statistical test. The difference between the good and bad RNGs, in a nutshell, is that the bad ones fail very simple tests whereas the good ones fail only very complicated tests that are hard to figure out or impractical to run."

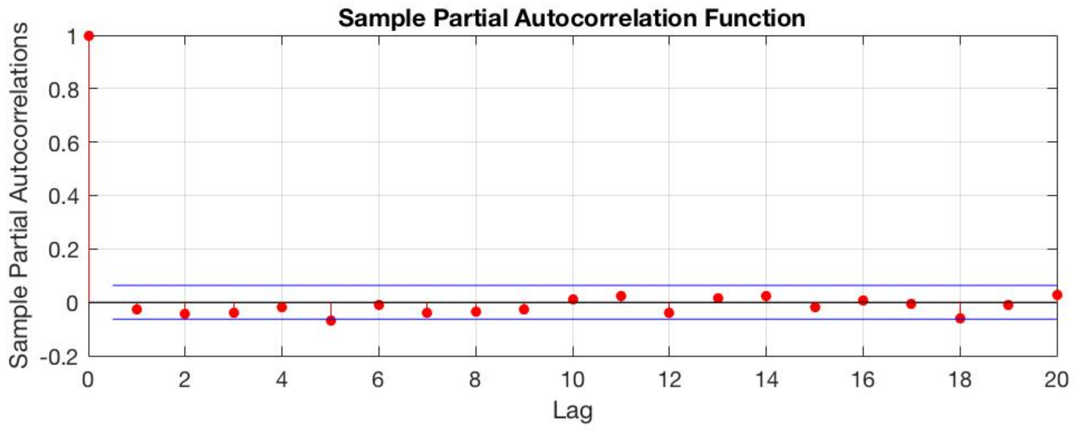
We now discuss empirical tests that can be used to statistically evaluate whether subsets of MT sequences reflect true random sequences. The more of these tests that typical MT sequences pass, the more confident we can be in their approximation to true randomness:

I – Serial Correlation

Serial correlation is a fairly intuitive statistical concept which can be used to simply evaluate the independence criterion in our notion of randomness. It refers to the degree by which current values of a sequence are correlated with previous values of the same sequence. For example, the strength of the correlation between all the terms of a sequence and all the terms two-terms behind them can be evaluated. If there is sufficient evidence for such a correlation, then the sequence is not truly random since a future value of the sequence can be predicted (albeit perhaps not perfectly) using values of the sequence which have already been observed.

A correlogram is a chart typically used to evaluate serial correlation. It visualizes computed autocorrelation coefficients for terms of the sequence various displacements away from each other (the correlation between terms of the sequence that displacement apart) as well as the partial autocorrelation coefficients, which measures the correlation between terms of the sequence specified displacements apart after controlling for correlations between the intermediate terms. A correlogram for an MT sequence (seed = 48118766, length = 1000) is provided below:





The blue lines in the plot denote the 95% Bartlett bands – under the null hypothesis of the sequence truly being random white noise (no serial correlation), it is expected that the autocorrelation and partial autocorrelation coefficients will exceed these bands only 5% of the time. The vast majority of the calculated coefficients fall within these bands in both plots, indicating that there is no evidence to reject the null hypothesis of the sequence exhibiting no serial correlation.

A Ljung-Box test can be used to provide a single numerical test statistic to test the null hypothesis of no serial correlation up to a specified lag k. The alternative hypothesis in this case is that some autocorrelation coefficient from a lag of 1 up to the specified lag k is non-zero. For an MT sequence with seed = 48118766, length = 1000, the corresponding p-values for the values of k = 5, 10, 15, and 20 are 0.4553, 0.2882, 0.5158, and 0.5143, all above the standard significance level of 5%. Hence, there is insufficient evidence to reject the null hypothesis of no serial correlation.

II – Batteries of Statistical Tests: (TestU01, Diehard Tests)

There are a large number of statistical tests that have been proposed for testing uniformity and independence of sequences of numbers. All of these tests have various strengths and weaknesses – some generators may fail some tests while passing others.[21] Thus, when evaluating an RNG it is important to consider a variety of statistical tests rather than just a single one. The question then becomes which tests to select. Researchers experienced with RNGs may know which specific patterns could be problematic for their particular applications and can apply tests which cater to those specifically, but many researchers lack this specialized knowledge. Many researchers face the issue of how to decide which and how many statistical tests should be used out of the many that have been proposed. To address this problem, several suites of statistical tests have been designed over the years that can be used to comprehensively and easily evaluate the randomness of RNG output.

One of the most prominent of these is the DIEHARD suite which was proposed in 1996 and distributed on CD-ROM. The MT creators noted in their original paper describing the algorithm that it passes the DIEHARD suite.[22] Since 1996, DIEHARD has largely been superseded by and improved upon by the TestU01 suite which was proposed in 2007. Many of the tests included in TestU01 are based on those included in DIEHARD and TestU01 also includes many other tests that have been proposed since DIEHARD was first made available.[23]

TestU01 consists of three pre-specified batteries of tests, each containing a subset of the tests that the researchers include in the package. These are "Small Crush" (10 tests), "Crush" (96 tests), and "Big Crush" (106 tests), listed here in increasing order of computational intensity (and hence time taken to run). MT19937 passes all but two of the tests across the various batteries – a great performance. The tests it fails are "linear complexity tests for bit sequences," which the researchers note are failed by all RNGs of the class that the MT belongs to (GFSR) since "their bit sequences obey linear recurrences by construction."[24] Hence, the MT failing these tests is not noteworthy.

The tests included in TestU01 can be broadly divided into three categories. The first, "tests on a single stream of n numbers," includes basic tests that assess autocorrelation (the same property assessed in the previous section) as well as tests that compare the distribution of the sequence being tested to that of a theoretical $U(0,1)$. This category also includes tests that look for "clustering" of sequence terms as well as tests that incorporate the lengths of increasing or decreasing subsequences.[25]

The second category is called "tests based on n subsequences of length t." These primarily examine the distribution of and distances between points within various partitions of the t-dimensional unit cube.[26]

[21] L'Ecuyer and Simard 2007, 3.
[22] Matsumoto and Nishimura 1998, 1.
[23] L'Ecuyer and Simard 2007, 2.
[24] L'Ecuyer and Simard 2007, 2.
[25] L'Ecuyer and Simard 2007, 9.
[26] L'Ecuyer and Simard 2007, 10.

The third category is called "tests that generate n subsequences of random length." These tests place the randomly generated numbers in various subsequences, as they are produced, until some criterion is met – for example, dividing the unit interval into a set number of equally spaced partitions and examining how many numbers must be drawn before there are an equal number of points in each partition. The number of draws needed for such a criterion to be met is then incorporated into a test statistic.[27]

Other tests include in the suite look at bit sequences specifically. These include "random walk tests, linear complexity tests, a Lempel-Ziv compression test, several Hamming weights tests, matrix rank tests, run and correlation tests, among others."[28]

Since MT passes the vast majority of the wide variety of empirical tests included in TestU01, we can safely deem it a good simulator of a true $U(0,1)$ generator.

4 – MT Pros and Cons

We now give an overview of good and bad qualities of the MT algorithm:

Pros:

I – Fast Computation

The MT algorithm makes "full use of the polynomial algebra over the two-element field"[29] to efficiently execute its recurrence. As described above, matrix multiplication can be done very efficiently using just bit operations – eliminating the need for extensive use of comparatively computationally expensive operations such as multiplication. Hence, the algorithm can produce a large volume of pseudorandom numbers in a short period of time. In a 2007 paper (applying TestU01 to several RNG algorithms), the time required to generate 10^8 random numbers from the MT on a 32-bit machine with clock speed 2.8 GHz was recorded as 4.3 seconds – towards the low end of the CPU time distribution for the various algorithms tested, and one of the few at the low end of this distribution that passed the overwhelming majority of the TestU01 tests.[30] This is a desirable property for simulation applications, for example, since less computational resources used for generating random numbers leaves more that can be used for the simulation itself – allowing faster results and/or more complex simulations using the same total amount of resources. This property also makes MT RNG more efficient than true RNG methods based on physical processes, which usually cannot match such an output speed.[31]

II – Small Working Memory

The MT algorithm only utilizes 624 machine-words of working memory in its most efficient implementations.[32] This is a desirable property for large simulations where multiple instances of

[27] L'Ecuyer and Simard 2007, 16.
[28] L'Ecuyer and Simard 2007, 24.
[29] Matsumoto and Nishimura 1998, 2.
[30] L'Ecuyer and Simard 2007, 28.
[31] Owen 2013, 4.
[32] Matsumoto and Nishimura 1998, 1.

the MT might need to be run at once. Potentially limited memory can then be used to store simulation programs and outputs instead of the RNG algorithms themselves.

III – Large Period

The MT period is on the order of 10^{6001}– far larger than the estimated number of atoms in the universe, which is on the order of 10^{80}.[33] This number is more than enough for practical purposes. At the computational speed quoted above, it would take more than 10^{5993}seconds to reach the end of the period. In years, this amount of time is on the order of about 10^{5985}.

IV – Passes Most Statistical Tests

As discussed above, the MT performs well empirically. It passes all of the TestU01 tests aside from the ones that specifically look for evidence of its nature as a linear recurrence. Hence, it produces almost indistinguishable simulations of draws from a true $U(0,1)$ and hence can one can safely approximate draws from this distribution using the MT for many applications.

Cons:

I – Not Good for Cryptography

The primary downfall of the MT is its deterministic nature as a simple linear transformation of a linear recurrence. Hence, after observing enough of the MT output, one can guess the current state of the algorithm and predict which numbers it will output next. This feature makes the raw MT output insufficient for applications that require complete unpredictability from the perspective of an outside observer, such as cryptography. However, MT output can be modified using a “hashing algorithm” to make it more suitable for such purposes.[34]

5 – Conclusion

The excellent mathematical properties and empirical performance of the MT come at a low computational resource cost. Hence, it is an efficient and suitable choice for many research applications involving simulation. The MT algorithm as-is should not be used for cryptographic applications, however, as explained above.

In the years since the publication of the original paper, the MT creators have developed another algorithm based on the original MT. Called the “Mersenne Twister for Graphic Processor” (MTGP), this newer version takes advantage of the wider adoption of GPUs in the years since the original MT to “provide better statistical quality for the same amount of storage”[35] and to generate multiple independent pseudo-RNG sequences simultaneously – up to “128 independent pseudorandom number sequences for each period.”[36] Hence, pseudo-RNG is continuing to evolve alongside processing power in order to achieve even greater efficiency and improved practical performance. While we will never be able to create an RNG that can be unequivocally proven to be truly random, modern pseudo-RNGs cater to most practical purposes and seem to be approaching the limits of humanity’s ability to distinguish between their outputs and true random sequences.

[33] From https://en.wikipedia.org/wiki/Observable_universe#Matter_content_.E2.80.93_number_of_atoms
[34] Matsumoto and Nishimura 1998, 5.
[35] Saito and Matsumoto 2012, 2.
[36] From http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MTGP/index.html

Bibliography

Haahr, Mads. " Introduction to Randomness and Random Numbers." RANDOM.ORG. Accessed December 15, 2016. https://www.random.org/randomness/.

L'ecuyer, Pierre, and Richard Simard. "TestU01: A C Library for Empirical Testing of Random Number Generators." ACM Transactions on Mathematical Software 33, no. 4, Article 22 (August 2007), 40 pages. doi:10.1145/1268776.1268777.

Matsumoto, Makoto, and Takuji Nishimura. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator." ACM Transactions on Modeling and Computer Simulation 8, no. 1 (January 1998): 3-30. doi:10.1145/272991.272995.

Matsumoto, Makoto, and Takuji Nishimura. Dept. Math., Hiroshima Univ. January 26, 2002. Accessed December 15, 2016. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/CODES/mt19937ar.c.

Matsumoto, Makoto, and Takuji Nishimura. "Mersenne Twister for Graphic Processors (MTGP)." Dept. Math., Hiroshima Univ. Accessed December 15, 2016. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MTGP/index.html.

Owen, Art B. Monte Carlo theory, methods and examples. 2013. Accessed December 15, 2016. http://statweb.stanford.edu/~owen/mc/.

Saito, Mutsuo, and Makoto Matsumoto. "Variants of Mersenne Twister Suitable for Graphic Processors." ACM Transactions on Mathematical Software 39, no. 2, Article 12 (February 2013), 20 pages. doi:10.1145/2427023.2427029.

Weisstein, Eric W. "Primitive Polynomial." Primitive Polynomial -- from Wolfram MathWorld. Accessed December 15, 2016. http://mathworld.wolfram.com/PrimitivePolynomial.html.

Weisstein, Eric W. "Irreducible Polynomial." Primitive Polynomial -- from Wolfram MathWorld. Accessed December 15, 2016. http://mathworld.wolfram.com/IrreduciblePolynomial.html.

Weisstein, Eric W. "Random Number." Random Number -- from Wolfram MathWorld. Accessed December 15, 2016. http://mathworld.wolfram.com/RandomNumber.html.