Random Number Generator with Elementary Cellular Automata in Matlab

Donald Bertucci*

Abstract: In this paper, I implement a random number generator that uniformly generates numbers between 0 and 1. I implemented the `rand_eca()` function to sample from a known simple yet chaotic system of Elementary Cellular Automata rule 30. I then show that my `rand_eca()` function is likely a uniform random distribution by using the χ^2 test. Finally, I implemented an algorithm to map the uniform `rand_eca()` to any another distribution (like standard normal). All the Matlab code is open source at https://github.com/xnought/rand-eca and shown in the Appendix.

1 Introduction

Elementary Cellular Automata (ECA) have extremely simple rules that transform the current state (a 1D list of on and off cells) to the next state and from that state to the next state and so on [2, 3, 4]. A simple example is shown on the left in Figure 1 for one time step. The simplicity might lead you to believe the system is predictable, but you would be wrong (see Figure 1 on the right). Some ECA update rules, like rule 30, lead to emerging complex patterns that are unpredictable and chaotic [2, 3, 4].

To be specific, at the current time step t you could not predict the state n steps into the future $t + n$ without computing all the steps in between (computationally irreducable)[2, 3, 4]. There are no shortcuts or linear patterns: you have to manually apply the rule n times to get to $t + n$ [2, 3, 4]. Since the more unpredictable a system is, the more seemingly random it is, I created a Matlab random uniform function from rule 30 which I called `rand_eca()`.

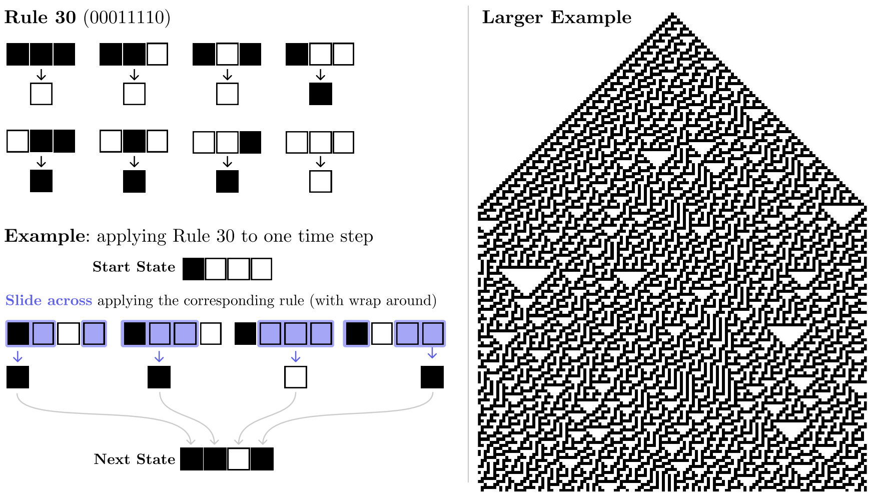

Figure 1: On the left I transform the 1D list of on (black) and off (white) cells from a **Start state** to a **Next state** with rule 30. On the right I apply rule 30 for hundreds of iterations starting from a 129 cell wide start state with 1 black cell in the middle.

*Oregon State University, donnybertucci.com

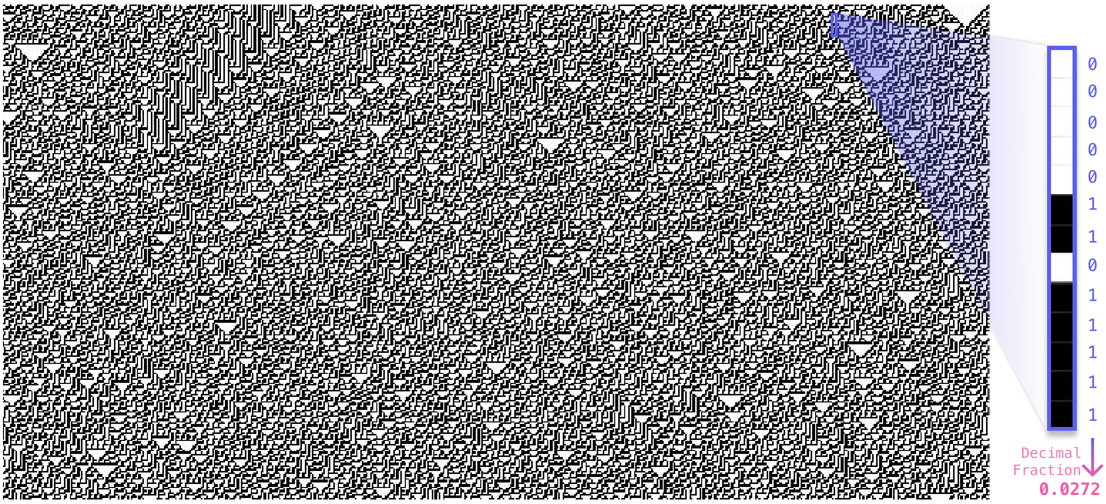

Figure 2: Running ECA rule 30 for 512 generations from the starting state of 513 cells with 1 in the middle. Here I only visualize row 257 to 513. On the right I show a zoomed-in column and how the binary representation can be converted to decimal.

2 Implementation of `rand_eca()`

In Matlab, I implemented the `rand_eca()` in two steps: running generations of ECA with rule 30 (Section 2.1) and converting the ECA output to numbers between 0 and 1 (Section 2.2).

2.1 Generating Elementary Cellular Automata

Given a starting state as a one-dimensional list of 1s (on states / black) and 0s (off states / white), I first implemented a function to produces the next state just like the left side in Figure 1.

I implemented a function called `rule30(vec3)` (Appendix A) which takes a vector with three cells to produce the ith cell in the next state. I then produced each index i in the next state in `next_state_rule30(current_state)` (Appendix B) by sliding the `rule30` function over the `current_state`.

Then to produce many iterations of rule 30, I implemented a `iterate_rule30(start_state, n_iterations)` (Appendix C) function which produced the matrix visualized on the right in Figure 1. All this function does is apply the `next_state_rule30` for `n_iterations` and saves the result in a matrix. To make things easier for visualization, I created a function called `visualize_rule30(start_state, n_iterations, visualize_from_i)` (Appendix D) which simply runs `iterate_rule30` and outputs a Matlab figure from rows `visualize_from_i` till the last row. For example, `visualize_rule30([zeros(1, 256) 1 zeros(1, 256)], 512, 257)` produced the Figure 2 and `visualize_rule30([zeros(1, 64) 1 zeros(1, 64)], 200, 1)` produced the Figure 1.

2.2 Extracting Numbers

Next, as shown in Figure 2 on the right, I can take a single column from the generations of rule 30 and interpret the column as number between $[0, 1)$. In my case I wanted 13 bits for each number, so I can take a 13 cell long column from the generation and interpret the white cells as 0 and the black cells as 1. First I can convert from base 2 to base 10 as

$$d(b) = b_1 \cdot 2^0 + b_2 \cdot 2^1 + \cdots + b_n \cdot 2^{n-1} \tag{2.1}$$

where b_i is the bit as index i in a bit string starting from left to right and n is the total number of bits in that bit string.

Then to force the decimal d into a fraction value, I'll divide the representation by the largest possible decimal of all $1s$ like $11\dots1$ which is just $2^n - 1$ in decimal (since 0 is all $00\dots0$ we -1 to 2^n possible combinations). Because I want the fractional decimal to never be 1 itself, I'll add 1 to the divisor as $2^n - 1 + 1 = 2^n$. Simplifying the normalized expression I get

$$\begin{aligned} f(b) = \frac{d(b)}{d(11\dots1)+1} &= \frac{b_1 \cdot 2^0 + b_2 \cdot 2^1 + \cdots + b_n \cdot 2^{n-1}}{(2^n - 1) + 1} \\ &= \frac{b_1 \cdot 2^0}{2^n} + \frac{b_2 \cdot 2^1}{2^n} + \cdots + \frac{b_n \cdot 2^{n-1}}{2^n} \\ f(b) &= b_1 \cdot 2^{-n} + b_2 \cdot 2^{n-1} + \cdots + b_n \cdot 2^{-1}. \end{aligned} \tag{2.2}$$

It's not difficult to see that the bounds of (2.2) is $[0, 1)$ for finite length bit strings.

In Matlab, I implemented the function `bits_to_fractions(bits, n, bits_per_number)` (Appendix E) which produces `n` numbers where each of them are between $[0, 1)$ using (2.2). `bits_to_fractions` takes `bits_per_number` number of bits in each conversion to decimal fraction. In other words, I chunk over the `bits` vector with a chunk size of `bits_per_number` and convert each to a decimal fraction. This will be how I can convert a single ECA column into multiple numbers.

2.3 Putting the pieces together for `rand_eca()`

Then, I combined the last two sections to convert every column of a rule 30 ECA generation into numbers between $[0, 1)$.

Others have taken similar approaches. For example, when extracting the middle column, this method has been shown to be aperiodic and therefore good for random number generation [1]. I instead take every single column, not just the middle column to speed up the computation.

The final function is called `rand_eca(rows, columns)` (Appendix F) where you can generate a matrix of shape (rows, columns) of random uniform values. In `rand_eca`, I chose 13 bits which corresponds to a single number and I started from a starting state of 512 zeros on the left, a single 1, and 512 zeros on the right (in total 1025 cells). I then warmed up the starting state by first iterating 512 times. This warmup essentially turns the starting pyramid (like in Figure 1) to completely populated (like in Figure 2).

Once I call the `rand_eca`, the next seed is changed to the last state/row computed from `rand_eca`. If you would like to reseed the the function, you can call `rng_eca(offset)` (Appendix H), like `rng_eca(0)` to reset the seed. The `offset` describes how many in addition to the warmup to initially generate the seed.

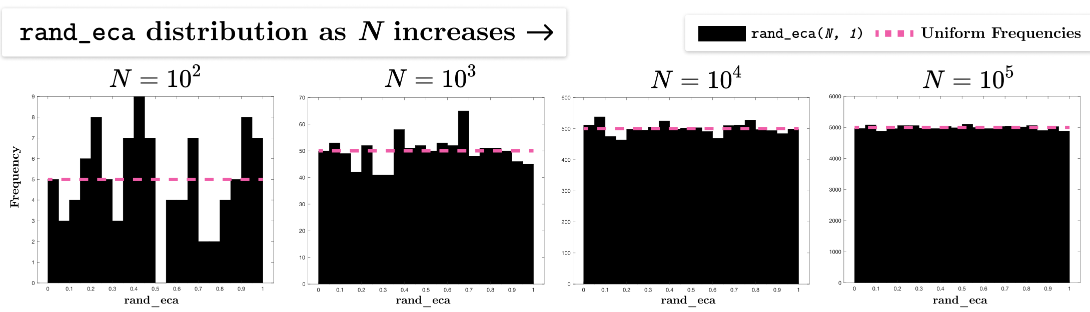

Figure 3: As I increase the numbers generated N, `rand_eca` approaches a uniform distribution. Each histogram has 20 bins/bars and `rand_eca` started from the same seed `rng_eca(0)` each time.

3 Test Compared to Random Uniform

Just to eye-ball how good rand_eca is, I can plot a histogram of generated numbers and see how well I approach a uniform distribution. In Figure 3, as I increase N generated numbers by a factor of 10 each time, I approach what appears to be a uniform distribution.

For a more rigorous test, I'll use the χ^2 test to test whether my distribution is a random uniform distribution. At large enough $N = 10^5$, given a null hypothesis that rand_eca for $n = 20$ bins has uniform counts at a $\alpha = 0.05$ significance level, I computed χ^2 with $n-1 = 20-1 = 19$ degrees of freedom.

First I computed the test using the observed bin counts O_i from rand_eca compared to the uniform expected bin counts (equal frequency) of E as

$$\sum_{i=1}^{n} \frac{(O_i - E)^2}{E} \tag{3.1}$$

where n is the total number of bins and i refers to the ith bin index (code in Appendix I and Appendix L). And since (3.1) is distributed like a χ^2 distribution with $n-1$ degrees of freedom, at a significance level of $\alpha = 0.05$, I can compare (3.1) to the theoretical critical value of 30.143 (area to the right of 30.143 is 0.05). If (3.1) is less than the critical value of 30.143, I will fail to reject the null hypothesis.

I observed that $\chi^2_{\alpha=0.05,df=19} = 15.2434$ with a p-value of 0.707 for rand_eca at $N = 10^5$ generated numbers. Since the observed test is less than the theoretical critical value, I do indeed fail to reject that rand_eca is distributed from a random uniform distribution. Just for reference, the built-in Matlab rand function (starting from rng(0) seed) for the same configuration above gets $\chi^2_{\alpha=0.05,df=19} = 23.8466$ with a p-value of 0.2021 which also passes the test, but is less evidence than the rand_eca function.

So for large enough N I've shown that the rand_eca function can be interpreted as a random uniform distribution. If you would like, run the paper_figures.mlx to reproduce these tests for yourself.

4 Sampling Other Distributions

The rand_eca function can produce a uniform distribution, but I cannot yet generate other important distributions. For example, If I wanted to sample from a standard normal distribution, I would have to symbolically invert the standard normal, then input my rand_eca values.

Since this requires symbolic derivation, I would have to solve the inverse function for all target distributions by hand, which is not feasible given how many distributions exist.

Instead I numerically approximated the inverse mapping from uniform to a target distribution. Some sampling transformations are shown in Figure 4. To do this, I started with a target

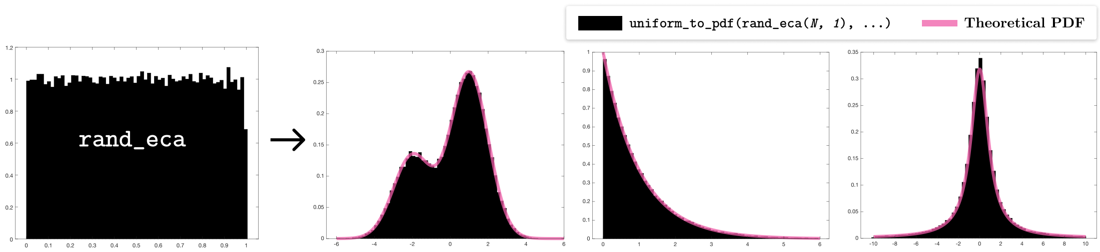

Figure 4: With $N = 10^5$ random numbers from rand_eca, I mapped to other probability density functions: mixed standard normals, exponential with $\lambda = 1$, and $1/(\pi(1 + x^2))$ for each of the plots. The black is the histogram and the pink is what the densities should ideally look like.

probability density function g, like the standard normal, and mapped the areas to a uniform distribution.

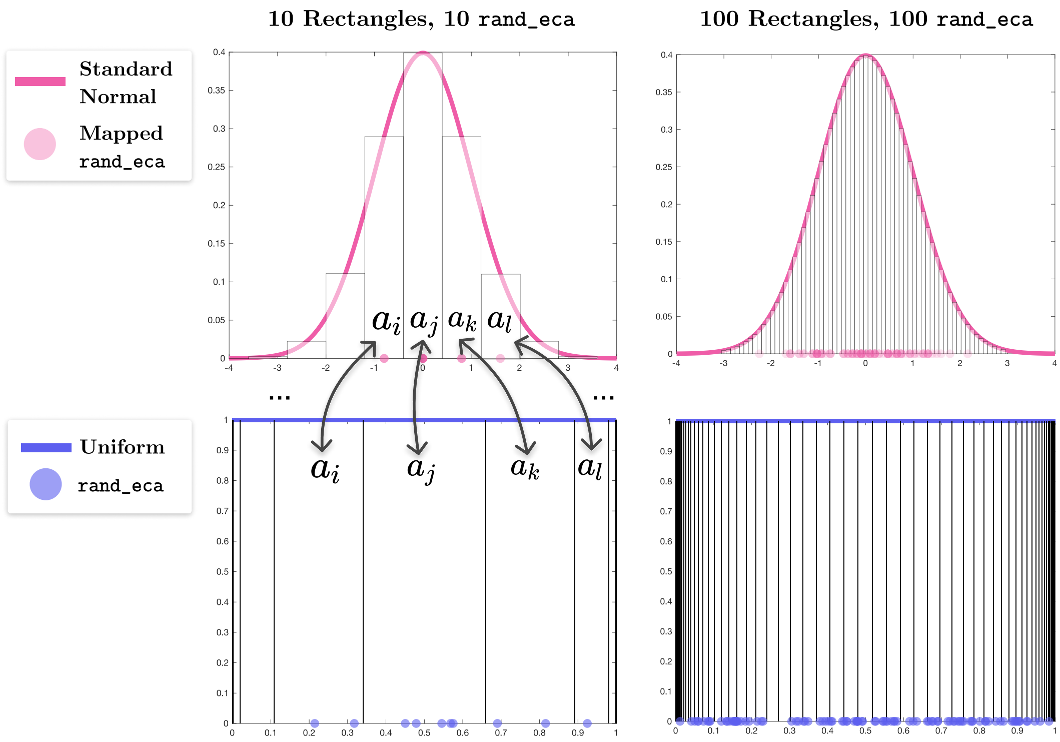

Figure 5: I can approximate the area (with rectangles) and map the areas back to uniform. On the left, I show a direct area mapping from the standard normal areas between -4 and 4 for ten rectangles directly to ten rectangles on the uniform distribution sized by the areas. Then I show for more points and more rectangles on the right.

I first divided up the g between a region $[a, b]$ into n sections I'll notate as $x_1, x_2, \ldots, x_n$. Then I created rectangles with a width of Δx (just $x_{i+1} - x_i$) and with a height of $g(x_i)$ across g. For all rectangles I then computed the areas per rectangle as $a_i = \Delta x \cdot g(x_i)$.

Next, I divided up the uniform distribution into rectangles sized with areas a_i. Since the `rand_eca` is between 0 and 1, I can simply divide up the x axis into accumulating a_i as shown in Figure 5 on the left. Now, I can take my random uniform sample from `rand_eca`, and figure out which ith rectangle each value fell into. Then map that directly onto the target g rectangle and use that x_i as the transformed number. As we increase the number of rectangle divisions, the more accurate the mapping becomes as shown in Figure 5.

I provided the `uniform_to_pdf(uniform, pdf, a, b, num_rectangles)` (Appendix M) Matlab implementation where I map the source random uniform numbers `uniform` to be distributed like the given anonymous `pdf` function. The approximation requires you to specify the x bounds $[a, b]$ of the `pdf` and the precision with `num_rectangles`.

5 Conclusion

In this paper I have implemented a random uniform function called `rand_eca` in Matlab by iterating Elementary Cellular Automata rule 30. I have also implemented a function called `uniform_to_pdf` that transform the uniform sampling to any other probability density function.

All the functions are open source at https://github.com/xnought/rand-eca. If you would like to reproduce the figures and statistical tests from this paper, see the paper_figures.mlx file.

References

[1] Erica Jen. "Aperiodicity in one-dimensional cellular automata". In: *Physica D: Nonlinear Phenomena* 45.1-3 (1990), pp. 3–18.

[2] Stephen Wolfram. "Cellular automata as models of complexity". In: *Nature* 311.5985 (1984), pp. 419–424.

[3] Stephen Wolfram. "Statistical mechanics of cellular automata". In: *Reviews of modern physics* 55.3 (1983), p. 601.

[4] Stephen Wolfram and M Gad-el-Hak. "A new kind of science". In: *Appl. Mech. Rev.* 56.2 (2003), B18–B19.

A rule30.m

```
function out_bit = rule30(vec3)
    % Applies rule 30 to a certain corresponding 3 cell states
    out_bit = xor(vec3(1), or(vec3(2), vec3(3)));
end
```

B next_state_rule30.m

```
function next_state = next_state_rule30(current_state)
    % Takes a 1D vector of size/shape (1, n)
    % Produces a 1D vector of size/shape (1, n) with rule30 applied

    n = length(current_state);
    next_state = zeros(1, n);

    % wrap around start
    next_state(1) = rule30([current_state(end) current_state(1:2)]);

    % slide window over current state
    for i=2:n-1
        window_start = i-1; window_end = i+1; % window of 3 cells to produce ith next state
        next_state(i) = rule30(current_state(window_start:window_end));
    end

    % wrap around end
    next_state(end) = rule30([current_state(end-1:end) current_state(1)]);
end
```

C iterate_rule30.m

```
function eca_generations = iterate_rule30(start_state, n_iterations)
    % from the start_state sized (1, state_width), iterate with rule30 n_iterations number of times
    % returns an matrix (n_iterations, state_width) with n_iterations past the start_state with rule30 applied to each row

    width = length(start_state);
    eca_generations = zeros(n_iterations, width);

    % generates next_state given previous_state starting with provided start_state
    current_state = start_state;
    for i=1:n_iterations
        current_state = next_state_rule30(current_state);
        eca_generations(i, :) = current_state;
    end
end
```

D visualize_rule30.m

```
function visualize_rule30(start_state, n_iterations, visualize_from_i)
    % Visualizes rule 30 in an image for n_iterations from the given start_state
    % only shows from offset to end

    eca_generations = iterate_rule30(start_state, n_iterations);
    % also include the start state in visualization
    to_visualize = [start_state;
                    eca_generations];
    % show 1s as black cells and 0s as white cells
    invert_colors = not(to_visualize(visualize_from_i:end, :));
    imshow(invert_colors, "InitialMagnification", 1000)
end
```

E bits_to_fractions.m

```
function decimal_fractions = bits_to_fractions(bits, n, bits_per_number)
    % converts a vector of bits sized into n decimal numbers chunked over with bits_per_number precision

    binary_fraction_powers = 2.^(-1:-1:-bits_per_number); % for the binary to fraction conversion 2^{-1}, 2^{-2}, ...
    decimal_fractions = zeros(1, n);
    for i=1:n
        % pick out chunk over bits by bits_per_number
        end_bit_index = i*bits_per_number;
        start_bit_index = end_bit_index - (bits_per_number - 1);
        nbits = bits(start_bit_index:end_bit_index);
```

```
        decimal_fractions(i) = sum(nbits .* binary_fraction_powers);
    end
end
```

F rand_eca.m

```
function rand_nums = rand_eca(rows, columns)
    % Computes random numbers uniformly [0, 1) using Elementary Cellular Automata Rule 30
    % you specifiy the size (rows, columns) of the matrix of random numbers you get

    % these values are set from rng_eca function
    global seed
    global bits_per_number
    global upper_memory_limit

    % initialize the seed if not found globally
    if isempty(seed)
       rng_eca(0); % initialize the seed value
    end

    % Iterate rule 30 to generate random numbers
    n = rows*columns;
    % Chunk over the timesteps instead of computing all at once to save memory
    [num_chunks, n_iterations_per_chunk, decimal_nums_per_chunk] = compute_chunks(n, bits_per_number, length(seed), upper_memory_limit);
    rand_nums = zeros(num_chunks, decimal_nums_per_chunk);
    for i=1:num_chunks
        % Generate elementary cellular automata
        eca_generations = iterate_rule30(seed, n_iterations_per_chunk);
        seed = eca_generations(end, :); % update seed with last ECA row

        % Convert the generated columns into fractions [0, 1)
        bits = reshape(eca_generations, 1, []);
        bits_to_fractions(bits, decimal_nums_per_chunk, bits_per_number);
        rand_nums(i, :) = bits_to_fractions(bits, decimal_nums_per_chunk, bits_per_number);
    end

    % Return matrix with specified shape (rows, columns)
    rand_nums = reshape(rand_nums, 1, []);
    rand_nums = reshape(rand_nums(1:n), rows, columns);
end
```

G compute_chunks.m

```matlab
function [num_chunks, n_iterations_per_chunk, decimal_nums_per_chunk] = num_iterations(total_numbers_to_generate, bits_per_number, seed_width, upper_memory_limit)
    % IMPORTANT: iterate_rule30() generates a (n_iterations, length(seed)) sized matrix
    % so if you want to iterate
    % put n_iterations into smaller chunks to limit memory use for large n_iterations

    % important high-level numbers
    total_bits_to_generate = total_numbers_to_generate*bits_per_number;
    num_eca_columns = seed_width;

    % ECA iterations and chunk size to limit memory consumptions
    % column must be atelast of length bits_per_number, but can be more
    n_iterations = max(bits_per_number, ceil(total_bits_to_generate/num_eca_columns));
    % break up the n_iterations into smaller chunks
    n_iterations_per_chunk = min(n_iterations, upper_memory_limit);
    num_chunks = ceil(n_iterations / n_iterations_per_chunk);
    % the count of decimal random numbers we get per chunk
    decimal_nums_per_chunk = ceil(total_numbers_to_generate / num_chunks);
end
```

H rng_eca.m

```matlab
function rng_eca(offset)
    % reseeds the rand_eca function for reproducability
    % time_offset allows you to change the seed time_offset iterations in the future

    % a single black square in the middle surrounded by white
    % there are seed_radius white cells on the left side then another seed_radius number of white cells on the right side
    padding = 512;
    start_state = [zeros(1, padding) 1 zeros(1, padding)];


    % when the start state goes past a certain number of iterations (padding number of times),
    % we get rid of the pyramid like pattern
    warmup = padding + offset;
    for i=1:warmup
        start_state = next_state_rule30(start_state);
    end

    % exposed globally so the rand_eca can access these things
```

```
    global seed
    global bits_per_number % numerical precision
    global upper_memory_limit

    seed = start_state;
    bits_per_number = 13;
    % we store upper_memory_limit*length(seed) numbers at any given time
    % must be multiple of bits_per_number
    upper_memory_limit = 32*bits_per_number;
end
```

I chi_squared_critical_value.m

```
function critical_value = chi_squared_critical_value(data, num_bins)
    N = length(data);

    % count bin frequncies
    bin_edges = (0:num_bins) ./ num_bins; % equally spaced num_bins from 0 to 1
    counts = count_bins(bin_edges, data);

    % compare versus true uniform counts should be
    uniform_per_bin_count = N / num_bins;
    true_uniform = zeros(1, num_bins) + uniform_per_bin_count;
    E = true_uniform;
    O = counts;
    % \sum_{i=1}^{num_bins} (O_i - E_i)^2 / E_i is ~ \Chi^2_{df=num_bins-1}
    critical_value = sum((O - E).^2 ./ O);
end
```

J count_bins.m

```
function counts = count_bins(bin_edges, data)
    counts = zeros(1, length(bin_edges) - 1);
    for i=1:length(data)
        bin_loc = find_bin(bin_edges, data(i));
        counts(bin_loc) = counts(bin_loc) + 1;
    end
end
```

K find_bin.m

```
function bin_loc = find_bin(bin_edges, number)
    for i = 1:(length(bin_edges)-1)
        % number fit within [bin_edge, bin_edge)
        if ( number >= bin_edges(i) ) && ( number < bin_edges(i+1) )
            bin_loc = i;
```

```
            break;
        end
    end
end
```

L chi_squared_test.m

```
function [observed_critical_value, theoretical_critical_value,
    p_value, passed_test] = chi_squared_test(data, bins)
    observed_critical_value = chi_squared_critical_value(data,
        bins);
    signficance_level = 0.05;
    degrees_of_freedom = bins - 1;
    theoretical_critical_value = chi2inv(1-signficance_level,
        degrees_of_freedom);
    passed_test = observed_critical_value <
        theoretical_critical_value;
    p_value = chi2cdf(observed_critical_value, degrees_of_freedom
        , "upper");
end
```

M uniform_to_pdf.m

```
function [transformed, areas, target_xs, target_ys] =
    uniform_to_pdf(uniform, pdf, a, b, num_rectangles)
    % compute target mapping areas and what x they correspond to
    dx = (b - a) / num_rectangles;
    target_xs = a:dx:(b-dx); % rectangle start coordinate x_i
    target_ys = pdf(target_xs); % pdf(x_i) or rectangle height
    areas = dx .* target_ys;  % rectangle widths times heights

    % then map the random uniform to those xs from the pdf
    % weighed by the pdf area for that rectangle
    r = reshape(uniform, [], 1); % column vector
    source_bins = cumsum(areas) ./ sum(areas); % the bins in the
        uniform distrubtion sized by the pdf areas

    % find what rectangle the uniform maps to in the pdf
    target_rectangles_indexes = zeros(1, length(r));
    for i=1:length(r)
        a_i = find(r(i) < source_bins, 1);
        if isempty(a_i)
            target_rectangles_indexes(i) = length(source_bins); %
                end bin
        else
            target_rectangles_indexes(i) = a_i;
        end
    end

    transformed = reshape(target_xs(target_rectangles_indexes),
        size(uniform)); % grab the pdf x_i coordinate from the
        rectangles
```

```
end
```