4DM4 Lab 1 Report
Linear Feedback Shift Register

Ashpan Raskar raskara 400185326
Ahnaf Bhuiyan bhuiya3 400198359

September 30, 2022

Contents

Part A **2**

A2 2

A3 2

A4 2

A5 and A6 2

Part B **3**

B2 3

B3 3

B4 4

B5 4

Extra Info **4**

Part A

A2

Yes, the Linear Feedback Shift Register (LFSR) does reach a steady state. It takes 4194303 (≈4.19 million) clock ticks for the LFSR to return back to its original state. This is also known as the period of the output stream.

A3

Included at the end of the file is the first page of the randomly generated numbers from the LFSR.

A4

The formula for the conditional probability of a 0-run and a 1-run of length k occuring is given by:

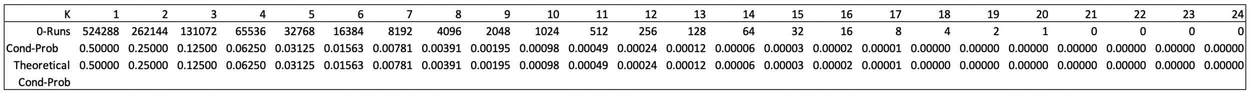
$$(\frac{1}{2})^k \tag{1}$$

Ideally the LFSR is used to create a perfectly random number using binary strings which are then converted to decimal. For a binary string to be completely random, each digit in the string must have a 50% chance of being a 1, and 50% chance for being a 0. The formula we have given follows this justification. For example, since a digit has a 50% chance to be a 1. Any consecutive digits after will also have a 50% chance of being 1, individually. Therefore for 1-run to be 3 digits long the probability can be calculated as such:

$$\frac{1}{2} * \frac{1}{2} * \frac{1}{2} \tag{2}$$

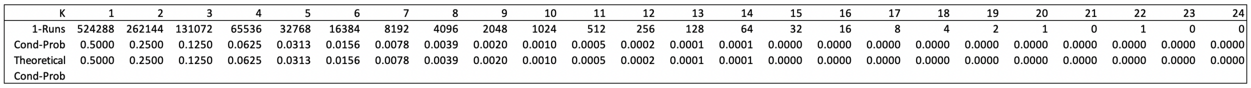
The same probablility can be applied to a 0-run. This means for any 1-run or 0-run of k-length, the theoretical probability can be calculated by the formula given in (1).

A5 and A6



K	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0-Runs	524288	262144	131072	65536	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1	0	0	0	0
Cond-Prob	0.50000	0.25000	0.12500	0.06250	0.03125	0.01563	0.00781	0.00391	0.00195	0.00098	0.00049	0.00024	0.00012	0.00006	0.00003	0.00002	0.00001	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
Theoretical Cond-Prob	0.50000	0.25000	0.12500	0.06250	0.03125	0.01563	0.00781	0.00391	0.00195	0.00098	0.00049	0.00024	0.00012	0.00006	0.00003	0.00002	0.00001	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Table 1: Table of 0-run lengths and their probabilities.



K	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1-Runs	524288	262144	131072	65536	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1	0	1	0	0
Cond-Prob	0.5000	0.2500	0.1250	0.0625	0.0313	0.0156	0.0078	0.0039	0.0020	0.0010	0.0005	0.0002	0.0001	0.0001	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Theoretical Cond-Prob	0.5000	0.2500	0.1250	0.0625	0.0313	0.0156	0.0078	0.0039	0.0020	0.0010	0.0005	0.0002	0.0001	0.0001	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 2: Table of 1-run lengths and their probabilities.

Comparing the theoretical and conditional probability shows the formula found in (1) is accurate and can be used to correctly estimate the chance of a 1 or 0-run of k length occuring. When analyzing Table 1 and Table 2 there seems to be no discrepencies between conditional and theoretical probability. Although for most cases there are no differences, as k increases, some probabilities begin to differ, and cannot be seen due to significant digits being cutoff on the tables. These discrepencies can be attributed to system limitations. The conditional probability is calculated through the division of the number of observed 0 or 1-runs of length k, with the sum of all runs which have occured. This calculation may cause floating point errors as k increases creating very small numbers, causing MATLAB to calculate a number slightly different from the theoretical probability for certain values of k.

Part B

B2

1. The space delimted file of random numbers was saved to a vector variable in matlab. This vector was of size 1×524287.

2. Then a triple nested for loop was used to iterate through the dimensions rows, columns, and depth from the image to be encrypted. In our case the dimensions were $427 \times 640 \times 8$. For the `RAND_matrix` vector of those dimensions, for every cycle in the loop, a random number was selected from the vector that has all the random numbers. In our case we did run out of random numbers, so we had to loop back from the random numbers vector. It is noted that doing so invalidates Shannon's One Time PAD principle, but this was done for ease of implementation, as increasing the size of the LFSR is very computationally heavy.

B3

The `RAND_matrix` was then used to XOR the image with the image vector `A` to encrypt the image. The encrypted image was then saved to a file.

1. First every element in the `RAND_matrix` vector was visited.

2. Then the function `bitxor` was used to XOR the `RAND_matrix` element with the corresponding element in the image vector `A`.

3. The result was then saved to the `A_encrypted` vector.

4. Finally the `A_encrypted` vector was displayed as an image using `image(uint8(A_encrypted))`.

B4

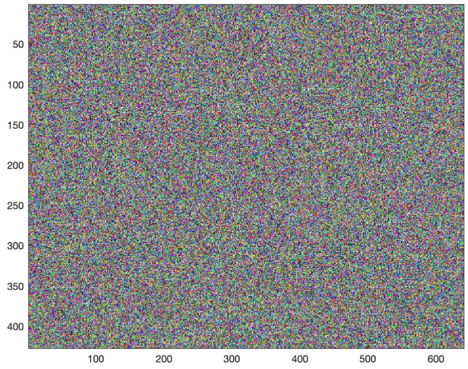


Figure 1: Encrypted image generated by XORing the `RAND_matrix` and image vectors.

B5

1. First every element in the `RAND_matrix` vector was visited.
2. Then the function `bitxor` was used to XOR the `RAND_matrix` element with the corresponding element in the encrypted image vector `A_encrypted`.
3. The result was then saved to the `A_decrypted` vector.
4. Finally the `A_decrypted` vector was displayed as an image using `image(uint8(A_decrypted))`.

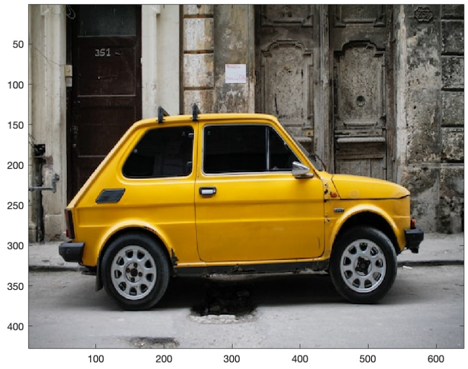


Figure 2: Decrypted image generated by XORing the `RAND_matrix` and `A_encrypted` vectors.

Extra Info

```
1 0 96 0 0 20 0 128 7 0 16 1 0 102 0 64 21 0 248 7 0 1 1 96 96 0 20 20 128 135
7 16 17 1 102 102 64 85 21 248 255 7 1 0 97 0 96 20 0 148 7 128 23 1 16 103 0
38 21 64 237 7 248 6 1 97 97 96 116 20 148 147 135 151 22 17 119 103 38 51 85
173 234 255 254 7 96 0 1 20 96 128 7 20 16 129 7 102 16 65 21 102 248 71 21 1
249 103 96 1 21 116 224 135 19 4 145 134 97 118 81 84 115 158 191 82 20 248
158 7 97 20 97 148 103 148 23 149 23 231 23 39 5 39 237 33 237 70 236 102 153
102 117 117 245 243 243 131 130 130 240 240 48 34 34 202 204 204 171 170 138
255 255 19 0 128 6 0 112 1 0 114 0 192 18 0 232 6 0 103 1 32 117 0 236 19 128
134 6 112 113 1 114 114 192 210 18 232 238 6 103 102 33 85 117 236 255 147 6
128 118 1 112 115 0 178 18 192 250 6 232 97 1 71 116 32 153 19 108 149 134 246
119 113 3 115 178 160 210 58 252 238 137 96 70 51 84 185 138 127 249 19 112
129 6 114 112 193 18 114 232 198 18 103 233 38 117 103 237 51 245 134 234 99
241 135 116 2 177 211 96 154 46 212 117 142 207 83 18 138 222 198 115 108 137
146 118 243 118 179 98 179 250 180 250 161 251 65 156 65 152 84 24 181 31 229
27 228 133 133 197 209 209 73 78 78 91 90 186 221 221 217 204 76 173 170 250
254 255 97 0 64 20 0 152 7 0 21 1 224 103 0 4 21 128 225 7 80 4 1 158 97 64 84
20 152 159 7 21 20 225 135 103 4 17 149 97 230 87 84 5 159 255 33 20 64 140 7
152 18 1 245 102 224 99 21 132 244 135 177 3 81 154 96 222 53 84 204 139 159
138 19 244 147 134 131 118 145 112 115 54 178 82 203 250 190 235 97 152 71 20
21 153 231 103 21 5 245 231 225 3 69 132 224 153 49 68 85 138 249 223 83 1 140
126 128 114 16 240 18 6 226 70 193 100 121 168 117 17 223 115 38 140 82 141
242 254 242 98 224 226 20 228 164 135 165 29 209 221 100 206 172 85 170 222
223 127 12 12 144 130 2 246 240 64 35 34 184 204 12 185 170 98 249 255 116 1
160 115 0 156 18 128 244 6 176 99 1 154 116 192 181 19 200 155 6 139 117 161
211 115 156 142 146 116 242 182 211 98 155 238 180 117 166 219 83 157 141 254
212 114 160 239 18 28 230 134 68 101 177 249 117 90 193 211 125 136 206 16 115
42 166 210 79 253 14 250 96 194 33 212 72 140 47 155 18 174 245 70 222 99 89
140 116 157 178 243 244 154 162 227 245 156 196 163 180 137 188 91 179 152 189
58 213 248 233 47 65 7 110 56 65 22 105 88 103 23 61 53 231 232 43 37 135 239
45 17 198 110 70 105 86 121 87 127 49 63 112 42 8 210 15 195 14 162 104 194 60
215 168 40 47 63 47 46 40 78 14 79 90 34 218 221 204 205 172 202 170 254 235
127 128 7 16 16 1 6 102 64 65 21 120 248 7 17 1 97 102 96 84 21 148 255 135 23
0 17 7 96 38 1 84 109 128 255 22 16 96 7 6 52 65 129 107 120 144 23 17 22 103
70 39 85 57 237 127 233 6 112 103 1 50 117 192 234 19 232 135 6 7 113 33 97
114 108 212 146 150 239 118 23 102 51 71 181 42 249 251 111 129 1 118 80 64 19
30 184 70 4 121 153 97 113 85 116 242 159 211 2 148 238 128 119 38 16 83 13
166 254 66 125 224 248 16 36 33 134 109 76 209 150 122 110 247 81 54 67 94 171
88 188 63 157 24 232 52 5 167 235 33 157 71 236 20 153 166 103 117 29 245 243
228 131 162 133 240 220 49 162 76 202 188 218 171 248 141 63 193 18 104 232 6
23 103 33 39 117 44 237 147 238 134 118 102 113 83 117 178 254 211 122 128 238
17 112 70 6 82 89 193 126 125 104 240 16 23 34 38 199 76 45 169 250 110 255 97
54 64 84 11 152 191 3 21 152 224 7 53 4 225 139 97 132 83 148 145 158 87 118
20 95 147 39 188 22 141 120 231 50 49 229 106 234 229 215 199 5 15 201 33 98
75 204 180 155 170 155 245 159 213 3 212 143 128 15 50 16 194 10 198 232 67 41
135 120 47 49 49 110 106 74 214 215 91 15 143 61 34 210 200 204 46 171 106 174
255 87 30 0 95 4 32 156 1 140 84 128 178 31 240 26 4 226 133 193 196 81 168 73
30 95 91 36 188 157 141 216 212 50 173 239 234 30 230 103 68 5 149 249 225 87
65 4 127 152 33 16 85 12 230 159 66 5 244 248 129 35 65 144 108 24 182 22 69
123 231 185 49 69 89 234 121 221 71 241 12 121 162 98 209 252 116 174 160 83
62 156 94 136 116 28 179 147 164 154 182 253 117 219 192 179 45 136 218 14 243
109 162 194 214 252 104 175 32 55 62 44 75 136 174 27 115 158 165 82 212 253
158 207 96 20 42 148 199 143 23 9 18 103 195 38 181 104 237 59 247 134 41 99
81 175 116 62 190 83 72 152 30 27 117 164 229 147 221 133 214 204 113 175 74
50 254 219 74 128 237 27 208 134 5 110 209 65 118 78 88 83 26 189 222 229 120
204 37 145 202 109 246 203 86 131 107 191 144 55 56 22 11 73 167 99 59 157 180
233 180 91 167 155 61 157 213 232 212 47 167 15 46 29 66 238 196 88 166 41 93
93 239 252 60 166 160 72 61 60 251 136 168 33 51 95 172 42 156 254 143 116 0
178 19 192 154 6 232 117 1 199 115 32 137 18 108 243 134 182 98 113 251 116
178 161 211 90 156 238 157 116 198 180 83 169 155 126 159 117 48 212 19 138
```

```
clear;

clc

S = zeros(1, 22); % Initialize the S vector

S(1,1) = 1; % Create the seed by setting the LSB to 1

DATA_OUT = zeros(1, 2^16); % Initialize a DATA_OUT vector to a large size
next_num = 1;

S_initial = S; % Create the initial S vector so we know when we have run for 1 period

found_period = 0;
period = 0;
disp(S)

zero_run_table = zeros(1,24); %Initialize vectors for counting the zeros and ones runs
ones_run_table = zeros(1,24);
zero_k_count = 0;
ones_k_count = 0;
theoretical_prob = 0.5.^(1:24);

for time=1:4.3e6
    ls_bit = S(1,1); % Store the LSB into a variable
    ms_bit = S(1, 22); % Store the MSB into a variable

    S(1, 22) = S(1, 1); % Set the next state of the MSB to the current value of the LSB
    S(1,1:20) = S(1,2:21); % Bit shift the bits from 2 to 21, to 1 to 20
    S(1, 21) = xor(ls_bit, ms_bit); % XOR the LSB and the MSB together and set that to the 21st bit

    DATA_OUT(1,next_num) = ls_bit; % Store the output into DATA_OUT
    next_num = next_num + 1;

    % If the zero k counter is between 1 and 24, and the LSB is 1,
    % increment the value on the table and reset the zero k counter
    if (zero_k_count > 0 && zero_k_count < 25 && ls_bit == 1)
        zero_run_table(zero_k_count) = zero_run_table(zero_k_count) + 1;
        zero_k_count = 0;
    end

    % If the ones k counter is between 1 and 24, and the LSB is 0,
    % increment the value on the table and reset the ones k counter

    if (ones_k_count > 0 && ones_k_count < 25 && ls_bit == 0)
        ones_run_table(ones_k_count) = ones_run_table(ones_k_count) + 1;
        ones_k_count = 0;
    end

    % If the LSB is 0, and the ones k counter is greater than 0, increment
    % the value on the table and reset the counter to start counting zeros
    if (ls_bit == 0)
        if (ones_k_count > 0)
           ones_run_table(ones_k_count) = ones_run_table(ones_k_count) + 1;
        end
        ones_k_count = 0;
        zero_k_count = zero_k_count + 1;

    % If the LSB is 1, and the zeros k counter is greater than 0, increment
    % the value on the table and reset the counter to start counting ones
    else
        if (zero_k_count > 0)
            zero_run_table(zero_k_count) = zero_run_table(zero_k_count) + 1;
        end
        zero_k_count = 0;
        ones_k_count = ones_k_count + 1;
    end

    fprintf("here is the state-vector at time %g\n", time);
    fprintf("%g, ", S);
    fprintf("\n\n");
    % Check if we have returned the S vector back to the origial state
    if (S == S_initial)
        fprintf("The state at time %g == the initial state; we are done\n", time);
        found_period = 1;
        period = time;
        break;
    end
end

if (found_period == 1)
    %Printing out final data after the period has been found
    fprintf("\nFound period = %g clock ticks, here are the random bits\n", period);
    fprintf("%g, ", DATA_OUT(1,1:period));
    fprintf("\n\n");

    fprintf("Here is a decimal representation\n");
    %Finding the number of total bytes in the period of the run
    num_bytes = floor(period/8);

    %Converting the DATA_OUT from an array of 8 bit binary numbers to its
    %decimal representation
    random_numbers = zeros(1, 2^16/8);
    for j=1:num_bytes
        start_index = (j-1)*8+1;
        end_index = start_index+8-1;

        BITS = DATA_OUT(1,start_index:end_index);

        integer = bits2num(BITS);
        random_numbers(1, j) = integer;
        fprintf("%g, ", integer);
    end
    fprintf("\n")
    fid = fopen("my_random_numbers.m", "w");
    fprintf(fid,"%3g ", random_numbers);
    fclose(fid);
else
   fprintf("DID NOT FIND PERIOD! \n");
end

%Creating the table for 0-runs and 1-runs occurences and probability
zeros_cond_prob(1:24) = zero_run_table(1:24)/sum(zero_run_table);
```

```
ones_cond_prob(1:24) = ones_run_table(1:24)/sum(ones_run_table);

zeros_stats = [4,24];
zeros_stats(1,1:24) = (1:24);
zeros_stats(2,1:24) = zero_run_table;
zeros_stats(3,1:24) = zeros_cond_prob;
zeros_stats(4,1:24) = theoretical_prob;

ones_stats = [4,24];
ones_stats(1,1:24) = (1:24);
ones_stats(2,1:24) = ones_run_table;
ones_stats(3,1:24) = ones_cond_prob;
ones_stats(4, 1:24) = theoretical_prob;

```

```
clear; clc;

%Reading DATA_OUT from the my_random_numbers.m file into rand_nums array
fileID = fopen('my_random_numbers.m','r');
formatSpec = '%f';
sizeA = [1 inf];
rand_nums = fscanf(fileID,formatSpec,sizeA);

%Opening the input image and converting it to a 3D array of pixels named A
A = imread("my_image_2.jpg");
image(uint8(A));
pause;
R_matrix = A(:,:,1); G_matrix = A(:,:,2); B_matrix = A(:,:,3);

%Initializing the RAND_matrix and A_encrypted arrays
[rows,cols,depth] = size(A);
RAND_matrix = zeros(rows,cols,depth);
A_encrypted = zeros(rows,cols,depth);

%Encrypting the image
%Iterating through the RAND_matrix and storing a value of rand_nums
%XORing the current indexed value of RAND_matrix and A, into A_encrypted
c = 1;
for i = 1:rows
    for j = 1:cols
        for k = 1:depth
            if c == (width(rand_nums))
                c = 1;
            else
                c = c + 1;
            end
            RAND_matrix(i, j, k) = rand_nums(c);
            A_encrypted(i, j, k) = uint8(bitxor(A(i,j,k), RAND_matrix(i,j,k)));
        end
    end
end

%Displaying the encrypted image
image(uint8(A_encrypted));
pause;

%Initializing A_decrypted array
A_decrypted = zeros(rows,cols,depth);

%Using the same steps as to encrypt, the image is decrypted
c = 1;
for i = 1:rows
    for j = 1:cols
        for k = 1:depth
            if c == (width(rand_nums))
                c = 1;
            else
                c = c + 1;
            end
            RAND_matrix(i, j, k) = rand_nums(c);
            A_decrypted(i, j, k) = uint8(bitxor(A_encrypted(i,j,k), RAND_matrix(i,j,k)));
        end
    end
end

%Displaying the decrypted image
image(uint8(A_decrypted))
```