COMPENG 4DM4 Lab 1 Report

Aaron Pinto
pintoa9

Raeed Hassan
hassam41

October 2, 2022

Exercise Part (A) - Generate Random Numbers

(A1) The LSFR is implemented in ‘`exerciseA.m`’, submitted in the pdf ‘`exerciseA.pdf`’. As we expect the period of the output data stream to be $2^n - 1$, we initialized an output vector `DATA_OUT` with size $2^{22} - 1$ for our 22-bit LSFR. The LSFR is stored in a vector `S`. In each clock tick we store the least significant bit or output in variable `LSB`, shift bits 2–21 into 1–20, use the MATLAB built-in `xor` function to XOR bit 22 and `LSB` and shift the result into bit 21, and shift `LSB` into bit 22. The output or `LSB` at each clock tick is stored into `DATA_OUT`.

(A2) For each clock-tick of the LSFR, we check if the current state of the LSFR is equal to the initial state. If these states are equal, then the LSFR will begin to repeat. The 22-bit LSFR reaches a steady-state after 4194303 clock-ticks, with the initial state of the LSFR appearing again after the 4194303 clock-ticks. This means the period of the LSFR is 4194303, and it can generate a pseudo-random output bit-stream that is 4194303 bits long.

(A3) The code to generate ‘`my_random_numbers.m`’ is implemented in ‘`exerciseA.m`’, submitted in the pdf ‘`exerciseA.pdf`’. The code stores the output bit-stream for one period from `DATA_OUT` and stores it into `BITS`, which we modify and then convert each byte from the bit-stream into a decimal number using MATLAB’s built-in `num2str` and `bin2dec` functions. These decimal numbers are then written to the file ‘`my_random_numbers.m`’.

(A4) For any bit 0, the conditional probability that the next bit in a perfectly random bit-stream will not be another 0 is 50%, therefore the probability of any 0-run of length 1 is 50%. Similarly, the probability of a 0-run of length 2 is 50% of the probability of a 0-run of length 1, or 25%, as there is a 50% probability of a run of two 0s being followed by a 1. We can generalize this and say that the conditional probability that a 0-run of length k occurs, given that we are only considering 0-runs, is equal to 2^{-k}.

This behaviour is the same for 1-runs, therefore the conditional probability that a 1-run of length k occurs, given that we are only considering 1-runs, is equal to 2^{-k}.

(A5) The number of conditional probability for 0-runs was determined by dividing the number of 0-runs of length k by the total number of 0-runs. The conditional probabilities are shown in Listing 1.

Listing 1: Conditional probability of 0-runs

```
>> zeroruns_table(3,:)

ans =

    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    0.0000         0         0
```

There are no discrepancies between the theoretical and experimental conditional probabilities, the conditional probabilities match what we expect from the theoretical values.

(A6) We see the same results for 1-runs as we do for 0-runs, as explained in A5. The conditional probabilities are shown in Listing 2.

Listing 2: Conditional probability of 1-runs

```
>> oneruns_table(3,:)

ans =

    0.5000    0.2500    0.1250    0.0625    0.0312    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    0.0000         0         0
```

There are no discrepancies between the theoretical and experimental conditional probabilities, the conditional probabilities match what we expect from the theoretical values.

Exercise Part (B) - A Simple Stream Cipher

(B1) For this experiment, we used an image of Dimorphos taken from the recent NASA DART mission. The image can be seen in Figure 1. This image was downloaded from the internet, resized to a size of 418x418 using the Image Resizer utility from Microsoft PowerToys. As the LSFR in Exercise A produces 4194303 random bits, or 524288 bytes when we store it in '`my_random_numbers.m`', we can only use pictures that contain 174762 (524288/3 random numbers for each R, G, B value for each pixel) or less pixels, which is approximately 418x418 if the image is a square. This image was read into MATLAB using the built-in `imread` function, storing the image and it's bit-map into the 3D matrix `A`.

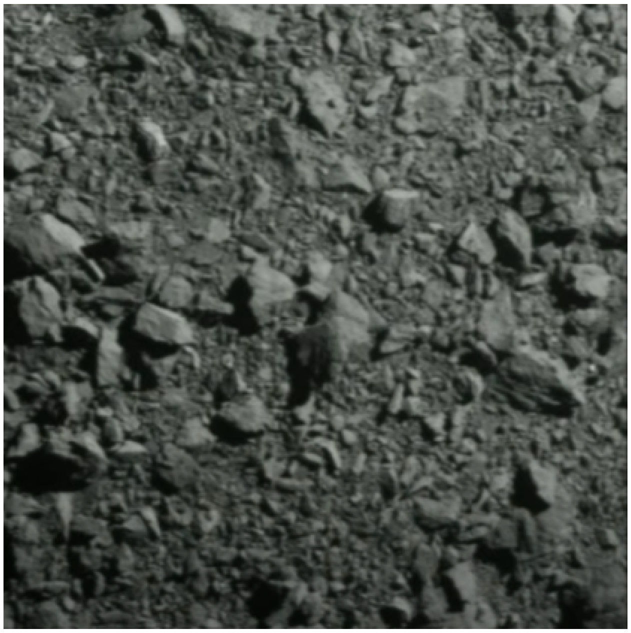

Figure 1: Image of Dimorphos from NASA DART spacecraft

(B2) The stream of pseudo-random numbers from the LFSR was read in by calling the 'my_random_numbers.m' script which stored these random decimal numbers. Once the script has loaded the random numbers, we create RAND_matrix with the following command:

```
RAND_matrix = RANDOM_DATA_OUT(1:numel(A));
```

We generate this matrix by just taking the same number of random numbers from the stored matrix as we have in the matrix bit-map of the image we read in A1. We do not create RAND_matrix as a 3D matrix as it is not necessary for our operations in the following steps.

(B3) We create the uint8 matrix A_encrypted in MATLAB with the same size and dimensions of A, which stores the image we read in. To perform the XOR operation, we iterate through all the elements of A and RAND_matrix accessing each element by their index, and performing a bitwise XOR operation using MATLAB's built-in bitxor function with the following command:

```
A_encrypted(i) = bitxor(A(i), RAND_matrix(i));
```

(B4) As the matrix A_encrypted is already of the type uint8, we can view the image without conversion by using the image function. The encrypted image can be seen in Figure 2. The image appears to be random noise as expected.

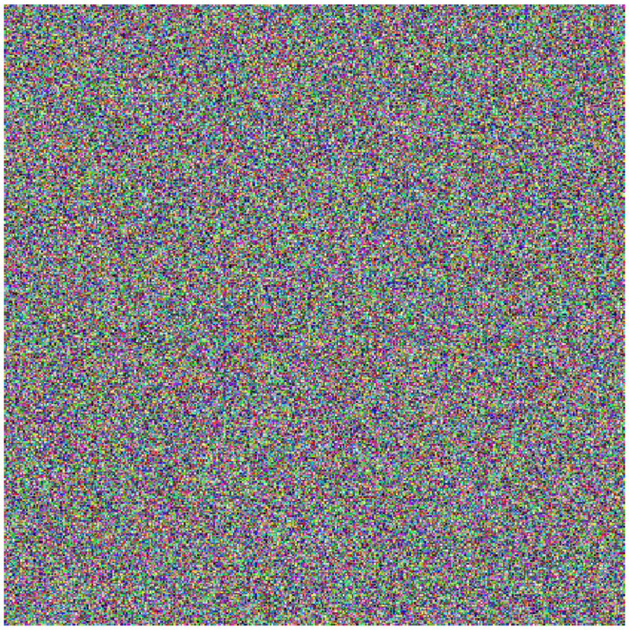

Figure 2: Encrypted Image

(B5) Similar to the process described in A3, We create the uint8 matrix A_decrypted in MATLAB with the same size and dimensions of A_encrypted. To perform the XOR operation, we iterate through all the elements of A_encrypted and RAND_matrix accessing each element by their index, and performing a bitwise XOR operation using MATLAB's built-in bitxor function with the following command:

```
A_decrypted(i) = bitxor(A_encrypted(i), RAND_matrix(i));
```

As the matrix A_decrypted is already of the type uint8, we can view the image without conversion by using the `image` function. The encrypted image can be seen in Figure 3. The image appears to match the original image in Figure 1 as expected.

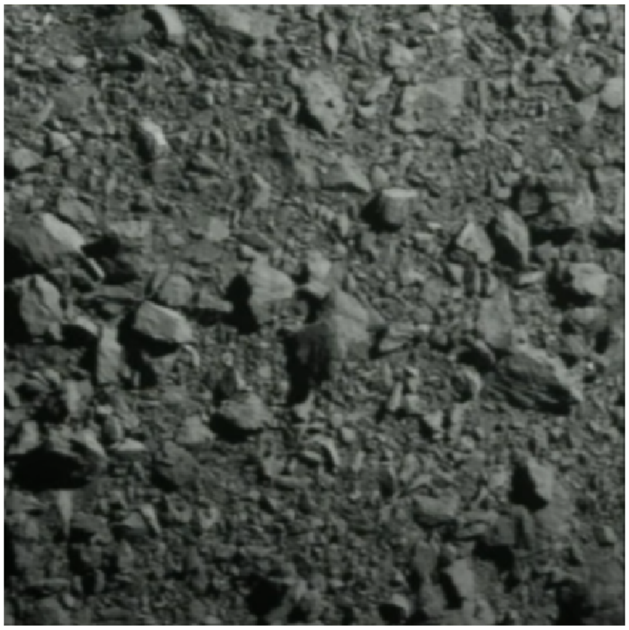

Figure 3: Decrypted Image

(B6) The script was modified to take an image of any size so that the user does not need to crop it. When inputting an image with too many pixels, the script will prompt the user asking if they wish to resize it to 418x418, resizing the image and proceeding with the processes described above if they accept and exiting the script otherwise. To select the input image, the user simply needs to change the variable `image_path` to the path of the image relative to the location of the script, and run the script.