Fast True Random Number Generation in Arduino using Atmospheric Noise

Parth Sastry (180260026), Karthik Dasigi (180260018)
December 6, 2020

1 ABSTRACT

In this project, we try to create a TRNG in Arduino, making use of atmospheric noise, that generates faster random numbers than Arduino's in-built `random()` function. We first look at a few naive methods of generating random numbers using the `analogRead()` reading from an Arduino pin. We show how they are bad sources of randomness, using ASCII Bitmaps generated using the `Serial.print()` function in Arduino. We then showcase our method, that combines these naive methods, to make the overall RNG resistant to external interference, and resistant to attacks, all the while showing reasonably high levels of entropy. Finally, we test out our method, by converting bit readings to floats, and subsequently running some well-known RNG tests, to see whether our method holds up in practice as well, along with theory.

2 PROJECT DETAILS, AND BACKGROUND

2.1 WHAT DOES A BAD RNG LOOK LIKE?

As a good theoretical exercise, let's look at what exactly a bad RNG looks like, and what the aforementioned bitmaps of such an RNG would look like.
We look at the Middle Square method proposed by John Von Neumann. He identified it as a bad PRNG, and showed how it could be very easily be susceptible to brute force attack by looking at the patterns of bits, but let's check it out.
The code fragment below includes the function that generates a sequence of `uint8_t` numbers, that can be compared to half of the `uint8_t` maximum value to give either a 0 or a 1, according

to which we shall output a specific character to display. The entire code is given below, we shall include the output in the results section.

```
int num_reads = 80;

uint16_t x = 0xb5ad; // any arbitrary seed
uint8_t compare = UINT8_MAX / 2;

inline static uint8_t ms() {
    x *= x;
    return x = (x>>8) | (x<<8);
}

void setup() {
  Serial.begin(19200);
  for (int iter1 = 0; iter1 < num_reads; iter1++) {
    for(int iter2 = 0; iter2 < num_reads; iter2++) {
      char c = (ms() > compare) ? ((char) 64) : ((char) 32);
      Serial.print(c);
    }
    Serial.print('\n');
  }
}

void loop() {}
```

See the Results where we have shown the Bitmap of this method, to see why exactly this is a horrible generator of random numbers, and what exactly our generator will have to NOT be, to pass off as a TRNG.

2.2 An ingenuous method to generate random bits

A good baseline, if somewhat naive, method to generate random bits, would be to use the `analogRead()` method on floating pins. It is well-known, however, that the level of entropy this method offers is very low. This is due to the fact that there are only 2^{10} possible readings, and the overall amplitude changes very slowly. It is also clear that this naive method is subject to external interference, and the periodic nature of surrounding voltages, will cause discernible patterns in the bits taken from this method.
The naive method here, is just to take the least significant bit of `analogRead()`, but this is flawed. We show here, using Bitmaps (which give one ASCII symbol for a 1, and another for a 0), how this output is periodic, and extremely susceptible to external interference.
The code fragment given below incorporates this method. We had to use some fancy data types so as to not exceed the internal memory of the Arduino. We shall be generating bitmaps for a few delay values (in the example code here, taken to be 64 microseconds), to show the periodic nature of this method.

```
uint8_t val_read;
const int num_reads = 80;
byte analogPin = 3;
int counter = 0;
float delay_val = 0.064;

uint8_t bit_array[10*80] = {0};

inline uint8_t get_bits(uint8_t data, int y){return ((data>>y) & 1);}

void setup() {
  Serial.begin(19200);
  for(int iter1=0; iter1<num_reads; iter1++)
  {
    for(int iter2=0; iter2<num_reads; iter2++)
    {
      val_read = analogRead(analogPin) & 1;
      bit_array[counter/8] += (val_read<<(counter%8));
      counter++;
      delay(delay_val);
    }
  }
}

void loop() {}
```

The bitmaps are generated via the code fragment given below. We print a certain character to the serial monitor if the bit is a 1, and a different character if the bit is a 0. Looking at the bitmap, thus, shows us if our RNG shows visually discernible patterns or not.

```
for (int iter1 = 0; iter1<num_reads;iter1++) {
    for(int iter2 = 0; iter2<num_reads/8; iter2++){
        for (int bits = 0; bits<8; bits++) {
          char c = get_bits(bit_array[iter2*num_reads + iter1], bits)
                ? ((char) 64) : ((char) 96);
          Serial.print(c);
        }
    }
    Serial.print('\n');
  }
}
```

The bitmaps, as before, are shown in the results section.

2.3 An improvement to the previous method : Von Neumann extractor

We can improve upon this method.
One of the earliest examples of a randomness extractor was given by John Von Neumann. This is the Von Neumann extractor, which takes 2 bits from the input stream (here the LSB of the `analogRead()` function). If the two bits match, then no output is generated. If they don't match, the first bit is stored in our `bit_array`. The Von Neumann extractor can be shown to produce a uniform output even if the distribution of input bits is not uniform so long as each bit has the same probability of being one and there is no correlation between successive bits. Such a transformation can help us get rid of the low frequency periodic components of the previous method. We sample two bits at a time, and store the first bit if and only if the two bits are not equal.
The code fragment to incorporate this is given below. We generate the bitmaps via the same code shown in the previous subsection.

```
uint8_t val_read;
uint8_t val_read1;
uint8_t val_read2;
const int num_reads = 80;
byte analogPin = 3;
int counter = 0;
float delay_val = 0.512;

uint8_t bit_array[10*80] = {0};

inline uint8_t get_bits(uint8_t data, int y){return ((data>>y) & 1);}

void setup() {
  Serial.begin(19200);
  int k = 0;
  for(int iter1=0; iter1<num_reads; iter1++)
  {
    for(int iter2=0; iter2<num_reads; iter2+=0)
    {
      val_read1=analogRead(analogPin) & 1;
      delay(delay_val);
      val_read2=analogRead(analogPin) & 1;
      if(val_read1!=val_read2)
      {
        val_read = val_read1;
        iter2++;
        bit_array[counter/8] += (val_read<<(counter%8));
        counter++;
      }
      delay(delay_val);
```

```
31        }
32      }
33  }
34
35  void loop() {}
```

The bitmaps generated are shown in the Results section. We can see how the Von Neumann extractor offers a good way to generate global randomness. It is, however, excruciatingly slow, as we sample the ADC twice for a single bit and we might ignore a large number of bits. Even so, this might be appropriate as a straightfoward way of generating true random numbers on the Arduino platform. Our task now becomes to improve upon these two previous methods, to incorporate the global randomness of the Von Neumann extractor with the local randomness of sampling `analogRead()` at a high speed.

2.4 The comprehensive method

Direct sampling is a good source of local randomness, and the Von Neumann extractor is a good source of global randomness. We will employ a combination of the two methods mentioned previously.
However, we will first address the two major drawbacks of the previously mentioned methods. These are as follows:

1. the entropy of bits other than the LSB is not used.
2. the slow speed of the Von-Neumann generator and the fact that we ignore so many input bits

We get around these drawbacks by employing the following methods.

2.4.1 Rotation of bits

We sample the analog pin 8 times, and rotate each sample's 8 least significant bits by an incrementing counter and XORing them together at the end.
In essence, we are shuffling the bits around. The rotation is done by the below code snippet:

```
1  byte rotate(byte b, int r) {
2      return (b << r) | (b >> (8-r));
3  }
```

And the code for XORing the 8 inputs is shown below:

```
1  for (int i=0; i<4; i++) {
2      delayMicroseconds(waitTime);
3      int leftBits = analogRead(analogPin);
4
5      delayMicroseconds(waitTime);
6      int rightBits = analogRead(analogPin);
```

```
7
8       finalByte ^= rotate(leftBits, i);
9       finalByte ^= rotate(rightBits, 7-i);
10  }
```

Our final byte is locally random. To incorporate global randomness from the Von Neumann extractor in our byte, we follow the method given below.

2.4.2 Seeding XOR-rotate using Von Neumann extractor

We define two functions pushLeftStack and pushRightStackRight that pushes bytes onto the leftStack and rightStack.

```
1  void pushLeftStack(byte bitToPush) {
2      leftStack = (leftStack << 1) ^ bitToPush ^ leftStack;
3  }
4  void pushRightStackRight(byte bitToPush) {
5      rightStack = (rightStack >> 1) ^ (bitToPush << 7) ^ rightStack;
6  }
```

Then we add the following code:

```
1  for (int j=0; j<8; j++) {
2      byte leftBit = (leftBits >> j) & 1;
3      byte rightBit = (rightBits >> j) & 1;
4
5      if (leftBit != rightBit) {
6          if (lastStack % 2 == 0)
7              pushLeftStack(leftBit);
8          else
9              pushRightStackRight(leftBit);
10     }
11 }
```

In essence, we use the LSB of leftBits and rightBits for our Von Neumann extractor.
The variables leftStack and the variable rightStack are now changed depending on the value of leftStack. These two variables give us our global randomness.

2.4.3 Tying it all together

We combine the local randomness with the global randomness along with XORShift-PRNG. We employ the PRNG to increase the overall entropy of our random number generator, thus improving our statistical properties.

```
1  lastByte ^= (lastByte >> 3) ^ (lastByte << 5) ^ (lastByte >> 4);
2  \\ XORShift PRNG
3  lastByte ^= finalByte;
```

```
\\ seeding PRNG with (local) TRNG
return lastByte ^ leftStack ^ rightStack;
\\ seeding again with (global) TRNG
```

In the above code, we first apply 8-bit XORShift PRNG on our last byte. Then we seed our generated PRNG with the TRNG generated from the XOR-rotate. This gives our byte local randomness.
Finally, we further seed our random byte with the TRNG generated using the Von Neumann extractor. This will give our byte global randomness.
The full code is shown below:

```
byte lastByte = 0;
byte leftStack = 0;
byte rightStack = 0;

byte rotate(byte b, int r) {
    return (b << r) | (b >> (8-r));
}

void pushLeftStack(byte bitToPush) {
    leftStack = (leftStack << 1) ^ bitToPush ^ leftStack;
}

void pushRightStackRight(byte bitToPush) {
    rightStack = (rightStack >> 1) ^ (bitToPush << 7) ^ rightStack;
}

// contrary to previous code, the random byte contains oldest value at
// MSB and newest value at LSB
byte getTrueRotateRandomByte() {
    byte finalByte = 0;
    byte lastStack = leftStack ^ rightStack;

    for (int i=0; i<4; i++) {
        delayMicroseconds(waitTime);
        int leftBits = analogRead(analogPin);

        delayMicroseconds(waitTime);
        int rightBits = analogRead(analogPin);

        finalByte ^= rotate(leftBits, i);
        finalByte ^= rotate(rightBits, 7-i);

            for (int j=0; j<8; j++) {
                byte leftBit = (leftBits >> j) & 1;
```

```
35                  byte rightBit = (rightBits >> j) & 1;
36
37                  if (leftBit != rightBit) {
38                      if (lastStack % 2 == 0)
39                              pushLeftStack(leftBit);
40                      else
41                              pushRightStackRight(leftBit);
42                  }
43              }
44      }
45
46      lastByte ^= (lastByte >> 3) ^ (lastByte << 5) ^ (lastByte >> 4);
47      lastByte ^= finalByte;
48
49      return lastByte ^ leftStack ^ rightStack;
50  }
```

2.5 Validating our RNG

How to quantify the performance of our RNG was a problem that we asked the solution to, on the forums, and Sahas posted a reply giving the example of the rank of binary matrices test. This led us to discovering the battery of Diehard tests developed by George Marsaglia, along with a few other tests to gauge the 'randomness' of a RNG. One of the Diehard tests, is the Runs test of randomness, which we have implemented here. The other test we are using is to calculate the Serial Correlation Coefficient of a sequence of random numbers, and seeing where it lies.
The concept and implementation behind both these tests is given below.

2.5.1 The Runs Test of Randomness

The first step in the runs test is to count the number of runs in the data sequence. There are several ways to define runs, however, in all cases the formulation must produce a dichotomous sequence of values. In our case, the values above the median are treated as positive and values below the median as negative. A run is defined as a series of consecutive positive or negative values.
The first step in applying this test is to formulate the null and alternate hypothesis.

- H_{null}: The sequence was produced in a random manner
- H_{alt}: The sequence was not produced in a random manner

We then calculate our test statistic, Z as follows.

$$Z = \frac{R - \bar{R}}{s_R}$$

where, R = number of observed runs, $\bar{R}$ = number of expected runs.

$$\bar{R} = \frac{2n_1 n_2}{n_1 + n_2} + 1$$

s_R = Standard deviation of the number of runs

$$s_R^2 = \frac{2n_1 n_2(2n_1 n_2 - n_1 - n_2)}{(n_1 + n_2)^2(n_1 + n_2 - 1)}$$

With n_1 and n_2 = the number of positive and negative values in the series.
Compare the value of the calculated Z-statistic with $Z_{critical}$ for a given level of confidence ($Z_{critical}$ = 1.96 for confidence level of 95%) . The null hypothesis is rejected i.e. the numbers are declared not to be random, if $|Z| > Z_{critical}$.
We implement the runs test on our generated byte array. We implement `quicksort` to find the median value, and perform the calculations as given above. The code for the Runs test, and the algorithm used to compute the Z-statistic is given below.

```
1  void swap(uint8_t* a, uint8_t* b)  {
2      uint8_t t = *a;
3      *a = *b;
4      *b = t;
5  }
6  int partition (uint8_t arr[], int low, int high)  {
7      uint8_t pivot = arr[high]; // pivot
8      int i = (low - 1); // Index of smaller element
9
10     for (int j = low; j <= high - 1; j++)
11     {
12         if (arr[j] < pivot)
13         {
14             i++;
15             swap(&arr[i], &arr[j]);
16         }
17     }
18     swap(&arr[i + 1], &arr[high]);
19     return (i + 1);
20 }
21
22 void quickSort(uint8_t arr[], int low, int high)  {
23     if (low < high)
24     {
25         int pi = partition(arr, low, high);
26         quickSort(arr, low, pi - 1);
27         quickSort(arr, pi + 1, high);
28     }
29 }
```

```
30
31 float getZstatistic(byte arr[]) {
32   byte temp_arr[num_reads*num_reads/8] = {0};
33   for (int k = 0; k < num_reads*num_reads/8; k++) {temp_arr[k] = arr[k];}
34
35   quickSort(arr, 0, num_reads*num_reads/8 - 1);
36   uint8_t median = (arr[num_reads*num_reads/16] + arr[num_reads*num_reads/16 - 1])/2;
37
38   int runs = 0;
39   double n1 = 0, n2 = 0;
40
41   for (int i = 1; i<num_reads*num_reads/8; i++) {
42       if ((temp_arr[i] >= median && temp_arr[i-1] < median) ||
43         (temp_arr[i] < median && temp_arr[i-1] >= median)) {runs++;}
44       if (temp_arr[i] >= median) {n1++;}
45       else {n2++;}
46   }
47
48   double runs_exp = (2*n1*n2/(n1 + n2))+1;
49
50   double stan_dev = sqrt((2*n1*n2*(2*n1*n2-n1-n2))/((n1+n2)*(n1+n2)*(n1+n2-1)));
51
52   double z = (runs-runs_exp)/stan_dev;
53
54   return z;
55 }
```

The results, and a screen recording of the code running on our PC is given in the results section. We see that $|Z| < Z_{critical} = 1.96$ every time we did the test, which means that the results of this test imply that we can say with 95% confidence that our RNG outputs random numbers.

2.5.2 Serial Correlation Coefficient

Most random numbers are generated by algorithms and not produced by physical processes. Because of this, we assume that there are dependencies between two successive numbers. A way to represent this fact is the “serial correlation coefficient". which will be high for severely correlated sequences, and low for sequences that don’t have dependencies in consecutive numbers.
The serial correlation coefficient C from a sequence $X_0, X_1, X_2, \ldots X_{N-1}$ of N random numbers is calculated as:

$$C = \frac{N(X_0X_1 + X_1X_2 + \cdots + X_{N-2}X_{N-1} + X_{N-1}X_0) - (X_0 + X_1 + \cdots + X_{N-1})^2}{N(X_0^2 + X_1^2 + \cdots + X_{N-1}^2) - (X_0 + X_1 + \cdots + X_{N-1})^2}$$

A correlation coefficient always lies between 1 and -1. When it is zero or very small, it indicates

that X_i and X_j are independent of each other. A "good" value of C will be between $\mu_N - 2\sigma_N$ and $\mu_N + 2\sigma_N$ which implies 95% confidence in our RNG, where

$$\mu_N = \frac{-1}{N-1}, \qquad \sigma_N = \frac{1}{N-1}\sqrt{\frac{N(N-3)}{N+1}}, \qquad N > 2$$

We calculate the Serial Correlation Coefficient for 400 floats between 0 and 1, where each float has been created from one byte generated from our RNG.
The full code block testing out our RNG with this is shown below.

```
1  const float waitTime = 0.512;
2  const int num_reads = 400;
3  const double num_reads_d = 400;
4  const byte analogPin = 3;
5  double double_array[num_reads] = {0};
6
7  byte lastByte = 0;
8  byte leftStack = 0;
9  byte rightStack = 0;
10
11 double byte_to_double(byte byte_convert)
12 {
13   return byte_convert / 256.0;
14 }
15
16 byte rotate(byte b, int r) {
17   return (b << r) | (b >> (8 - r));
18 }
19
20 void pushLeftStack(byte bitToPush) {
21   leftStack = (leftStack << 1) ^ bitToPush ^ leftStack;
22 }
23 void pushRightStackRight(byte bitToPush) {
24   rightStack = (rightStack >> 1) ^ (bitToPush << 7) ^ rightStack;
25 }
26
27 // contrary to previous code, the random byte contains oldest value
28 // at MSB and newest value at LSB
29 byte getTrueRotateRandomByte() {
30 byte finalByte = 0;
31
32 byte lastStack = leftStack ^ rightStack;
33
34 for (int i = 0; i < 4; i++) {
35     delayMicroseconds(waitTime);
```

```
36      int leftBits = analogRead(analogPin);
37
38      delayMicroseconds(waitTime);
39      int rightBits = analogRead(analogPin);
40
41      finalByte ^= rotate(leftBits, i);
42      finalByte ^= rotate(rightBits, 7 - i);
43
44      for (int j = 0; j < 8; j++) {
45        byte leftBit = (leftBits >> j) & 1;
46        byte rightBit = (rightBits >> j) & 1;
47
48        if (leftBit != rightBit) {
49          if (lastStack % 2 == 0) {
50            pushLeftStack(leftBit);
51          } else {
52            pushRightStackRight(leftBit);
53          }
54        }
55      }
56
57    }
58    lastByte ^= (lastByte >> 3) ^ (lastByte << 5) ^ (lastByte >> 4);
59    lastByte ^= finalByte;
60
61    return lastByte ^ leftStack ^ rightStack;
62  }
63
64  double get_c(double arr[])
65  {
66    double sum_duo = arr[0] * arr[num_reads - 1];
67    double sum_squares = arr[num_reads - 1] * arr[num_reads - 1];
68    double sum = arr[num_reads - 1];
69    double c;
70    for (int i = 0; i < num_reads - 1; i++)   {
71      sum_duo += arr[i] * arr[i + 1];
72      sum_squares += arr[i] * arr[i];
73      sum += arr[i];
74    }
75    c = ((num_reads * sum_duo)-(sum*sum))/((num_reads*sum_squares)-(sum * sum));
76    return c;
77  }
78
79  void setup() {
```

```
80    Serial.begin(19200);
81    for (int iter = 0; iter < num_reads; iter++)
82    {
83      double_array[iter] = byte_to_double(getTrueRotateRandomByte());
84    }
85    double mu = -1/(num_reads_d-1);
86    double sigma = (-1*mu)*sqrt((num_reads_d)*(num_reads_d-3)/(num_reads_d+1));
87    Serial.print(mu-2*sigma);
88    Serial.print(' ');
89    Serial.print(mu+2*sigma);
90    Serial.print('\n');
91    Serial.print(get_c(double_array));
92  }
93
94  void loop() {}
```

The results for calculating the serial correlation coefficient are given in the Results section, we print out the 95% confidence interval, and then the calculated correlation coefficient. We see that we consistently get the calculated value within or very close to the 95% confidence interval, showing that this test implies that we can say with 95% confidence that our RNG outputs random numbers.

3 Block Diagram of Components involved, and Contribution

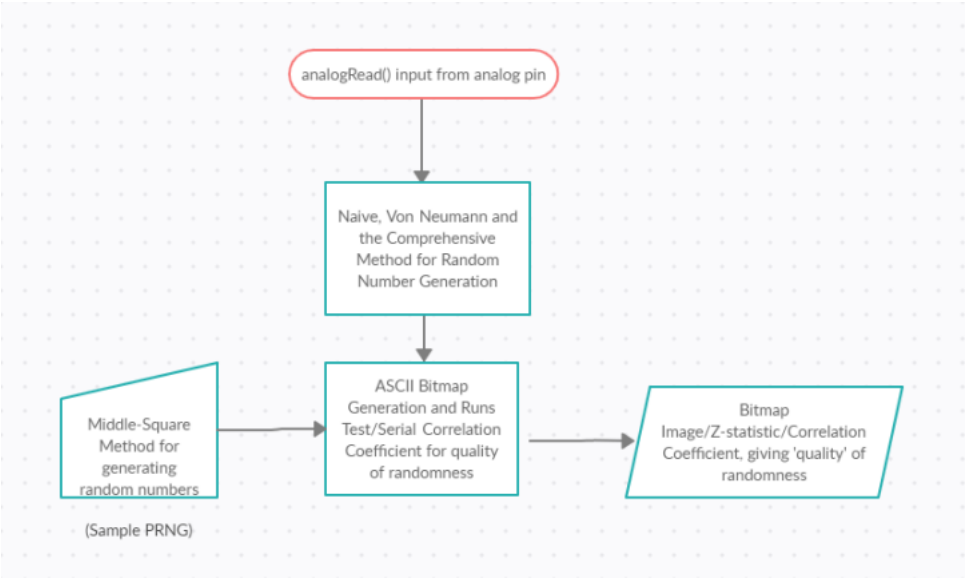

Figure 3.1: Components of the Project

The contribution of each of the team members is listed here.

- Karthik: was responsible for coding the Naive, Von-Neumann and the Final Method for generating random numbers, was responsible for writing code for the Serial Correlation Coefficient calculation.
- Parth: was responsible for coding the baseline Middle-Square PRNG, was responsible for coding the method for ASCII Bitmap generation, along with writing code for computing the Z-statistic for the Runs test for randomness.

4 Components Needed For Project

Our project is largely theoretical, requiring byte manipulation to generate random numbers with high entropy and high speed. As such, we only need an Arduino UNO board, and an interface, which is a laptop or a desktop.

5 Results and Validation

This section is comprised of the bitmaps generated for the different generator methods. The screen recordings showing the test results for the Runs test and the Serial Correlation Coefficient have been uploaded to the Google drive mentioned in the submission template.

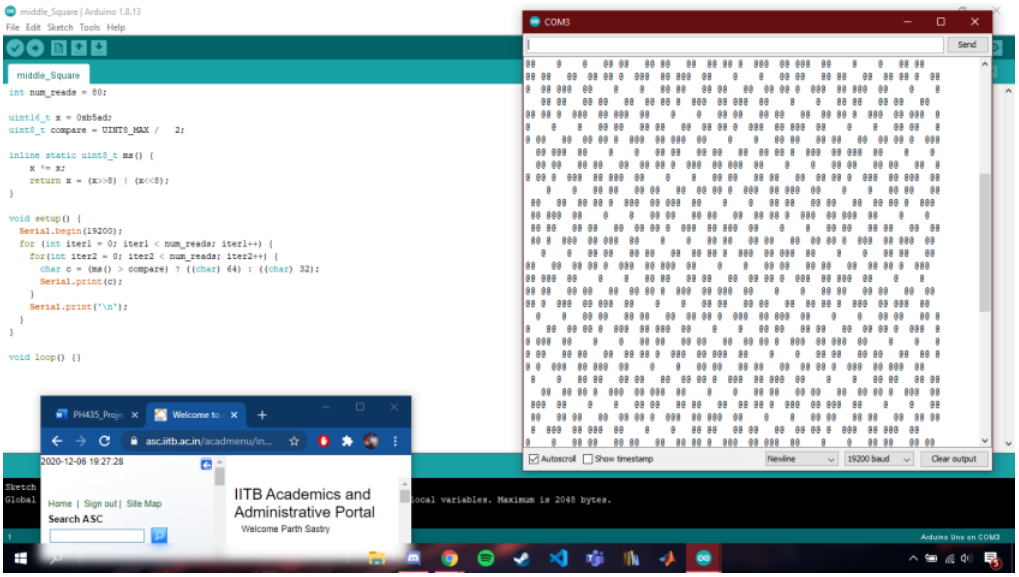

Figure 5.1: Bitmap for the Middle Square PRNG. Some diagonal patterns are clearly discernible, showing the bad nature of this RNG

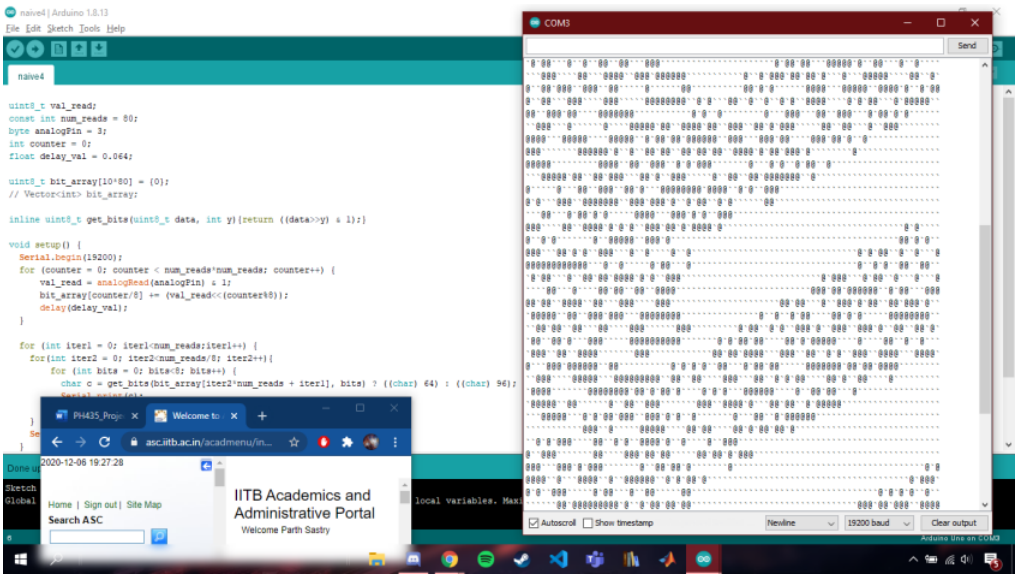

Figure 5.2: Bitmap for the Naive method (section 2.2), with a delay of 64 μs (sampling the LSB of analogRead()). Patterns can be seen, revealing this to be a bad method of generating Random numbers

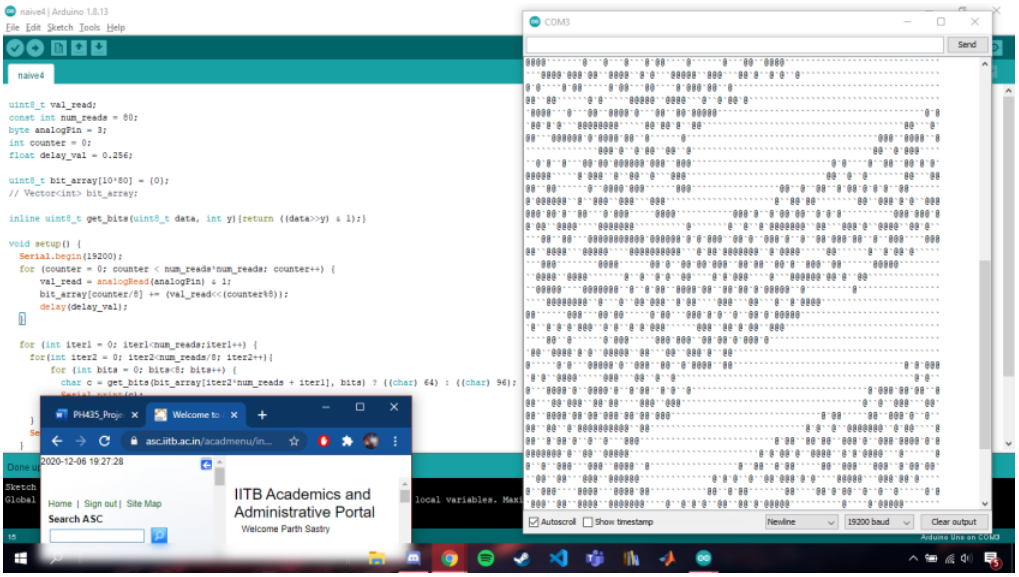

Figure 5.3: Bitmap for the Naive method, with a delay of 256 μs. Similar patterns to the previous delay can be seen, implying that our sampling is limited by some condition that isn't the delay. In any case, this is either due to external noise or some internal sampling of the Arduino UNO, and is a terrible method for Random number generation

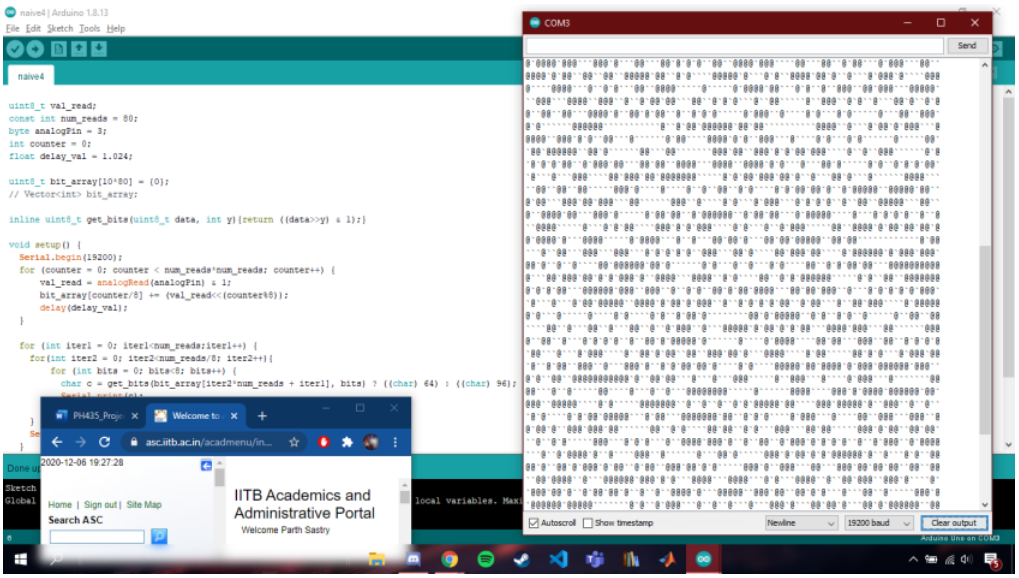

Figure 5.4: Bitmap for the Naive method, with a delay of 1024 μs. The previous patterns are gone, but we've already established that this is a bad method for random number generation

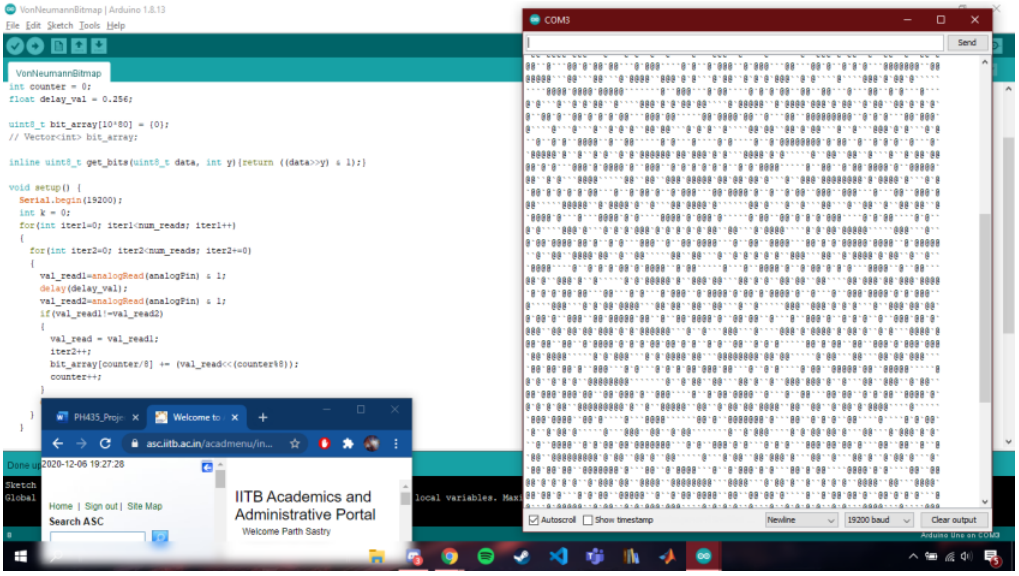

Figure 5.5: Bitmap for the Von Neumann extractor (section 2.3), with a delay of 256 μs. This is a good source of Global Randomness, but is very slow, requiring multiple samplings and delays for generating a single bit

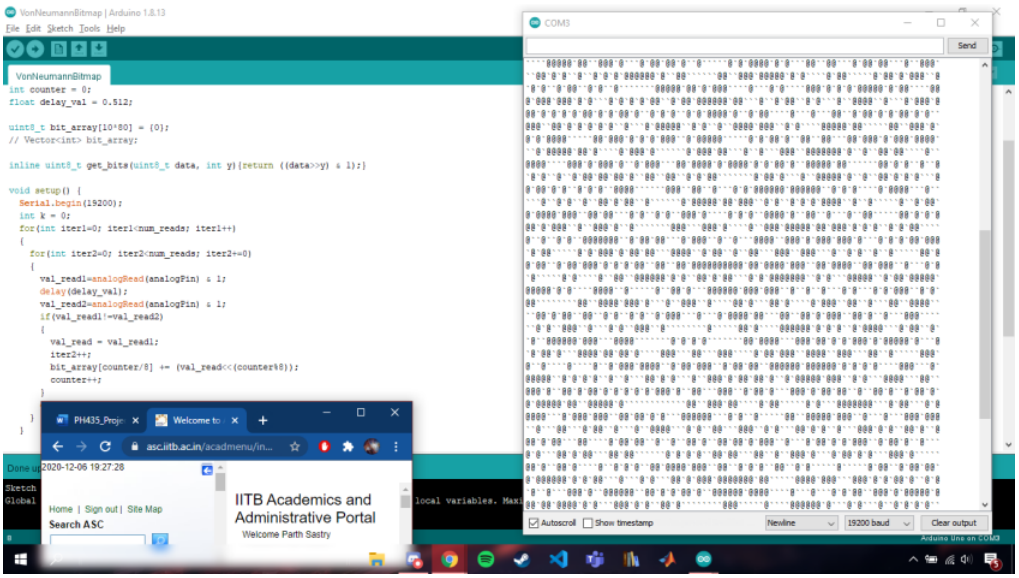

Figure 5.6: Bitmap for the Von Neumann extractor (section 2.3), with a delay of 512 μs. This takes even more time than the previous delay value

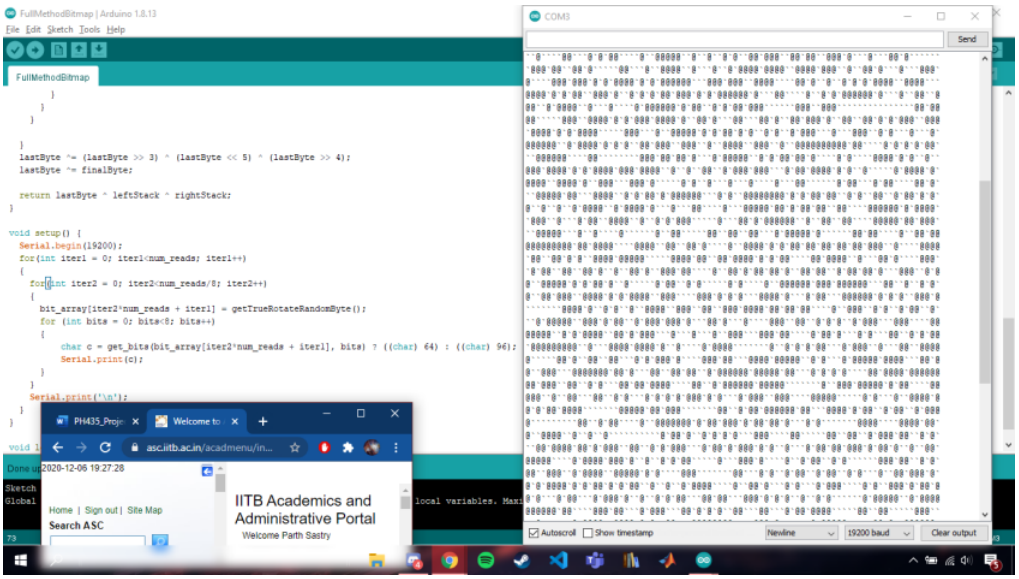

Figure 5.7: Bitmap for our comprehensive method for generating True Random Numbers. No discernible patterns are visible, and it runs way faster than the Von Neumann extractor, generating 8 bits at a time, with lesser sampling delays

The results of the quantitative validation tests are shown in the form of video recordings, that both Karthik (my teammate) and I have uploaded to the Google Drive link mentioned in the template.

As a failsafe, we have also uploaded the zipped folder to a private drive link, given below -
Link to Project Screen Recordings

6 Acknowledgements

The idea for the XORshift method that's at the heart of our Random Number Generator was first conceived of, by George Marsaglia in a paper titled 'XORshift RNGs'. The link to the paper is given below -
XORshift RNGs

The idea of Bitmap generation for gauging randomness, we got from Bo Allen's blog page -
Blog Page

The idea for the project came from a GitHub repository that referred to the earlier George Marsaglia paper and mentioned some pseudo-code for the method.
GitHub repository