# GROUP NUMBER: 30

ROLL NUMBER 1: 200260039 NAME 1: Rehmat Singh Chawla
ROLL NUMBER 2: 200260006 NAME 2: Aneesh Anand Kamat

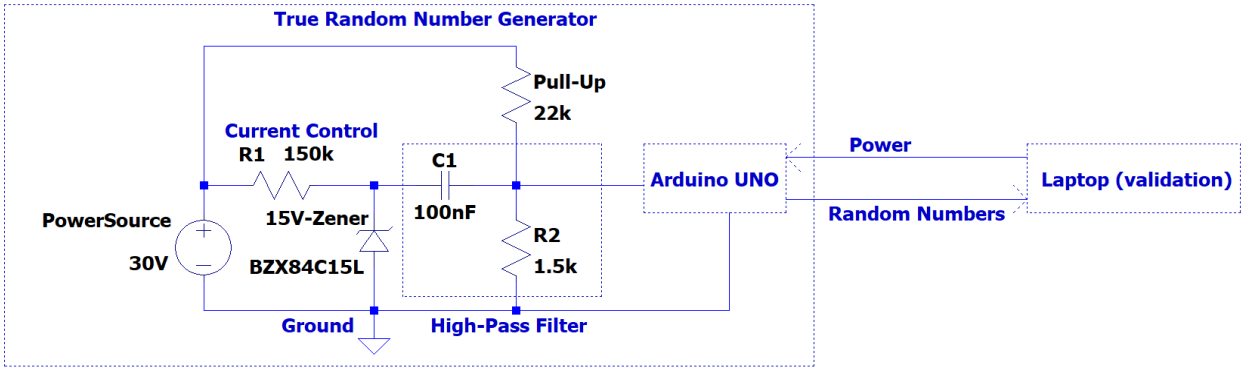
**TITLE:** A True Random Number Generator using the Zener Avalanche effect and the Watchdog Timer

## ABSTRACT:

We attempt to create a true random number generator on the Arduino UNO with two different approaches - reading the analog voltage fluctuations in the Zener Avalanche effect and sampling the system clock using the clock drift in the Watchdog timer[1]. The raw data generated is then processed to remove biases and correlations between the bits before being concatenated into random bitstrings. The generated bitstrings are validated through a combination of visual and statistical tests we've written and also pass the standardized NIST statistical test suite.

## PROJECT DETAILS:

We use the avalanche effect of a 15V Zener diode in 30V reverse bias as a good source of entropy[2] and measure the fluctuations in the potential using the ADC on the Arduino. The current control resistor ensures that there is an adequate supply of charge carriers to observe the avalanche effect. We use a high-pass filter with a very low cut-off frequency to remove the DC 15V and then pull up the potential slightly (2V) to allow the ADC to measure the negative fluctuations as well. We then sample the last 8 bits of the measurement and implement an iterative ROT-XOR algorithm (described below) and concatenate the output to generate a truly random bitstring.



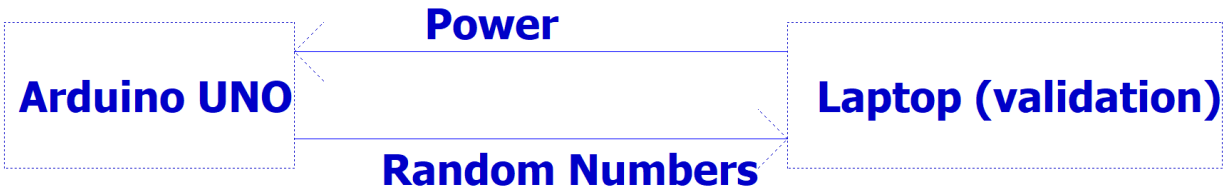

The iterative ROT-XOR algorithm (rotation followed by XOR-ing) rotates the first byte by 1 bit (LSB -> MSB) and then XORs the result with the second byte. The resultant byte is again rotated by 1 bit and XOR-ed with the third byte. After 8 iterations we declare the final byte as random and concatenate it to our bitstring. The ROT-XOR algorithm concentrates the entropy in each byte and eliminates any correlation between bits in the same position in successive bytes. This approach decreases the bitrate, but it produces truly random bitstrings that pass a variety of randomness tests.

---

[1] J. Hlaváč, R. Lórencz and M. Hadáček, "True random number generation on an Atmel AVR microcontroller," 2010 2nd International Conference on Computer Engineering and Technology, 2010, pp. V2-493-V2-495, doi: 10.1109/ICCET.2010.5485568.

[2] http://www.reallyreallyrandom.com/zener/why-its-random/

Due to the high-frequency fluctuations observed in the avalanche effect, this method still achieves a bitrate of the order of 1000 bits per second. The next approach details a method to create random numbers by extracting randomness from a weak entropy source - the clock drift between the RC oscillator-based Watchdog Timer and the system clock. The resultant bitrate is much lower than the avalanche effect approach however it is entirely built on-board the Arduino and requires no external components save for a power supply (currently provided by the laptop).




The Watchdog Timer (WDT) is used to reset the system should it “hang” or fall into a recursive loop. The software is programmed to periodically reset the WDT and should it fail to, it is assumed that the system has hung and the WDT resets the system when it is high. We change the settings of the WDT such that it triggers an interrupt when high, instead of resetting the system. This interrupt service routine (ISR) samples the count of the system clock. The count is stored in two 8-bit registers, we sample the lower register (as it varies much faster and will be more affected by jitters in the time period of the WDT) which gives us one byte (8 bits) to work with after every interrupt.

We observe that the higher significant bits (bits 7 and 8 in particular) of the raw bytes we obtain are not flipping in consecutive interrupts i.e., the rate at which we generate the bytes is of an order with the rate at which they change. Another point to note is that the LSB is likely to be non-random due to its dependence on the code being executed. This is because when an interrupt is triggered, the current instruction must be completed before the interrupt is serviced, and since the code is fairly simple, it may have deterministic effects on the LSB.

We again implement the iterative ROT-XOR algorithm and concatenate the final bytes to create a random bitstring. The bitrate here is significantly limited by a limitation of the WDT - the time period of the WDT can be a minimum of 16 ms, the operating condition we are using. Thus, we are limited to one sampled byte every 16 ms and no faster - it is this bitrate which is then decreased by a factor of 8 while processing.

The proposed generator runs independently of external components and only requires a power supply once the required code has been uploaded to it. Currently, it is being powered through the laptop’s USB port as we require the random bitstring to be sent to the laptop for testing. However, a 9V battery-powered generator would have roughly 30 hours of functionality (a 9V battery has a capacity of the order of 0.5 Ah, and an idle Arduino Uno would draw about 15mA).

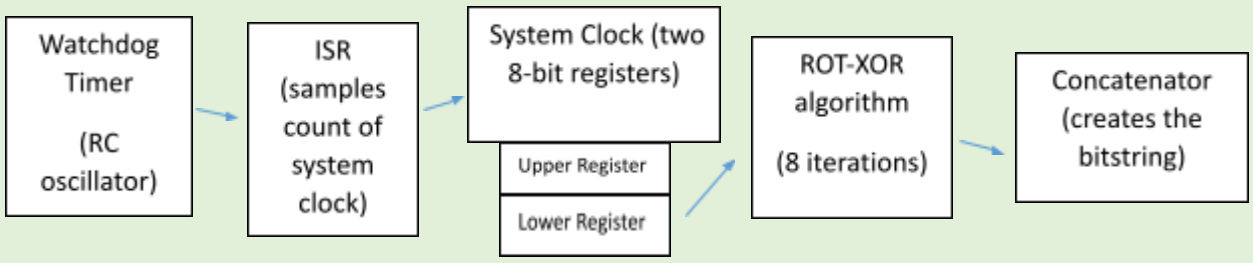
However, as the Arduino does not need to be continuously switched on, it should take less than a minute – we have observed a random bit generation rate of 60 bits/second – to generate a true random number 1000 bits long. If this functionality were to be added - the user could encrypt their message, which will generally also be less than a minute’s work. Hence, one 9V battery can last the Arduino thousands of new random numbers and thousands of encrypted messages. Pretty nifty for an all-in-one on-board generator.

We have encoded and validated the randomness of the bitstring produced using a suite of tests based on several different principles as follows:

- Frequency mono-bit test
- Frequency block test
- Runs test
- Spectral (fourier) test
- Histogram visual test
- Time series visual test
- Fourier visual test
- Bitmap visual test
- Pi test

Furthermore, we have tested our random numbers using the NIST suite of tests, and have achieved very positive results.

The **algorithm flowchart** is as below and the program code is included in the appendix to the report.




The work distribution is enlisted below as our block diagram is not descriptive of the stages in this project:

- Rehmat -
  - Researching documentation for the use of two-timers to generate random numbers
  - Researching documentation for the tests to validate the randomness of our TRNG
  - Encoding the WDT approach and the ROT-XOR algorithm
  - Encoding the frequency monobit and blocks test, fourier test, pi test
  - Assembling the circuitry to explore the zener avalanche effect entropy source
  - Writing the results and references sections for the final project report
- Aneesh -
  - Researching documentation for the use of thermistors and zener avalanche effect as sources of entropy and their viability for generating random numbers
  - Implementing Pyserial to read the serial register to Python on a laptop for validation
  - Encoding the Zener diode approach
  - Encoding the runs test, bitmap test and matrix rank test
  - Analysing the zener circuit design, testing and optimising component values
  - Writing the abstract and project details in the final project report

**MAIN COMPONENTS NEEDED TO BUILD THE PROJECT:**

The only components required for the Watchdog timer approach are the Arduino UNO and a laptop with the Arduino IDE installed.

The Zener Avalanche effect approach requires a breadboard, 3 resistors (150k, 22k, 1.5k), a 100nF capacitor, a 15V Zener diode and a 30V power supply in addition to the Watchdog timer requirements.

# RESULTS:

We have successfully created two True Random Number Generators on the Arduino.

The Watchdog Timer approach generates 60 random bits per second and our dataset of 0.22 million bits passes 9 tests that we have encoded ourselves based on varied principles. It has also performed impressively when tested on the NIST suite of tests (14/15 tests passed, and the one which failed did so due to lack of sufficient data - it has a prerequisite of 1 million bits to be run).

The Zener-Avalanche approach has a bitrate of about 800 bits per second and our dataset of 1.8 million bits passes 6 of the 9 tests we have encoded, along with performing well on NIST's test suite, passing 7/15 tests. The ones which failed seem to all be due to the same non-random behaviour we identified in the plot of the discrete fourier transform of the bitstream of this dataset.

We have also tested for comparison 0.2 million bits generated from the Pseudo-Random Number Generator incorporated in the 'random' module for Python. All these results are given below.

The tests rely on a set of bytes, which is then turned into both a set of 8-bit integers and a set of bits, on which the tests are then conducted.

### Watchdog Timer Approach

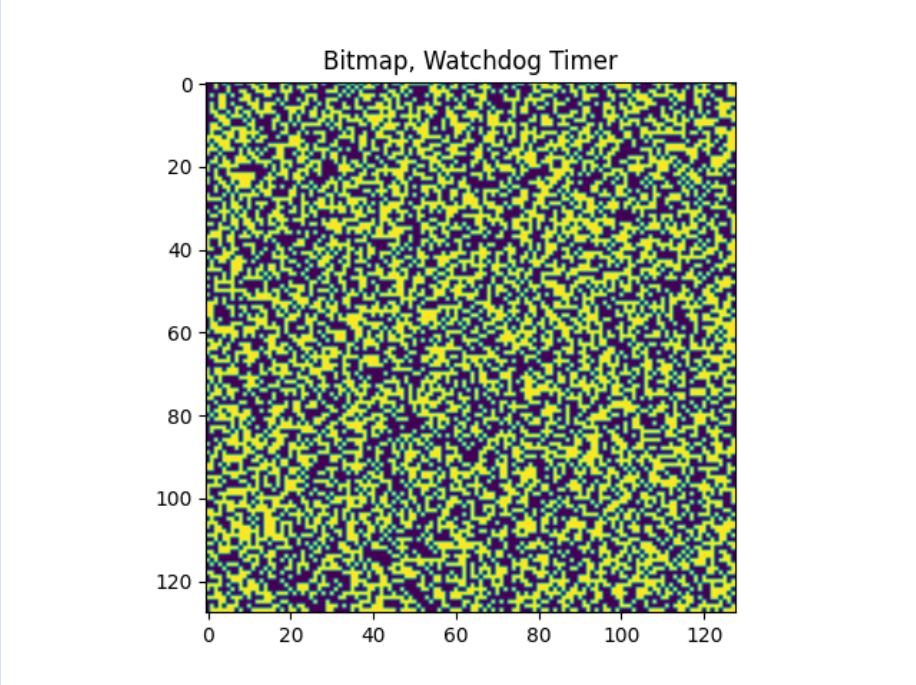
The results of our tests (the visual results follow after NIST's results):

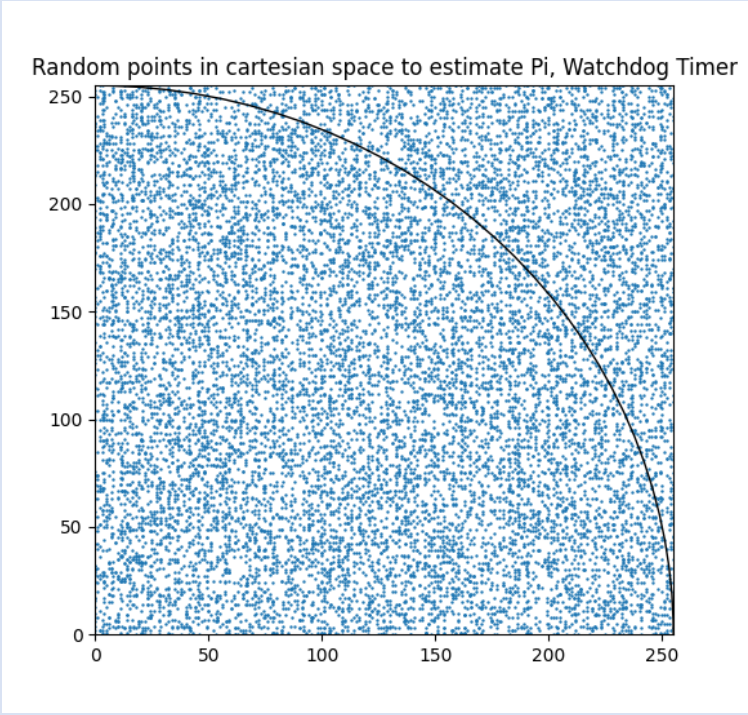
```
Number of bits :  223104
Proportion of ones :  0.4987136044176707  Expected proportion :  0.5 ± 0.0042342482556298085
Integers' Mean :  126.52499282845669  Standard deviation :  73.86931177153768  Expected STD :  73.61215932167728
Shannon Entropy per bit :  0.9999952252120003  bits, Min-Entropy per byte :  7.701267663667912
Frequency monobit test : passed
s =  1.215229249365755  p-value =  0.2242786676115436
Frequency block test : passed
Chi Squared =  151.219736087206  p-value =  0.0788539987776804
Runs Test : passed
v =  111552  distance from expected v (scaled) =  0.0022108059592526566  p-value =  0.997505376677382
Fourier Transform test : passed
d =  -2.028289820954884  p-value =  0.04253067724247794
Calculated value of Pi :  3.1612162937464143
Actual Pi :  3.141592653589793
```

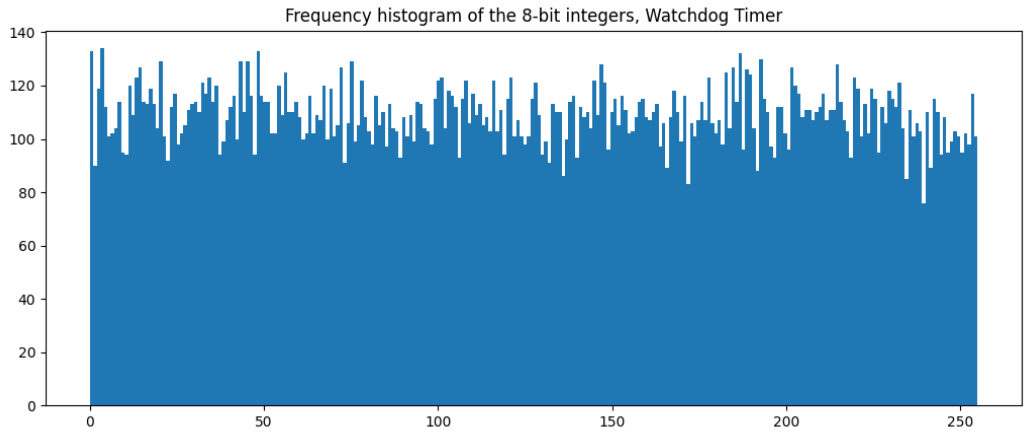
The results of NIST's test suite:

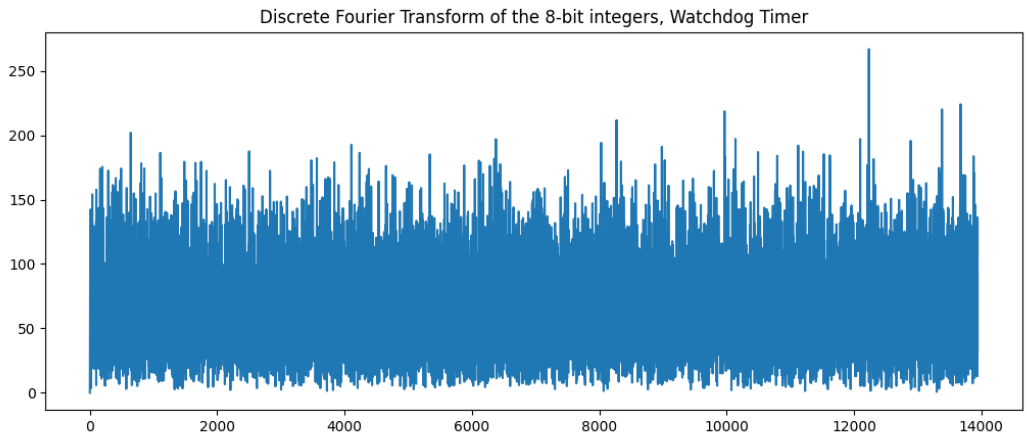
```
2.01. Frequency Test:                                                   (0.2242786676115438, True)
2.02. Block Frequency Test:                                             (0.014921792167682003, True)
2.03. Run Test:                                                                (0.997505376677382, True)
2.04. Run Test (Longest Run of Ones):                           (0.2699118682702816, True)
2.05. Binary Matrix Rank Test:                                          (0.9986710820494098, True)
2.06. Discrete Fourier Transform (Spectral) Test:       (0.04253067724247794, True)
2.07. Non-overlapping Template Matching Test:           (0.012554254759718218, True)
2.08. Overlappong Template Matching Test:                       (0.8550056056054129, True)
2.09. Universal Statistical Test:                                       (-1.0, False)
2.10. Linear Complexity Test:                                           (0.597056105774274, True)
2.11. Serial Test:                                                                      ((0.29066412717470547, True), (0.531524088197987, True))
2.12. Approximate Entropy Test:                                         (0.5136542944370146, True)
2.13. Cumulative Sums (Forward):                                        (0.06097462998211882, True)
2.13. Cumulative Sums (Backward):                                       (0.4432373465317165, True)
2.14. Random Excursion Test:
               STATE                xObs                    P-Value                         Conclusion
               '-4'          3.690202611656907       0.5948226330518233          True
               '-3'          7.103905882352942       0.21302668856653087         True
               '-2'          7.307915758896151       0.19872879518512265         True
               '-1'          7.588235294117647       0.18043652785295622         True
               '+1'          2.0588235294117645      0.8409472843768087          True
               '+2'          3.57516339869281        0.612046214663584           True
               '+3'          7.352941176470588       0.19568678642521128         True
               '+4'          2.428571428571429       0.7872118138839401          True
2.15. Random Excursion Variant Test:
               STATE          COUNTS                  P-Value             Conclusion
               '-9.0'                5       0.6176848467379198          True
               '-8.0'                1       0.4786398352857738          True
               '-7.0'                3       0.5054672911892402          True
               '-6.0'                7       0.6050949464540122          True
               '-5.0'                15      0.9089743072571725          True
               '-4.0'                20      0.8458148386257579          True
               '-3.0'                16      0.9388649898237753          True
               '-2.0'                19      0.8430219763697249          True
               '-1.0'                22      0.3911725228101395          True
               '+1.0'                10      0.2299490567942134          True
               '+2.0'                5       0.23476366282766925         True
               '+3.0'                2       0.24995961262448674         True
```

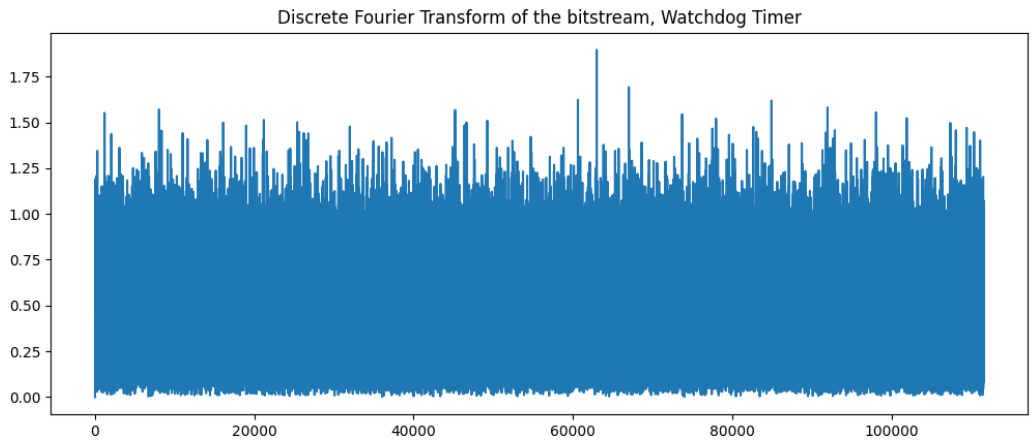
The visual results of our tests:




Random points in cartesian space to estimate Pi, Watchdog Timer
0
50
100
150
200
250


Frequency histogram of the 8-bit integers, Watchdog Timer
0
20
40
60
80
100
120
140
0
50
100
150
200
250


Discrete Fourier Transform of the 8-bit integers, Watchdog Timer
0
50
100
150
200
250
0
2000
4000
6000
8000
10000
12000
14000




Note that for these fourier transforms, the x-axis is the frequency, though not in Hertz, but rather in the inverse units of the indices of the data - i.e. the period given by the inverse of a frequency f corresponds to a periodic signal in the data repeating every 2*pi/f datapoints. It is simple to convert these to seconds and Hz if the bitrate is known.

### Zener-Avalanche Approach

Our tests:

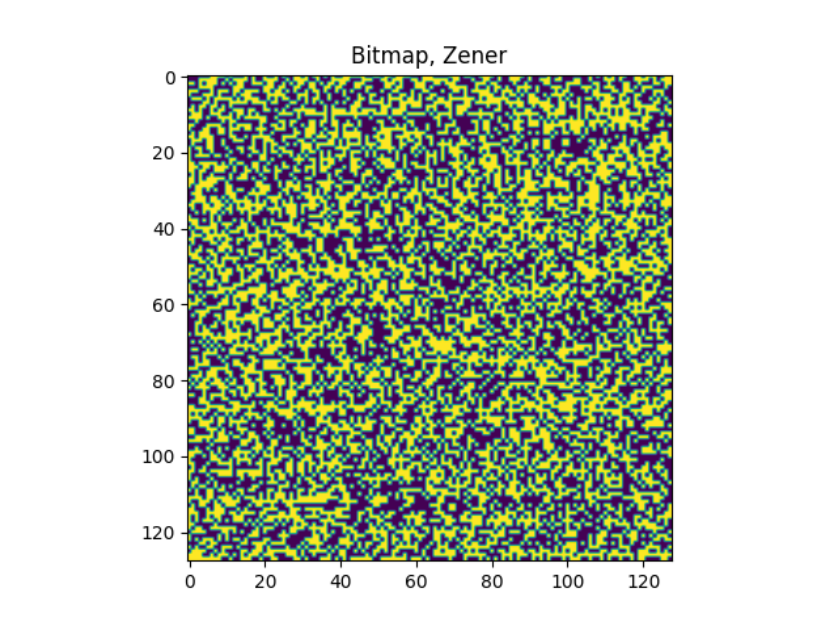
```
Number of bits :  1891016
Proportion of ones :  0.499916975847904  Expected proportion :  0.5 ± 0.00145439507065809
Integers' Mean :  127.60406469326541  Standard deviation :  73.96455654694495  Expected STD :  73.61215932167728
Shannon Entropy per bit :  0.9999999801109778  bits, Min-Entropy per byte :  7.852139709647588
Frequency monobit test : passed
s =  0.22834002609332013  p-value =  0.8193819100190329
Frequency block test : failed
Chi Squared =  248.96933595072102  p-value =  8.543702412211474e-10
Runs Test : passed
v =  943741  distance from expected v (scaled) =  1.8171783340296188  p-value =  0.010173426350782867
Fourier Transform test : failed
d =  -114.12277762128058  p-value =  0.0
Calculated value of Pi :  3.1303347209530576
Actual Pi :  3.141592653589793
```

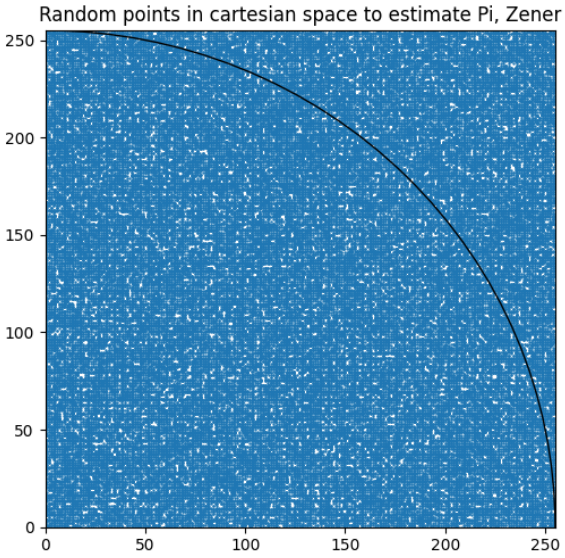
NIST’s tests:

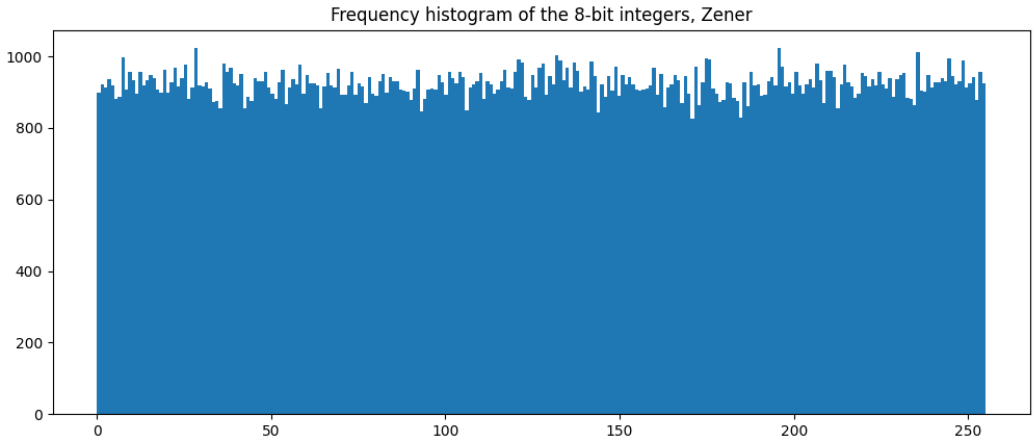
```
2.01. Frequency Test:                                                   (0.39866940560418573, True)
2.02. Block Frequency Test:                                             (0.0, False)
2.03. Run Test:                                                                 (0.003103563042356409, False)
2.04. Run Test (Longest Run of Ones):                           (3.794203920746071e-102, False)
2.05. Binary Matrix Rank Test:                                          (0.03359825686730552, True)
2.06. Discrete Fourier Transform (Spectral) Test:       (0.0, False)
2.07. Non-overlapping Template Matching Test:           (7.244253015969236e-54, False)
2.08. Overlappong Template Matching Test:                       (1.7113930794062272e-33, False)
2.09. Universal Statistical Test:                                       (0.8007517194746459, True)
2.10. Linear Complexity Test:                                           (0.1341324454265104, True)
2.11. Serial Test:                                                              ((0.0, False), (0.0, False))
2.12. Approximate Entropy Test:                                         (0.0, False)
2.13. Cumulative Sums (Forward):                                        (0.7392843240380698, True)
2.13. Cumulative Sums (Backward):                                       (0.47362915614421996, True)
2.14. Random Excursion Test:
                STATE                   xObs                            P-Value                         Conclusion
                '-4'            4.792471727415386               0.4417287207774636              True
                '-3'            4.421098191933241               0.49050912096248656             True
                '-2'            11.412266694139666              0.04379167011803299             True
                '-1'            11.656467315716272              0.03981054556861886             True
                '+1'            2.2684283727399164              0.8108925485878126              True
                '+2'            7.617696045605179               0.1786019117996686              True
                '+3'            7.850634770514606               0.16466182925860165             True
                '+4'            10.53530373007538               0.061412993145069684            True
2.15. Random Excursion Variant Test:
                STATE           COUNTS                          P-Value                 Conclusion
                '-9.0'                  744             0.8729634329652352              True
                '-8.0'                  745             0.8594843749398571              True
                '-7.0'                  757             0.7810675433568735              True
                '-6.0'                  777             0.6446833774308426              True
                '-5.0'                  777             0.6101689962923422              True
                '-4.0'                  758             0.6974837772090909              True
                '-3.0'                  797             0.35763659946964543             True
                '-2.0'                  771             0.4285328279318582              True
                '-1.0'                  699             0.5979070519386456              True
                '+1.0'                  746             0.4764603827069238              True
                '+2.0'                  823             0.11332814026833549             True
                '+3.0'                  880             0.057601009582170466            True
                '+4.0'                  838             0.23558583825861557             True
                '+5.0'                  825             0.35145934527670475             True
                '+6.0'                  857             0.27253464851273834             True
                '+7.0'                  851             0.33432723093709316             True
                '+8.0'                  769             0.7335226324967417              True
                '+9.0'                  743             0.8780042806680459              True
```

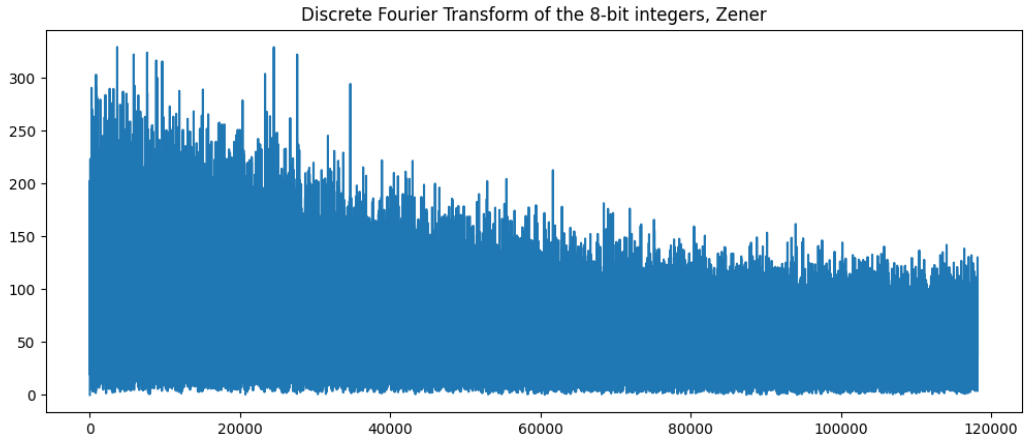
So the Zener approach fails a significant amount of the tests, notable among which is the Spectral test, from which we have plotted the fourier transform below, and it is illustrative in showing the behaviour which accounts for most of these fails.

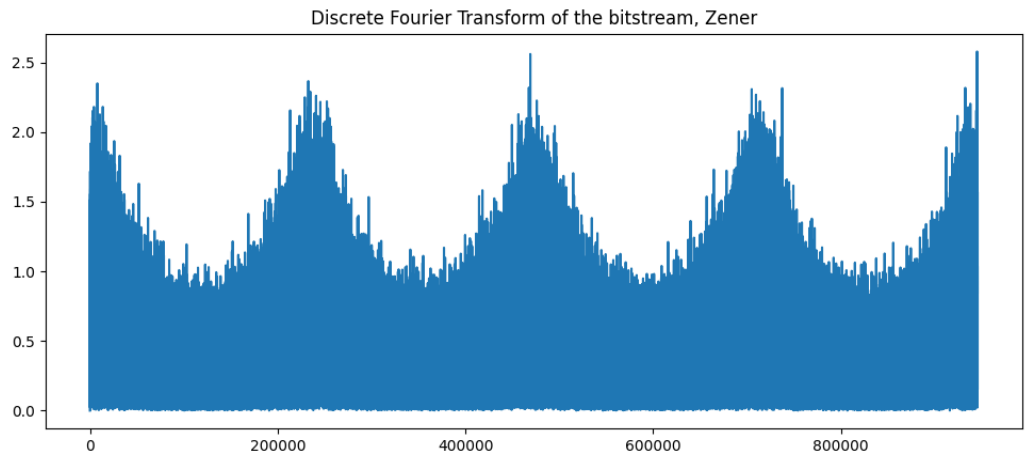
Visual tests:




Random points in cartesian space to estimate Pi, Zener
0
50
100
150
200
250


Frequency histogram of the 8-bit integers, Zener
0
200
400
600
800
1000
0
50
100
150
200
250


Discrete Fourier Transform of the 8-bit integers, Zener
0
50
100
150
200
250
300
0
20000
40000
60000
80000
100000
120000




Here, we observe an unusual periodicity in the fourier transform of the bitstream, and the fourier transform of the 8-bit integers obtained from the bytes is skewed towards lower frequencies (i.e. large scale variations in the data).

We are not completely sure as to the origins of these artifacts yet.

**Pseudo-Random Numbers**

Our tests:

```
Number of bits :  200000
Proportion of ones :  0.501125  Expected proportion :  0.5 ± 0.00447213595499958
Integers' Mean :  127.39536  Standard deviation :  73.82742912001203  Expected STD :  73.61215932167728
Shannon Entropy per bit :  0.9999963481750964  bits, Min-Entropy per byte :  7.554358038935622
Frequency monobit test : passed
s =  1.0062305898749053  p-value =  0.3143046604738541
Frequency block test : passed
Chi Squared =  144.05377720870678  p-value =  0.15732762580097648
Runs Test : passed
v =  99972  distance from expected v (scaled) =  0.08694331156979679  p-value =  0.9021416154478987
Fourier Transform test : passed
d =  -0.02051956704170308  p-value =  0.9836289031200308
Calculated value of Pi :  3.13312
Actual Pi :  3.141592653589793
```

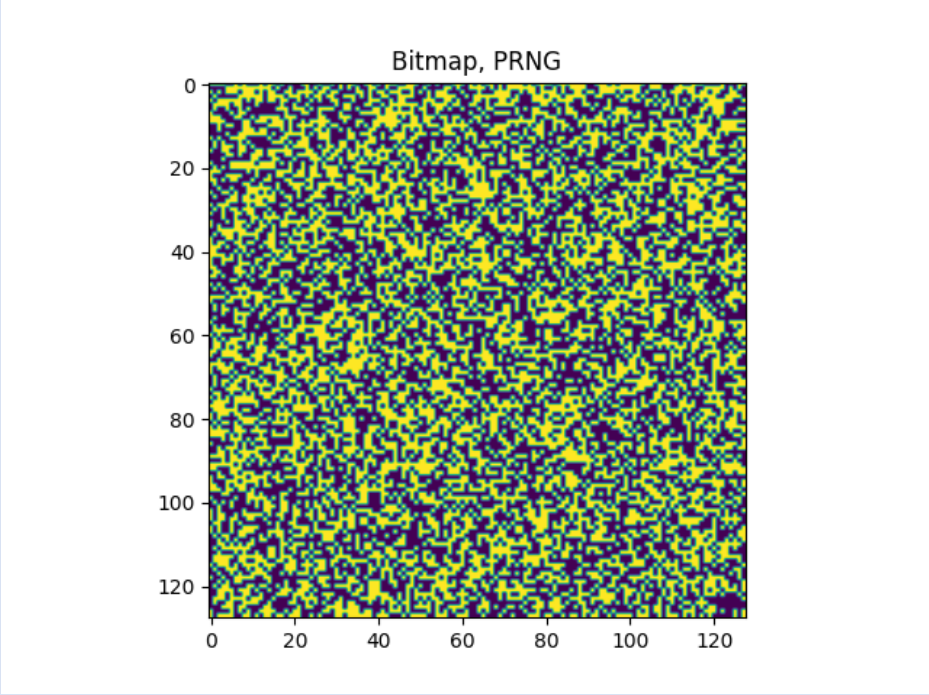
Despite the Zener approach's flawed spectral characteristics, it is impressive that the entropy per byte is higher for the Zener diode than either the Watchdog Timer, or the PRNG. Furthermore, the Watchdog Timer outperforms the PRNG.

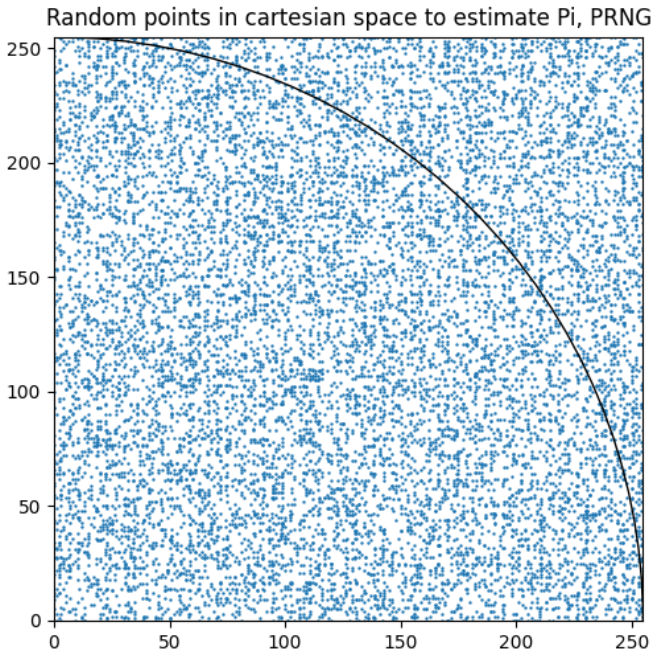
While comparing the results for our RNGs and the PRNG may be instructive, it is worth stressing that the aim of this project was not to beat a standard PRNG in passing randomness tests, but rather to create a TRNG, which has a fundamentally different region of applications than a PRNG - they are used in cryptographic applications where security is crucial and PRNGs are not applicable, and have little application in other areas (areas where true randomness is not necessary and perceived randomness will suffice) because of their comparative inefficiency.

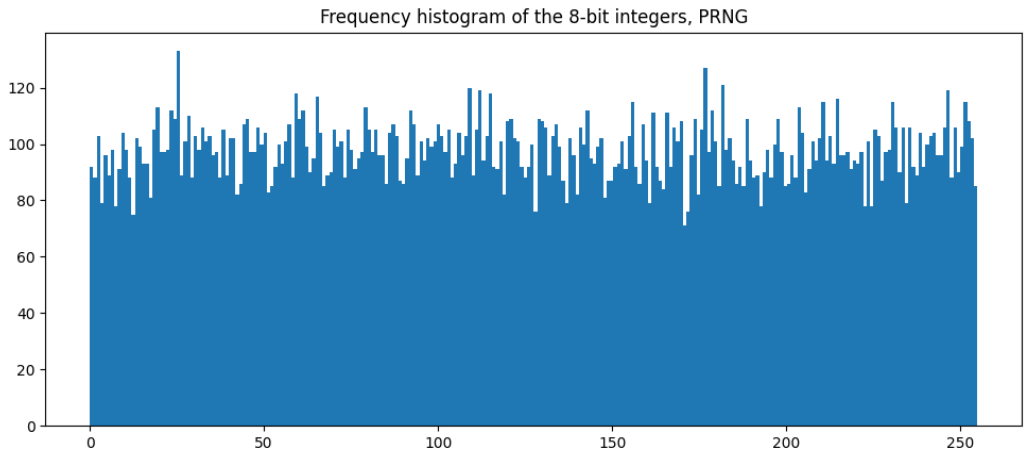
NIST's suite of tests:

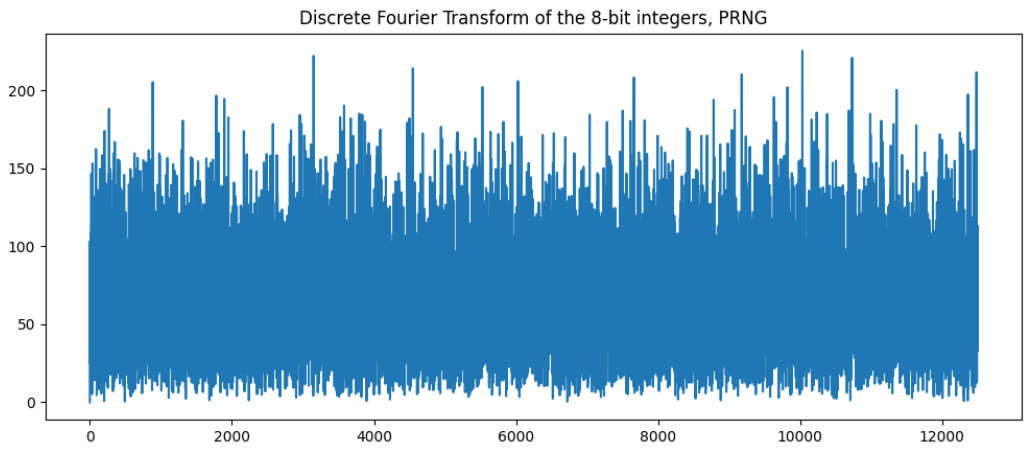
```
2.01. Frequency Test:                                                (0.31430466047385397, True)
2.02. Block Frequency Test:                                          (0.13173025879889186, True)
2.03. Run Test:                                                                (0.9021416154478987, True)
2.04. Run Test (Longest Run of Ones):                        (0.45516351517107223, True)
2.05. Binary Matrix Rank Test:                                       (0.38412149157727166, True)
2.06. Discrete Fourier Transform (Spectral) Test:        (0.9836289031200308, True)
2.07. Non-overlapping Template Matching Test:            (0.5372999626328658, True)
2.08. Overlappong Template Matching Test:                    (0.4400263967966669, True)
2.09. Universal Statistical Test:                                    (-1.0, False)
2.10. Linear Complexity Test:                                        (0.33098978519977507, True)
2.11. Serial Test:                                                             ((0.979815295257154, True), (0.8614384110332339, True))
2.12. Approximate Entropy Test:                                      (0.8140021132456853, True)
2.13. Cumulative Sums (Forward):                                     (0.11410877279001497, True)
2.13. Cumulative Sums (Backward):                                    (0.2489810065104647, True)
2.14. Random Excursion Test:
                STATE                   xObs                    P-Value                         Conclusion
                '-4'            2.747801711833552           0.7387982298956115              True
                '-3'            1.103442696629213           0.9537984937178208              True
                '-2'            5.838673879872381           0.3222345429887124              True
                '-1'            1.9887640449438202          0.8506975519989097              True
                '+1'            6.595505617977528           0.25250214857635317             True
                '+2'            3.7396310167845748          0.5874774822591173              True
                '+3'            1.4529258426966287          0.9184373931769197              True
                '+4'            4.527734230587442           0.4761812910704576              True
2.15. Random Excursion Variant Test:
                STATE           COUNTS                  P-Value                 Conclusion
                '-9.0'                  170             0.14089024960635127             True
                '-8.0'                  169             0.12156789378838208             True
                '-7.0'                  137             0.31835845383488526             True
                '-6.0'                  110             0.6350836228944423              True
                '-5.0'                  109             0.6172949293633999              True
                '-4.0'                  103             0.6916514708131072              True
                '-3.0'                  84              0.8668976716121827              True
                '-2.0'                  71              0.4360173098755147              True
                '-1.0'                  71              0.1772865301518678              True
                '+1.0'                  86              0.8220885715698659              True
                '+2.0'                  77              0.6035579771302728              True
                '+3.0'                  69              0.5026018639414771              True
                '+4.0'                  66              0.5146711178440291              True
                '+5.0'                  80              0.8220885715698659              True
                '+6.0'                  77              0.7862443124601404              True
                '+7.0'                  56              0.49270499349216745             True
                '+8.0'                  48              0.4275064729658107              True
                '+9.0'                  45              0.42378739486935724             True
```

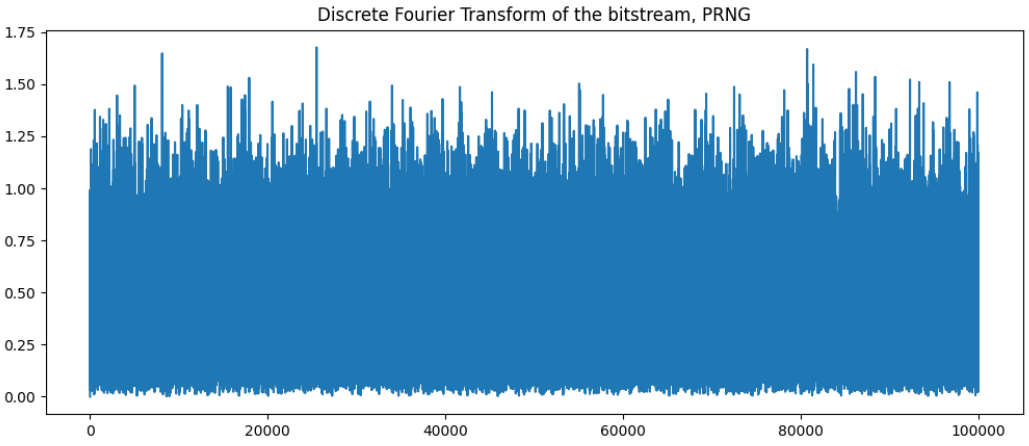
Visual tests:




Random points in cartesian space to estimate Pi, PRNG
0
50
100
150
200
250


Frequency histogram of the 8-bit integers, PRNG
0
20
40
60
80
100
120
0
50
100
150
200
250




## REFERENCES:


1. J. Hlaváč, R. Lórencz and M. Hadáček, "True random number generation on an Atmel AVR microcontroller," 2010 2nd International Conference on Computer Engineering and Technology, 2010, pp. V2-493-V2-495, doi: 10.1109/ICCET.2010.5485568.
2. Arduino Hardware True Random Generator, https://gist.github.com/endolith/2568571
3. TRNG Using a Zener diode in breakdown, https://github.com/nicolacimmino/TRNG
4. NIST Special Publication 800-90B, Recommendation for the Entropy Sources Used for Random Bit Generation, https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90B.pdf
5. Randomness extractors - Lecture from Masayrk University, https://www.fi.muni.cz/~xbouda1/teaching/current/IV111/prednasky/lecture9.pdf
6. Randomness Extractors, https://people.seas.harvard.edu/~salil/pseudorandomness/extractors.pdf

7. NIST Special Publication 800-22, A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications, https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-22r1a.pdf
8. Implementation of NIST's test suite, https://github.com/stevenang/randomness_testsuite
9. ENT — Fourmilab Random Sequence Tester, https://github.com/Fourmilab/ent_random_sequence_tester

# APPENDIX:

**Random Number Generation using Watchdog Timer (Arduino):**

```
/*
`* PH 435 Project
 * True Random Number Generator
 * Using the jitter in the time period of the Watchdog Timer
 *
 * Group 30
 * Rehmat Singh Chawla
 * Aneesh Anand Kamat
 */


#include <stdint.h>
#include <avr/interrupt.h>
#include <avr/wdt.h>

boolean valueWaiting = false;
byte value;
byte result;

unsigned int count = 0;
const unsigned int stringLength = 8;


void setup() {
  Serial.begin(9600);
  WDT_setup(); // Sets up the Watchdog timer
}

void loop() {
  if (valueWaiting) {

    result = rotate(result, 1); // Spread randomness around
    result ^= value;
    count++;

    if (count==stringLength){
       binPrint(result);
       count = 0;
    }
    valueWaiting = false;
  }
}

void WDT_setup() {
  noInterrupts(); // Due to the security features of the WDT,
  // its settings can be changed for only a few clock cycles after enabling changes
  // Hence noInterrupts is necessary.

  MCUSR &= ~bit(WDRF); // Clearing WDT Reset Flag in the MCU Status Register

  // Enable changes to WDE (Setting WDCE and WDE to 1 simultaneously)
  WDTCSR |= bit(WDCE) | bit(WDE);

  // Put WDT into interrupt mode (by setting WDIE 1 and WDE 0)
  // So instead of resetting system, it executes the interrupt handler
```

```
  // Set shortest prescaler (time-out) value = 2048 cycles (~16 ms)
  WDTCSR = bit(WDIE);

  interrupts();
}


ISR(WDT_vect) { // Interrupt Handler for the Watchdog
  value = TCNT1L; // Sample the count of the timer
  // TCNT1 = 0; If using TimerOne instead
  valueWaiting = true;
}

byte rotate(const byte value, int shift) { // Rotate a given number by some number of places in
binary
  if ((shift &= sizeof(value)*8 - 1) == 0)
    return value;
  return (value << shift) | (value >> (sizeof(value)*8 - shift));
}

byte getTotalXOR(byte x){ // XORs all the bits of a byte to get a bit
  byte result = 0;

  for (int i=0; i<10; i++){
    result ^= (x>>i)&1;
    // x = 10101000, x>>2 = 00101010, x>>5 = 00000101
    // 1 = 0000001, 0101001x & 0000001 = 0000000x
    }
  return result;
}

void binPrint(int input) { // Print an integer in 8-bit binary
  for (unsigned int mask = 0x80; mask; mask >>= 1) {
    Serial.print( (mask & input) ? '1' : '0' );
  }
  Serial.println();
}

void intPrint(int* bitList, int listLength) {
  int result=0;
  for (int i=0; i<listLength; i++) {
    if (bitList[i]) {
      result += int(bit(i));
    }
  }
  Serial.print(result);
}
```

## Random Number Generation using Zener Diode (Arduino):

```
/*
 * PH 435 Project
 * True Random Number Generator
 * Using the Avalanche Effect in a Zener Diode in Breakdown Region
 *
 * Group 30
 * Rehmat Singh Chawla
 * Aneesh Anand Kamat
 */

int sensorPin = A0;
int value;
int thisBit;
int lastBit;
byte result;

const int bitStringLength = 8;
int count = 0;
int bitString[bitStringLength];
//int change = 0;

void setup() {
  Serial.begin(9600);
}
```

```
void loop() {
  // Want delay to avoid correlation between consecutive measurements
  // analogRead takes ~100 us, sufficient delay
  value = analogRead(sensorPin);

  result = rotate(result, 1); // Spread randomness around
  delay(1);
  result ^= value;
  count++;

  if (count==bitStringLength){
     binPrint(result);
     count = 0;
  }
}

int getLSB(int x){
  return bitRead(x, 0); // Takes LSB
}


int getTotalXOR(int x){
  int result = 0;

  for (int i=0; i<10; i++){
    result ^= (x>>i)&1; // x = 10101000, x>>2 = 00101010, x>>5 = 00000101
    // 1 = 0000001, 00000101 & 0000001 = 00000001
    }
  // XORs all the bits of sensorValue to get thisBit

  return result;
}

byte rotate(const byte value, int shift) { // Rotate a given number by some number of places in
binary
  if ((shift &= sizeof(value)*8 - 1) == 0)
    return value;
  return (value << shift) | (value >> (sizeof(value)*8 - shift));
}

void binPrint(int input) { // Print an integer in 8-bit binary
  for (unsigned int mask = 0x80; mask; mask >>= 1) {
    Serial.print( (mask & input) ? '1' : '0' );
  }
  Serial.println();
}
```

## Code for testing the Random Numbers (Python code, run on laptop):

```
import numpy as np
from scipy.special import gammainc
from scipy.fft import *
import matplotlib.pyplot as plt
from math import *
from nist_tests import testsBatteryNIST

with open("timer_random_numbers_rotxor.csv","r") as f:
       data = f.read()

byteStrings = data.rstrip("\n").split("\n")
# Now numbers should have the random numbers in 8-bit strings
numbers = [int(x, 2) for x in byteStrings]
# print(byteStrings[:5])
# print(numbers[:5])

quantityBytes = len(byteStrings)
elementSize = len(byteStrings[0])
numBits = quantityBytes*elementSize


bits = []
for byte in byteStrings:
       for x in byte:
              bits.append(int(x))
```

```
bitsString = ''.join(byteStrings)

def vonNeumannAlgorithm(bitsArray):
	length = len(bitsArray)
	newArray = []
	for i in range(length//2):
		if (bitsArray[2*i] ^ bitsArray[2*i+1]):
			newArray.append(bitsArray[2*i])
	return newArray

def makeByteStringArray(bitsArray):
	newArray = []
	for i in range(len(bitsArray)//8):
		temp = ""
		for j in range(8):
			temp += str(bitsArray[8*i+j])
		newArray.append(temp)
	return newArray

def sanityChecks(bitsArray, intArray):
	numBits = len(bitsArray)
	numInts = len(intArray)
	propOnes = sum(bitsArray)/numBits
	print("Number of bits : ",numBits)
	print("Proportion of ones : ",propOnes, " Expected proportion :
",0.5,"±",2/sqrt(numBits))
	mean = sum(intArray)/numInts
	print("Integers' Mean : ", mean," Standard deviation : ",np.std(intArray), " Expected
STD : ", 255/sqrt(12))
	entropyPerBit = -propOnes*log2(propOnes) - (1-propOnes)*log2(1-propOnes)
	minEntropyPerByte = -log2(max(np.histogram(intArray, bins=256)[0])/numInts)
	print("Shannon Entropy per bit : ",entropyPerBit, " bits, Min-Entropy per byte : ",
minEntropyPerByte)


# Frequency monobit test
def freqMonobitTest(bitsArray):
	length = len(bitsArray)
	s = abs(2*bitsArray.count(1) - length)/np.sqrt(length) # The test statistic
	# erfc is the Complementary Error Function, imported from math
	p = erfc(s/np.sqrt(2)) # p-value

	print("Frequency monobit test : ", end="")
	if p>0.01:
		print("passed")
	else:
		print("failed")
	print("s = ",s," p-value = ",p)

def freqBlockTest(bitsArray, numBlocks):
	length = len(bitsArray)
	blockLength = length//numBlocks
	bitBlocks = [bitsArray[blockLength*i : blockLength*(i+1)] for i in range(numBlocks) ]
	propOnes = np.array([sum(x)/blockLength for x in bitBlocks])
	chiSquared = 4*blockLength*(sum( (propOnes - 1/2)**2 ))
	p = 1 - gammainc(numBlocks/2, chiSquared/2)

	print("Frequency block test : ", end="")
	if p>0.01:
		print("passed")
	else:
		print("failed")
	print("Chi Squared = ",chiSquared," p-value = ",p)

# Calculating Pi using random numbers and seeing accuracy
def valueOfPiTest(intArray, makePlot = True, label = ""):
	length = len(intArray)
	numCords = length//2
	xCords = intArray[:numCords]
	yCords = intArray[numCords: 2*numCords]
	pointsInCircle = 0

	for i in range(numCords):
		if (xCords[i]**2 + yCords[i]**2 <= 255**2):
			pointsInCircle += 1
```

```
        piValue = 4*(pointsInCircle/numCords)

        print("Calculated value of Pi : ", piValue)
        print("Actual Pi : ", pi)

        if makePlot:
                plt.figure()
                fig, ax = plt.subplots()
                ax.scatter(xCords, yCords, s=0.5)
                # fig = plt.gcf()
                # ax = fig.gca()
                arc = plt.Circle((0,0), 255, fill = False)
                ax.add_patch(arc)
                ax.set_xlim(0,255)
                ax.set_ylim(0,255)
                if label:
                        ax.set_title("Random points in cartesian space to estimate Pi, " + label)
                else:
                        ax.set_title("Random points in cartesian space to estimate Pi")

# Runs Test
def runsTest(bitsArray):
        length = len(bitsArray)

        propOnes = sum(bitsArray)/length

        prereq = bool(abs(propOnes-0.5) < 2/np.sqrt(length))

        v = 1 #initializing the test statistic

        if prereq:
                for i in range(length-1):
                        if bitsArray[i] != bitsArray[i+1]:
                                v += 1

                p = erfc((abs(v -
2*length*propOnes*(1-propOnes)))/(2*np.sqrt(2*length)*propOnes*(1-propOnes))) #p-value
                # erfc is the Complementary Error Function, imported from math

                print("Runs Test : ", end="")
                if p>0.01:
                        print("passed")
                else:
                        print("failed")
                print("v = ",v," distance from expected v (scaled) = ",(abs(v -
2*length*propOnes*(1-propOnes)))/(2*np.sqrt(2*length)*propOnes*(1-propOnes))," p-value = ",p)
        else:
                print("Frequency test prerequisite was failed, so Runs test not run")

def longestRunsBlockTest(bitsArray):
        length = len(bitsArray)
        if length>=128:
                if length>=6272:
                        if length>=75*10**4:
                                blockLength = 10**4
                        else:
                                blockLength = 128
                else:
                        blockLength = 8
        else:
                print("Insufficient number of bits for Runs Block Test")
                return

        numBlocks = length//blockLength
        bitBlocks = [bitsArray[blockLength*i : blockLength*(i+1)] for i in range(numBlocks) ]
        propOnes = np.zeros(numBlocks)
        freqArray = np.zeros(7)

        for i in range(numBlocks):
                bitBlock = bitBlocks[i]
                propOnes[i] = sum(bitBlock)/blockLength
                index = longestRunIndex(bitBlock, blockLength)
                if index == -1:
                        print("Something went wrong, longestRunIndex returned -1")
                        return
```

```
                freqArray[index] += 1

        probArray = []
        if blockLength == 8:
                probArray = [0.2148, 0.3672, 0.2305, 0.1875]
        elif blockLength == 128:
                probArray = [0.1174, 0.2430, 0.2493, 0.1752, 0.1027, 0.1124]
        elif blockLength == 10**4:
                probArray = [0.0882, 0.2092, 0.2483, 0.1933, 0.1208, 0.0675, 0.0727]

        chiSquared = 0
        K = len(probArray)-1
        N_values = {3 : 16, 5 : 49, 6 : 75}
        N = N_values[K]

        for i in range(K+1):
                chiSquared += (freqArray[i] - N*probArray[i])**2/(N*probArray[i])

        p = 1 - gammainc(K/2, chiSquared/2)

        print("Longest Runs Block test : ", end="")
        if p>0.01:
                print("passed")
        else:
                print("failed")
        print("Chi Squared = ", chiSquared, " p-value = ", p)
        print(blockLength, K, probArray, freqArray)

def longestRunIndex(bitsArray, blockLength):
        lengthOfCurrentRun = 0
        lengthOfLongestRun = 0
        for i in range(blockLength):
                if bitsArray[i]==1:
                        lengthOfCurrentRun += 1
                else:
                        if lengthOfCurrentRun>lengthOfLongestRun:
                                lengthOfLongestRun = lengthOfCurrentRun
                        lengthOfCurrentRun = 0
        if lengthOfCurrentRun>lengthOfLongestRun:
                lengthOfLongestRun = lengthOfCurrentRun


        if blockLength == 8:
                if 0<lengthOfLongestRun<4:
                        return lengthOfLongestRun-1
                elif lengthOfLongestRun >= 4:
                        return 3

        elif blockLength == 128:
                if 0<lengthOfLongestRun<=4:
                        return 0
                elif 4<lengthOfLongestRun<9:
                        return lengthOfLongestRun - 4
                elif lengthOfLongestRun >= 9:
                        return 5

        elif blockLength == 10**4:
                if 0<lengthOfLongestRun<=10:
                        return 0
                elif 10<lengthOfLongestRun<16:
                        return lengthOfLongestRun - 10
                elif lengthOfLongestRun >= 16:
                        return 6

        else:
                print("Invalid block length entered.")
                return -1

def runsInBlock(bitsArray, blockLength): # Returns the tabulated frequencies
        freqArray = np.zeros(7)
        lengthOfCurrentRun = 0
        bitsArray.append(0)

        if blockLength == 8:
                for i in range(blockLength+1):
```

```
                        if bitsArray[i]==1:
                                lengthOfCurrentRun += 1
                        elif 0<lengthOfCurrentRun<4:
                                freqArray[lengthOfCurrentRun-1] += 1
                        elif lengthOfCurrentRun >= 4:
                                freqArray[3] += 1

        if blockLength == 128:
                for i in range(blockLength+1):
                        if bitsArray[i]==1:
                                lengthOfCurrentRun += 1
                        elif 0<lengthOfCurrentRun<=4:
                                freqArray[0] += 1
                        elif 4<lengthOfCurrentRun<9:
                                freqArray[lengthOfCurrentRun-4] += 1
                        elif lengthOfCurrentRun >= 9:
                                freqArray[5] += 1

        if blockLength == 10**4:
                for i in range(blockLength+1):
                        if bitsArray[i]==1:
                                lengthOfCurrentRun += 1
                        elif 0<lengthOfCurrentRun<=10:
                                freqArray[0] += 1
                        elif 10<lengthOfCurrentRun<16:
                                freqArray[lengthOfCurrentRun-10] += 1
                        elif lengthOfCurrentRun >= 16:
                                freqArray[6] += 1

        return freqArray

def matrixRankTest(bitsArray):
        pass

def fourierTransformTest(bitsArray):
        length = len(bitsArray)
        M = abs(fft(2*np.array(bitsArray) - 1)[:length//2])
        T = sqrt(length*log(1/0.05))
        ExpectedNumPeaks = 0.95*length/2
        ActualNumPeaks = np.count_nonzero(M<T)
        d = 2*(ActualNumPeaks - ExpectedNumPeaks)/sqrt(length*0.95*0.05)
        p = erfc(abs(d)/sqrt(2))

        print("Fourier Transform test : ", end="")
        if p>0.01:
                print("passed")
        else:
                print("failed")
        print("d = ",d," p-value = ",p)


def testsBattery(bitsArray, intArray, makePlots = True, label="", bitmapSize = 128):
        sanityChecks(bitsArray, intArray)
        freqMonobitTest(bitsArray)
        freqBlockTest(bitsArray, 128)
        runsTest(bitsArray)
        # longestRunsBlockTest(bitsArray) # Not workin yet
        fourierTransformTest(bitsArray)
        valueOfPiTest(intArray, makePlots, label)
        # matrixRankTest(bitsArray, 8) Yet to be encoded

        if makePlots:
                if len(bitsArray)>=(bitmapSize**2):
                        plt.figure()
                        plt.imshow(np.reshape(bitsArray[:bitmapSize**2],
(bitmapSize,bitmapSize)))
                        if label:
                                plt.title("Bitmap, " + label)
                        else:
                                plt.title("Bitmap")
                else:
                        print("Not enough numbers for Bitmap - decrease bitmap size")
                plt.figure()
                plt.plot([*[0],*abs(rfft(bitsArray, norm = "ortho")[1:])])
                # Get rid of the 0th term because it is essentially the sum of the array
```

```
		# And so is very high, but does not represent a frequency
		# Note that these frequencies are not in Hz
		if label:
			plt.title("Discrete Fourier Transform of the bitstream, " + label)
		else:
			plt.title("Discrete Fourier Transform of the bitstream")
		plt.figure()
		plt.plot([*[0],*abs(rfft(intArray, norm = "ortho")[1:])])
		if label:
			plt.title("Discrete Fourier Transform of the 8-bit integers, " + label)
		else:
			plt.title("Discrete Fourier Transform of the 8-bit integers")
		plt.figure()
		plt.plot(numbers[:])
		plt.grid(True)
		if label:
			plt.title("Time-series of 8-bit integers, " + label)
		else:
			plt.title("Time-series of 8-bit integers")
		plt.figure()
		plt.hist(intArray, bins = 256)
		if label:
			plt.title("Frequency histogram of the 8-bit integers, " + label)
		else:
			plt.title("Frequency histogram of the 8-bit integers")


# Running Tests

if __name__ == "__main__":
	makePlots = True
	applyVN = False

	if applyVN:
		vnBits = vonNeumannAlgorithm(bits)
		vnByteStrings = makeByteStringArray(vnBits)
		vnNumbers = [int(x, 2) for x in vnByteStrings]

		print("Before Von Neumann : ")
		testsBattery(bits, numbers, makePlots, "Before Von Neumann")

		print("\nAfter Von Neumann : ")
		testsBattery(vnBits, vnNumbers, makePlots, "After Von Neumann")

	else:
		testsBattery(bits, numbers, makePlots)
		testsBatteryNIST(bitsString)

	plt.show()
```

**Code for testing Pseudo-Random Numbers from the Random module (Python code, run on laptop):**

```
from random import getrandbits
from tests import *
from nist_tests import testsBatteryNIST


n = 2000000 # Number of bits you want

bitsString = bin(getrandbits(n))[2:]

assert len(bitsString) == n

byteStrings = []
for i in range(n//8):
    byteStrings.append(bitsString[8*i:8*(i+1)])

numbers = [int(x, 2) for x in byteStrings]
bits = [int(x) for x in bitsString]

testsBattery(bits, numbers, True)
testsBatteryNIST(bitsString)
```