Image encryption:

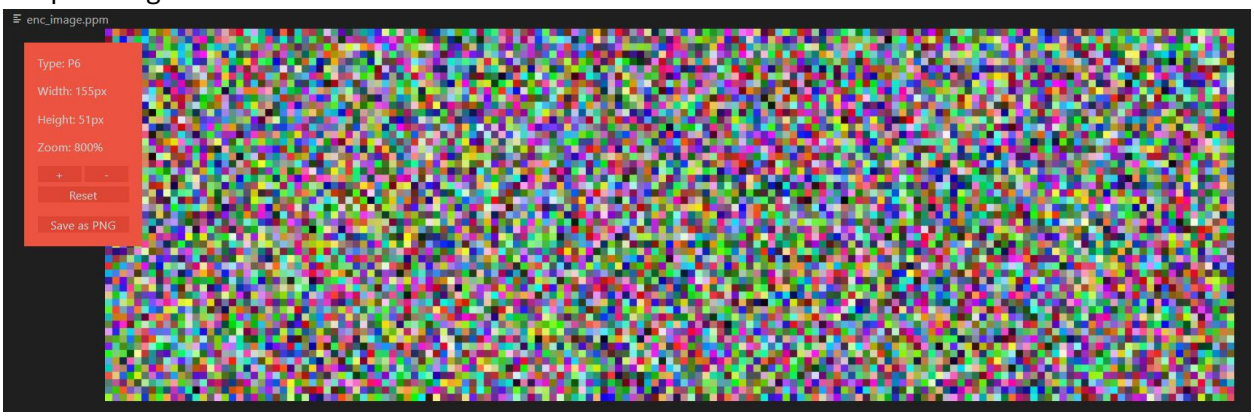
I first copied my encrypt function from the last homework and changed it to encrypt_block(). It now only takes a plaintext bitvector as an input. I moved key initialization to the init() function and that sped up my implementation considerably.

In my ctr_aes_image() function, I open the plaintext image as a bitvector until I get 3 newline characters to know that I get through the header. Then I start a while(more to read) loop and read from the plain image 128 bits at a time, padding if the block is ever less than 128 bits. Then I encrypt the block, increment the initialization vector, xor the current chunk with the output of the encrypt_block function, and then write that result to the output file.

Code:

```
def ctr_aes_image(self, iv, image_file, enc_image):
    # iv (bitvector): 128-bit init vector
    # image_file (string): input .ppm file name
    # enc_image (string): output .ppm file name
    f2 = open(enc_image, 'wb')
    '''
    f = open(image_file, 'rb')
    f2.write(f.readline()) # write 3 lines for header
    f2.write(f.readline())
    f2.write(f.readline())
    '''
    # the encryption algorithm encrypts a b-bit integer produced by th
    # what is encrypted is the XOR of the encryption of the integer an
    # of the plaintext
    bv_image = BitVector( filename=image_file) # cursor already moved
    j = 0
    while j < 3:
        temp = bv_image.read_bits_from_file(8)
        if(temp.get_bitvector_in_ascii()=='\n'):
            j += 1
        temp.write_to_file(f2)
    while (bv_image.more_to_read):
        current_chunk = bv_image.read_bits_from_file(128)
        if current_chunk.size < 128:
            print("PADDED!!!")
            current_chunk.pad_from_right(128 - current_chunk.size)
        if current_chunk._getsize() > 0:
            enc_bv = self.encrypt_block(iv)
            iv = BitVector(intVal = (iv.int_val() + 1), size = 128) #i
            output = current_chunk^enc_bv
            print(output.get_bitvector_in_hex(), end="\n")
            output.write_to_file(f2)
    f2.close()
```

Output image:



Random number generation:

Following the diagram from the 10.6 lecture notes, I generated 5 pseudorandom numbers. I used 2 lists for the seeds and the random numbers. In a for loop iterating from 0 to totalNum, I generate an output from the encoded datetime XORed with seed[i], then I generate a generate a second output from datetime XORed with the random number that was generated for that iteration, then I make the next seed by encrypting that output. This function uses the same encrypt_block() function from the image encryption portion of this assignment. Then I write those 5 random numbers to the output file.

Numbers generated:

```
aubre@Aubrey_IdeaPad MINGW64 ~/OneDrive - purdue.edu/Desktop/
ECE404/HW05 (master)
$ python AES.py -r 5 key.txt random_numbers.txt
331374527193731622526773163027689011175
262633037080229609278739248627548891 87
621388110439928640615094882415799550 8
317525806849049200816126045738729418009
240080400546264647934751409092776671804
```