Relatório T1 - Modelagem e Simulação

Membros:

- Manoella Rockembach, 19102193
- Rodrigo Ferraz Souza, 19103563

1 - Introdução

Neste relatório contém algumas das principais funções de geração de números pseudo aleatórios assim como a implementação em python de duas funções: a função dos métodos quadrados e a função Hash SHA256, a qual foi desenvolvida pela dupla. Este trabalho tem o objetivo de analisar as duas funções implementadas, tendo em vista a pesquisa apresentada e importantes fatores como eficiência, o tamanho do período de números gerados e a correlação entre os mesmos.

Dessa forma poderemos medir e comparar as funções implementadas quando a sua usabilidade em áreas como estatística, amostragem aleatória ou em simulações. Áreas nas quais pode-se tirar grande proveito de uma função de geração de números aleatórios ótima.

2 - Pesquisa sobre Funções Geradoras

2.1 - Geração de números pseudo aleatórios por relação recursiva

A geração de números pseudo aleatórios geralmente ocorre por uma relação recursiva entre os valores. Relação na qual o próximo número da sequência é uma função do último ou dos dois últimos números gerados. Ou seja, dado um valor inicial x, chamado de semente, escolhido para dar início a sequência de números, o próximo valor é gerado da aplicação da função criada no último número gerado.

Uma das características desse método é ser determinístico, ou seja, conhecendo a função f, podemos gerar a mesma sequência de números pseudo aleatórios sempre que fornecermos o mesmo valor de semente. Além disso, os números gerados são periódicos, e a sequência acabará se repetindo eventualmente.

Esse método é útil se você precisar repetir a mesma sequência de números novamente. E é eficiente para produzir muitos números em um curto espaço de tempo.

2.2 - Métodos de geração:

2.2.1 - Método congruente:

$$x_n = a^n mod\, m$$

Desenvolvido pelo Prof. D. H. Lehmer, em 1951. Baseia-se na ideia que os restos de sucessivas potências de um número possuíam boas características de aleatoriedade. Sua função obtinha o n-ésimo número de uma sequência, tomando o resto da divisão da n-ésima potência de um inteiro a por um outro inteiro m.

2.2.2 - Método Congruente Multiplicativo:

$$x_n = ax_{n-1}\, mod\, m$$

O parâmetro a é chamado de multiplicador e o parâmetro m de módulo. Os valores escolhidos por Lehmer para estas variáveis foram a = 23 e m = 108 + 1, valores escolhidos pela facilidade de implementação no ENIAC, que era uma máquina de oito dígitos decimais.

2.2.3 - Método Congruente Linear:

$$x_n = (ax_{n-1} + b)\, mod\, m$$

Os valores de xn são inteiros entre 0 e m-1, as constantes a e b são positivas. De maneira geral, a escolha dos valores de a, b, e m afetam o período e a auto correlação na sequência.

2.2.4 - Método de quadrados médios:

Esse método inventado por John Von Neumann, tem uma vantagem em relação aos métodos anteriores pois a quantidade de números gerados antes que a sequência comece a repetir não depende mais do módulo da função usada para gerar a sequência de números aleatórios, dessa forma a extensão de números até as repetições começarem é maior.

Neste método, a partir de uma semente, esse número é então elevado ao quadrado, e os dígitos do centro são usados como próximo elemento da sequência, repetindo o processo. Caso o número de dígitos que fique à esquerda seja maior que os que ficam à direita não há problema, simplesmente fixamos para qual lado vamos fazer o corte.

A aleatoriedade da sequência e o seu período dependem apenas da semente inicial.

3 - Análise e Implementação

3.1 - Função desenvolvida com Hash SHA256 desenvolvida

A técnica que utilizamos para fazer um programa que gere números pseudo aleatórios foi:

```
import hashlib
import time
import math
seed = 1
def hexstr_to_dec(num):
    m = {
        "0":0,
        "1":1,
        "2":2,
        "3":3,
        "4":4,
        "5":5,
        "6":6,
        "7":7,
        "8":8,
        "9":9,
        "a":10,
        "b":11,
        "c":12,
        "d":13,
        "e":14,
        "f":15,
    }
    return m[num]

def rand_sha():
    global seed
    seed = seed + ((float(time.time())))/seed
    #print("seed: ", seed)
    string_seed = hashlib.sha256(int(seed**3).to_bytes(16, 'little',
signed=False)).hexdigest().lower()
    #print("HEX: ",string_seed)
    sum = 0
    for l in string_seed:
        sum+= hexstr_to_dec(l)
    result = 1.01 + seed
    for l in string_seed:
        #print(l)
        result = hexstr_to_dec(l)*sum + result/sum
        #print('parcial result: ',result)
        #print()
    return int(result*10000 * seed**2)
```

- Usamos o timestamp do momento da execução como seed
    - O primeiro da seed valor é 1 nos testes
    - `seed = seed + ((float(time.time())))/seed`
- Geramos uma hash em SHA256 da conversão da seed em string
- Somamos cada dígito em hexadecimal para obtermos a soma de todos os números.

- Então inicializamos a variável result
    - `result = 1.01 + seed`

- Com a soma em mãos executamos a seguinte operação para cada letra da hash gerada
    - `result = hexstr_to_dec(l)*sum + result/sum`
- Então retorna-se
    - `int(result*10000 * seed**2)`

3.2 - Função dos Quadrados Médios

A parte inteira do timestamp, um número de 10 dígitos, é usado como a semente inicial do algoritmo. A partir disso, a semente é elevada ao quadrado e são escolhidos os dez números do meio como a próxima semente. Isso é feito sucessivamente sempre que a função é chamada.

```
import math
import time

seed = int(time.time())
seed_size = len(str(seed))
start = math.floor(seed_size/2)
end = start + seed_size
def mid_square():
    global seed
    global start
    global end

    seed_number = seed
    seed_number=int(str(seed_number*seed_number).zfill(seed_size*2)[start:end])
    seed = seed_number
    return seed_number
```

3.3 - Como foi feita a análise das funções

Para obtermos resultados concretos, que serão discutidos no tópico a seguir, foi desenvolvido um programa em python, disponível no Github no arquivo main.py, que chama as funções e as testa.

Para selecionar qual método desejamos testar, foi utilizado a seguinte abordagem:

```
        if __name__ == "__main__":
            test = input("Which method do you want to visualize?\n1 - SHA256\n2 - Python
random()\n3 - Mid Square\nSelect: ")
            test_type = input("Which test type do you want to do?\n1 - Test Until Repeat\n2 - Noise
Generator\n3 - Print Table\nSelect: ")
            algs = [rand_sha,python_random,mid_square]
            randomgen = algs[int(test)-1]
```

Para uma inspeção mais detalhada do algoritmo desenvolvido, recomendamos os próprios códigos fontes que estão indicados no link acima.

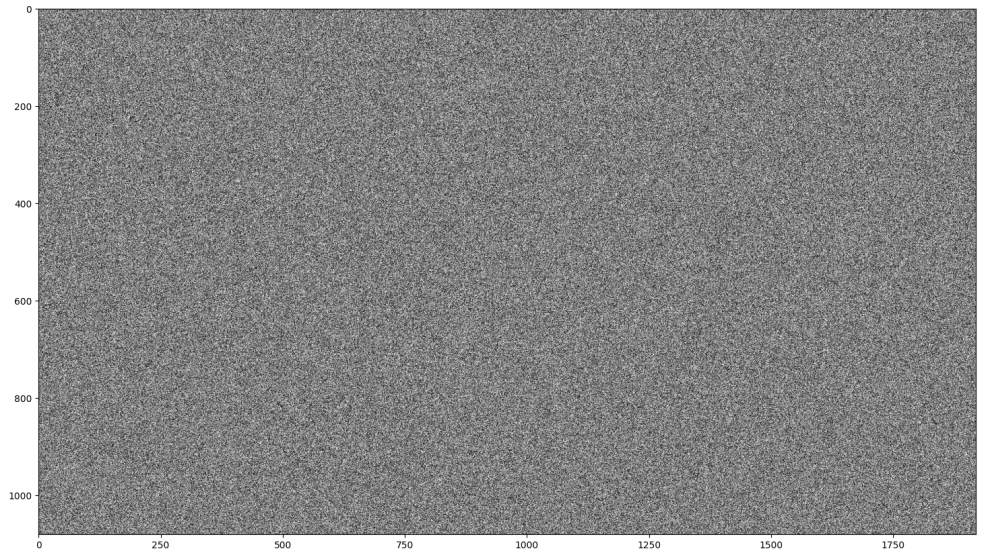
4 - Resultados Obtidos

4.1 - Função Hash SHA256 desenvolvida

4.1.1 - Testes para avaliar a aleatoriedade

Para analisar bem este problema foi utilizado duas técnicas, que também serão utilizadas no algoritmo tradicional discutido mais abaixo para fins de comparação. E são elas:

- Gerar números infinitamente e guardá-los em uma lista e, caso seja gerado um repetido, o loop é interrompido e mostra-se em tela quantas iterações se passaram até que tal fato ocorresse.
    - Nos nossos testes o algoritmo gera, em média, mais de 10 000 000 números antes de repetir
- Gerar números aleatórios entre 0 e 255 e inseri-los em uma matriz 1920x1080 e exibir em tela a imagem em escala de cista que isto representa, mostrando um ruído, para avaliarmos, graficamente, a existência, ou não, de algum padrão na geração de números
    - O resultado do nosso algoritmo é o que segue:



4.1.2 - Tabela de números aleatórios SHA256

Table of random Numbers

71625216	58920960	12708608	46790912	19842304	22092032	89723904	44460800	25578240	89303808
64494336	27912704	05607936	06818816	35761408	07901440	45508864	89186304	54319872	56304128
09362688	14270464	61636608	79773184	09547520	50213632	85314816	72028416	79479552	65274880
22857216	76155136	60060928	61983744	97699072	25860352	07942912	47660032	32594432	96043264
95585024	22152960	91574016	71457024	41858560	43540992	26468096	23044864	36399616	45299200
17320704	19283712	17276672	70327552	88512768	15933696	71881728	61341952	00284416	64909568
66936576	44321280	38432256	46233088	48948224	05181440	06632960	90914048	04675840	17055232
82858752	83826176	87471616	56776192	56467968	71696128	76441344	68881664	63358720	92221696
67897600	73821696	81658368	72228608	93589248	29381632	12300544	89442048	14242048	80822528
45810432	76055040	99499264	01275392	52206336	59083776	39735552	77112576	66392064	60425984

4.2 - Função dos quadrados médios

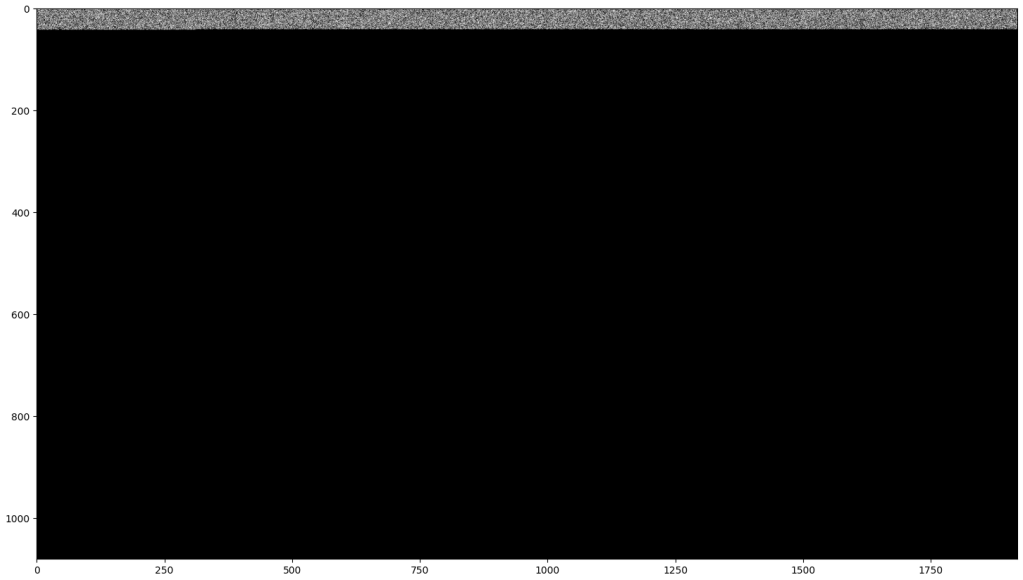
4.2.1 - Testando para avaliar sua aleatoriedade

Os mesmos testes que foram feitos no tópico anterior foram retomados aqui, para que possamos comparar a performance dos códigos.

Os testes foram feitos utilizando uma seed gerada com o timestamp do momento do teste, portanto tinham 10 dígitos.

Mesmo com uma semente relativamente única como o timestamp de 10 dígitos, a sequência de números gerados é pequena se comparada à outra função implementada, e é comum a semente se tornar cada vez mais pequena até que atinja o valor de zero e não gere mais números.

- Gerando números até que um repetido seja gerado.
    - em média foram geradas 500 números até que um fosse repetido
- Gerando o noise de 1920x1080



4.2.2 - Tabela de números aleatórios

```
Table of random Numbers
--------------------------------------------------------------------------------
| 88836947 | 41713522 | 14015176 | 23051583 | 99236788 | 79400925 | 00468908 | 09782747 | 94371388 | 59588730
| 79567430 | 97799168 | 28468614 | 85035830 | 85423837 | 95543278 | 03767709 | 54464311 | 80577727 | 80412884
| 17615131 | 43670401 | 31189235 | 60803798 | 78698512 | 70877910 | 93121259 | 48710777 | 87055959 | 95217974
| 05817726 | 27443358 | 35818983 | 01513431 | 67592733 | 09891544 | 93866427 | 28753177 | 42337875 | 58206595
| 06747014 | 37305979 | 24982691 | 71976496 | 84815764 | 73114229 | 50552822 | 80046121 | 13530871 | 44846700
| 66065008 | 17116820 | 69015269 | 38827551 | 08439166 | 23807227 | 58098574 | 20219008 | 67378845 | 79637535
| 01479808 | 19978317 | 09763501 | 66297517 | 59253603 | 40486684 | 47579813 | 50710051 | 99580724 | 40493923
| 51729999 | 79763965 | 34621125 | 04472962 | 79157890 | 22475492 | 20173406 | 29555096 | 05788995 | 76234631
| 02023637 | 93747067 | 86515711 | 57938498 | 99683504 | 05097697 | 49321147 | 98161413 | 14232021 | 74848217
| 22707880 | 38478140 | 37312578 | 30396770 | 55396264 | 81108651 | 22494670 | 80401784 | 83428703 | 20138842
```

5 - Considerações Finais

Conforme verificado no tópico de resultados, o algoritmo que propusemos é mais eficiente que o dos quadrados médios, contudo, não é exatamente determinístico, já que ele depende do momento em que foi executado. Claro que, se soubermos a timestamp de uma execução, poderíamos tentar prever quais foram suas saídas, contudo, a mesma função pode ter sido chamada mais de uma vez em um único timestamp, o que não haveria como supor nesta revisão de suas saídas.

Pela sua implementação extremamente simples, seria possível alterar o código para que ele se tornasse mais determinístico, por exemplo, setando a seed manualmente com uma função de seed, que poderia ser o timestamp, e então remover de sua execução a inclusão de um novo timestamp a cada execução.

Contudo, por este não ser um código pensado para ser o melhor gerador de números aleatórios existente, existem algumas falhas nele. Por exemplo: Ele gera números astronômicos, na casa de 1000000 * 10^20. Portanto, caso um dia esta proposta seja revisitada, é aconselhável que uma das alterações seja uma solução para esta questão.

Quanto à implementação do método dos quadrados médios, a única coisa que podemos mudar para tentar melhorá-lo é a escolha da semente original. Como mencionado na seção de pesquisa deste relatório, a aleatoriedade da sequência e o seu período dependem apenas da semente inicial. Portanto, para gerarmos uma maior quantidade de número aleatórios, devemos ter cuidado no momento de escolhermos a semente original.

Escolhemos o timestamp pois este corresponde ao número de segundos desde a meia-noite do dia 01/01/1970 no fuso horário UTC sem considerar os segundos bissextos, dessa forma toda vez que compilamos o código temos uma semente diferente e única.

Comparado a sementes mais simples e de menos dígitos, o timestamp gera uma grande quantidade de números pseudo aleatórios antes da sequência voltar a se repetir. Na verdade, com este tipo de semente o que realmente acontece é que as sementes vão se tornando cada vez menores até que atinja o valor zero. Pois da forma como implementamos, se a semente atual tem menos dígitos que a original, completamos os dígitos que faltam com zeros à esquerda.

6 - Referências

CRUISE, Brit. **Khan Academy**, 2017. Geradores de números pseudoaleatórios. Disponível em: https://www.youtube.com/watch?v=f4sE1r3UL4E&ab_channel=KhanAcademyBrasil. Acesso em 13 jul. 2021

ANDRADE, Domingas; LEONARDO, Gabriel Moraes; FILIZZOLA, Renato Italo. **Avaliação de Desempenho (UFRJ)**, 2014. Geração de números pseudo-aleatórios uniformes. Disponível em:https://www.youtube.com/watch?v=f-TPFzSRa0A&ab_channel=Avalia%C3%A7%C3%A3odeDesempenho%28UFRJ%29. Acesso em 13 jul. 2021

MIRANDA, Paulo A. V.; **USP**, 2014. Números pseudo-aleatórios http://www.vision.ime.usp.br/~pmiranda/mac110_1s14/EPs/ep01/numeros_aleatorios.html . Acesso em 13 jul. 2021

SANTOS, Sandra Augusta. Geração de Números Aleatórios. Campinas: Unicamp, 2016. 23 slides, color. Disponível em: https://www.ime.unicamp.br/~sandra/MS614/handouts/GeracaoNumerosAleatorios.pdf. Acesso em: 13 jul. 2021.

STAFUSA, Vitor. **stackoverflow**, 2015. O que é o timestamp do unix? Disponível em: https://pt.stackoverflow.com/questions/70473/como-%C3%A9-feito-o-c%C3%A1lculo-do-timestamp. Acesso em 31 jul. 2021