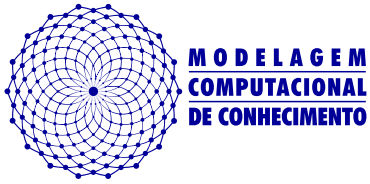

Dissertação de Mestrado

Geradores de Números Aleatórios, um estudo comparativo utilizando a linguagem R

Marcelo Queiroz de Assis Oliveira
marceloqao@gmail.com

Orientadores:

Dr. Alejandro C. Frery
Dr. Heitor Soares Ramos Filho

Maceió, Março de 2015

Marcelo Queiroz de Assis Oliveira

Geradores de Números Aleatórios, um estudo comparativo utilizando a linguagem R

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-Graduação em Modelagem Computacional de Conhecimento do Instituto de Computação da Universidade Federal de Alagoas.

Orientadores:
Dr. Alejandro C. Frery

Dr. Heitor Soares Ramos Filho

Maceió, Março de 2015

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecária: Fabiana Camargo dos Santos

T693n Oliveira, Marcelo Queiroz de Assis.

Geradores de Números Aleatórios, um estudo comparativo utilizando a linguagem R / Marcelo Queiroz de Assis Oliveira. – 2015
73 f. : il.

Orientadores: Alejandro C. Frery.
Heitor Soares Ramos Filho.
Dissertação (Mestrado em Modelagem Computacional de Conhecimento) – Universidade Federal de Alagoas. Instituto de Computação. Maceió, 2015.

Bibliografia: f. 69–74.

1. Geradores de Números Aleatórios. 2. Testes Teóricos. 3. Testes Estatísticos.

CDU: 004.932:004.852 – VERIFICAR –

Membros da Comissão Julgadora da Dissertação de Mestrado de Marcelo Queiroz de Assis Oliveira, intitulada “Geradores de Números Aleatórios, um estudo comparativo utilizando a linguagem R”, apresentada ao Programa de Pós-Graduação em Modelagem Computacional de Conhecimento da Universidade Federal de Alagoas em 30 de marco de 2015, às 10h00min, na sala de aula do Mestrado em Modelagem Computacional de Conhecimento. Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Curso de Mestrado em Modelagem Computacional de Conhecimento do Instituto de Computação da Universidade Federal de Alagoas, aprovada pela comissão examinadora que abaixo assina.

Dr. Alejandro C. Frery – Orientador
Instituto de Computação
Universidade Federal de Alagoas

Dr. Heitor Soares Ramos Filho – Orientador
Instituto de Computação
Universidade Federal de Alagoas

Dr. Osvaldo Anibal Rosso – Examinador
Pesquisador Visitante – Instituto de Computação
Universidade Federal de Alagoas

Dr. Raydonal Ospina Martínez – Examinador
Departamento de Estatística
Universidade Federal de Pernambuco

Maceió, Março de 2015

RESUMO

Este trabalho busca comparar alguns geradores de números aleatórios bem conhecidos pela comunidade acadêmica. Geradores de números aleatórios são amplamente utilizados em algoritmos probabilísticos como "Simulated Annealing", Algoritmos Genéticos e GRASP, assumindo assim um importante papel na utilização destes. Primeiramente falaremos sobre o histórico dos geradores de números aleatórios e de geradores em ambientes paralelos. Em seguida faremos a avaliação de alguns geradores segundo as abordagens clássica e numa segunda abordagem baseada na teoria da informação, procurando, desta forma, uma melhoria nos geradores de números pseudoaleatórios. Na sequência, discutiremos a avaliação paralela dos geradores utilizando três geradores disponíveis no R através dos trinta e um testes disponíveis no pacote "Dieharder", além de realizar um teste baseado da teoria da informação, encontrar regiões de confiança no plano (H,C) (Entropia x Complexidade) utilizando como dados de entrada as sequências aleatórias oriundas dos geradores testados. Como contribuição, proporemos uma ferramenta web para realizar os testes de sequências aleatórias.
Palavras-chave: Geradores de Números Aleatórios. Testes Teóricos. Testes Estatísticos.

ABSTRACT

This work aims to compare some Random Numbers Generators well known by the Scientific Community. Random Numbers Generators (RNG's) are used in probabilistic algorithms, such the Simulated Annealing, Genetics algorithms and GRASP. Also, the use of a robust RNG method is key factor to the success of a simulation. To introduce, some theoretic concepts about generators will be showed **Keywords**: Random Number Generators. Theoretical tests.

Statistical Tests.

AGRADECIMENTOS

Marcelo Queiroz de A. Oliveira

LISTA DE FIGURAS

LISTA DE TABELAS

Lista de Equações

SUMÁRIO

1

Introdução

1.1 Definição do Problema

1.2 Revisão Bibliográfica

1.3 Contribuições

1.4 Riscos

Neste capítulo tratamos de aspectos introdutórios do trabalho como a Definição do Problema, uma Revisão Bibliográfica, Contribuições e Riscos envolvidos no mesmo, no próximo capítulo trataremos da metodologia utilizada do desenvolvimento do mesmo.

2

Fundamentação

ESTE capítulo tem como objetivo apresentar a revisão bibliográfica necessária para a realização deste trabalho.

2.1 Quão bons são os geradores de Números Pseudoaleatórios Disponíveis no R ?

No R há alguns geradores de números pseudoaleatórios bem como um pacote de testes, conhecido como Dieharder, para aferir a sua qualidade. Há por outro lado, uma metodologia para análise de séries temporais baseada em ferramentas da teoria da informação que se mostrou útil para avaliar geradores de números pseudoaleatórios. A proposta é avaliar os geradores disponíveis no R comparando os testes disponíveis no Dieharder com o teste baseado em ferramentas da teoria da informação.

2.2 O que são PRNGs

2.2.1 Geradores de Números Aleatórios?

A necessidade de números aleatórios e pseudoaleatórios surge em inúmeras aplicações, como aborda **?** :

- *Simulação* - Quando um computador é usado para simular fenômenos naturais, números aleatórios são necessários para fazer as coisas de forma realística. Simulação abrange diversas áreas, desde o estudo de física nuclear (onde particulas são submetidas a colisões aleatórias) até pesquisa operacional (como, por exemplo, a taxa de pessoas que entram num aeroporto em intervalos aleatórios).

- *Amostragem* - É praticamente impossível examinar todos os possíveis casos, porém uma amostra aleatória provê um palpite sobre como é o comportamento típico do fenômeno em questão.
- *Análise Numérica* - Técnicas elaboradas para a solução de complexos problemas numéricos foram desenvolvidas utilizando-se números aleatórios.
- *Programação de Computadores* - Valores aleatórios são uma ótima fonte de dados para testar a eficácia de algoritimos computacionais.
- *Tomada de Decisão* - Existem relatos de que diversos executivos tomam suas decisões lançando moedas ou atirando dardos. Também há rumores de que professores universitários lançam suas notas de forma similar. Em alguns momentos é importante tomar decisões de forma não influenciada por qualquer agente externo.
- *Criptografia* - Uma fonte de bits não viesada é essencial para diversos tipos de comunicações seguras, quando os dados precisam ser mantidos em sigilo.
- *Estética* - Um pouco de aleatoriedade faz com que gráficos e músicas geradas por computador aparentem ser menos artificiais.
- *Diversão* - Rolar dados, embaralhar cartas, girar rodas de roletas, etc., são passatempos facinantes para muitos. Estes usos tradicionais de números aleatórios sugeriram o nome "Método de Monte Carlo".

Existem dois tipos básicos de geradores utilizados para produzir sequências aleatórias: geradores de números aleatórios (RNGs) e geradores de números pseudoaleatórios (PRNGs). Ambos podem produzir um fluxo de zeros e uns, que podem ser divididos em blocos ou subcorrentes de números aleatórios. Uma seqüência de bits aleatórios poderia ser interpretada como o resultado dos lançamentos de uma moeda justa, imparcial com os lados que são identificados como "0" e "1", onde cada caso tem uma probabilidade de exatamente ½. Além disso, o resultado de qualquer lançamento anterior não influencia futuros lançamentos. A moeda imparcial justa é, portanto, o gerador de fluxo de bits aleatórios perfeito. Todos os elementos da sequência são gerados independentemente uns dos outros, e o valor do elemento seguinte na sequência não pode ser previsto, independentemente do número de elementos que já foram produzidos. Obviamente, a utilização de moedas imparciais para fins de criptografia, modelagem computacional ou simulação é impraticável. No entanto, a saída hipotética de um gerador de uma seqüência realmente aleatória serve como um ponto de referência para a avaliação dos geradores de números aleatórios e pseudoaleatórios.

Números aleatórios e pseudoaleatórios gerados para quaisquer aplicações devem ser imprevisíveis. No caso de PRNG, se a semente é desconhecida, o próximo número na sequência de saída deve ser imprevisível, apesar de qualquer conhecimento de números aleatórios

anteriores na sequência. Esta propriedade é conhecida como imprevisibilidade para a frente. Também não deve ser viável para determinar a semente conhecendo os valores gerados (ou seja, a imprevisibilidade para trás também é obrigatória). Nenhuma correlação entre uma semente e qualquer valor gerado a partir da mesma devem ser evidentes; cada elemento da sequência deve parecer ser o resultado de um evento aleatório independente, cuja probabilidade é de 1/2. Para garantir a imprevisibilidade para a frente, o cuidado deve ser exercido na obtenção de sementes. Os valores produzidos por um PRNG são completamente previsíveis se a semente e algoritmo de geração são conhecidos. Uma vez que em muitos casos

2.2.2 Propriedades desejadas de um gerador ideal

Um gerador de números pseudoaleatórios deve possuir algumas propriedades que garantam sua qualidade para um leigo a construção de um gerador de números aleatórios pode parecer uma tarefa simples e alguns programadores tem demonstrado ser relativamente fácil escrever programas que gerem sequências de números aparentemente imprevisíveis. Entretanto, é bastante complexo escrever um bom programa que gere sequências satisfatórias, ou seja, uma sequência virtualmete infinita de números aleatórios estatisticamente independentes entre 0 e 1. Pois sequências aparentemente imprevisíveis não são necessariamente aleatórios

- *D. H. Lehmer (1951)*: "Uma sequência aleatória é uma vaga noção baseada na ideia de uma sequência onde cada termo é imprevisível e cujos dígitos passam em um certo número de testes, tradicionais com estatísticas ou dependendo do uso no qual a sequência será utilizada."

- *J. N. Franklin (1962)*: "A sequência é aleatória se possuir todas as propriedades compartilhadas por todas as infinitas sequências de amostras independentes de variáveis aleatórias sobre a distribuição uniforme.

A afirmação de Franklin essencialmente generaliza a de Lehmer ao dizer que a sequência precisa satisfazer *todos* os testes estatísticos. Esta definição não é precisa e uma interpretação sensata leva-nos a concluir que uma sequência aleatória simplesmente não existe! Portanto, iniciemos com a primeira e menos restritiva afirmação de Lehmer e tentemos torná-la mais precisa. O que realmente queremos é uma pequena lista de propriedades matemáticas, cada uma delas satisfeita por nossas noções intuitivas de uma sequência aleatória; além disso, a lista precisa ser completa o bastante para que qualquer sequência que a satisfaça possa ser considerada aleatória **??**.

2.2.3 Histórico

2.2.4 Geradores disponíveis no R

- a
- b
- c

2.3 Verificação clássica das propriedades dos geradores

2.3.1 Repetibilidade, portabilidade, eficiência computacional

2.3.2 Testes

Seja $\boldsymbol{X} = (X_1, X_n)$ uma amostra a respeito da qual temos uma conjectura que queremos verificar. Essa conjectura é a respeito dos parâmetros que caracterizam a distribuição da amostra, ou a respeito de parâmetros que caracterizam a distribuição de atributos relacionados à distribuição da amostra. Chamaremos "hipótese nula" àquela que convimos em não rejeitar a não ser que obtenhamos suficiente evidência para isso; a denotaremos H_0. Por vezes precisaremos da "hipótese alternativa", que denotaremos H_1.

Classicamente, um teste de hipótese se baseia em uma estatística de teste T que depende exclusivamente da amostra $\boldsymbol{X}$, isto é, $T(\boldsymbol{X})$, e é construída de tal forma que adota valores "pequenos" sob H_0 e "cresce" conforme "se afasta" de H_0. Idealmente, conhecemos a distribuição de T sob a hipótese nula e, com isso, somos capazes de aferir a probabilidade de observarmos valores "grandes" mesmo sob H_0. Definimos, assim, o p-valor do teste baseado em $T(\boldsymbol{X})$ para o valor observado η como $\Pr_{H_0}(T(\boldsymbol{X}) \geq \eta)$. O procedimento básico consiste em rejeitar a hipótese nula ao nível de significância $100(1-\alpha)\%$ se o p-valor for inferior a α, e em não rejeitá-la caso contrário. Mais modernamente, não se fala em "rejeição", reporta-se o p-valor, deixando a decisão para o leitor. Desta forma, chamaremos de "valor crítico" o conjunto de valores, definidos pelo leitor tais que, quando assumidos pela estatística de teste T levam à rejeição da hipótese, nula H_0.

Diversos testes foram desenvolvidos ao longo do tempo com o objetivo de testar a aleatoriedade de sequências de números. Como "aleatoriedade" é uma noção vaga, cada teste procura identificar uma ou algumas falhas dos dados. Por esse motivo é que há classes ou conjuntos de testes que, aplicados de forma criteriosa, permitem aferir o quanto uma sequência se afasta da hipótese de aleatoriedade. A partir dessa análise, pode se ter uma idea do comportamento global do gerador que a produziu.

Sob a conjectura de aleatoriedade, cada estatística de teste T tem uma distribuição, que pode ser determinada ou de forma exata ou de forma aproximada, mas sempre precisa ser

conhecida. Com ela é possível informar o p-valor de uma determinada amostra.

2.3.3 Testes anterior

2.3.4 Testes Diehard

2.3.5 Testes NIST

Fundado em 1991, o NIST (Instituto Nacional de Padrões e Tecnologia) é uma agência não regulatória do Departamento de Comércio dos Estados Unidos da América (EUA) que tem por missão promover a inovação e a competitividade nos EUA através da ciência de medidas, padrões e tecnologia de forma a alavancar a segurança econômica e melhorar a qualidade de vida do povo americano. A Divisão de Segurança de Computadores (CSD) e o Centro de Pesquisa em Segurança Computacional (CSRC) facilitam a ampla disseminação de práticas e ferramentas de segurança da informação, provendo recursos para a definição de padrões além de identificar recursos de segurança na web para suportar usuários na industria, governo e academia. CSRC é o portal de acesso primário para se ter acesso às publicações de segurança de computadores, padrões e instruções, além de outras informações relacionadas a segurança. Desde 1997, o Grupo de Trabalho Técnico em Geração de Números Aleatórios (RNG-TWG) tem trabalhado no desenvolvimento de uma bateria de testes estatísticos apropriados para a avaliação de geradores de números aleatórios e pseudoaleatórios utilizados em aplicações criptográficas. Os principais objetivos do grupo são:

- Desenvolvimento de uma bateria de testes estatísticos para detectar não aleatoriedade em sequencias binárias construidas através de geradores de números aleatórios e pseudoaleatórios utilizados em aplicações criptográficas;
- Produzir documentação e uma implementação em software destes testes;
- Prover auxílio no uso e aplicação destes testes.

Um total de quinze testes estatísticos foram desenvolvidos, implementados e avaliados. A seguir fazemos um breve descritivo acerca dos testes.

- **Frequency (Monobits) Test** - O foco do teste está na proporção de $zeros$ e uns em toda a sequência. O propósito deste teste é determinar em que momento o número de $zeros$ e uns numa sequência é aproximadamente o mesmo como é o esperado para uma sequência realmente aleatória. O teste avalia a proximidade da porção de uns a ¡, ou seja, o número de $zeros$ e uns numa sequência deve ser o mesmo.
- **Test For Frequency Within A Block** - O teste avalia a proporção de $zeros$ e uns em blocos de M bits. O propósito principal deste teste é determinar se a frequência de uns em um bloco de M bits é aproximadamente $M/2$.

- **Runs Test** - Dada uma sequência ininterrupta de zeros ou de uns, chamaremos esta sequência de “rodada” e de “k” tamanho da sequência. O objetivo do teste é determinar a proporção de rodadas de zeros e uns de diversos tamanhos “k” necessária para caracterizar uma sequência com aleatória. Em particular, este teste determina quando a oscilação enter estas substrings é muito rápida ou muito lenta.

- **Test For The Longest Run Of Ones In A Block** - O teste tem como objetivo determinar quando o comprimento da maior rodada de uns na sequência testada é equivalente ao tamanho da maior rodada esperada em uma sequência aleatória. Note que uma irregularidade no tamanho esperado no valor da maior rodada de uns implica que também há uma irregularidade no tamanho esperado da rodada de $zeros$ mais longa.

- **Random Binary Matrix Rank Test** - O teste avalia a categoria das submatrizes disjuntas da sequência toda. O objetivo do teste é verificar a ocorrência de dependência linear entre substrings de tamanho fixo da sequência original.

- **Discrete Fourier Transform (Spectral) Test** - O teste foca nos picos da `Transformada Rápida de Fourrier` com o propósito de detectar característica periódicas.

- **Non-Overlapping (Aperiodic) Template Matching Test** - O teste observa o número de ocorrências de substrings predefinidas com o propósito de rejeitar sequências que possuam muitas ocorrências de um dado padrão aperiódico. Para tal uma janela de m bits é usada para procurar por um padrão específico de "m"bits.

- **Overlapping (Periodic) Template Matching Test** - O teste, assim com o o anterior, procura por substrings predefinidas de rodadas de uns com um dado tamanho. O objetivo do teste é rejeitar sequências que mostrem um desvio do número esperado de uns de um dado tamanho. Note que, quando há um desvio no número de uns, este desvio também ocorre com o número de $zeros$. Da mesma forma que o teste anterior, uma janela de m bits é usada para detectar a presença de padrões.

- **Maurer’s Universal Statistical Test** - O teste observa o número de bits entre padrões casados com o objetivo de detectar ou não se a sequência pode ser significativamente comprimida sem a perda de informação. Uma sequência que pode ser totalmente comprimida é considerada como não aleatória.

- **Linear Complexity Test** - O teste focaliza no tamanho de um registro gerador de informação com o objetivo de determinar quando ou não a sequência é complexa o suficiente para ser considerada aleatória. Sequências aleatórias são caracterizadas por um registro gerador de informação longo.

- **Serial Test** - O teste detem o foco sobre a frequência de cada bit e de todos os padrões sobrepostos de m bits em toda a sequência, com o propósito de determinar quando o

número de ocorrências de padrões sobrepostos de 2m m bits é o mesmo do esperado para sequências aleatórias, sabendo que o padrão pode ser sobreposto.

- **Approximate Entropy Test** - O teste se detém na frequência de cada um dos padrões sobrepostos de m bits com o objetivo de comparar a frequência de dois blocos sobrepostos de tamanhos consecutivos/adjacentes (m e $m+1$) com os valores esperados para uma sequência aleatória.

- **Cumulative Sum (Cusum) Test** - O teste verifica a distância máxima do "passeio aleatório" definido pela soma dos dígitos ajustados (-1, $+1$) na sequência com o objetivo de determinar quando a soma cumulativa da sequência parcial que ocorre nas sequências testadas é muito grande ou muito pequeno em relação ao comportamento esperado para sequências aleatórias. Esta sequência cumulativa pode ser considerada como um "passeio aleatório". Para uma sequência aleatória, o "passeio aleatório" deve ser próximo a $zero$ e para sequências não aleatórias a excursão máxima do "passeio aleatório" se distancia de zero e tende a ser grande.

- **Random Excursions Test** - O teste verifica o número de ciclos que possuem exatamente K visitas em uma soma cumulativa do tipo "passeio aleatório". A soma cumulativa é encontrada se a soma parcial das sequências de (0, 1) está ajustada a (-1, $+1$). Uma "jornada aleatória" de um "passeio aleatório" consiste em uma sequência de n passos de tamanho unitário tomados aleatoriamente iniciando e terminando na origem. O objetivo deste teste é determinar se o número de visitas a um estado em um "passeio aleatório" excede o esperado para uma sequência aleatória.

- **Random Excursions Variant Test** - O teste verifica o número de vezes que um estado em particular ocorreu na soma cumulativa de um "passeio aleatório" com o objetivo de detectar desvios do número esperado de ocorrências de vários estados do "passeio aleatório".

2.3.6 Testes Dieharder

Como substituta à suíte de testes anteriores, RGB reescreveu-os numa linguagem portável como C e acrescentou ao conjunto de testes disponíveis no Diehard alguns outros testes do NIST e mais alguns de sua autoria, listados na tabela 2.1 e explicados abaixo.

- **"Birthdays" test (modificado). Id(0)** - Este teste baseia-se no Paradoxo do Aniversário Cada teste determina o número de intervalos que combinam 512 "aniversários" (por padrão) tomados num "ano" fictício de 24 bits (por padrão). Este processo é repetido (por padrão) 100 vezes e o resultado é acumulado em um histograma. Intervalos repetidos podem ser distribuidos em uma distribuição Poisson se o gerador em questão

for aleatório o suficiente, e em uma Chi Quadrado com o p-valor avaliado relativamente à hipótese nula. É recomendado rodar este teste próximo ou com exatamente 100 amostras por p-valor com `-t 100`. Dois parametros adicionais foram incluidos. No Diehard, nms=512, porém isto pode ser variado e todas as fórmulas de Marsaglia continuam a funcionar. Pode ser ajustado para valores diferentes com `-x nmsvalue`. Similarmente, o parâmetro `nbits`pode ser 24, mas podemos fazê-lo assumir qualquer valor desde que seja menor ou igual a `rmax_bits = 32`. E pode ser atribuido qualquer valor com o parâmetro `-y nbits`. Ambos são padrão para os valores do Diehard se as opções `-x e -y` não forem utilizadas.

- **Diehard Overlapping 5-Permutations Test. Id(1)** - O teste procura por uma sequência aleatória de 10^6 inteiros. Cada conjunto de cinco inteiros consecutivos pode estar em um dos 120 estados, para as 5! possíveis combinações de cinco números. Portanto, cada um dos $5^o, 6^o, 7^o, \ldots$ fornecem um estado. Assim, como milhares de transições de estado são observadas, contadores cumulativos são gerados a partir do número de ocorrências de cada estado. Logo, a forma quadrática na inveresa fraca da matriz de covariância 120×120 produz um teste equivalente ao do teste da taxa de verossimilhança, onde as 120 células vêem da distribuição normal (assintoticamente) especificada com a específica matriz de covariância 120×120 (com rank 99). Esta versão usa 1.000.000 de inteiros, duas vezes. Note que o Dieharder executa o teste 100 vezes por padrão e não 2.

- **Diehard 32x32 Binary Rank Test. Id(2)** - Neste teste uma matriz binária 32×32 é formada com cada linha contendo um inteiro aleatório de 32 bits. O rank é determinado e pode assumir valores de 0 a 32, ranks menores de 29 são raros e suas ocorrências são acumuladas no rank 29. Os ranks são calculados para as 40.000 matrizes e um teste chi quadrado é realizado nas contagens para os ranks $32, 31, 30 e \leqslant 29$. Como de costume na bateria de testes, o teste é repetido e um teste KS (Kolmogorov Smirnov) é aplicado aos $p - valores$ obtidos afim de verificar se eles são uniformes.

- **Diehard 6x8 Binary Rank Test. Id(3)** - Neste teste cada uma dos $seis$ inteiros aleatórios de 32 bits do gerador submetido ao teste tem um $byte$ escolhido e os $seis$ bits resultantes formam uma matriz binária 6×8 cujo rank é determindado. Este rank pode ser de 0 a 6, porém ranks $0, 1, 2, 3$ são raros e seus valores são armazendados juntamente daqueles de rank 4. São encontrados os ranks para 100.000 matrizes aleatórias e um testes chi quadrado é aplicado aos contadores para os ranks $6, 5 e \leqslant 4$. Como sempre, o teste é repetido e um teste KS é aplicado ao $p - valor$ resultante para verificar se eles são uniformmes.

- **Diehard Bitstream Test. Id(4)** - O arquivo sob teste é visto como um fluxo de bits. Chamaremo-os $b_1, b_2, \ldots$. Considere um alfabeto de duas "letras", 0 e 1 e pense no fluxo de bits como uma sucessão de "palavras" de 20 letras com sobreposição. Então, a

primeira palavra é $b_1\ b_2 \ldots\ b_{20}$, o segundo é $b_2\ b_3 \ldots\ b_{21}$ e assim por diante. O teste conta o número de palavras de 20 letras ou 20 bits ausentes numa string de $2^{(21)}$ sobrepondo palavras de 21 letras. Dado que existem 2^{20} possíveis palavras de 20 letras, para uma string realmente aleatória de $2^{21}+19$ bits, o número de palavras ausentes "j" deve ser (ou bem próximo de) normalmente distribuido com média 141.909 e $\sigma = 428$. Logo, $(j - 141909)/428$ deve ser a variação (z score) que nos leva a um $(p - value)$ uniforme [0.1). O teste é repetido vinte vezes. Note que o teste é repetido 100 vezes por padrão no $dieharder$, mas o tamanho da amostra é fixo ($t - samples$ não devem/podem ser modificados do padrão), neste teste σ requer o uso de amostras sobrepostas, e tais amostras não são independentes. Caso queira usar a versão sem sobreposição do teste, $\sigma = 290$ precisa ser utilizado.

- **Diehard Overlapping Pairs Sparse Occupance (OPSO). Id (5)** - O teste considera palavras de 2 letras em um alfabeto de 1024 letras, cada letra é determinada por 10 bits específicos em uma sequência inteira de 32 bits a ser testada. O mesmo gera 2^{21} palavras (sobrepostas) de 2 bits (de um total de $2^{21}+1$ "teclas pressionadas" e conta o número de palavras ausentes, ou seja, palavras de 2 letras que não aparecem em toda a sequência. Esta contagem precisa ser muito próxima de normalmente distribuida com média 141909, σ290. Logo $(ausentes - 141909)/290$ deve ser a variação padrão normal. O teste toma 32 bits do arquivo a ser testado por vez e usa um conjunto específico de dez bits consecutivos. Então reinicia o arquivo para os próximos dez bits e assim por diante. Note que $2^{21} = 2097152$, $tsamples$ não podem ser variados.

- **Diehard Overlapping Quadruples Sparce Occupancy (OQSO) Test. Id(6)** - Similarmente ao teste anterior, exceto pelo fato de considerar palavras de 4 letras em um alfabeto de 32 letras, cada letra determinada por uma determinada string de 5 bits consecutivos do arquivo testado. O número médio de palavras ausentes numa sequência de 2^{21} palavras de 4 letras, ($2^{21}+3$ "teclas pressionadas") é novamente 141909, com $\sigma = 295$. A média é baseada na teoria, σ vem de exaustiva simulação. Note que $2^{21} = 2097152$, tsamples não podem ser variados.

- **Diehard DNA Test. Id(7)** - O teste considera um alfabeto de 4 letras `C`, `G`, `A`, `T`, determinadas por dois bits significativos na sequência de inteiros aleatórios que está sendo testada. Avaliando palavras de 10 letras, assim como no OPSO e OQSO, existem 2^{20} possíveis palavras, e o número médio de palavras faltantes em uma sequência de 2^{21} sobrepondo palavras de 10 letras ($2^{21}+9$ "teclas pressionadas") é 141909. O desvio padrão $\sigma = 339$ foi determinado assim como para OQSO, por simulação.

- **Diehard Count the 1s (stream) (modified) Test. Id(8)** - Neste teste o arquivo avaliado é tratado como um fluxo de bytes (quatro para cada inteiro de 32 bits). Cada byte pode conter de $0a8$ uns, Com probabilidades $1, 8, 28, 56, 70, 56, 28, 8, 1 sobre 256$. Desta forma,

deixemos que o fluxo de bytes forneça uma string de 5 palavras com sobreposição, onde cada "letra" assume os valores A, B, C, D, E. As letras são determinadas pela quantidades de "uns" em um byte: 0, 1 ou 2 representam A, 3 representa B, 4 representa C, 5 representa D e 6, 7 ou 8 representam E. Desta forma, temos 5 teclas sendo pressionadas com diversas probabilidades (37,56,70,56,37$sobre$256). Existem 5^5 palavras de 5 letras, e para uma sequência de palavras de 5 letras de tamanho 256.000 com sobreposição, a contagem é realizada a partir da frequência com a qual cada palavra ocorre. A forma quadrática da contagem de células na matriz de covarância inversa fraca provê um teste chi quadrado: $Q5 - Q4$ a diferença das somas de Pearson de $(OBS - EXP)^2/EXP$ nas contagens para células de 5 letras e 4 letras.

- **Diehard Count the 1s Test (byte) (modified). Id(9)** - Este é o teste "Conte os uns" para bytes específicos. Considera o arquivo testado como um fluxo de inteiros de 32 bits. Para cada inteiro, um byte específico é escolhido, digamos que os bits de $1a8$ à esquerda. Como cada byte pode conter de $0a8$ uns, com probabilidades $1, 8, 28, 56, 70, 56, 28, 8, 1sobre256$. Então tomemos os bytes específicos dos sucessivos inteiros proverem uma sequência de palavras de 5 letras com sobreposição, onde cada letra podendo assumir os valores A, B, C, D, E. Sendo as letras determinadas pela quantidade de uns em um dado byte desta forma: $0, 1, ou2 \rightarrow A, 3 \rightarrow B, 4 \rightarrow C, 5 \rightarrow D, and6, 7ou8 \rightarrow E$. Desta forma, temos 5 teclas sendo pressionadas com diversas probabilidades (37,56,70,56,37$sobre$256). Existem 5^5 palavras de 5 letras, e para uma sequência de palavras de 5 letras de tamanho 256.000 com sobreposição, a contagem é realizada a partir da frequência com a qual cada palavra ocorre. A forma quadrática da contagem de células na matriz de covarância inversa fraca provê um teste chi quadrado: $Q5 - Q4$ a diferença das somas de Pearson de $(OBS - EXP)^2/EXP$ nas contagens para células de 5 letras e 4 letras. Nota: O teste é executado em rodadas sobre amostras em todo o espaço de $0 - 31$ bits, logo, se há algum problema com alguma sequência em particular, há chance deste problema ser detectado. Pode-se imaginar problemas com sequências ímpares e não com sequências pares. O parâmetro "tsamples" e "psamples" podem ser livremente variados, porém é sensato manter "tsamples" $\gg$ 100.000 para obter-se um resultado confiável num teste KS.

- **Diehard Parking Lot Test (modified). Id(10)** - Este teste verifica a distribuição das tentativas de estacionar um "carro quadrado" de tamanho 1 em um estacionamento que mede 100×100 sem causar nenhum acidente. O resultado é plotado em $n \times k$ onde n é o número de tentativas e k o números de tentativas que foram bem sucedidas, ou seja, não houve um acidente de se colocar um carro onde já havia um (sobreposição), comparado com o resultado esperado de um conjunto perfeitamente randômico de coordenadas de vagas. Isto não é realmente conhecido no campo teórico, então ao invés de comparar com $n = 12.000$ onde k deve ter média 3523 com $\sigma = 21.9$ e é muito próxima

da normal. Então, $(k-3523)/21.9$ é uma variável normal padrão, a qual convertida a um $p-valor$ uniforme, provê valores para um teste KS com 100 amostras padrão.

- **Diehard Minimum Distance (2d Circle) Test Id(11)** - O teste repete 100 vezes: escolhe $n = 8.000$ pontos aleatórios em um quadrado de lados 10.000. Encontra d, a distância mínima entre os $(n^2-n)/2$ pares de pontos. Se os pontos são realmente independentes e uniformes, então d^2, o quadrado da distância mínima deve ser distribuido exponencialmente (ou muito próximo disto) com média 0.995. Então $1-exp(-d^2/0.995)$ deve ser uniforme em [0,1) e um teste KS nos 100 valores resultantes serve como um teste de uniformidade para pontos aleatórios no quadrado. $numbers = 0 mod 5$ são apresentados, porém o teste KS é baseado no conjunto completo de 100 escolhas dentre os 8.000 pontos no quadrado de 10.000×10.000. Este teste usa um número fixo de amostras (tsamples é ignorado). Ele também usa o valor padrão de 100 psamples no teste KS final.

- **Diehard 3d Sphere (Minimum Distance) Test. Id(12)** - Neste teste são escolhidos 4.000 pontos em um cubo de aresta 1.000. Em cada ponto, centralize uma esfera grande o suficiente para alcançar o ponto mais próximo. Então, o volume da menor esfera é exponencialmete distribuido com média $120\pi/3$. E o raio ao cubo é exponencial com média 30. (A média é obtida através de exaustivas simulações). O teste gera 4.000 esferas repetidas por 20 vezes. Cada raio mínimo inscrito no cubo leva-nos a uma variável aleatória uniforme com média $1-exp(-r^3/30)$, então é realizado um teste KS nos 20 p-valores. Este teste ignora "tsamples" e roda o padrão de 100 psamples para usar no teste KS final.

- **Diehard Squeeze Test. Id(13)** - Random integers are floated to get uniforms on [0,1). Starting with $k = 2^{31}$=2147483647, the test finds j, the number of iterations necessary to reduce k to 1, using the reduction $k = ceiling(k \times U)$, with U provided by floating integers from the file being tested. Such j's are found 100.000 times, then counts for the number of times j was $\leq 6, 7, \ldots, 47, \geq 48$ are used to provide a chisquare test for cell frequencies.

- **Diehard Sums Test Id(14)** - Integers are floated to get a sequence $U(1)$,$U(2)$,... of uniform [0,1) variables. Then overlapping sums, $S(1) = U(1) + \cdots + U(100)$, $S2 = U(2) + \cdots + U(101)$,... are formed. The S's are virtually normal with a certain covariance matrix. A linear transformation of the S's converts them to a sequence of independent standard normals, which are converted to uniform variables for a KSTEST. The pvalues from ten KSTESTs are given still another KSTEST. Comments At this point I think there is rock solid evidence that this test is completely useless in every sense of the word. It is broken, and it is so broken that there is no point in trying to fix it. The problem is that the transformation above is not linear, and doesn't work. Don't use it. For what it

is worth, rgb_lagged_sums with ntuple 0 tests for exactly the same thing, but scalably and reliably without the complication of overlapping samples and covariance. Use it instead.

- **Diehard Runs Test. Id(15)** - This is the RUNS test. It counts runs up, and runs down, in a sequence of uniform [0,1) variables, obtained by floating the 32-bit integers in the specified file. This example shows how runs are counted: .123,.357,.789,.425,.224,.416,.95 contains an up-run of length 3, a down-run of length 2 and an up-run of (at least) 2, depending on the next values. The covariance matrices for the runs-up and runs-down are well known, leading to chisquare tests for quadratic forms in the weak inverses of the covariance matrices. Runs are counted for sequences of length 10.000. This is done ten times. Then repeated. In Dieharder sequences of length tsamples = 100000 are used by default, and 100 p-values thus generated are used in a final KS test.

- **Diehard Craps Test. Id(16)** - This is the CRAPS TEST. It plays 200.000 games of craps, finds the number of wins and the number of throws necessary to end each game. The number of wins should be (very close to) a normal with mean 200000p and variance 200000p(1-p), with p=244/495. Throws necessary to complete the game can vary from 1 to infinity, but counts for all > 21 are lumped with 21. A chi-square test is made on the number-of-throws cell counts. Each 32-bit integer from the test file provides the value for the throw of a die, by floating to [0,1), multiplying by 6 and taking 1 plus the integer part of the result.

- **Marsaglia and Tsang GCD Test. Id(17)** - 10^7 tsamples (default) of uint rands u, v are generated and two statistics are generated: their greatest common divisor (GCD) (w) and the number of steps of Euclid's Method required to find it (k). Two tables of frequencies are thus generated - one for the number of times each value for k in the range 0 to 41 (with counts greater than this range lumped in with the endpoints). The other table is the frequency of occurrence of each GCD with k is be distributed approximately binomially, but this is useless for the purposes of performing a stringent test. Instead four "good" RNGs (gfsr4, mt19937_1999, rndlxs2, taus2) were used to construct a simulated table of high precision probabilities for k (a process that obviously begs the question as to whether or not THESE generators are "good" wrt the test). At any rate, they produce very similar tables and pass the test with each other's tables (and are otherwise very different RNGs). The table of probabilities for the gcd distribution is generated dynamically per test (it is easy to compute). Chisq tests on both of these binned distributions yield two pvalues per test, and 100 (default) pvalues of each are accumulated and subjected to final KS tests and displayed in a histogram.

- **STS Monobit Test. Id(100)** - Very simple. Counts the 1 bits in a long string of random uints. Compares to expected number, generates a p-value directly from erfc(). Very ef-

fective at revealing overtly weak generators; Not so good at determining where stronger ones eventually fail.

- **STS Runs Test. Id(101)** - Counts the total number of 0 runs + total number of 1 runs across a sample of bits. Note that a 0 run must begin with 10 and end with 01. Note that a 1 run must begin with 01 and end with a 10. This test, run on a bitstring with cyclic boundary conditions, is absolutely equivalent to just counting the 01 + 10 bit pairs. It is therefore totally redundant with but not as good as the rgb_bitdist() test for 2-tuples, which looks beyond the means to the moments, testing an entire histogram of 00, 01, 10, and 11 counts to see if it is binomially distributed with p = 0.25.

- **STS Serial Test. Id(102)** - Accumulates the frequencies of overlapping n-tuples of bits drawn from a source of random integers. The expected distribution of n-bit patterns is multinomial with p = $2^{(-n)}$ e.g. the four 2-bit patterns 00 01 10 11 should occur with equal probability. The target distribution is thus a simple chisq with $2^n - 1$ degrees of freedom, one lost due to the constraint that: p_00 + p_01 + p_01 + p_11 = 1 With overlap, though the test statistic is more complex. For example, given a bit string such as 0110100111000110 without overlap, it becomes 01|10|10|01|11|00|01|10 and we count 1 00, 3 01s, 3 10s, and 1 11. WITH overlap we get all of these patterns as well as (with cyclic wrap): 0|11|01|00|11|10|00|11|0 and we count 4 00s, 4 01s, 4 10s, and 3 11s. There is considerable covariance in the bit frequencies and a simple chisq test no longer suffices. The STS test uses target statistics that are valid for overlapping samples but which require multiple orders to generate. It is much easier to write a test that doesn't use overlapping samples and directly checks to ensure that the distribution of bit ntuples is consistent with a multinomial distribution with uniform probability $p = 1/2^n$, e.g. 1/8 for n = 3 bit, 1/16 for n = 4 bit NON-overlapping samples, and the rgb_bitdist is just such a test. This test doesn't require comparing different orders. An open research question is whether or not test sensitivity significantly depends on managing overlap testing software RNGs where it is presumed that generation is cheap and unlimited. This question pertains to related tests, such as overlapping permutations tests (where non-overlapping permutation tests are isomorphic to non-overlapping frequency tests, fairly obviously). This test does all the possible bitlevel tests from n=1 to n=24 bits (where n=1 is basically sts_monobit, and n=2 IMO is redundant with sts_runs). However, if I understand things correctly it is not possible to fail a 2 bit test and pass a 24 bit test, as if 2 bits are biased so that (say) 00 occurs a bit too often, then 24 bit strings containing 00's MUST be imbalanced as well relative to ones that do not, so we really only need to check n=24 bit results to get all the rest for free, so to speak.

- **RGB Bit Distribution Test. Id(200)** - Accumulates the frequencies of all n-tuples of bits in a list of random integers and compares the distribution thus generated with the

theoretical (binomial) histogram, forming chisq and the associated p-value. In this test n-tuples are selected without WITHOUT overlap (e.g. 01|10|10|01|11|00|01|10) so the samples are independent. Every other sample is offset modulus of the sample index and ntuple_max. This test must be run with -n ntuple for ntuple > 0. Note that if ntuple > 12, one should probably increase tsamples so that each of the 2^{ntuple} bins should end up with an average of around 30 occurrences. Note also that the memory requirements and CPU time requirements will get quite large by e.g. ntuple = 20 - use caution when sampling the distribution of very large ntuples.

- **THE GENERALIZED MINIMUM DISTANCE TEST. Id(201)** - This is the generalized minimum distance test, based on the paper of M. Fischler in the doc directory and private communications. This test utilizes correction terms that are essential in order for the test not to fail for large numbers of trials. It replaces both diehard_2dsphere.c and diehard_3dsphere.c, and generalizes the test itself so that it can be run for any d = 2,3,4,5. There is no fundamental obstacle to running it for d = 1 or d 5, but one would need to compute the expected overlap integrals (q) for the overlapping d-spheres in the higher dimensions. Note that in this test there is no real need to stick to the parameters of Marsaglia. The test by its nature has three controls: n (the number of points used to sample the minimum distance) which determines the granularity of the test - the approximate length scale probed for an excess of density; p, the usual number of trials; and d the dimension. As Fischler points out, to actually resolve problems with a generator that had areas 20% off the expected density (consistently) in $d = 2, n = 8000$ (Marsaglia's parameters) would require around 2500 trials, where p = 100 (the old test default) would resolve only consistent deviations of around 1.5 times the expected density. By making both of these user selectable parameters, dieharder should be able to test a generator pretty much as thoroughly as one likes subject to the generous constraints associated with the eventual need for still higher order corrections as n and p are made large enough.

- **RGB Permutations Test. Id(202)** - This is a non-overlapping test that simply counts order permutations of random numbers, pulled out n at a time. There are n! permutations and all are equally likely. The samples are independent, so one can do a simple chisq test on the count vector with n! - 1 degrees of freedom. This is a poor-man's version of the overlapping permutations tests, which are much more difficult because of the covariance of the overlapping samples.

- **RGB Lagged Sums Test. Id(203)** - This package contains many very lovely tests. Very few of them, however, test for lagged correlations – the possibility that the random number generator has a bitlevel correlation after some fixed number of intervening bits. The lagged sums test is therefore very simple. One simply adds up uniform deviates

sampled from the rng, skipping lag samples in between each rand used. The mean of tsamples samples thus summed should be 0,5*tsamples. The standard deviation should be sqrt(tsamples/12). The experimental values of the sum are thus converted into a p-value (using the erf()) and a ks-test applied to psamples of them.

- **The Kolmogorov-Smirnov Test Test. Id(204)** - This test generates a vector of tsamples uniform deviates from the selected rng, then applies an Anderson-Darling or Kuiper KS test to it to directly test for uniformity. The AD version has been symmetrized to correct for weak left bias for small sample sets; Kuiper is already ring-symmetric on the interval. The AD code corresponds roughly to what is in R (thanks to a correction sent in by David Bauer). As always, the test is run pvalues times and the (same) KS test is then used to generate a final test pvalue, but the real purpose of this test is to test ADKS and KKS, not to test rngs. This test clearly reveals that kstests run on only 100 test values (tsamples, herein) are only approximately accurate; their pvalues are distinctly high-biased (but less so than Kuiper or KS before the fix). This bias is hardly visible for less than 1000 trivals (psamples, herein) but will constently cause failure for -t 100, -p 10000 or higher. For -t 1000, it is much more difficult to detect, and the final kstest is approximately valid for the test in question.

- **DAB Byte Distribution Test. Id(205)** - Extract n independent bytes from each of k consecutive words. Increment indexed counters in each of n tables. (Total of 256 × n counters.) Currently, n=3 and is fixed at compile time. If n>=2, then the lowest and highest bytes will be used, along with n-2 bytes from the middle. If the generator's word size is too small, overlapped bytes will be used. Current, k=3 and is fixed at compile time. Use a basic chisq fitting test (chisq_pearson) for the p-value. Previous version also used a chisq independence test (chisq2d); it was found to be slightly less sensitive. I envisioned this test as using a small number of samples and large number of separate tests. Experiments so far show that keeping -p 1 and increasing -t performs best.

- **DCT (Frequency Analysis) Test. Id(206)** - This test performs a Discrete Cosine Transform (DCT) on the output of the RNG. More specifically, it performs tsamples transforms, each over an independent block of ntuple words. If tsamples is large enough, the positions of the maximum (absolute) value in each transform are recorded and subjected to a chisq test for uniformity/independence. [1] (A standard type II DCT is used.) If tsamples is smaller than or equal to 5 times ntuple then a fallback test will be used, whereby all DCT values are converted to p-values and tested for uniformity via a KS test. This version is significantly less sensitive, and is not recommended. Power: With the right parameters, this test catches more GSL generators than any other; however, that count is biased by each of the randomNNN generators having three copies. Limitations: ntuple is required to be a power of 2, because a radix 2 algorithm is used to calculate the

DCT. False positives: targets are (mostly) calculated exactly, however it will still return false positives when ntuple is small and tsamples is very large. For the default ntuple value of 256, I get bad scores with about 100 million or more tsamples (psamples set to 1). [1] The samples are taken as unsigned integers, and the DC coefficient is adjusted to compensate for this.

- **DAB Fill Tree Test. Id(207)** - This test fills small binary trees of fixed depth with words from the the RNG. When a word cannot be inserted into the tree, the current count of words in the tree is recorded, along with the position at which the word would have been inserted. The words from the RNG are rotated (in long cycles) to better detect RNGs that may bias only the high, middle, or low bytes. The test returns two p-values. The first is a Pearson chi-sq test against the expected values (which were estimated empirically). The second is a Pearson chi-sq test for a uniform distribution of the positions at which the insert failed. Because of the target data for the first p-value, ntuple must be kept at the default (32).

- **DAB Fill Tree 2 Test. Id(208)** - Bit version of Fill Tree test. This test fills small binary trees of fixed depth with "visited"markers. When a marker cannot be placed, the current count of markers in the tree and the position that the marker would have been inserted, if it hadn't already been marked. For each bit in the RNG input, the test takes a step right (for a zero) or left (for a one) in the tree. If the node hasn't been marked, it is marked, and the path restarts. Otherwise, the test continues with the next bit. The test returns two p-values. The first is a Pearson chi-sq test against the expected values (which were estimated empirically. The second is a Pearson chi-sq test for a uniform distribution of the positions at which the insert failed. Because of the target data for the first p-value, ntuple must be kept at the default (128).

- **DAB Monobit 2 Test. Id(209)** - Block-monobit test. Since we don't know what block size to use, try multiple block sizes. In particular, try all block sizes of 2^k words, where k=0 . . . n. The value of n is calculated from the word size of the generator and the sample size used, and is shown as ntuple.

2.4 Verificação das propriedades com ferramentas da teoria da informação

2.4.1 aaa

Neste capítulo tratamos da Revisão Bibliográfica realizada para o desenvolvimento do trabalho, no capítulo seguinte tratamos da metodologia utilizada do desenvolvimento do mesmo.

Tabela 2.1: Testes disponíveis no Dieharder

Seq	Nome	ID
1	diehard birthdays	0
2	diehard operm5	1
3	diehard rank 32x32	2
4	diehardrank 6x8	3
5	diehard bitstream	4
6	diehard opso	5
7	diehard oqso	6
8	diehard dna	7
9	diehard count 1s stream	8
10	diehard count 1s byte	9
11	diehard parking lot	10
12	diehard 2dsphere	11
13	diehard 3dsphere	12
14	diehard squeeze	13
15	diehard sums	14
16	diehard runs	15
17	diehard craps	16
18	marsaglia tsang gcd	17
19	sts monobit	100
20	sts runs	101
21	sts serial	102
22	rgb bitdist	200
23	rgb minimum distance	201
24	rgb permutations	202
25	rgb lagged sum	203
26	rgb kstest test	204
27	dab bytedistrib	205
28	dab dct	206
29	dab filltree	207
30	dab filltree2	208
31	dab monobit2	209

3

Metodologia

ESTE capítulo tem como objetivo apresentar os materiais e métodos utilizados no trabalho. Como principal objetivo, este capítulo visa fornecer subsídios suficientes e para que o mesmo possa ser reproduzido e a continuidade do mesmo na pesquisa e desenvolvimento dos problemas deixados em aberto possa ser alcançada.

3.1 Materiais e Métodos

Em relação a fundamentação teórica, utilizou-se como principal fonte de pesquisa a área de indexação de periódicos científicos ISI *Web of Knowledge*, onde foram obtidas a grande maioria das referências, usando como parâmetros o fator de impacto dos periódicos pesquisados, a quantidade de citações de cada publicação, o grau de relevância para o tema pesquisado e o nível de produtividade (fator-H) dos autores envolvidos. O apoio em livros, surveys, lecture notes e ferramentas complementares de busca, como o *google acadêmico* foram utilizadas para complementar esta pesquisa.

Para organizar, catalogar e facilitar a consulta a todo material obtido, as referências foram gerenciadas com a ferramenta *Mendeley*. Um gerenciador de referências bibliográficas multiplataforma gratuito que permite organizar de maneira centralizada vários vínculos entre as referências utilizadas, bem como visualizar e anotar as mesmas dentro da própria ferramenta. Quanto à editoração eletrônica do trabalho, fez-se uso da plataforma LaTeX, com editor de textos *Kile* de código aberto. Este trabalho foi desenvolvido num equipamento com as seguintes configurações:

Arquitetura	`Intel i7 64 bits`
S.O.	Linux Mint 17.1 - kernel 3.13.0 – 43 – $generic$
Editor	Kile versão 2.1.3 e LaTeXtexlive 2013.20140215 – 1

Do ponto de vista técnico deste trabalho, com ênfase em Geradores de Números Pseudoaleatórios, é utilizada a plataforma de análise estatística R . Esta plataforma foi desenvolvida originalmente por Ross Ihaka e Robert Gentleman, com o intuito de ser uma linguagem de código aberto voltada para a análise estatística e, consequentemente, a precisão numérica com fortes características funcionais (**?**). Por este motivo, ela será utilizada para a geração e manipulação das sequências pseudoaleatórias, análise dos dados e geração dos gráficos desse trabalho. A precisão numérica desta ferramenta, sendo aferida por **?**, é adequada para essa abordagem.

As ferramentas utilizadas no desenvolvimento deste trabalho, são preferencialmente multiplataforma e código aberto com licença de uso *GNU General Public License* (GPL).

Todos esses aplicativos, métodos e informações obtidas, forneceram grandes contribuições para o traçado da linha mestra deste trabalho, indicando que o mesmo está na fronteira do conhecimento produzindo um estado da arte fidedigno aos temas e ferramentas adotadas para norteá-lo.

Neste capítulo tratamos da metodologia utilizada do desenvolvimento do trabalho, no capítulo seguinte analisaremos os impactos esperados com a realização do mesmo.

4

Resultados Esperados

UMA vez com a modelagem concluída e os dados para simulação, serão realizados os experimentos afim de comprovar a eficácia do modelo proposto.

4.1 Resultados Esperados

Neste capítulo tratamos os resultados esperados com a realização do trabalho enquanto que no próximo faremos a conclusão alcançada com o mesmo.

5

Conclusão

A análise dos dados deve evidenciar a importância de trabalhar a redução dos dados numa rede de sensores, visando a redução do consumo de energia e maximizando o tempo de vida da rede.

5.1 Impactos Esperados

A análise dos dados deve evidenciar a importância de trabalhar a redução dos dados numa rede de sensores, visando a redução do consumo de energia e maximizando o tempo de vida da rede.

Apêndice A

Ambiente Reprodutível e Computacional

O?

REFERÊNCIAS BIBLIOGRÁFICAS

Almiron, M. G., Almeida, E. S. & Miranda, M. N. (2009), 'The reliability of statistical functions in four software packages freely used in numerical computation', *Brazilian Journal of Probability and Statistics* **23**(2), 107–119.

Knuth, D. E. (1998), *The Art of Computer Programming, Volume 2: (2Nd Ed.) Seminumerical Algorithms*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Este trabalho foi redigido em LaTeX utilizando uma modificação do estilo IC-UFAL. As referências bibliográficas foram preparadas no Mendeley e administradas pelo BibTeX com o estilo LaCCAN. O texto utiliza fonte Fourier-GUT*enberg* e os elementos matemáticos a família tipográfica Euler Virtual Math, ambas em corpo de 12 pontos.