Danilo Barreto Cavalcanti

Aplicação de Computação Estocástica à Detecção de Bordas em Imagens Digitais

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica - PPGEE, da Universidade Federal da Paraíba - UFPB, como requisito parcial para a obtenção do título de Mestre em Engenharia Elétrica.

Universidade do Federal da Paraíba – UFPB

Centro de Energias Alternativas e Renováveis

Programa de Pós-Graduação em Engenharia Elétrica

Orientador: Cícero da Rocha Souto

João Pessoa, PB, Brasil

2019

Catalogação na publicação
Seção de Catalogação e Classificação

C376a Cavalcanti, Danilo Barreto.
Aplicação de Computação Estocástica à Detecção de Bordas em Imagens Digitais / Danilo Barreto Cavalcanti. - João Pessoa, 2019.
88 f.

Orientação: Cícero da Rocha Souto.
Coorientação: Waslon Terllizzie Araujo Lopes.
Dissertação (Mestrado) - UFPB/CEAR.

1. Computação Estocástica. 2. Processamento de imagens. 3. Detecção de bordas. 4. Eficiência energética. I. Souto, Cícero da Rocha. II. Lopes, Waslon Terllizzie Araujo. III. Título.

UFPB/BC

UNIVERSIDADE FEDERAL DA PARAÍBA – UFPB
CENTRO DE ENERGIAS ALTERNATIVAS E RENOVÁVEIS – CEAR
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA - PPGEE

A Comissão Examinadora, abaixo assinada, aprova a Dissertação

APLICAÇÃO DE COMPUTAÇÃO ESTOCÁSTICA À DETECÇÃO DE BORDAS EM IMAGENS DIGITAIS

Elaborado por

DANILO BARRETO CAVALCANTI

como requisito parcial para obtenção do grau de
Mestre em Engenharia Elétrica.

COMISSÃO EXAMINADORA

PROF. DR. CÍCERO DA ROCHA SOUTO (Presidente)

PROF. DR. WASLON TERLLIZZIE ARAUJO LOPES

PROF. DR. JUAN MOISÉS MAURICIO VILLANUEVA

PROF. DR. LUCÍDIO DOS ANJOS FORMIGA CABRAL

João Pessoa/PB, 23 de julho de 2019

Dedico este trabalho à minha família, principalmente a minhas irmãs, que estão sempre me apoiando e me inspirando em suas lutas.

Agradecimentos

Agradeço a todos com quem tive contato desde o início do meu mestrado.

Agradeço à minha família pelas inumeráveis de formas de apoio no passado, presente e futuro.

Agradeço ao professor Antonio Augusto pelo trabalho passional em Eletrônica e pelo rico estágio realizado ao fim da graduação no RFWild.

Agradeço ao professor Waslon Terllizzie por ter assumido minha orientação em uma situação complicada e pelo me auxílio continuo desde então.

Agradeço aos professores Cícero Souto e Alexsandro Virgínio pelo apoio na reta final.

Agradeço a toda a equipe da Coordenadoria em Engenharia Elétrica da Universidade Federal da Paraíba e aos membros do Centro de Energias Alternativas e Renováveis por sempre reduzirem qualquer problema burocrático que veio a aparecer, e por todo o auxílio rápido oferecido para as mais diversas questões.

Agradeço a Joabe Brasil por toda a perícia técnica e bom humor durante os trabalhos no RFWild, tornando todo o trabalho mais agradável.

Agradeço aos professores do bloco CI e à banca examinadora da anterior qualificação deste trabalho, que me forneceram várias ideias e direcionamentos para complementar meus estudos.

Agradeço ao grupo do LAMEP/UFPB, principalmente aos professores Moisés, Gustavo e Rafael, pela presteza em me fornecer condições para realizar simulações de alto desempenho nas fases finais deste trabalho.

Agradeço ao programa FAPESQ/CAPES/BRASIL pela concessão de incentivos de estudos para pós-graduação.

Agradeço a todos os demais amigos, colegas e orientadores que participaram e participam do processo de criação deste trabalho.

Resumo

Este trabalho compara o algoritmo de detecção de bordas em imagens em tons de cinza baseado em operadores de Sobel em duas implementações: utilizando lógica digital convencional e utilizando Computação Estocástica. Esse último caso se refere a uma forma não convencional de organizar números e operá-los, tratando-os como probabilidades. A área possui vantagens em relação a robustez a ruídos, utilização de área de circuitos integrados e consumo de potência, ao se comparar com a Eletrônica Digital CMOS amplamente utilizada, para certas aplicações. São feitas implementações em *software* (Python) e em *hardware* (Verilog/FPGA) de ambos os casos. Mostra-se que a ordem de grandeza da diferença dos resultados obtidos pelas abordagens é de dois bits. É mostrado que a estocástica é mais robusto a ruídos suaves, e com distribuições de erro menos discrepantes entre 1% e 5% de taxas de erro de bits. Finalmente, são mostradas a quantidade de portas lógicas utilizadas em cada versão e sua respectiva potência requerida. Isso estabelece diretrizes de custo-benefício para projetos de Eletrônica Digital visando eficiência energética.

Palavras-chave: Computação Estocástica. processamento de imagens. detecção de bordas. eficiência energética.

Abstract

This work compares the grey image edge detection algorithm based on Sobel operators considering two implementations: conventional digital logic and Stochastic Computing. The latter refers to an unconventional way of organising and operating numbers, treating them as probabilities. There are advantages in noise robustness, integrated circuit area utilisation and power consumption, when compared to CMOS Digital Electronics, in certain applications. This work implemented and evaluated filters in software (Python) and hardware (Verilog/FPGA). It is shown that the order of magnitude of the difference between results in both versions is of two bits. It is shown that the stochastic approach is more robust to soft errors, while also having less discrepant error distributions between 1% and 5% of error bit rates. Finally, the number of logical gates and power required for each implementation is presented. This stablishes guidelines of cost-benefit for energy efficient Digital Electronics projects.

Keywords: Stochastic Computing. image processing. edge detection. energy efficiency.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

A/D	Analógico-digital
BF	Função booleana (*Boolean Function*)
BP	*Bipolar*
CI	Independente a correlação (*Correlation Independent*)
D/A	Digital-analógico
FPGA	Matriz de portas programável em campo (*Field-Programmable Gate Array*)
HDL	Linguagem de descrição de *hardware* (*Hardware Description Language*)
IBP	Bipolar inversa (*Inverse Bipolar*)
IoT	Internet das Coisas (*Internet of Things*)
LDPC	Checagem de Paridade de Baixa Densidade (*Low-Density Parity Check*)
PFM	Modulação por frequência de pulso (*Pulse Frequency Modulation*)
MAE	Erro médio absoluto (*Mean Absolute Error*)
MReSC	Arquitetura reconfigurável baseada em lógica estocástica multivariável (*Multivariable Reconfigurable architecture based on Stochastic logiC*)
PTM	Matrizes de transferência probabilísticas (*Probabilistic Transfer Matrices*)
ReSC	Arquitetura reconfigurável baseada em lógica estocástica (*Reconfigurable architecture based on Stochastic logiC*)
SC	Circuito estocástico (*Stochastic Circuit*)
SF	Função estocástica (*Stochastic Function*)
SN	Número estocástico (*Stochastic Number*)
SNG	Gerador de números estocásticos (*Stochastic Number Generator*)
STRAUSS	Uso de transformada espectral em síntese de circuitos estocásticos (*Spectral TRAnsform Use in Stochastic circuit Synthesis*)

TT	Tabela-verdade (*Truth-Table*)
UP	*Unipolar*

Sumário

1 INTRODUÇÃO

A diversidade do número de sistemas eletrônicos na sociedade motiva uma diversidade de abordagens para implementá-los. A título de ilustração, uma simples calculadora pode estar inserida em um ambiente com excesso de recursos, como um microcomputador pessoal, ou estar embutida em um dispositivo dedicado, como um forno a micro-ondas ou uma rede de sensores sem fio. No primeiro caso, a utilização ótima dos recursos de cálculo não é tão relevante quanto a aplicação em si, pois os recursos disponíveis são dimensionados para problemas muito mais complexos e com ordem de tempo de execução muito maior do que as operações aritméticas simples da calculadora. Para o segundo caso, a ação de operar em dados pode ser o ponto crítico de funcionamento do sistema, dado que seus recursos energéticos e computacionais são limitados. Também não costuma haver possibilidade de reprogramação, ou seja, as soluções são implementadas em circuitos dedicados altamente especializados e otimizados para executarem as operações sendo robustos a influências externas e com baixo consumo de energia. Este trabalho é motivado pela necessidade de implementar algoritmos em sistemas embarcados visando o equilíbrio ótimo entre três fatores: desempenho, utilização de recursos e robustez a ruídos externos.

Desde os primórdios da Eletrônica e dos sistemas eletrônicos, observa-se a evolução da Computação. Os modelos mais utilizados atualmente são binários e se baseiam em uma notação ponderada. Os seus elementos fundamentais são os bits, que podem assumir apenas valores 0 ou 1 individualmente. A partir de seu ordenamento e de codificação, como por exemplo o padrão IEEE 754 (KAHAN, 1996), que define aritmética de ponto flutuante, é possível representar dados e efetuar operações sobre eles. Embora esse padrão seja amplamente utilizado, ele não é perfeito. Operações no domínio digital estão sujeitas a erros, seja por conversão entre domínios (analógico-digital, digital-analógico) ou por limitações de representação. A despeito da existência de fontes de erro, a ampla utilização da aritmética de ponto flutuante em sistemas eletrônicos no cotidiano mostra que muitas vezes o erro produzido é negligenciável ou controlável por sistemas auxiliares (HAZUCHA; SVENSSON, 2000), (BAUMANN, 2002), (KIM et al., 2015).

Um padrão de Computação alternativo, proposto em meados de 1960, denominado Computação Estocástica (do inglês, *Stochastic Computing*) propunha a utilização de bits em um sistema sem ponderação onde cadeias representariam probabilidades e a aritmética probabilística poderia ser utilizada para implementar expressões e algoritmos (GAINES, 1967). Esse padrão não se viu competitivo frente à lógica digital tradicional, em ascensão, e manteve-se em desuso até o início do século XXI. Uma das notáveis aplicações que contribuiu para o ressurgimento de estudos em Computação Estocástica foi a decodificação

de códigos LDPC (*Low-Density Parity-Check codes*) (GAUDET E REPLAY 2003). Em resumo, um código LDPC é um código linear de correção de erro, aplicável na transmissão de mensagens em canais ruidosos. Esse código é relevante em Telecomunicações por permitir a aproximação da capacidade do canal ao limite teórico (o limite de Shannon) (Shannon, 1959), (VERDÚ et al., 1994). Sua aplicação não era muito presente até recentes avanços na Eletrônica (por volta de 2010), que permitiram a implementação das matrizes esparsas e das unidades de cálculo do código em SoCs (*System on Chips*) e seu estabelecimento nos padrões atuais de transmissão sem-fio (Wi-Fi). Os códigos LDPC são utilizados no padrão 5G de telecomunicações sem fio (BAE et al., 2019). A aplicação Computação Estocástica na implementação de um algoritmo considerado complexo utilizando lógica digital convencional chamou atenção para características que poderiam beneficiar outros sistemas.

Os sistemas mais compatíveis com Computação Estocástica são combinacionais. A revisão bibliográfica do estado atual da área, feita por Alaghi, Qian e Hayes (2018), destaca desafios em implementar circuitos sequenciais, como circuitos com malhas de realimentação. Até nos dias atuais, a decodificação de códigos LDPC se mantém como um problema pertinente a ser estudado e melhorado, como no trabalho de Leduc-Primeau et al. (2019). Ademais, há estudos como os de Ranjbar, Salehi e Najafi (2015) e Lee et al. (2017), que avaliam abordagens novas ao Processamento Digital de Imagens tradicional. Em especial, o trabalho de Onizawa et al. (2017) apresenta um algoritmo sofisticado, baseado em Computação Estocástica, para implementar filtros de áudio para uso em implantes cocleares. Esses trabalhos representam formas inovadoras de integrar os princípios de Computação Estocástica, conseguindo relações de custo-benefício bastante elevadas em termos de consumo de energia e utilização de área de circuito integrado. Esses resultados enfatizam a aplicabilidade de Computação Estocástica em situações em que há restrição de recursos disponíveis, principalmente envolvendo processamento de sinais.

A teoria da área ainda não possui consolidação em forma de livros didáticos. Desta forma, são vistas pertinentes as contribuições na forma de ferramentas e metodologias para reproduzir e validar resultados quanto a desempenho e utilização de recursos, em relação a uma implementação dual determinística.

1.1 Objetivos

1.1.1 Objetivo Geral

O objetivo deste trabalho é utilizar Computação Estocástica para o problema de detecção de bordas em imagens digitais por Filtro de Sobel a fim de oferecer uma metodologia, e comparar e contrastar características de desempenho, custo-benefício e robustez a ruídos suaves, em relação à implementação convencional.

1.1.2 Objetivos Específicos

1. Oferecer ferramentas e metodologias para implementação de um sistema que utiliza Computação Estocástica;
2. Implementar Filtro de Sobel 3×3 utilizando sistemas convencionais e estocásticos;
3. Comparar resultados da operação de filtragem apresentando métricas objetivas;
4. Avaliar robustez a ruídos;
5. Avaliar utilização de portas lógicas e dissipação de potência em ambas as abordagens;
6. Estimar desperdício de recursos ao utilizar conversores binário-estocástico, em vez de conceber um circuito estocástico puro.

1.1.3 Motivação

O trabalho é motivado por características do problema e sua pertinência como tecnologia. Por exemplo, o sistema possui múltiplas entradas e, muitas vezes, possui requerimentos de tempo real em suas aplicações. Além disso, áreas como Engenharia Biomédica e o mercado de dispositivos móveis (telefones celulares, robôs, etc.) se beneficiam das condições de custo-benefício oferecidas por essa metodologia alternativa de computação.

A análise da aplicação de Computação Estocástica em um Filtro de Sobel fornece uma base da comparação para outras aplicações que envolvem Processamento de Sinais. Isso pode possibilitar a criação de um fluxo de projeto dedicado para sistemas embarcados de processamento de sinais com ênfase em eficiência de uso de recursos, em troca de exatidão.

2 REVISÃO BIBLIOGRÁFICA

A aplicação efetiva de Computação Estocástica na área de Processamento Digital de Sinais requer alguns conhecimentos prévios. Inicialmente, é necessário entender a natureza do algoritmo convencional a ser adaptado. Em seguida, precisa-se compreender os principais conceitos de Computação Estocástica para avaliar sua aplicabilidade. Há sistemas completamente conversíveis, aproximáveis ou incompatíveis. Uma vez que são observadas ambas as bases, é requerido conhecimento em Eletrônica Digital para entender como realizar fisicamente um circuito estocástico como circuito integrado, garantindo suas vantagens e desvantagens intrínsecas.

2.1 Detecção de Bordas em Imagens com Operadores de Sobel

A detecção de bordas em imagens digitais é um problema que consiste basicamente em buscar uma forma de enfatizar fronteiras em áreas de interesse e é compreendido como uma filtragem da imagem. Isso é comumente implementado de duas maneiras: filtros espaciais ou filtros em frequência (GONZALEZ; WOODS, 2006).

Filtros no domínio da frequência requerem a existência do espectro da imagem *a priori* para serem aplicados, além do eventual retorno ao domínio do espaço de cores da imagem. Por esses motivos, optou-se pela utilização de filtros no domínio espacial.

Filtragem espacial é um tipo de processamento digital de imagem que ocorre pela operação de convolução entre um operador (matriz representando o filtro) e uma imagem. A operação de convolução pode ser representada por

$$R = F * A \tag{2.1}$$

em que R é a matriz resultante, F é uma matriz representando um operador e A é uma imagem digital, representada por uma matriz de *pixels*. O operador $*$ representa a operação de convolução.

Duas das formas mais comuns de destacar contornos em uma imagem são a utilização das derivadas de primeira e de segunda ordem. A técnica para as de primeira ordem se baseia no cálculo de uma imagem gradiente e as de segunda ordem no cálculo de uma imagem Laplaciano. A vantagem do primeiro caso é a necessidade de cálculo de uma derivada de ordem menor, porém introduzindo uma operação não linear. Analogamente, o segundo caso opera na imagem de forma linear, mas precisa da definição da segunda derivada em cada ponto da imagem (GONZALEZ; WOODS, 2006).

Embora a adoção de uma técnica não linear em um sistema linear não seja intuitiva, ela é justificada e utilizada em alguns operadores comuns, como o de Sobel (FARID; SIMONCELLI, 1997). O objetivo da utilização desses operadores é a aproximação do gradiente da imagem. O gradiente de uma função $f(x, y)$ arbitrária e seus respectivos g_x e g_y são definidos por

$$\nabla f \equiv \text{grad}(f) \equiv \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \tag{2.2}$$

Para esta aplicação, f é a matriz de *pixels* que representa a imagem no domínio digital.

A imagem gradiente é obtida tomando a magnitude do gradiente da imagem em cada par (x, y). O valor de cada elemento de imagem $M(x, y)$, representado por

$$M(x, y) = \text{mag}(\nabla f) = \sqrt{g_x^2 + g_y^2} \tag{2.3}$$

é a intensidade do gradiente para essa posição, tomando a norma euclidiana como valor do vetor.

Levando em consideração que ambos os componentes do gradiente são derivadas, eles são lineares. Logo, é possível aproximá-los por operadores discretos em forma matricial e calculá-los como uma soma de produtos. Uma única operação não linear (para cada *pixel*) ocorre ao final desse processo, pelo cálculo da norma para obter a imagem gradiente M (GONZALEZ; WOODS, 2006).

Uma aproximação comumente aplicada à Equação (2.3) é

$$M(x, y) \approx |g_x| + |g_y| \tag{2.4}$$

que pode tornar menos onerosas as operações de elevar ao quadrado e então aplicar raiz quadrada ao resultado da soma. Essa aproximação reduz o custo computacional do cálculo da imagem gradiente, mas mantém sua linearidade.

Outra importante característica desse tipo de processamento é a presença ou ausência de isotropia nas operações. As derivadas, embora lineares, não são operações isotrópicas. Ou seja, são variáveis de acordo com a rotação do operador. A norma euclidiana, porém, é uma operação isotrópica. Quando a aproximação tal qual pela Equação (2.3) é feita, a isotropia é perdida, de forma geral. A despeito disso, as aproximações discretas para as derivadas parciais do gradiente na Equação (2.2) são isotrópicas para ângulos múltiplos de 90°, independentemente da posterior adoção da Equação (2.3) ou (2.4). Esse resultado é relevante, pois possibilita aplicar uma operação menos custosa, do ponto de vista computacional, sem perda da isotropia da imagem gradiente (GONZALEZ; WOODS, 2006).

A partir deste ponto, define-se uma região 3×3 de uma imagem de forma genérica, como representado na Figura 1(a). O elemento z_5 representa a intensidade do *pixel* central de uma sub-região da imagem ao qual o filtro será aplicado. Os demais valores da matriz são obtidos de acordo com a vizinhança de z_5.

A Figura 1(b) apresenta os operadores de Sobel g_x e g_y. Observa-se que a definição de g_x e g_y varia de acordo com a referência adotada. Este trabalho adota a convenção de que g_x destaca variações ao longo do eixo x, horizontal, e g_y do eixo y, vertical, presente no trabalho de Sobel (2014).

Os componentes do gradiente da Equação (2.2), g_x e g_y, são aproximados, respectivamente, por

$$g_x = \frac{\partial f}{\partial x} = (z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7) \tag{2.5}$$

e

$$g_y = \frac{\partial f}{\partial y} = (z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3) \tag{2.6}$$

A imagem gradiente M é aproximada por

$$M(x, y) \approx |(z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7)| + |(z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3)| \tag{2.7}$$

substituindo os termos do operador de Sobel, provenientes das Equações (2.5) e (2.6).

É necessário, então, abordar o problema de condições de fronteira. Como a operação de filtragem opera com um *pixel* de referência central convencionado z_5, o algoritmo não está bem definido para as primeiras e últimas linhas e colunas da matriz. Como a dimensão das imagens trabalhadas é elevada (ao menos 100×100) e a região de interesse dessa operação se concentra em sub-regiões tendendo ao centro, e não às extremidades das

Figura 1 – Matrizes importantes para aplicação do método de Sobel. (a): matriz representando uma área 3×3 em volta do pixel z_5. (b): máscaras de Sobel

.

$$\begin{bmatrix} z_1 & z_2 & z_3 \\ z_4 & z_5 & z_6 \\ z_7 & z_8 & z_9 \end{bmatrix}$$

(a)

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

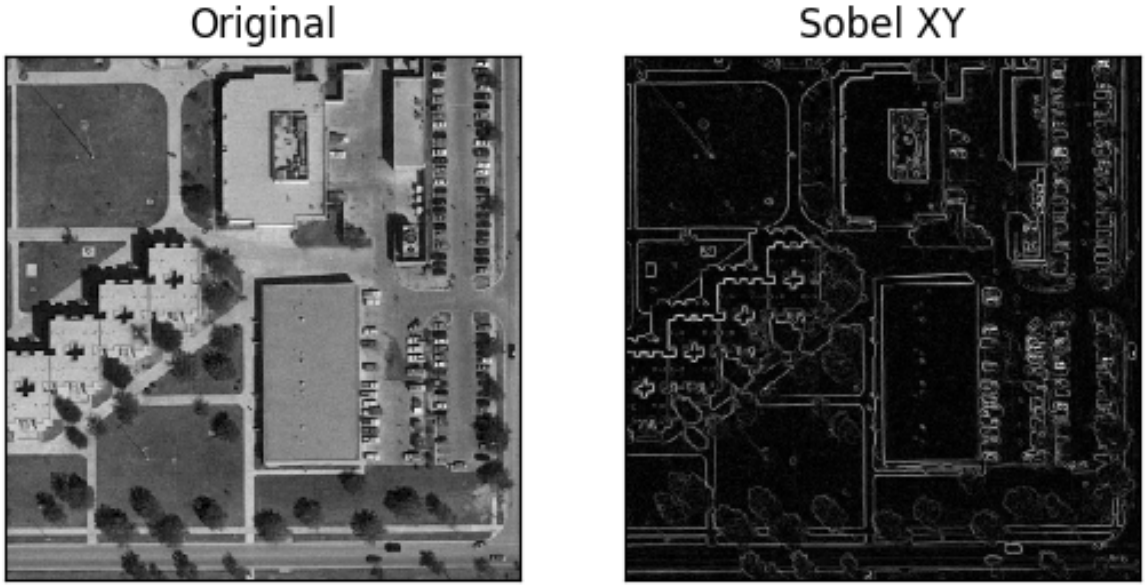
(b) Sobel

Fonte: Gonzalez e Woods (2006, p. 166) Adaptada pelo autor.

imagens, a abordagem adotada neste trabalho será de excluir o cálculo do operador nessas fronteiras.

Uma vez que as expressões estão definidas, define-se o foco deste trabalho como a detecção de bordas para imagens em tons de cinza com 8 bits de profundidade. Isso significa que cada pixel possuirá um valor de intensidade de branco (tom de cinza) na faixa [0, 255]. A Figura 2(a) apresenta uma imagem típica que atende a essa especificação adjacente a sua respectiva imagem gradiente (Figura 2(b)), obtida pela Equação (2.7).

Figura 2 – Imagem original e imagem gradiente após a aplicação dos operadores de Sobel 3×3.



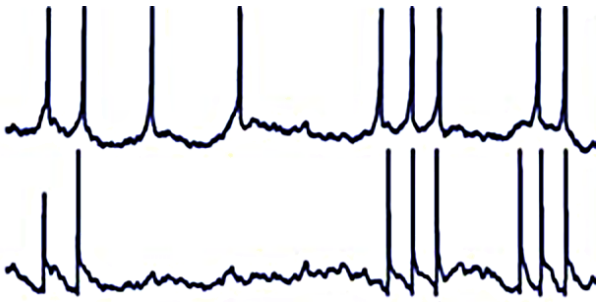
Fonte: Produzida pelo autor.

2.2 Computação Estocástica

A Computação Estocástica aborda a utilização dos elementos lógicos tradicionais, como portas AND, OR, ou NOT, para implementar funções probabilísticas. Esse modelo de computação foi inspirado no sistema nervoso humano. Os sinais são codificados em cadeias de bits mais longas do que o convencional (128 a 1024 bits, por exemplo). Para ilustrar, um sinal biológico qualquer está representado como cadeia de bits na Figura 3. Esse tipo de representação pode ser entendido como modulação por frequência de pulso (PFM – *Pulse Frequency Modulation*), onde cada pulso é representado por um *bit* 1 e instantes de repouso por um *bit* 0 (ROCHELLE, 1962) (ALAGHI, 2015, p. 2, 105).

Essas cadeias de bits são denominadas de números estocásticos (SN), cujo valor representa a probabilidade da ocorrência de um *bit* 1 em qualquer posição da cadeia. Por

Figura 3 – Sinal biológico representado como cadeia de bits.



1100111000001001 0111011100000000011

Fonte: Alaghi (2015, p. 4).

exemplo, um SN X contendo 50% de bits 1 possui valor $p_x = 0,50$. Diferentemente da notação convencional ponderada, SN não levam em consideração a posição dos bits para determinar seu valor. Importa apenas a relação entre número de bits 1 e o tamanho total da cadeia. Para ilustrar, todos os números a seguir representam $p = 0,50$:

- 10101010
- 1100
- 11010010
- 111000
- 0011010110011100

Esses exemplos também servem para destacar a redundância presente no sistema. Dois números estocásticos distintos e de mesmo tamanho podem representar um mesmo valor. Esse fato pode ser benéfico para certos tipos de ruído em sistemas, porém traz consigo implicações sobre o comportamento do erro em uma sequência de operações.

Um SN e seu valor são comumente representados utilizando o mesmo símbolo, para simplificar a notação (ALAGHI, 2015, p. 13). Por exemplo, um SN X de valor $p_x = 0,25$ pode ser escrito como $X = 0,25$. Subentende-se, pelo contexto, quando trata-se da cadeia de bits ou da probabilidade associada.

Duas operações básicas relevantes em Computação Estocástica são a multiplicação e a soma escalonada. Como dois dados SN representam probabilidades, uma multiplicação de duas entradas pode ser entendida como a probabilidade de ter bits de valor 1 simultaneamente em ambas as cadeias, dada uma posição. A forma mais simples de representar um SN é na região de 0 a 1, na chamada representação unipolar (UP), em que o valor do

número equivale à razão do número de bits 1 (N_1) dividido pelo tamanho (N) da cadeia:

$$p_x = \frac{N_1}{N} \tag{2.8}$$

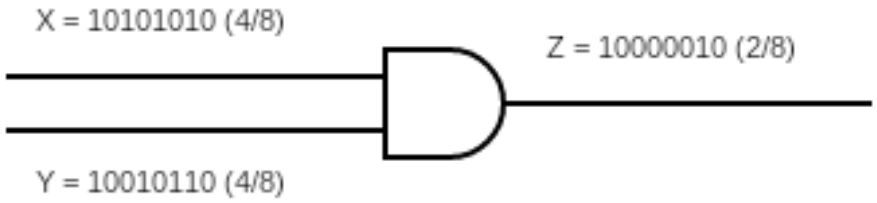
Logo, pode-se efetuar a operação com uma única porta AND, para a representação, com pulsos de relógio tantos quantos o número de bits da cadeia de entrada. A cadeia de bits da saída representa a multiplicação das entradas.

As Figuras 4 e 5 apresentam dois SN X e Y, ambos com valor $0,50$ com 8 bits de representação. Observa-se que a Figura 4 apresenta o resultado esperado da operação, enquanto a Figura 5 ilustra um erro, considerando o contexto de números estocásticos.

As operações efetuadas com números estocásticos são ideais em ambientes que apresentam uma situação de compromisso entre exatidão e eficiência no uso de recursos (área de chip, energia) (ALAGHI, 2015, p. 8). As grandes cadeias de bits características provêm imunidade a ruídos que alterem bits individualmente. Além disso, pares de erros de mudança de bit podem resultar em erro nulo, pois ao contrário da notação binária ponderada convencional, cada *bit* influencia em apenas $\frac{1}{N}$ o valor de uma cadeia de tamanho N.

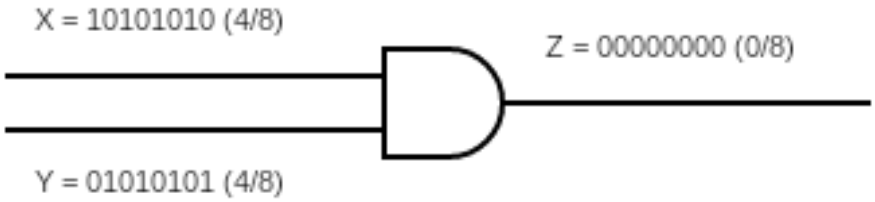
Computação Estocástica envolve várias áreas de estudo. A concepção de um circuito integrado estocástico (*stochastic circuit* – SC) possui estudos presentes tanto nas tecnologias de fabricação (CMOS, *memristors*, etc.) quanto nas metodologias de projeto. Isso inclui síntese automática de funções lógicas (ReSC, STRAUSS) (QIAN et al., 2011a), (ALAGHI; HAYES, 2015), tipos de representação (unipolar, bipolar e bipolar inversa, abordadas na subseção 2.2.2), geração de números estocásticos (*Stochastic Number Generators* – SNG), algoritmos para diminuição de erro (correlação, flutuações aleatórias), entre outros. Os tópicos mais relevantes para auxiliar na síntese e análise do circuito estocástico em questão são abordados nas subseções seguintes.

Figura 4 – Multiplicação em Computação Estocástica.



Fonte:Produzida pelo autor.

Figura 5 – Multiplicação em Computação Estocástica errônea por causa de números correlacionados. A correlação existe porque $Y = \overline{X}$. Ou seja, Y é perfeitamente correlacionado negativamente a X.



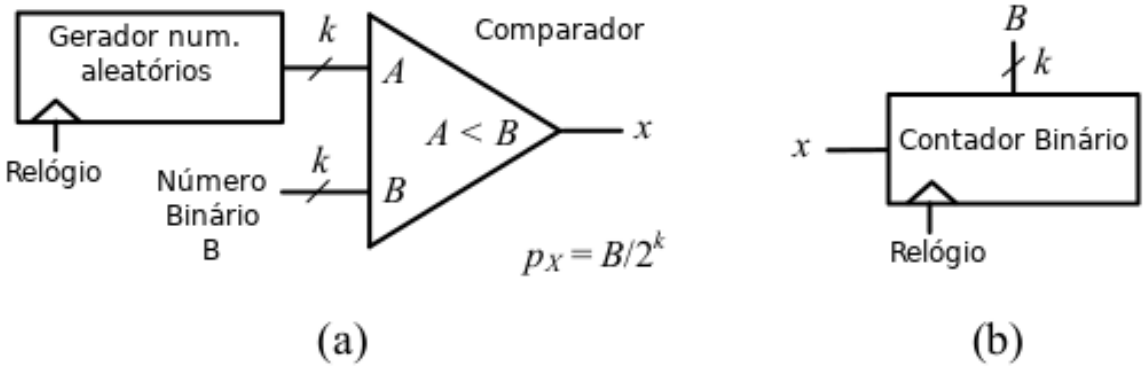
Fonte: Produzida pelo autor.

2.2.1 Geração de números estocásticos

A Computação Estocástica, em muitas aplicações, situa-se entre o domínio digital tradicional e o analógico. Como ainda não há padrões de conversores A/D (analógico-digitais) e respectivos D/A (digital-analógicos) para o domínio estocástico, a conversão de números binários convencionais em estocásticos torna-se uma tarefa comum. Sendo assim, é necessário abordar o problema da geração números estocásticos adequados para diversas aplicações.

O exemplo apresentado da Figura 5 demonstra a importância de controlar quais números irão operar entre si, e como eles são gerados. O método mais tradicional para conversão de domínios é apresentado na Figura 6. Embora o tópico sempre sugira que aleatoriedade plena é um alvo a ser buscado, ele não representa, necessariamente, o caso ideal. Fontes pseudoaleatórias ou até mesmo fontes determinísticas são capazes de produzir conversores adequados para a maioria das aplicações referenciadas na literatura (ALAGHI, 2015, p. 24), (ALAGHI; QIAN; HAYES, 2018, p. 4 – 5).

Figura 6 – Circuitos lógicos conversores: (a) circuito conversor binário-estocástico, (b) circuito conversor estocástico-binário

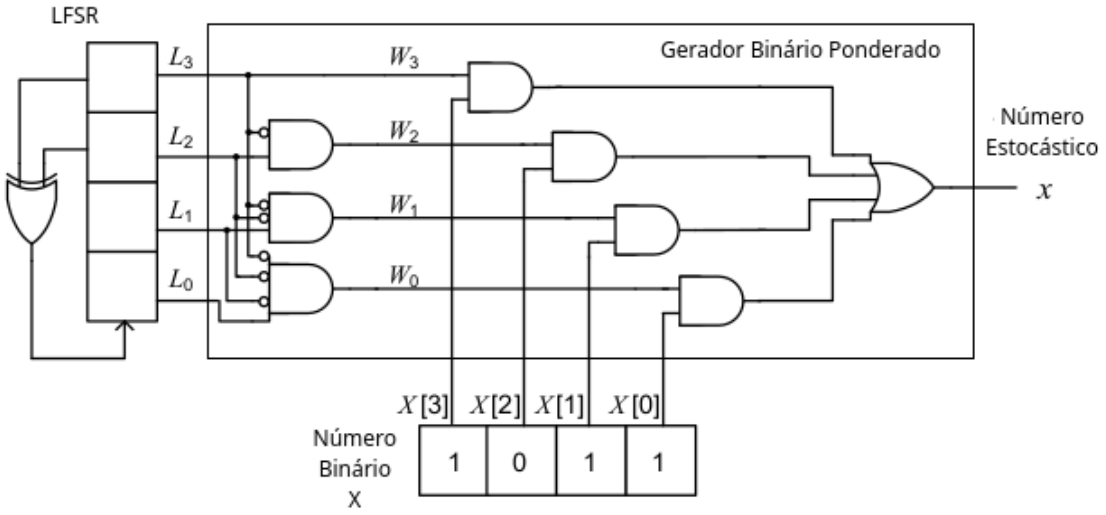


Fonte: Alaghi (2015, p. 20) Traduzida e adaptada pelo autor.

O circuito conversor estocástico-binário pode ser implementado por meio de um contador. A partir da contagem de bits iguais a 1 na cadeia de entrada e dado o tamanho da mesma, é possível chegar ao valor original. Exemplo: se um número binário $x = (011)_b = 3$ for convertido para a forma mínima estocástica unipolar, deve-se usar 8 bits, tal que a precisão é mantida a mesma ($2^3 = 8$). Um dos possíveis números que representa x é $X = 01010100$, que possui $p_x = \frac{3}{8}$. Ao contar o número de bits 1, obtém-se o valor original: 3.

A mesma simplicidade não é encontrada nos conversores binário-estocástico, denominados geradores de números estocásticos (*Stochastic Number Generators – SNG*). Um exemplo relevante é o conversor ponderado proposto por (GUPTA; KUMARESAN, 1988). Ele é composto por duas unidades principais: um registrador de deslocamento linear (*linear feedback shift register – LFSR*) e um circuito de ponderação. A Figura 7 apresenta uma versão de 4 bits do conversor.

Figura 7 – SNG ponderado proposto por Gupta e Kumaresan



Fonte: Alaghi (2015, p. 22). Texto traduzido pelo autor.

Ao acompanhar seu funcionamento a cada pulso de relógio, pode-se observar, por exemplo, para um número binário de entrada $x = (1011)_b = 11$, a sequência da Tabela 1. Não há menção na proposição original desse sistema sobre a forma correta de inserir o estado $L = (0000)_b$ artificialmente. Entretanto, há soluções na forma de circuitos auxiliares que lidam com tal problema, como aplicado em Alaghi e Hayes (2015). Uma observação interessante é que essa forma só possui um *bit* igual a 1 em qualquer coluna para os valores de W. Ou seja, se um dos bits de W assume valor 1, os demais certamente estarão em 0.

Um destaque muito interessante é a possibilidade de gerar números estocásticos perfeitos (ou seja, passíveis de utilização em SC) substituindo o LFSR por um contador binário simples, totalmente determinístico. Isso pode ser observado na Tabela 2. O valor

de L realiza contagem binária de 0 a 15. Mesmo sendo determinística, essa técnica pode ser utilizada para realizar operações com SN, pois é possível garantir que não haverá correlação entre os dois números simplesmente deslocando a sequência completamente em uma unidade. Ou seja, o contador binário começaria em 1, e não em 0. Correlação em números estocásticos é definida e discutida na subseção 2.2.3.

Tabela 1 – Cadeias de bits produzidas por uma situação do SNG de Gupta e Kumaresan

Sinal	Cadeia de bits	Valor
L_3	0 0 1 0 1 0 1 1 1 1 0 0 0 0 1 1	$\frac{8}{16}$
L_2	0 1 0 1 0 1 1 1 1 0 0 0 0 1 1 0	$\frac{8}{16}$
L_1	1 0 1 0 1 1 1 1 0 0 0 0 1 1 0 0	$\frac{8}{16}$
L_0	0 1 0 1 1 1 1 0 0 0 0 1 1 0 0 1	$\frac{8}{16}$
W_3	0 0 1 0 1 0 1 1 1 1 0 0 0 0 1 1	$\frac{8}{16}$
W_2	0 1 0 1 0 1 0 0 0 0 0 0 0 1 0 0	$\frac{4}{16}$
W_1	1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0	$\frac{2}{16}$
W_0	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0	$\frac{1}{16}$
x	1 0 1 0 1 0 1 1 1 1 0 1 1 0 1 1	$\frac{11}{16}$

Fonte: Alaghi (2015, p. 23)

Tabela 2 – Cadeias de bits produzidas por uma situação do SNG de Gupta e Kumaresan de forma determinística

Sinal	Cadeia de bits	Valor
L_3	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1	$\frac{8}{16}$
L_2	0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1	$\frac{8}{16}$
L_1	0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1	$\frac{8}{16}$
L_0	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1	$\frac{8}{16}$
W_3	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1	$\frac{8}{16}$
W_2	0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0	$\frac{4}{16}$
W_1	0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0	$\frac{2}{16}$
W_0	0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0	$\frac{1}{16}$
x	0 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1	$\frac{11}{16}$

Fonte: Alaghi (2015, p. 23)

Embora a técnica apresentada seja promissora, os pontos negativos provenientes da adição da várias portas lógicas adicionais, além do crescimento exacerbado do circuito com o crescimento do tamanho da cadeia de bits, faz com que essa opção não seja viável para o uso em detecção de bordas em imagens digitais.

2.2.2 Tipos de representação

SN ditos unipolares (UP) assumem valores apenas no intervalo [0, 1], sendo esse o domínio de toda a aritmética probabilística. Há outras notações para expandir as possibilidades de funções efetuadas dessa com Computação Estocástica, tais como a forma

bipolar (BP) e a bipolar inversa (IBP), cujos números pertencem aos intervalos [-1, 1] e [1, -1], respectivamente. A Tabela 3 apresenta os possíveis valores representados por uma cadeia de 8 bits nas formas unipolar, bipolar e bipolar inversa, excluindo representações redundantes.

Tabela 3 – Cadeias de bits representativas de tamanho 8 e seus valores em diferentes formatos.

Cadeia	UP	BP	IBP
00000000	0	-1	$+1$
00000001	1/8	$-3/4$	3/4
00000011	2/8	$-2/4$	2/4
00000111	3/8	$-1/4$	1/4
00001111	4/8	0	0
00011111	5/8	1/4	$-1/4$
00111111	6/8	2/4	$-2/4$
01111111	7/8	3/4	$-3/4$
11111111	1	1	-1

Fonte: Alaghi (2015, p. 14)

A representação IBP será a principal adotada, já que ela engloba números negativos no seu domínio e possui utilidade para a modelagem dos algoritmos de síntese automática de circuitos estocásticos. A Tabela 4 apresenta como obter o valor numérico de dado SN, sendo N_1 o número de bits 1 nela, N_0 o de bits 0 e N o tamanho total da cadeia.

Tabela 4 – Cadeias de bits representativas de tamanho 8 e seus valores em diferentes formatos.

Formato	Valor numérico	Domínio	Relação ao valor unipolar p_x
Unipolar (UP)	N_1/N	[0, 1]	p_x
Bipolar (BP)	$(N_1 - N_0)/N$	[-1, +1]	$2p_x - 1$
Bipolar inversa (IBP)	$(N_0 - N_1)/N$	[+1, -1]	$1 - 2p_x$

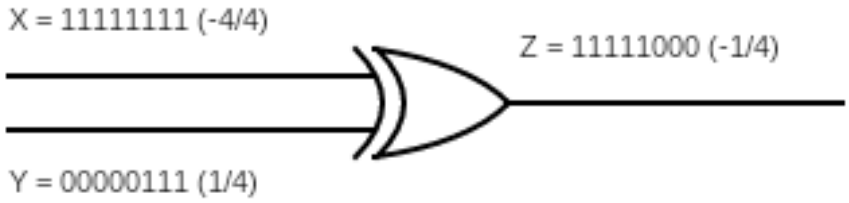
Fonte: Alaghi (2015, p. 14) Traduzida e adaptada pelo autor.

A Figura 8 ilustra uma porta XOR efetuando a multiplicação de dois números na representação IBP. Tal como a porta AND foi substituída pela porta XOR nesse caso, alterações mais significativas podem ocorrer para operações distintas. Isso implica diferentes circuitos de diferentes áreas e características de consumo podem efetuar a mesma operação e há uma oportunidade de otimização nesta etapa. Esses circuitos ainda apresentam o mesmo caráter estocástico e estão sujeitos às mesmas fontes de erro, como exemplificado na Figura 9.

2.2.3 Correlação

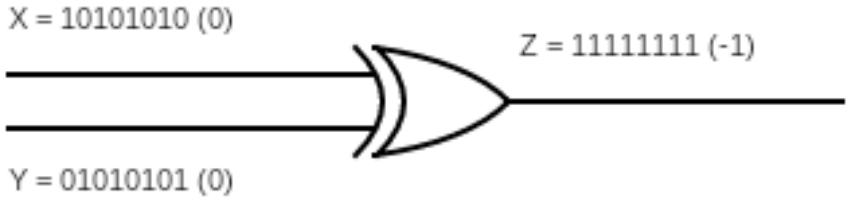
A correlação é uma medida de Estatística que quantifica a dependência entre duas ou mais variáveis (BUSSAB; MORETTIN, 2010). Historicamente, Computação

Figura 8 – Multiplicação em Computação Estocástica em representação bipolar inversa



Fonte: Produzida pelo autor.

Figura 9 – Multiplicação em Computação Estocástica em representação bipolar inversa com números correlacionados



Fonte: Produzida pelo autor.

Estocástica utilizava a premissa de operar com números descorrelacionados. Ou seja, números representando uma correlação nula entre eles mesmos. Entretanto, estudos mais recentes exploraram a correlação em Computação Estocástica. Alaghi e Hayes (2013a), por exemplo, definem SCC (*Stochastic Computing Correlation*), uma medida de correlação apropriada para a área.

O coeficiente de correlação de Pearson, dado por

$$\rho = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2} \cdot \sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}} = \frac{\mathrm{cov}(X, Y)}{\sqrt{\mathrm{var}(X) \cdot \mathrm{var}(Y)}} \tag{2.9}$$

é a principal medida de correlação entre variáveis no meio científico (BUSSAB; MORETTIN, 2010). Essa definição não é apropriada para Computação Estocástica devido a restrições nos valores das cadeias de bits. Por exemplo, $\rho = +1$ implica cadeias de bits idênticas (ALAGHI; HAYES, 2013a, p. 3).

A definição matemática de SCC é dada por

$$\mathrm{SCC}(X, Y) = \begin{cases} \frac{p_{x \wedge y} - p_x p_y}{\min(p_x, p_y) - p_x p_y}, \text{ se } p_{x \wedge y} > p_x p_y \\ \frac{p_{x \wedge y} - p_x p_y}{p_x p_y - \max(p_x + p_y - 1, 0)}, \text{ caso contrário} \end{cases} \tag{2.10}$$

em que $X \wedge Y$ é a operação AND bit a bit, cujo resultado é um SN de valor $p_{X \wedge Y}$. $p_x p_y$ é o valor da multiplicação dos valores individuais X e Y tomados como descorrelacionados.

É possível utilizar uma notação simplificada para calcular SCC, como sugerido por Alaghi e Hayes (2013a). A partir de dois SN X e Y, toma-se:

- a como o número de bits 1 sobrepostos entre ambos;
- b o número de bits 1 de X sobrepostos a bits 0 de Y;
- c o número de bits 0 de X sobrepostos a bits 1 Y;
- d o número de bits 0 sobrepostos entre ambos;
- n o número de bits da cadeia, ou seja, $n = a + b + c + d$;

Então, é possível calcular o coeficiente $\mathrm{SCC}(X, Y)$ utilizando.

$$\mathrm{SCC}(X,Y) = \begin{cases} \frac{ad-bc}{n\cdot\min(a+b,a+c)-(a+b)(a+c)}, \text{ se } ad > bc \\ \frac{ad-bc}{(a+b,a+c)-n\cdot\max(a-d,0)}, \text{ caso contrário} \end{cases} \tag{2.11}$$

Enquanto a correlação de Pearson é normalizada pela variância das cadeias de bits, a SCC é normalizada tal que cadeias de bits com máxima, ou mínima, sobreposição de 1s e 0s levem a $SCC = +1$ ou $SCC = -1$, respectivamente, independente dos valores dos SN. A Tabela 5 apresenta alguns exemplos de pares de SN X e Y e seus respectivos valores SCC e ρ.

Tabela 5 – Alguns SN e seus coeficientes SCC e de Pearson

Números estocásticos	$SCC(X,Y)$	$\rho(X,Y)$
X = 11110000 Y = 11001100	0	0
X = 11110000 Y = 11110000	+1	+1
X = 11110000 Y = 00001111	-1	-1
X = 11111100 Y = 11110000	1	0,58
X = 11111100 Y = 00001111	-1	-0,58
X = 11111100 Y = 11100001	0	0
X = 11000000 Y = 11111100	1	0,33

Fonte: (ALAGHI; HAYES, 2013a, p. 4). Traduzida e adaptada pelo autor.

A presença de qualquer valor de correlação diferente de zero nas entradas de um circuito estocástico tradicionalmente indicaria a existência de um erro por correlação na operação. Com a adoção da teoria de SCC, é possível caracterizar a resposta de SC mesmo sob a presença de correlação. Desta forma, ela se torna uma ferramenta de projeto, mesmo que ainda seja uma fonte de erro válida. Por exemplo, uma porta AND de duas entradas que equivale à função $X \cdot Y$ para $SCC = 0$ torna-se $\min(p_x, p_y)$, para $SCC = +1$. Há, ainda, outra classe de circuitos estocásticos chamados independentes a correlação (CI – *correlation independent*), cujas funções não são afetadas pelos valores SCC de suas entradas.

Os estudos em correlação em números estocásticos também são aprofundados com a introdução das matrizes de transferência probabilística (PTM – *Probabilistic Transfer Matrices*), introduzidas por Krishnaswamy et al. (2008) e utilizadas na teoria de SCC de Alaghi e Hayes (2013a). Entretanto, o conhecimento do conceito de SCC e sua utilidade para a geração de circuitos estocástico bastam para o escopo deste trabalho. Isso é justificado na subseção 2.2.5.

2.2.4 Fontes de erro

Um ponto negativo de utilizar Computação Estocástica é a controlabilidade dos erros em operações. Diferentemente da metodologia determinística, a abordagem probabilística possui erros inerentes à sua construção. De acordo com (ALAGHI; HAYES, 2013b) e, mais recentemente, (ALAGHI; QIAN; HAYES, 2018), as principais fontes de erro são: flutuações aleatórias nos padrões das cadeias de bits dos SN; independência insuficiente entre cadeias ou presença de correlação; e erros de aproximação e quantização.

As flutuações aleatórias vêm do fato de que um SN qualquer X, com valor p_x, representa, em cada um de seus bits, realizações independentes de um processo estocástico de Bernoulli com probabilidade p_x. Logo, o valor do número estocástico possui distribuição binomial com variância $p_x(1 - p_x)/N$, em que N é o tamanho da cadeia de bits. Sendo assim, o valor do número representado flutua, tendendo ao valor esperado p_x apenas para comprimentos N suficientemente grandes. Como mencionado por Alaghi (2015, p. 86), os geradores de números estocásticos são os principais componentes de SC que podem ajudar a minimizar erros provenientes desta fonte.

A utilização de bons SNG também reduz erros por correlação. Essa classe de erros faz com que um determinado SC não exiba em sua saída o resultado esperado de uma operação, assim como representado nas Figuras 5 e 9. Além disso, mesmo com bons geradores, a execução de operações sucessivas acaba por introduzir correlação no sistema. Uma possível solução para esse problema seria descorrelacionar os sinais de forma forçada. Para alcançar isso, pode-se converter os números em questão para a forma binária convencional e, então, novamente para a forma estocástica. Embora seja um método muito custoso em relação a número de portas lógicas e eficiência energética, ele é capaz de efetivamente eliminar esse tipo de erro (ALAGHI, 2015, p. 85 – 87).

Ao trabalhar no domínio de Computação Estocástica, funções que possivelmente estão representadas nos números reais deverão ser aproximadas para a faixa de 0 a 1 e quantizadas de acordo com o número de bits do sistema. A operação de soma, por exemplo, precisa ser substituída por uma soma escalonada, já que a soma de dois elementos do universo probabilístico, que poderia resultar em valores além dos possíveis, não pode ser (por exemplo, $1 + 1 = 1 \neq 2$).

Uma outra observação importante é que a precisão regular dos números estocásticos é muito aquém da habitual da Computação convencional. Precisões de 32 a 64 bits em ponto flutuante requereriam 2^{32} a 2^{64} bits de representação estocástica por SN representado. Além de gerarem circuitos muito grandes, o tempo de processamento total acabaria superando as vantagens oferecidas pela Computação Estocástica. Mesmo assim, circuitos estocásticos são muito paralelizáveis e a possibilidade de aumentar a velocidade de processamento de sequências tão grandes pela adoção de paralelismo não pode ser descartada.

Embora circuitos estocásticos apresentem os pontos negativos citados, é possível que eles se sobressaiam com fontes de erro presentes no mundo atual, os chamados *soft errors* (traduzidos como ruídos ou ruídos suaves), tais como no estudo caso de Li e Lilja (2011) e Qian et al. (2011b). Esses erros acontecem devido a falhas externas, muitas vezes incontroláveis durante a fase de projetos. Por exemplo, se um sistema está transmitindo uma mensagem e algum fenômeno físico altera o valor de alguns de seus bits, isso ocasionará um erro temporário, mas não caracterizará o dispositivo de transmissão como não confiável.

A vantagem de utilizar a abordagem estocástica em ambientes sujeitos a ruído está em sua inerente tolerância a eles. Isso está presente nos trabalhos de Gaudet e Rapley (2003), sobre decodificação de códigos LDPC; de Lee et al. (2017), contendo sistemas de processamento de imagem baseados em redes neurais; e de Chen, Alaghi e Hayes (2014), que analisa o desempenho dos SC em taxas de erro extremas de forma generalizada. Dois fatores que contribuem para essa robustez a ruídos aleatórios são relacionados à construção dos SN.

Primeiramente, por não serem representados de forma ponderada, uma falha em um bit qualquer de uma cadeia pode ser corrigida por uma falha em outro bit distinto, uma vez que a chance de falha em quaisquer bits é a mesma. O segundo fator se refere ao fato de que o erro de mudança de bit em um sistema convencional tem um peso associado. Isso significa que a magnitude do erro em caso de falha pode ser possivelmente catastrófica e é necessário levá-la em consideração durante o projeto. No caso estocástico, a mudança de qualquer bit implica em um erro de cálculo de $1/N$. Como os circuitos estocásticos já são idealizados para trabalhar com uma tolerância a erros dessa magnitude, eles acabam não influenciando no funcionamento do sistema de forma significativa.

2.2.5 Síntese lógica

Historicamente, a concepção de circuitos estocásticos foi abordada de forma *Ad Hoc*. Isso está presente tanto em Gaudet e Rapley (2003) quanto em Ranjbar, Salehi e Najafi (2015), o que traz consigo seu conjunto de vantagens e desvantagens. O debate sobre implementar circuitos de forma manual ou automática é semelhante ao debate de desenho de leiaute de Eriksson et al. (2003). A abordagem manual, *Ad Hoc*, possui ajuste fino de um projetista sobre como um sistema deve ser interconectado. A automática, porém,

oferece ferramentas matemáticas (algoritmos e heurísticas) que possibilitam redução do tempo de projeto até o circuito poder ser testado de fato.

Um dos assuntos mais importantes em Computação Estocástica é a síntese de funções booleanas (BFs – *boolean functions*) a partir de funções estocásticas (SFs – *stochastic functions*). Isso pode ser definido como um problema de síntese lógica.

Inicialmente, estudava-se o comportamento estocástico de portas lógicas e então associavam-se umas com as outras para definir o circuito lógico equivalente. Por exemplo, a função estocástica $F(X, Y, Z) = 0,5 \cdot (X + Y) \cdot Z$, em formato unipolar, utilizaria um bloco lógico representando soma escalonada por $0,5$ associado às entradas X e Y, seguido de uma porta AND que realizaria a multiplicação por Z. A aplicação dessa metodologia para problemas mais complexos implica analisar várias relações entre portas lógicas existentes e elementos canônicos, seguido de estudos sobre os SNG e possíveis fontes de erro.

Há duas abordagens de síntese lógica automáticas na literatura: ReSC (QIAN et al., 2011a) e STRAUSS (ALAGHI; HAYES, 2015). ReSC (*Reconfigurable architecture based on Stochastic logiC* – Arquitetura reconfigurável baseada em lógica estocástica) foi o primeiro método, baseado em aproximar funções reais em polinômios de Bernstein e, então, implementá-los em uma topologia fixa e reprogramável. STRAUSS (*Spectral TRAnsform Use in Stochastic circuit Synthesis* – Uso de transformada espectral em síntese de circuitos estocásticos) sucede ReSC em forma de extensão, baseando-se em transformadas espectrais aplicadas a funções booleanas.

Alguns dos principais estudo de transformadas espectrais aplicadas a design de circuitos digitais são os trabalhos de Hurst, Miller e Muzio (1985) e Karpovsky, Stankovic e Astola (2008). A transformada em questão é denominada transformada discreta de Walsh com ordenamento de Hadamard, dada por

$$\overrightarrow{F} = \frac{1}{2^n} H_n \times \overrightarrow{f} \tag{2.12}$$

é um caso particular da transformada de Fourier (ALAGHI, 2015, p. 40). O termo F é a transformada de Fourier em questão e f é um vetor tabela-verdade (TT – *Truth-Table*) que representa uma função booleana.

O termo H_n é definido recursivamente:

$$H_0 = [+1] \text{ e } H_n = \begin{bmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{bmatrix} \tag{2.13}$$

É possível demonstrar que uma SF pode ser expressa analogamente à forma de soma de mintermos (ALAGHI, 2015, p. 40 – 44), comum à teoria de circuitos lógicos. A soma é dada por

$$F_1(X_1, X_2, ..., X_n) = \sum_{i=0}^{2^n-1} C_i S_i \tag{2.14}$$

F_1 representa uma SF de n SN de entrada e C_i representa o espectro de f_1. Retomando a analogia a circuitos lógicos convencionais, S_i são todas as combinações lineares das entradas: $\overline{X_1X_2...X_n}$, $\overline{X_1X_2...}X_N$, etc. A forma de representação bipolar inversa é bastante relevante, pois é compatível com a aritmética requerida pela transformada de Walsh.

A título de exemplo, demonstra-se como obter a porta XOR como elemento multiplicador IBP, como representado na Figura 8, com esse conjunto de ferramentas espectral. A tabela-verdade da porta XOR é representado em forma vetorial por $f_{xor} = [+1 - 1 - 1 + 1]^T$. O cálculo de sua transformada é dado por

$$\overrightarrow{F_{xor}} = \frac{1}{4}H_2 \times \overrightarrow{f_1} = \frac{1}{4}\begin{bmatrix} +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 \end{bmatrix} \times \begin{bmatrix} +1 \\ -1 \\ -1 \\ +1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{2.15}$$

Logo, $F_{xor} = 0 \cdot 1 + 0 \cdot X_2 + 0 \cdot X_1 + 1 \cdot X_1X_2 = X_1X_2$. Esse resultado também denota o comportamento estocástico $\widehat{F}_{xor}$ de f_{xor} (ALAGHI, 2015, p. 41). Assim, ilustra-se que a transformada de Walsh representa o comportamento estocástico na forma IBP de uma dada função booleana. Isso indica que a transformada inversa de Walsh pode ser empregada em ferramentas de síntese automática para, dada uma SF $\widehat{F}$, obter uma aproximação, se necessária, F e, então, obter uma expressão booleana em forma de tabela-verdade $\overrightarrow{f}$.

O STRAUSS está disponível em forma de código-aberto em paǵina GitHub (<https://github.com/arminalaghi/scsynth/>, acessado em julho de 2019) para a plataforma Octave. Ele contém uma implementação do algoritmo ReSC, o STRAUSS original e uma variação chamada STRAUSS assimétrico.

Este trabalho adota o mecanismo de síntese automática do STRAUSS, baseado na teoria de transformadas espectrais, a fim de comparar o desempenho de um circuito detector de bordas em imagens digitais feito de forma *Ad Hoc* a um feito de forma automatizada.

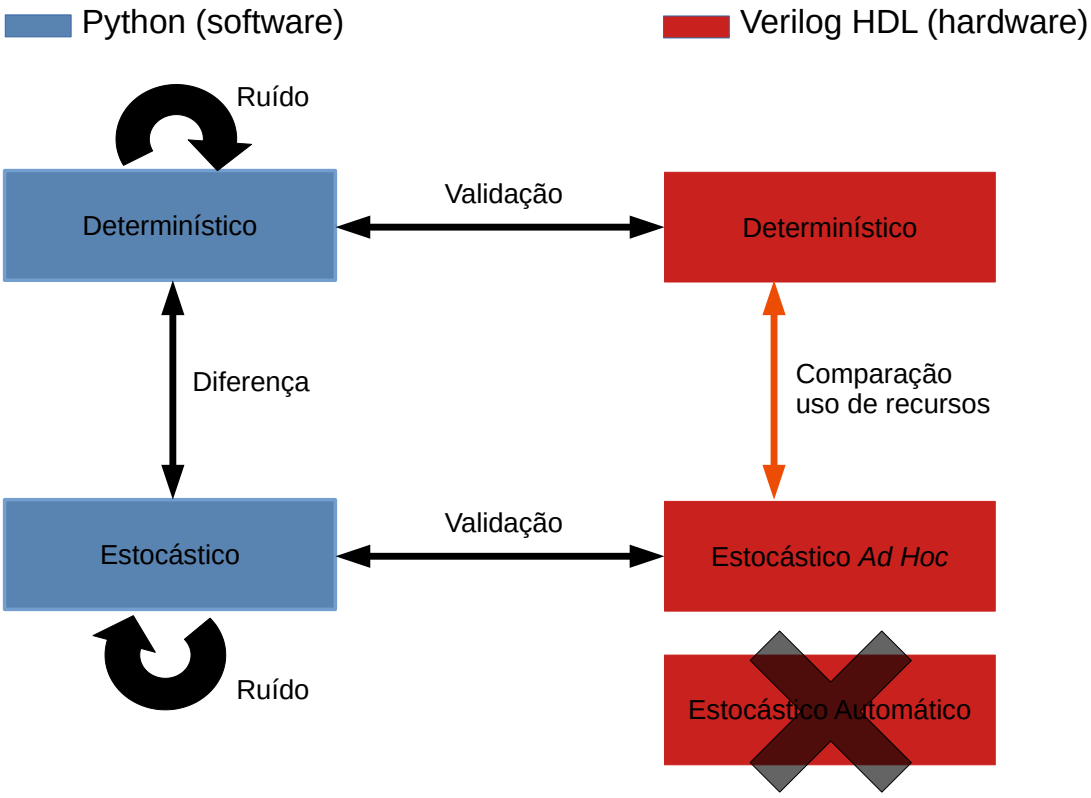
3 METODOLOGIA ADOTADA

Para alcançar os objetivos deste trabalho, a seguinte metodologia foi adotada:

1. Implementação um algoritmo de detecção de bordas por filtro de Sobel 3x3 determinístico e outro estocástico em linguagem de programação Python;
2. Execução algoritmos sobre um banco de dados e extrair imagens de bordas destacadas de referência;
3. Simulação taxas de erros de bits em ambos os sistemas e compará-los com seus resultados de referência a fim de determinar diferenças em robustez a ruído;
4. Repetição de metodologia para simular de descrição de *hardware* em Verilog, simulando em Icarus Verilog;
    - O algoritmo estocástico possui implementação manual e automática (STRAUSS).
5. Avaliação dos resultados em *hardware* com a referência obtida em *software*;
6. Adaptação conjunto de testes em Verilog para implementação em FPGA;
    - Este passo fornece uma estimativa de uso de recursos para o caso dos algoritmos aplicados como circuitos integrados dedicados.
7. Obtenção dados de utilização de recursos (portas lógicas utilizadas, potência dissipada) pelas abordagens em *hardware*;
8. Estimação da compatibilidade de cada abordagem com as tecnologias disponíveis.

A Figura 10 resume a metodologia em um diagrama. Implementações em *software* e em *hardware* foram feitas e validadas entre si. Isso possibilitou comparar diferenças de desempenho e utilização de recursos. A abordagem estocástica por síntese automática teve de ser descartada devido à impossibilidade de produzir um circuito estocástico com 8 entradas no espaço de tempo deste trabalho. Finalmente, a análise de ruído foi avaliada de forma relativa, medindo o desvio entre uma situação sem erros com outra possuindo crescente taxa de erro de bits assumindo distribuição de Bernoulli. Esses procedimentos estão detalhados nas Seções seguintes.

Figura 10 – Diagrama sintético da metodologia adotada.



Fonte: Produzida pelo autor.

3.1 Modelagem em *Software*

Python é a linguagem de programação utilizada como base para o trabalho. Ela foi escolhida devido a sua facilidade de programação e distribuição, aliadas a um grande número de pacotes para desenvolvimento. Além disso, ela propicia um ambiente gratuito e reprodutível para compartilhar resultados de desenvolvimento. Ademais, ela está sendo utilizada amplamente, por exemplo, em serviços de *Big Data* (CIELEN; MEYSMAN; ALI, 2016).

Os principais pacotes adicionais utilizados com Python 3.6.7 são:

- Bitarray;
- Matplotlib;
- NumPy;
- OpenCV;
- Ray.

O pacote *bitarray* auxilia o tratamento dos números como conjuntos de bits. Isso é particularmente útil para a simulação de taxas de erros de bit específicas e para implementar funções estocásticas.

O *matplotlib* é usado na produção de gráficos e imagens, o que ajuda a produzir figuras ilustrativas dos resultados ou manipular o carregamento e salvamento de dados.

O pacote *NumPy* oferece um ambiente para operações de álgebra linear, oferecendo várias funções pré-construídas e precisão de 64 bits de ponto flutuante comumente utilizado em outras aplicações.

OpenCV consiste de um grande esforço comunitário para implementar algoritmos de visão computacional. Ele é utilizado principalmente para fazer interface com formatos diversos de imagens e para validar os algoritmos de detecção de bordas implementados, por possuir suas próprias implementações validadas mundialmente.

Ray é um módulo de processamento paralelo que auxilia na utilização eficiente de recursos computacionais. Ele foi adotado para acelerar a simulação do circuito estocástico proposto em Python, uma vez que é uma tarefa altamente paralelizável. Como ele só está disponível para sistemas baseados em Linux e MacOS, uma implementação sem o Ray também foi feita.

Os filtros que foram implementados em *software* e *hardware* utilizam uma modificação da Equação (2.7):

$$M(x,y) = \frac{|(\frac{1}{4}(z_3+z_6+z_6+z_9)) - (\frac{1}{4}(z_1+z_4+z_4+z_7))|}{2} + \frac{|(\frac{1}{4}(z_7+z_8+z_8+z_9)) - (\frac{1}{4}(z_1+z_2+z_2+z_3))|}{2} \tag{3.1}$$

A finalidade é garantir que os resultados estarão na faixa de valores [0, 255], permitindo que sejam representados e exibidos como uma imagem de 8 bits em tons de cinza. Para cada par de coordenadas de um pixel da imagem (x, y), há uma respectiva região 3×3 (Figura 1(a)) utilizada no cálculo. Observa-se que esta implementação desconsidera as condições de borda, retornando uma imagem de dimensão $(N-2) \times (M-2)$ pixels. Isso significa, efetivamente, que a imagem gradiente possui duas linhas e duas colunas a menos que a original. Porém, essa decisão simplifica os circuitos propostos.

3.1.1 Implementação Determinística

A implementação segue a Equação (3.1) sem observações especiais quanto a execução das operações (soma, subtração, multiplicação e valor absoluto). A imagem foi carregada como matriz de pixels com número em ponto flutuante e, ao final, foi convertida para o formato de inteiro de 8 bits sem sinal, para garantir que não havia erros de aproximação e que o resultado pudesse ser transformado em uma imagem.

É natural destacar a representação numérica utilizada (ponto flutuante e, então, número inteiro, sem sinal, de 8 bits), pois erros de aproximação da aritmética computacional convencional também estão associados a essas escolhas. Como Computação Estocástica utiliza outra abordagem, optou-se por minimizar quaisquer erros computacionais na implementação determinística. Desta forma, os resultados desta versão do algoritmo servem como um ponto de referência absoluta, o que torna qualquer divergência observada no resultado estocástico como um erro da natureza dessa realização.

Ainda a fim de validar os resultados determinísticos, a Equação (2.7) foi implementada e testada contra a versão pronta da biblioteca OpenCV. Isso resultou nas mesmas matrizes de saída, com exceção do tratamento que a versão da OpenCV fornece em relação às bordas (o que produz duas linhas e duas colunas adicionais em relação à saída deste trabalho).

3.1.2 Implementação Estocástica

O programa que simula o SC associado foi baseado no trabalho de Ranjbar, Salehi e Najafi (2015), com algumas modificações. As mesmas portas lógicas são utilizadas, porém o trabalho de referência faz uso de filtros diagonais de Sobel, e este usa os verticais e horizontais. O presente trabalho também buscou aprofundar alguns detalhes de implementação que carecem explicações. Como apresentado na seção 2.2, circuitos lógicos podem desempenhar diferentes funções aritméticas de acordo com a representação utilizada e a SCC das entradas entre si. A implementação *Ad Hoc* em *software* e *hardware* utiliza dessas primitivas para produzir a Equação (3.1).

3.1.2.1 Soma Ponderada

A função estocástica de soma ponderada pressupõe representação unipolar e entradas de sinal descorrelacionadas entre si. Ela é representada por

$$Z = S_1 \cdot (X_1 + X_2) \tag{3.2}$$

e é comumente aplicada como um multiplexador 2:1, onde sua entrada de seleção representa o fator de ponderação. A análise do comportamento estocástico dessa função por PTM mostra que não há restrições de correlação entre a entrada de seleção e as demais (ALAGHI; HAYES, 2013a).

É possível estender esses resultados para uma soma ponderada com quatro termos, utilizando

$$Z = S_1 \cdot S_2 \cdot (X_1 + X_2 + X_3 + X_4) \tag{3.3}$$

que pode ser implementada por um multiplexador 4:1 com suas entradas de sinal descorrelacionadas entre si. Neste caso, a restrição das entradas de seleção é que elas são descorrelacionadas entre si, tal como seria necessário na aplicação de SC distintos.

3.1.2.2 Valor Absoluto da Diferença

É possível realizar a operação

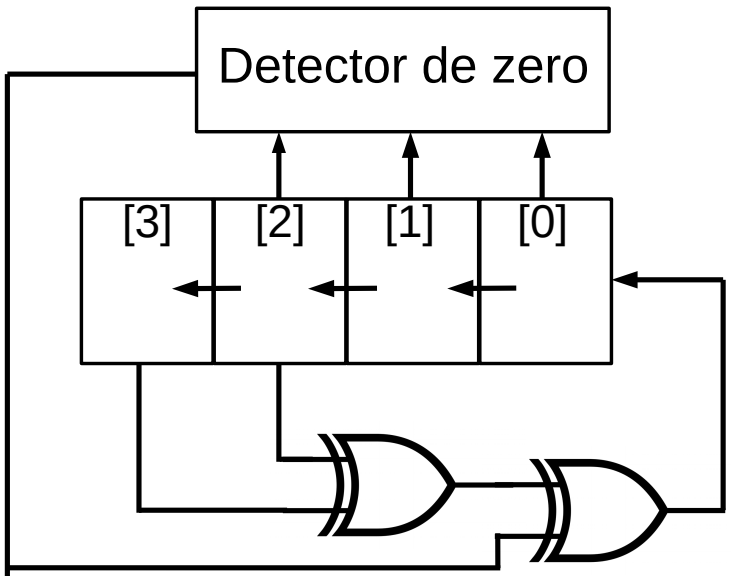
$$|X_1 - X_2| \tag{3.4}$$

com uma porta XOR com ambas as entradas com $SCC = 1$ entre si. Destaca-se a relevância desse resultado, uma vez que a consideração histórica de que deveria-se utilizar $SCC = 0$ necessita de mais portas lógicas (ALAGHI; HAYES, 2013a).

3.1.2.3 Conversor Binário-Estocástico

Os SNG necessários foram implementados com simples diretivas de comparação (tal como representado na Figura 6, tratada na subseção 2.2.1). Os geradores de números aleatórios são LFSR utilizados na ferramenta STRAUSS (ALAGHI; HAYES, 2015). A Figura 11 representa um exemplo de circuito LFSR de 4 bits tal como gerado pela ferramenta. O utilizado neste trabalho possui 8 bits, diferindo-se apenas na operação booleana realizada para definir a sequência produzida.

Figura 11 – Exemplo de LFSR de 4 bits.



Fonte: Produzida pelo autor.

Observa-se que a conversão para números nas extremidades do intervalo considerado ([0, 255])sempre incluirá erros. Ao juntar esse fato à distribuição de ocorrências de valores nas extremidades do intervalo em imagens, nota-se que esses erros de representação muitas vezes não existem, simplesmente pelo fato dos valores residirem no interior do intervalo (por exemplo: [10, 240]). Algumas soluções, tais como trabalhos no intervalo probabilístico $[0,20;\ 0,80]$, existem, mas não há consenso (ALAGHI; QIAN; HAYES, 2018).

Destaca-se o circuito conversor como fonte de erro, porém finaliza-se a discussão ressaltando que, mesmo com a representação e o algoritmo de conversão adequados, ainda haveria erros provenientes de flutuações aleatórias de acordo com a implementação do gerador de números aleatórios adotado. Logo, os esforços são direcionados para minimizar erros nos geradores de números aleatórios.

3.1.2.4 Conversor Estocástico-Binário

Essa operação é feita por um acumulador simples. Um circuito estocástico fornece apenas um bit de saída por resultado de operação. Logo, a obtenção do número binário correspondente a um SN é feita ao acumular (somar) esse bit de saída a uma variável (registrador, no caso de *hardware*) que guarda o valor total. Ao fim de toda a cadeia ser processada, a saída do acumulador terá o valor binário correspondente.

3.1.2.5 Filtro de Sobel

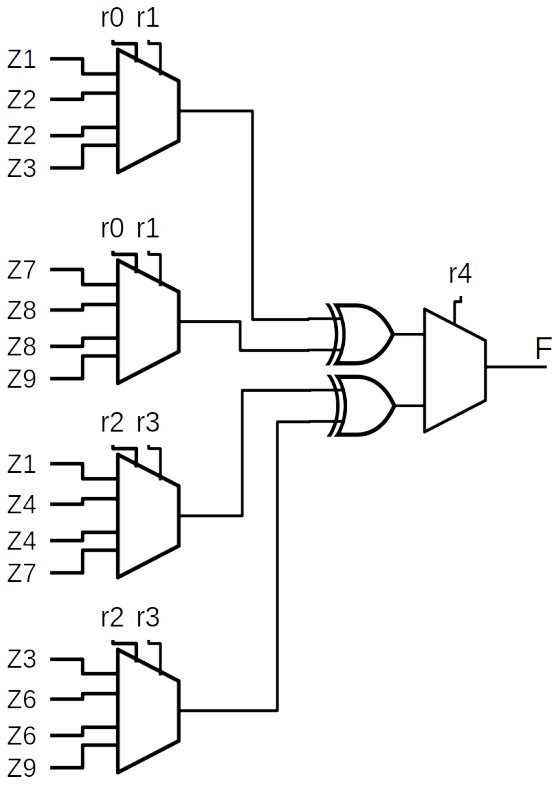
O circuito que implementa a Equação (3.1) está representado na Figura 12. Para cumprir com os requisitos de correlação das portas XOR, os multiplexadores 4:1, em pares, possuem entradas correlacionadas entre si. Por exemplo, $r0$, $r1$, $Z1$, $Z2$, $Z2$ e $Z3$ possuem SNG descorrelacionados entre si, mas, respectivamente, correlacionados ($SCC = 1$) às entradas $r0$, $r1$, $Z7$, $Z8$, $Z8$ e $Z9$ do multiplexador relacionado à mesma porta XOR. Nota-se que os SNG de um par são descorrelacionados do outro par. Também observa-se que qualquer entrada repetida, embora represente o mesmo número estocástico ($Z2$ ou $Z8$, por exemplo), utiliza um SNG distinto. Ou seja, são utilizados 13 SNG diferentes e descorrelacionados entre si.

O estudo de referências bem-sucedidas para circuitos estocásticos com múltiplas variáveis apresenta pouca ou nenhuma demonstração da garantia sobre uma correlação nula ($SCC = 0$) entre várias entradas. Esse tópico, em especial, é um dos correntes desafios para o crescimento de Computação Estocástica (ALAGHI; QIAN; HAYES, 2018).

Este trabalho contribuiu com o sistema de síntese automática STRAUSS, em repositório público (disponível em <https://github.com/arminalaghi/scsynth/>, acessado em julho de 2019), e utiliza a abordagem aceita por sua metodologia: utilizar a função de pareamento de Cantor (SZUDZIK, 2006) para produzir números iniciais únicos em cada LFSR de cada SNG.

Cada entrada ou saída do circuito da Figura 12 representa apenas um bit de sua respectiva cadeia estocástica. Isso significa que a operação de detecção de bordas requer um processo repetitivo para obter 255 entradas e transformá-las em 255 bits de saída em F. Ao final das iterações, a acumulação de todos os valores obtidos em F representam o valor do pixel equivalente à região 3×3 considerada.

Figura 12 – Diagrama do circuito estocástico que aplica as máscaras de Sobel sobre uma região de pixels 3x3 e resulta em 1 bit que compõe o número estocástico F.



Fonte: Produzida pelo autor.

3.2 Modelagem em *Hardware*

Uma das principais áreas de aplicação deste trabalho é em Microeletrônica. As primeiras iniciativas para fazer projetos circuitos digitais utilizando ferramentas CAD eram por meio de desenhos personalizados. Um projetista de circuito desenhava representações de elementos condutores, semi-condutores e resistivos a fim de replicar seu circuito esquemático (SEDRA et al., 2016). Com os avanços da Integração em Larga Escala (*Very Large Scale Integration* – VLSI), os projetos não poderiam mais ser realizados manualmente devido à grande demanda de tempo atribuída à presença de milhões – a bilhões – de transistores em apenas um circuito, além da grande dimensionalidade da análise da influência de vários componentes entre si (DENNING; LEWIS, 2017). Logo, ferramentas de síntese de circuitos e geração de leiaute automáticas foram elaboradas e amplamente utilizadas. Esse mesmo histórico é visto na realidade de circuitos estocásticos.

Os primeiros projetos utilizando Computação Estocástica baseavam-se em arranjos de portas lógicas manualmente a fim de produzir as funções booleanas que produzissem suas respectivas funções estocásticas de interesse (GAINES, 1967), (GAUDET; RAPLEY, 2003), (RANJBAR; SALEHI; NAJAFI, 2015). Entretanto, esse método se torna cada vez menos eficiente à medida que funções mais complexas são adotadas. Como o problema em

estudo de detecção de bordas não alcança níveis elevados de complexidade, duas opções são consideradas:

1. realizar o projeto de forma *Ad Hoc*, com um circuito especializado tal como o de Ranjbar, Salehi e Najafi (2015);
2. utilizar uma ferramenta automática de síntese como o STRAUSS, que possui embasamento matemático para chegar a soluções ótimas ou sub-ótimas.

Em ambos os casos, a linguagem de descrição de *hardware* (HDL) Verilog pode ser usada para descrever componentes lógicos e suas interconexões. Em seguida, ferramentas de simulação verificam o correto funcionamento das topologias obtidas.

A abordagem determinística é feita sem observações especiais. Os algoritmos de síntese em Verilog, de forma geral, conseguem compreender instruções com todas as operações aritméticas básicas (adição, subtração, multiplicação e divisão). Isso possibilitou uma implementação muito próxima à em *software*.

3.2.1 Implementação Determinística

O módulo, feito em Verilog, recebe 8 entradas com 8 bits de tamanho, a fim de receber os valores dos pixels da região necessária para o cálculo (excluindo o pixel central, desnecessário). O circuito é síncrono, contando com entradas auxiliares para *clock* e *reset*. A saída também é em 8 bits.

O circuito implementado foi validado por meio de um teste de unidade em conjunto com a implementação em Python. Uma mesma imagem de entrada produziu uma matriz para cada metodologia, e ambas possuíram exatamente os mesmos valores.

3.2.2 Implementação Estocástica

Este módulo segue as mesmas observações apontadas na subseção 3.1.2. A maior divergência se refere à utilização de uma máquina de estados para representar o laço da versão em Python. Esta máquina de estados baseia-se na descrição de *hardware* gerada pelo STRAUSS. A Figura 13 representa o diagrama de estados da versão final do projeto. Há três estados: *IDLES* (ocioso), *INITS* (início) e *RUNNINGS* (executando). Os nomes remetem a significados em inglês, com adição da consoante "S" indicando estado (*state*).

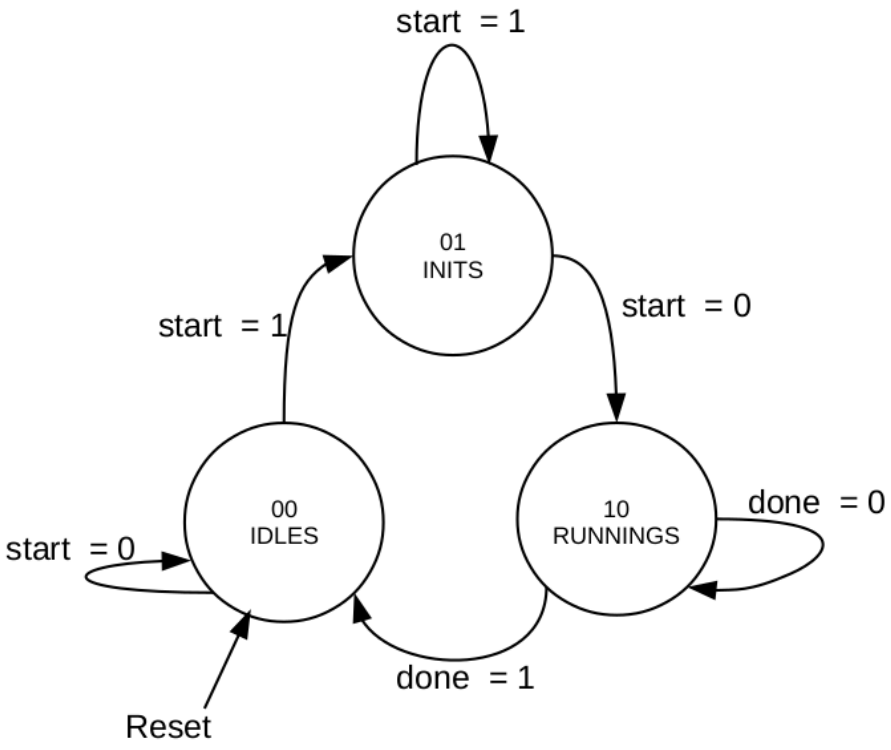
Uma vez iniciado o sistema, é enviado um sinal *reset* para estabelecer condições iniciais a todo o circuito, incluindo a máquina de estados. Quando há uma nova imagem a ser processada, a atribuição de um bit 1 ao registrador *start* sinaliza o início um novo ciclo de processamento. No estado *INITS*, todas as grandezas estocásticas são reiniciadas (LFSR, entradas e saídas). Nesta situação, os pixels da região 3×3 a ser processada devem

ser mantidos até a máquina retornar ao estado ocioso. O retorno do registrador *start* ao valor 0 inicia o processamento, fazendo com que o novo estado seja *RUNNINGS*.

Nesse estado, todas as entradas (8 pixels e 5 constantes) passam por um passo de conversão binário-estocástico. Ou seja, para cada entrada, é associado um valor 0 ou 1 a seu equivalente estocástico de acordo com o seu valor e o do seu LFSR associado. Cada bit estocástico é conectado ao circuito de processamento, que é puramente combinacional. Este núcleo é a parcela do circuito responsável por fornecer a cadeia de bits de saída, que são representam um pixel de 8 bits ao fim do processo. A cada repetição, os LFSR são conjuntamente incrementados em um passo.

Como um circuito estocástico opera em forma de cadeia, o estado *RUNNINGS* se repete por 255 intervalos de tempo (pulsos de *clock*). Os bits de saída durante cada uma das repetições são acumulados em um registrador de 8 bits. Quando a última repetição é realizada, um registrador *done* sinaliza o fim da operação, assumindo o valor 1. Isso também torna o valor do acumulador válido para ser utilizado pelo módulo principal como valor calculado do pixel da imagem gradiente para a região considerada. No intervalo seguinte, *done* retorna ao valor 0, o que leva a máquina novamente ao estado ocioso (*IDLES*).

Figura 13 – Diagrama de estados para circuito estocástico que realiza a operação de detecção de bordas em imagens 3×3.



Fonte: Produzida pelo autor.

Embora também existam estruturas de laço em HDL, elas não representam *hardware*

sintetizável e são utilizadas apenas em ambientes de simulação. Logo, a implementação estocástica proposta pode ser validada e sintetizada em *hardware*.

Analisar o circuito projetado revela três partes essenciais: geradores de números estocásticos, função estocástica e máquina de estados.

A primeira parte minimiza erros provenientes de correlação. Como também pode representar uma parcela considerável de circuito (tantos SNG quanto necessário), é a principal região a ser observada para projetar um circuito de forma eficiente.

A segunda parte realiza a função matemática considerada. Ela representa o maior ganho direto de eficiência de recursos ao se considerar Computação Estocástica, em vez da abordagem convencional. Os circuitos de detecção de bordas, por exemplo, dispensam elementos multiplicadores e somadores de 8 bits em troca de multiplexadores e portas lógicas básicas, como aborda o trabalho de Ranjbar, Salehi e Najafi (2015).

A terceira parte representa a latência do SC. O circuito deve produzir bits estocásticos tanto quanto for necessário para a precisão considerada. Isso significa que a temporização do circuito é definida pelo tempo requerido pela parte combinacional para realizar a função estocástica multiplicado pelo número que vezes que essa operação precisa ser efetuada. Observa-se que não há necessidade de processar toda a cadeia estocástica por uma única máquina de estados. Entretanto, isso impõe outros problemas como maior utilização de área e potência, além da necessidade de duplicar os SNG de forma a manter as considerações de correlação entre implementações paralelas. Um caso extremo dessa observação é a eliminação completa da necessidade da máquina de estados. Se o circuito estocástico somado aos de conversão puderem efetuar paralelamente todos os bits da cadeia estocástica, a latência do circuito será definida somente pela latência desses elementos componentes. Novamente, a desvantagem está em gerenciar um número potencialmente elevado de geradores de números estocásticos e minimizar os erros por correlação do conjunto.

Foram consideradas duas abordagens para projetar o circuito que realiza a função estocástica.

3.2.2.1 Abordagem *Ad Hoc*

O circuito e parâmetros utilizados na modelagem em *software* são aplicados de forma direta, pois ela é baseada em elementos lógicos (neste caso multiplexadores e portas XOR). Os LFSR utilizados para os 13 SNG necessários seguem a mesma construção do STRAUSS.

Semelhantemente ao caso determinístico, os resultados foram validados com a implementação em Python.

3.2.2.2 Abordagem Automática

Buscou-se utilizar as observações feitas na Subseção 2.2.5. O código do repositório denominado *scsynth*, disponível em <https://github.com/arminalaghi/scsynth/> (visitado em julho de 2019) apresenta duas abordagens para síntese automática de circuitos estocásticos: ReSC e STRAUSS. A primeira baseia-se em aproximar funções matemáticas a polinômios de Bernstein e escrever estruturas fixas em Verilog que reproduzam esses resultados. A segunda utiliza a teoria de transformadas espectrais para obter um conjunto de funções lógicas que implemente uma função matemática. A ferramenta utiliza o programa Octave.

A ferramenta foi estudada e notou-se que o tempo para produzir circuitos automaticamente elevara-se abruptamente com um aumento no número de entradas consideradas. Também observou-se limitações. O algoritmo STRAUSS só havia sido implementado para funções com uma variável (polinômios, por exemplo), e, mesmo se houvesse sido para múltiplas, possuiria tempo de processamento proibitivo.

Foi experimentada a utilização da abordagem MReSC (*Multivariable Reconfigurable architecture based on Stochastic logiC*). Porém, mesmo com ajustes de parâmetros, o tempo para gerar um circuito aceitável ainda mostrou-se proibitivo. Por exemplo, um computador operando a 4,2 GHz esteve em execução durante duas semanas e não terminou a operação.

Foram feitas contribuições ao projeto a fim de contribuir para este trabalho, fornecendo uma fonte alternativa de comparação para circuitos estocásticos, e para trabalhos futuros. Uma delas foi realizar o perfilamento do código e destacar as áreas mais lentas. A partir disso, identificou-se a possibilidade de reduzir em, no mínimo, quatro vezes o tempo de processamento do algoritmo MReSC ao utilizar processamento paralelo. Mesmo com essa contribuição, a possibilidade de aceleração não está disponível em todos os sistemas operacionais (apenas Linux e MacOS). Ademais, constatou-se a existência de um comando sem possibilidade de paralelismo e que contribui com boa parcela do tempo de processamento total do algoritmo. Algumas outras contribuições foram integradas ao repositório público, porém concluiu-se não ser aplicável utilizar o algoritmo para funções de múltiplas variáveis, mesmo que simples.

Há uma metodologia de síntese de circuitos estocásticos alternativa em desenvolvimento, que resolve muitos dos problemas da área (tempo de processamento, aproximação da solução, geração de SNG), porém não há nenhuma ferramenta disponível ao público para utilizá-la (LEE et al., 2018).

3.3 Conjuntos de Testes

A metodologia para testar ambas as abordagens é idêntica. Algumas imagens são carregadas e processadas por todas as quatro alternativas. Em seguida, comparam-se os resultados de Python contra os de Verilog, respectivamente, a fim de garantir que as implementações se correspondem. A forma de carregar imagens utiliza diretivas de alto nível, como uma função pronta de carregar imagens da biblioteca OpenCV. Entretanto, os trechos referentes a processamento, que eventualmente podem se tornar circuitos, foram ser escritos em código sintetizável. Foram usadas as imagens marcadas como *aerial* da base de Bowyer, Kranenburg e Dougherty (2001).

As versões em *software* utilizam os *front-ends* disponíveis nos Apêndices C e D O código utilizado para simular a implementação em Verilog estão disponíveis nos Apêndices E e F. Os parâmetros referentes ao tamanho da imagem a ser processada e o arquivo a ser lido são definidos a priori.

Uma vez que os testes de unidade foram bem-sucedidos, foi possível introduzir *soft errors* na saída das funções em Python, por simplicidade, para analisar a robustez a ruído do sistema.

Todos os códigos-fonte necessários para reproduzir, validar e identificar eventuais falhas nos testes estão disponíveis em repositórios públicos nas plataformas GitHub e GitLab (<https://github.com/danilo-bc/edge-detect> e <https://gitlab.com/danilo-bc/edge-detect>, acessados em julho de 2019).

3.4 Características Físicas dos Circuitos

Ao garantir que a metologia estocástica produz circuitos que realizam a função matemática prevista, é possível estudar as diferenças nas implementações físicas de cada um. Buscou-se obter a utilização de recursos em três cenários: circuito determinístico, circuito estocástico com circuitos auxiliares conversores, e circuito estocástico puro. Essa distinção entre os dois últimos se dá a partir do dado de que os circuitos auxiliares aos SC (LFSR, SNG, B2S etc.) compõem cerca de 80% da área total de circuito (e, por conseguinte, também cerca de 80% do consumo de energia quando em operação). O trabalho de Lee et al. (2017), por exemplo, sugere economizar tais recursos ao dispensar circuitos de conversão entre domínios.

O grande problema de obter esses dados é o fato de que as tecnologias para fabricar circuitos integrados são proprietárias e requerem contratos e parcerias com empresas do ramo. Buscou-se alternativas educacionais que pudessem ser atemporais, mas todas estão vinculadas a contratos. Como a Universidade Federal da Paraíba possui placas para desenvolvimento em FPGA da Intel/Altera, optou-se por esse caminho.

Ao utilizar tais placas, é possível obter arquivos que auxiliam na síntese para FPGA. Embora os algoritmos para síntese em FPGA não sejam exatamente os mesmos de síntese para circuitos integrados, acredita-se poder comparar os resultados de ambas as implementações, determinística e estocástica, uma vez que estarão sob as mesmas condições. Destaca-se a ressalva de que circuitos para aritmética convencional podem acabar sendo otimizados internamente pelas ferramentas, o que pode não ser verdade para SC.

O número de elementos lógicos (unidade genérica do FPGA) utilizados, assim como as características de potência consumida durante processamento, serão obtidos pelas ferramentas de relatórios internos do programa Quartus II 12.1 sp1 da Intel. A síntese foi direcionada para o FPGA *Cyclone II*, relacionado à placa DE-02.

3.5 Métricas de Desempenho

Inicialmente, considerou-se avaliar a robustez a ruído de ambos os sistemas utilizando técnicas tradicionais de aferição de desempenho para algoritmos de processamento de imagem. Uma imagem adulterada com ruído do tipo sal e pimenta, por exemplo, processada pelo algoritmo determinístico seria avaliada de acordo com a diferença para o resultado obtido a partir da imagem sem alterações (VERMA; ALI, 2013). Entretanto, a maior contribuição de Computação Estocástica está em sua aplicação em um ambiente com certa taxa de erro de bits. Ou seja, os erros precisam ser simulados a nível de bits, em vez de pixels.

Imagens-resultado respectivas para cada implementação são comparadas umas com as outras por cálculo de erro absoluto médio (MAE):

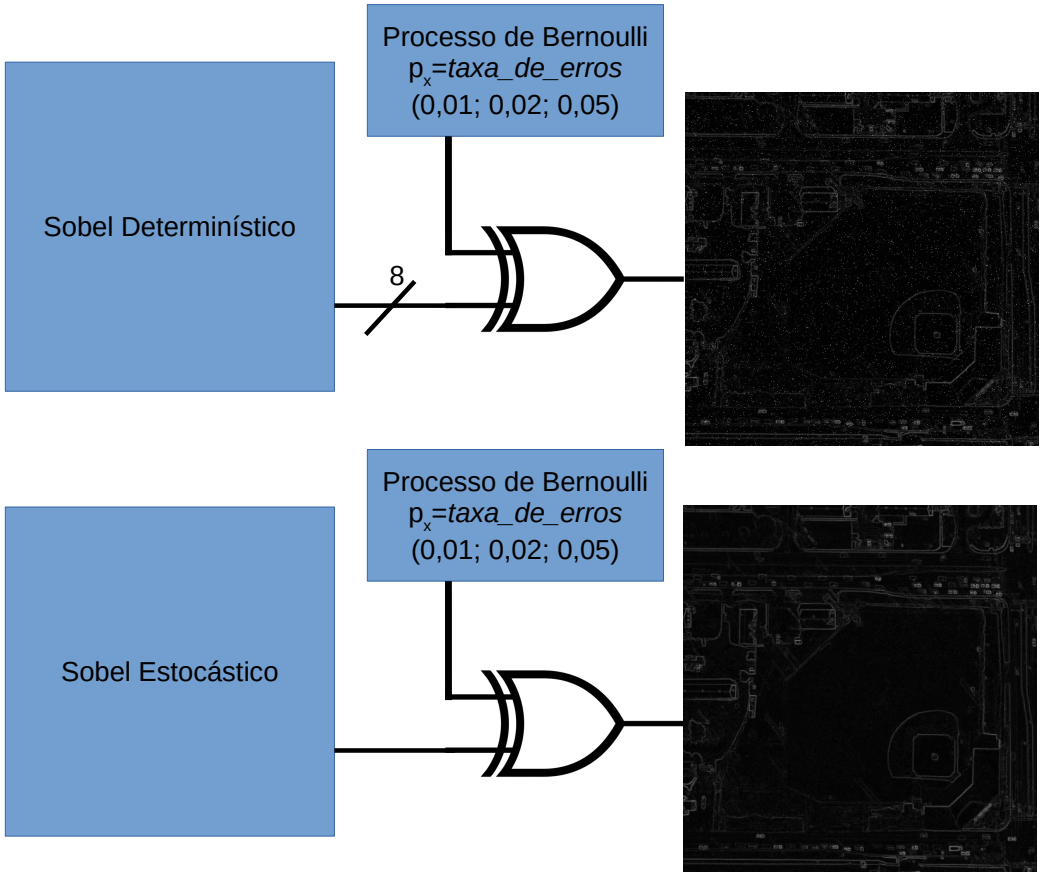
$$MAE = \frac{\sum_{i=0}^{N} |F_{det_i} - F_{stoch_i}|}{N} \tag{3.5}$$

Essa análise representa o erro proveniente da adoção da metodologia estocástica.

Em seguida, são introduzidos *soft errors* em cada bit de cada pixel-resultado. Isso é feito em Python com um parâmetro de função que conecta cada bit de saída a uma função XOR de duas entradas. Uma das entradas recebe a saída real, enquanto a outra recebe uma realização de um processo de Bernoulli, representando o erro. Isso está representado na Figura 14. Cada um dos 8 bits determinísticos, por pixel, é operado por uma porta XOR individualmente, a Figura 14 apenas simplifica a representação. Analogamente, cada um dos 255 bits estocásticos que compõem cada pixel da imagem-resultado são avaliados. Nota-se que o erro produzido pelo circuito determinístico varia de acordo com a posição do bit afetado, enquanto que o estocástico sempre erra 1/256, independente da posição do bit. Ademais, erros aos pares, em Computação Estocástica, podem cancelar a si mesmos. A desvantagem explícita nessa comparação é que há menos bits determinísticos em relação

a estocásticos. Isso significa que uma taxa de erro de bits percentual afeta mais bits na implementação estocástica, mesmo que o efeito seja menor.

Figura 14 – Introdução de ruídos suaves para análise de robustez a ruído.



Fonte: Produzida pelo autor.

Essa abordagem é semelhante à presente em Alaghi (2015, p. 90 – 97), porém, no trabalho em questão, portas XOR são introduzidas nos vários estágios do circuito, inclusive a entrada. Como a alteração dos pixels de entrada vai contra as premissas adotadas, espera-se obter resultados semelhantes, porém não idênticos. A conclusão do trabalho de referência é que o erro do circuito estocástico é inicialmente maior (para taxas de erro de bit menores a 0,01%), mas cresce mais lentamente que o do circuito determinístico. Isso ocorre até cerca de 2%, onde os erros calculados são muito próximos, o que levou à conclusão de que ambos os circuitos estão operando em ruído, e não sinal. Essa observação significa que os circuitos estocásticos devem permitir uma relação sinal-ruído mais estreita que o convencional.

Observa-se que essa abordagem leva em consideração um erro com distribuição de Bernoulli. Os ruídos suaves, que Computação Estocástica possui robustez, podem ser modelados como ruídos de canal. Desta forma, essa metodologia não trata de erros grosseiros, como uma imagem de entrada ruidosa a ser restaurada, e sim de erros que podem acontecer por falhas eletromagnéticas de acordo com a propensão a ruídos do

ambiente no qual o sistema está operando, ou do canal de transmissão de seus dados, após cada operação. Uma análise mais extensa sobre ruídos suaves em Computação Estocástica está presente no trabalho de Chen, Alaghi e Hayes (2014).

A partir dessas observações, foram simuladas taxas de erros de bit de 1%, 2% e 5% a fim de confirmar essas observações. Os resultados de cada uma das respectivas abordagens foram comparado aos seus casos sem erro para cada imagem. A grandeza erro quadrático médio (MSE), dada por

$$MSE = \frac{\sum_{i=0}^{N}(F_{ref} - F_{erro})^2}{N} \tag{3.6}$$

Essa análise auxilia a análise por MAE ao prover uma estimativa da variância do erro.

Como a influência de um erro de bit estocástico é menor do que a de um erro convencional (devido à ponderação dessa abordagem), espera-se que a variância do erro (estimada por MSE) seja menor. Entretanto, o aumento da quantidade de bits utilizados na operação estocástica pode se contrapor às suas vantagens intrínsecas. Ou seja, há a distinção entre, possivelmente, vários erros numerosos, da abordagem estocástica, contra poucos erros de alto valor, da abordagem determinística.

4 RESULTADOS

4.1 Desempenho

O algoritmo de detecção de bordas em imagens utilizando operador de Sobel convencional opera de acordo com a referência (OpenCV). Isso é verdade tanto em Python quanto em Verilog. Os resultados de desempenho comparativo dessa abordagem com a estocástica para o banco de images utilizado estão apresentados na Tabela 6. O MAE obtido em cada caso possui ordem de grandeza representando dois bits de precisão. Isso indica utilidade desta implementação em aplicações que permitem erros com limite superior igual a dois bits.

Tabela 6 – Erro absoluto médio (MAE) para o conjunto de imagens considerado.

Imagem	MAE	Imagem	MAE
airfield	1.857	baseball	2.257
buildings	2.894	homes	3.342
largebuilding	3.395	mainbuilding	2.905
pool_tennis	2.253	school	2.604
series	2.244	woods	3.247

4.2 Robustez a Ruídos

A análise de robustez a ruídos apresenta o MAE entre os resultados originais e após inserção de taxas de erro de bit. Foram testadas as taxas 1%, 2% e 5%. Os resultados determinísticos estão apresentados na Tabela 7, e os estocásticos na Tabela 8.

Em todas as situações testadas, a média dos erros calculados se diferencia na mesma ordem de grandeza, nos décimos, sendo o erro estocástico sempre menor. Isso reforça a hipótese de que o número de bits utilizados na abordagem estocástica é relevante quando há um erro de taxa de bit percentual. Espera-se que existam muito mais erros na versão estocástica. Entretanto, erros aos pares que ocorrem em um mesmo SN são cancelados. De forma contrastante, a versão convencional possui menos erros, porém não existe a possibilidade de cancelamento de erros, e eles podem ser mais significativos (uma mudança de bit mais significativo, por exemplo).

É possível combinar os comentários anteriores com a análise dos MSE obtidos. As Tabelas 9 e 10 apresentam, respectivamente, os resultados determinístico e estocástico. Observa-se que, desta vez, os resultados diferem em uma ordem de grandeza. Isso significa

Tabela 7 – Erro absoluto médio (MAE) para análise de robustez a ruídos do circuito determinístico.

Imagem	MAE (1%)	MAE (2%)	MAE (5%))
airfield	2.50	5.07	12.53
baseball	2.56	5.09	12.58
buildings	2.51	5.09	12.53
homes	2.54	5.10	12.58
largebuilding	2.53	5.06	12.56
mainbuilding	2.55	5.04	12.49
pool_tennis	2.52	5.02	12.53
school	2.53	5.02	12.61
series	2.57	5.10	12.58
woods	2.49	5.09	12.56
Valor médio	2.53	5.07	12.55

Tabela 8 – Erro absoluto médio (MAE) para análise de robustez a ruídos do circuito estocástico.

Imagem	MAE (1%)	MAE (2%)	MAE (5%))
airfield	2.43	4.84	11.57
baseball	2.37	4.71	10.80
buildings	2.33	4.61	11.70
homes	2.32	4.57	10.70
largebuilding	2.32	4.57	11.20
mainbuilding	2.34	4.67	11.70
pool_tennis	2.36	4.68	12.27
school	2.36	4.66	12.48
series	2.33	4.61	12.21
woods	2.31	4.56	11.79
Valor médio	2.35	4.64	11.64

que a distribuição de erro da abordagem convencional é mais dispersa do que a estocástica. Isso também implica maior facilidade em corrigir os erros da versão estocástica. A Figura 18 auxilia a análise. Mesmo com MAE próximos, nota-se que as imagens estocásticas se aproximam da imagem-objetivo, subjetivamente.

4.3 Utilização de Recursos Físicos

O número de elementos lógicos de cada implementação foi obtido ao compilar os projetos utilizando a ferramenta Quartus II 12.1 sp1 da Intel/Altera. A estimativa de potência consumida durante o funcionamento dos módulos ocorreu no mesmo ambiente,

Tabela 9 – Erro quadrático médio (MSE) para análise de robustez a ruídos do circuito determinístico.

Imagem	MSE (1%)	MSE (2%)	MSE (5%))
airfield	216.66	450.70	1172.66
baseball	224.98	454.72	1178.86
buildings	217.45	451.70	1171.39
homes	220.72	452.51	1177.94
largebuilding	220.40	449.91	1175.74
mainbuilding	221.83	449.14	1168.74
pool_tennis	218.68	442.98	1174.60
school	221.72	446.37	1183.18
series	224.14	455.03	1182.51
woods	217.47	455.40	1178.68
Valor médio	220.40	450.85	1176.43

Tabela 10 – Erro quadrático médio (MSE) para análise de robustez a ruídos do circuito estocástico.

Imagem	MSE (1%)	MSE (2%)	MSE (5%))
airfield	8.40	28.43	177.83
baseball	8.10	27.30	143.08
buildings	7.86	26.38	159.54
homes	7.81	25.99	138.26
largebuilding	7.82	26.05	152.02
mainbuilding	7.92	26.74	165.70
pool_tennis	8.05	27.00	174.75
school	8.03	26.90	184.20
series	7.87	26.42	169.25
woods	7.75	25.99	162.89
Valor médio	7.96	26.72	162.75

utilizando a ferramenta *PowerPlay Power Analyzer* com configurações padrão. Adicionalmente, foram isolados os componentes puramente estocásticos do circuito proposto. Isso permite estimar a vantagem em utilização de recursos ao dispensar o domínio binário convencional e, com isto, os circuitos conversores. Os resultados estão exibidos na Tabela 11.

Observa-se que a implementação estocástica exibiu, de fato, melhores resultados quanto a utilização de recursos físicos. A estocástica utiliza 42,86% de recursos lógicos e 98,78% de potência, ambos em relação à determinística. Entretanto, esse resultado está aquém do estimado pela literatura, especialmente em questão de potência. A situação

Tabela 11 – Características físicas de ambas as implementações do filtro de Sobel e razão entre os resultados.

Implementação	Est.	Det.	Est. Pura	Est./Det. (%)	Est. P./Det. (%)	Est. P./Est. (%)
Número de Elementos Lógicos	15	32	7	42,86	21,88	46,67
Potência Total Dissipada (mW)	164,12	166,14	161,73	98,78	97,35	98,54

é melhor ao analisar a abordagem hipotética estocástica pura frente à determinística. Utilizam-se 21,88% do número de elementos lógicos e 97,35% da potência. Finalmente, nota-se a alta utilização de recursos por parte dos circuitos conversores da implementação estocástica. A análise relativa das implementações com ausência e com presença desses circuitos destaca que aproximadamente metade do circuito (46,67%) é voltado para circuitos que desempenham a função numérica de interesse, enquanto que o resto destina-se a SNG e B2S.

4.4 Associação de Resultados e Aplicabilidade

Os resultados de desempenho associados aos de utilização de recursos permitem estabelecer diretrizes para a escolha de abordagem para criar um circuito integrado. Listam-se as seguintes vantagens e desvantagens a serem levadas em consideração:

- Circuito determinístico
  - **Desempenho:** é referencial adotado cientificamente em relação ao Filtro de Sobel 3×3. Imprescindível se a aplicação requerer uma resposta exata;
  - **Robustez a ruídos suaves:** inadequado para ambientes ruidosos. Requer circuitos ou códigos auxiliares para mitigar erros;
  - **Utilização de recursos físicos:** ideal para situações em que não há restrições a serem consideradas.
- Circuito estocástico
  - **Desempenho:** relevante em situações que permitem respostas aproximadas ou um erro máximo por operação. Também adequado para operações em faixa;
  - **Robustez a ruídos suaves:** adequada para ambientes ruidosos. Sua distribuição de erro possivelmente permite sua utilização sem circuitos auxiliares para mitigação de erros;

– **Utilização de recursos físicos:** adequado quando há restrições em termos de utilização de recursos físicos, mas também há flexibilidade quanto à exatidão do resultado do circuito.

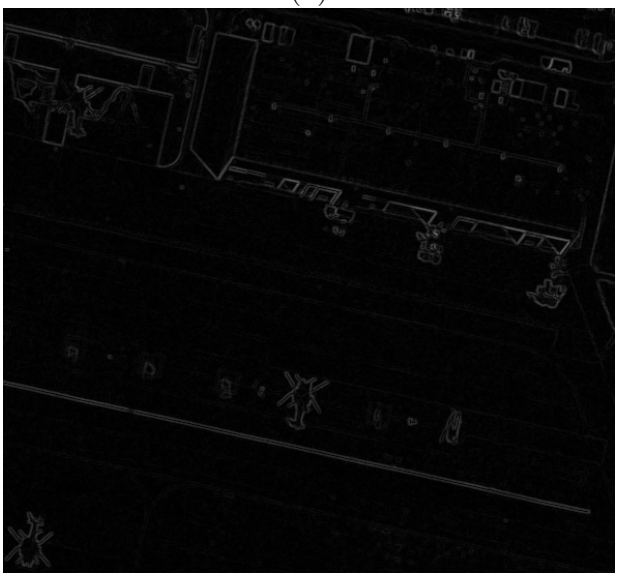
Destaca-se a importância de haver futura uma análise de utilização de recursos a nível de circuito integrado. A placa para desenvolvimento em FPGA fornecida (DE-2), por exemplo, possui níveis-base de consumo de potência que não são facilmente isoláveis. Um exemplo desse efeito, presente na Tabela 11, está na diferença entre abordagens ser da ordem de mW, enquanto o valor reportado ser de centenas de mW. Outra motivação de buscar uma análise como circuito integrado é a interferência do algoritmo de síntese para FPGA. Como os programas de síntese são otimizados de acordo com sua aplicação, as conexões já presentes no FPGA e seu design interno influenciam no número de elementos lógicos utilizados.

A metodologia deste trabalho permitiu criar um repositório público para validar os resultados e oferecer possibilidade de colaborações futuras. O repositório *edge-detect* (<https://github.com/danilo-bc/edge-detect> ou <https://gitlab.com/danilo-bc/edge-detect>, acessados em julho de 2019) possui as implementações e conjuntos de testes utilizados neste trabalho. Como vários trabalhos usados como referência trazem apenas expressões matemáticas ou pseudocódigos, acredita-se que a existência de implementações publicamente disponíveis em Python e Verilog contribuam com reprodutibilidade. Esse fator é importante tanto como base científica para estudos derivados, quanto para facilitar a confirmação de diversas premissas teóricas.

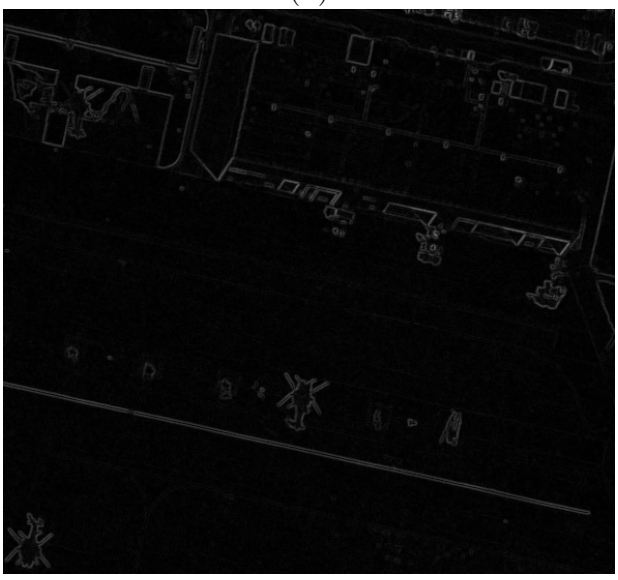
Adicionalmente, foi possível colaborar no repositório em que o trabalho de Alaghi e Hayes (2015) está disponível (<https://github.com/arminalaghi/scsynth/>, acessado em julho de 2019). A colaboração se deu na forma de revisão de código e proposta de funcionalidades. Ao estudar e aplicar os fundamentos de Computação Estocástica, assim como na utilização do repositório *scsynth*, foram destacadas falhas de implementação que levavam a erros ou a um funcionamento lento no programa de síntese automática. As contribuições foram revisadas, avaliadas e incorporadas pelo administrador do repositório. Entre outras mudanças, destaca-se a adoção da função de pareamento de Cantor (SZUDZIK, 2006) para síntese de MReSC, e a introdução do pacote de processamento paralelo, que reduziu o tempo de síntese de circuitos estocásticos de 4 a 6 vezes, em média. Esse resultado não afeta o circuito sintetizado diretamente, mas contribui com o fluxo de projeto ao diminuir o tempo necessário para sintetizá-lo. Destaca-se que esse resultado não pôde ser diretamente aplicado a este trabalho devido ao seu grande número de entradas (8 pixels), o que aumentou consideravelmente o espaço de solução do algoritmo de síntese, então não houve nenhum circuito gerado pelo procedimento.

Figura 15 – (a) Imagem “airfield” e resultados (b) determinístico e (c) estocástico.

(a)



(b)

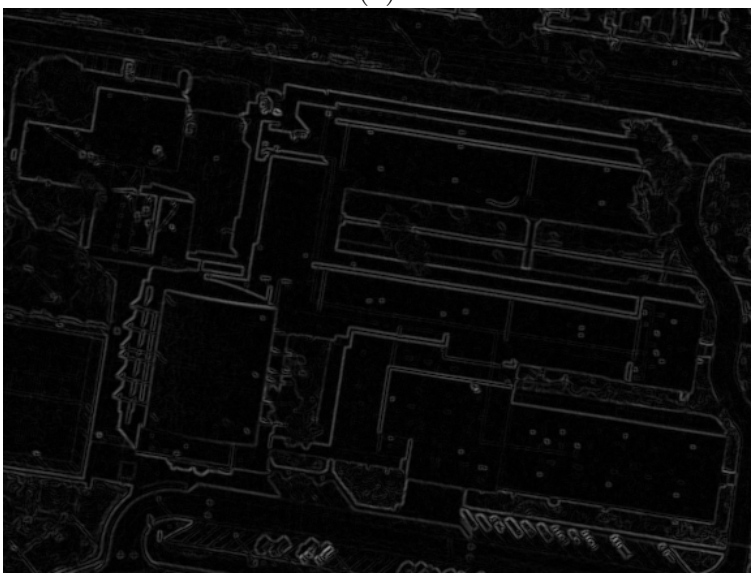


(c)

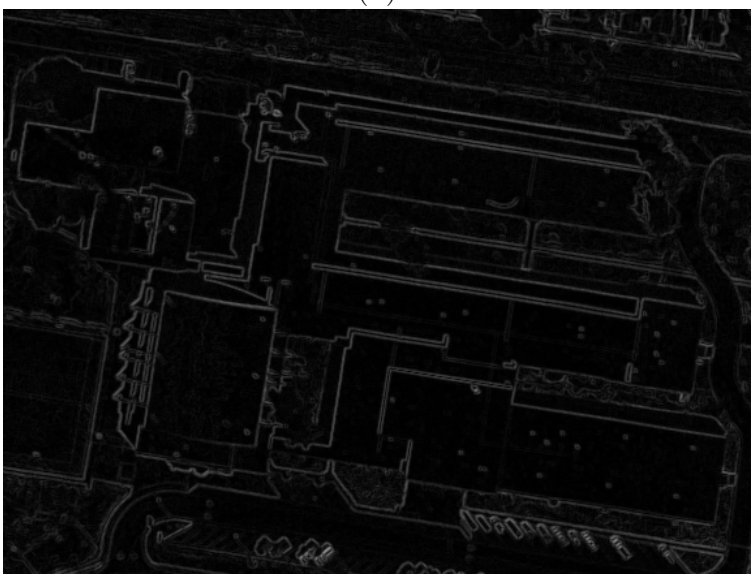
Fonte: Produzidas pelo autor.

Figura 16 – (a) Imagem “school” e resultados (b) determinístico e (c) estocástico.

(a)



(b)



(c)

Fonte: Produzidas pelo autor.

Figura 17 – (a) Imagem “woods” e resultados (b) determinístico e (c) estocástico.

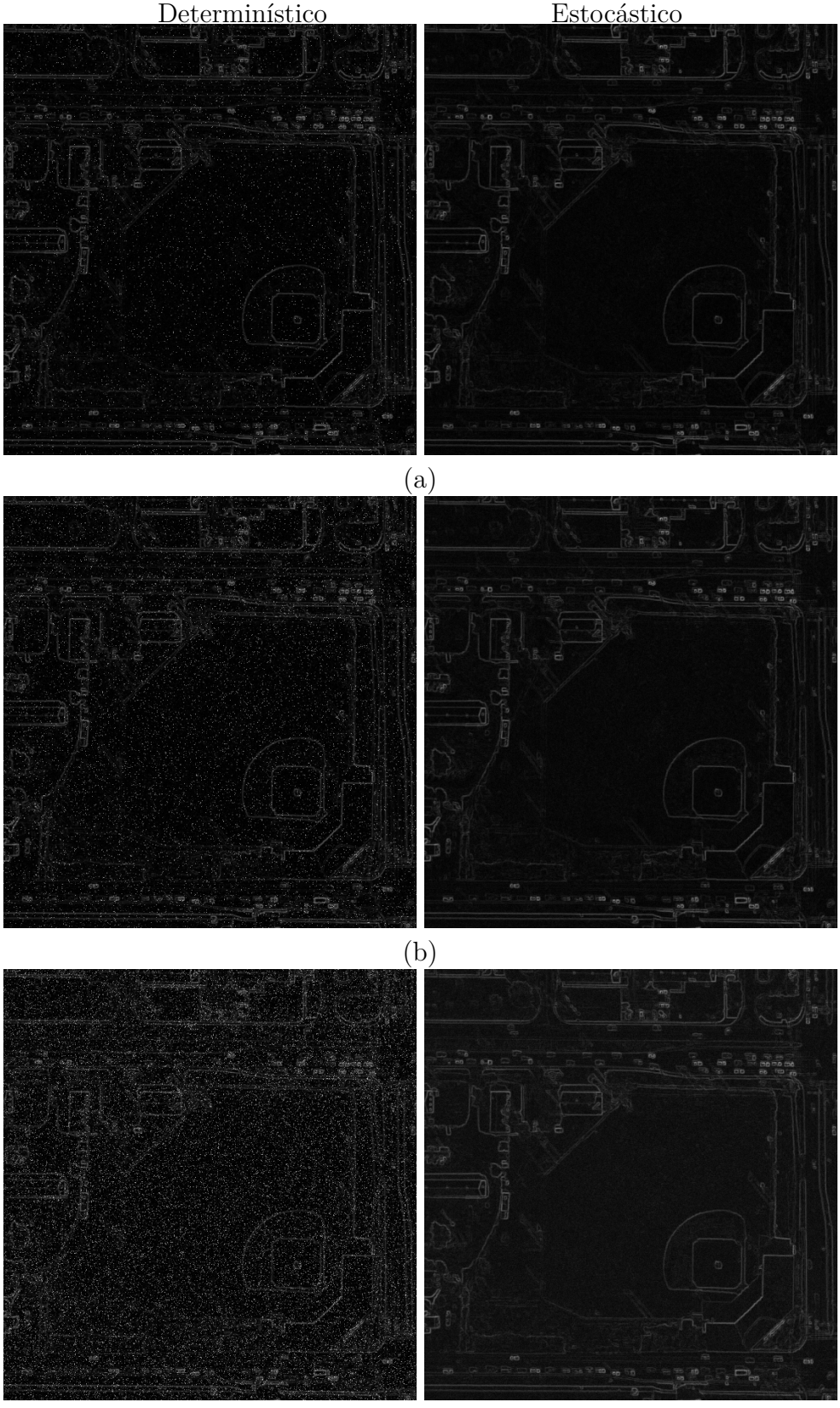
(a)

(b)

(c)

Fonte: Produzidas pelo autor.

Figura 18 – Imagem “baseball” após inserção de ruídos de (a) 1%, (b) 2% e (c) 5% para abordagens determinística e estocástica.



Fonte: Produzidas pelo autor.

5 CONCLUSÃO E PROPOSTAS PARA TRABALHOS FUTUROS

Este trabalho aborda os principais elementos de Computação Estocástica e os aplica ao problema de detecção de bordas em imagens digitais. Na Seção seguinte, destacam-se as contribuições mais importantes frente a outros trabalhos. Na Seção 5.2, são destacadas propostas para trabalhos derivados.

5.1 Principais Contribuições

O algoritmo convencional de Sobel foi implementado, validado e aplicado para servir de base de comparação a um algoritmo alternativo, estocástico. O algoritmo proposto realiza a operação de detecção de bordas utilizando circuitos diferentes aos da versão determinística. Foi possível destacar as diferenças entre as implementações determinística e estocástica utilizando métricas objetivas: MAE, MSE, número de elementos lógicos utilizados e potência consumida. Ambas as abordagens possuem adaptações em linguagens de descrição de *hardware* e podem ser sintetizadas em FPGA.

Foram validados os resultados de ambas as versões, *software* ou *hardware*, nas suas abordagens respectivas. Ou seja, as determinísticas correspondem entre si, e o mesmo ocorre para as estocásticas. As implementações em *software* auxiliam na análise de desempenho das operações realizadas com ou sem adição de ruídos suaves. As feitas para *hardware* oferecem ferramentas de análise para implementações físicas. O circuito estocástico utiliza menos elementos lógicos que o determinístico, com potência também menor. Destaca-se que a parte do circuito estocástico responsável por conversões de domínio representa aproximadamente metade da utilização de elementos lógicos.

A implementação estocástica se sobressai quanto a robustez a ruídos suaves. Os resultados indicam que seu erro possui menos discrepância que a abordagem determinística, o que favorece sua utilização em ambientes ruidosos, em consonância com o trabalho de Chen, Alaghi e Hayes (2014).

Houve contribuição do trabalho na criação de um fluxo de projeto para produção de circuitos estocásticos e avaliação de suas aplicações mediante requisitos de robustez a ruído e utilização de recursos físicos. Além disso, destaca-se a contribuição no repositório de síntese automática de circuitos *scsynth*, baseado em Alaghi e Hayes (2015). Ademais foram criados repositórios públicos para permitir reprodutibilidade e avanços em estudos na área, buscando utilizar ferramentas gratuitas e de código aberto sempre que possível.

Os valores de correlação requeridos pelos SNG e a estutura da máquina de estados são explicitados, o que complementa o trabalho de Ranjbar, Salehi e Najafi (2015). Isso, somado ao repositório criado, valida a topologia adotada para o filtro de Sobel e garante reprodutibilidade de resultados.

5.2 Propostas para Continuação do Trabalho

Este trabalho abordou alguns dos fundamentos de Computação Estocástica aplicados à síntese de circuitos de processamento de sinais. É possível listar algumas frentes para continuar a pesquisa:

- Aplicar a metodologia para outros circuitos de detecção de bordas. O grau de dificuldade do problema de produzir um circuito estocástico varia drasticamente de acordo com o número de entradas e de operações matemáticas realizadas. Isso implica mudanças relevantes na análise de correlação de seus componentes, o que pode mudar o número de conversores necessários, melhorando a eficiência do sistema;
- Experimentar outros níveis de precisão. Há limites nas vantagens oferecidas por Computação Estocástica devido a um aumento linear na precisão do circuito ocasionar um aumento exponencial do número de bits necessário para cada cadeia estocástica. Entretanto, o trabalho de Onizawa et al. (2017) oferece uma solução inteligente para otimizar o número de bits estocásticos utilizado dinamicamente, garantindo alta eficiência e desempenho;
- Testar SNG alternativos. Boa parte do desempenho de um circuito estocástico depende das relações de correlação entre seus SN. Além disso, eles representam uma parcela significativa do circuito físico;
- Estudar a aplicação de circuitos estocásticos puros, sem necessidade de elementos conversores para domínio binário convencional. Há situações em que conversores A/D conseguem produzir bons números estocásticos e dispensar a presença do meio determinístico. Trabalhos como o de Lee et al. (2017) sugerem benefícios em aliar circuitos estocásticos puros a sistemas não-lineares, como redes neurais artificiais;
- Refazer a análise física a nível de circuito integrado a fim de obter resultados mais significativos para projeto de sistemas embarcados. Essa análise foi impedida pela falta de ferramentas (*software*) e recursos (arquivos de tecnologia) para síntese e análise de projetos de circuitos integrados, que são disponibilizadas sob contrato com as respectivas empresas. Ao que se sabe, não existem arquivos de tecnologia disponíveis gratuitamente sem contrato, mesmo que para fins acadêmicos;

- Produzir um circuito de decodificação de LDPC, seguindo essa metodologia. Os resultados de Leduc-Primeau et al. (2019) e a presença de códigos LDPC no padrão 5G de telecomunicações ressalta a importância da tecnologia;
- Estudar aplicabilidade em sistemas com fusão ou enxame de sensores;
- Avaliar relevância na área de Internet das Coisas (IoT)

Referências

ALAGHI, A. *The Logic of Random Pulses: Stochastic Computing*. Tese de Doutorado — University of Michigan, 2015. Disponível em: <https://deepblue.lib.umich.edu/handle/2027.42/113561>. Citado 11 vezes nas páginas 20, 21, 22, 23, 24, 25, 26, 29, 31, 32 e 46.

ALAGHI, A.; HAYES, J. P. Exploiting correlation in stochastic circuit design. In: *2013 IEEE 31st International Conference on Computer Design (ICCD)*. [S.l.: s.n.], 2013. p. 39–46. ISSN 1063-6404. Citado 5 vezes nas páginas 27, 28, 29, 36 e 37.

ALAGHI, A.; HAYES, J. P. Survey of stochastic computing. *ACM Trans. Embed. Comput. Syst.*, ACM, New York, NY, USA, v. 12, n. 2s, p. 92:1–92:19, maio 2013. ISSN 1539-9087. Disponível em: <http://doi.acm.org/10.1145/2465787.2465794>. Citado na página 29.

ALAGHI, A.; HAYES, J. P. Strauss: Spectral transform use in stochastic circuit synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 34, n. 11, p. 1770–1783, Nov 2015. ISSN 0278-0070. Citado 6 vezes nas páginas 22, 24, 31, 37, 52 e 57.

ALAGHI, A.; QIAN, W.; HAYES, J. P. The promise and challenge of stochastic computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 37, n. 8, p. 1515–1531, Aug 2018. ISSN 0278-0070. Citado 5 vezes nas páginas 15, 23, 29, 37 e 38.

BAE, J. H. et al. An overview of channel coding for 5G NR cellular communications. *APSIPA Transactions on Signal and Information Processing*, Cambridge University Press, v. 8, 2019. Citado na página 15.

BAUMANN, R. The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction. In: IEEE. *Electron Devices Meeting, 2002. IEDM'02. International*. [S.l.], 2002. p. 329–332. Citado na página 14.

BOWYER, K.; KRANENBURG, C.; DOUGHERTY, S. Edge detector evaluation using empirical ROC curves. *Computer Vision and Image Understanding*, Elsevier, v. 84, n. 1, p. 77–103, 2001. Citado na página 44.

BUSSAB, W. d. O.; MORETTIN, P. A. *Estatística Básica*. [S.l.]: Saraiva, 2010. Citado 2 vezes nas páginas 26 e 27.

CHEN, T.-H.; ALAGHI, A.; HAYES, J. P. Behavior of stochastic circuits under severe error conditions. *it-Information Technology*, De Gruyter Oldenbourg, v. 56, n. 4, p. 182–191, 2014. Citado 3 vezes nas páginas 30, 47 e 57.

CIELEN, D.; MEYSMAN, A.; ALI, M. *Introducing Data Science: Big Data, Machine Learning, and More, Using Python Tools*. 1st. ed. Greenwich, CT, USA: Manning Publications Co., 2016. ISBN 1633430030, 9781633430037. Citado na página 34.

DENNING, P. J.; LEWIS, T. G. Exponential laws of computing growth. ACM, 2017. Citado na página 39.

ERIKSSON, H. et al. Full-custom vs. standard-cell design flow-an adder case study. In: IEEE. *Design Automation Conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific*. [S.l.], 2003. p. 507–510. Citado na página 30.

FARID, H.; SIMONCELLI, E. P. Optimally rotation-equivariant directional derivative kernels. In: SPRINGER. *International Conference on Computer Analysis of Images and Patterns*. [S.l.], 1997. p. 207–214. Citado na página 18.

GAINES, B. R. Stochastic computing. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. New York, NY, USA: ACM, 1967. (AFIPS '67 (Spring)), p. 149–156. Disponível em: <http://doi.acm.org/10.1145/1465482.1465505>. Citado 2 vezes nas páginas 14 e 39.

GAUDET, V.; RAPLEY, A. Iterative decoding using stochastic computation. v. 39, p. 299 – 301, 03 2003. Citado 2 vezes nas páginas 30 e 39.

GONZALEZ, R. C.; WOODS, R. E. *Digital Image Processing (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN 013168728X. Citado 3 vezes nas páginas 17, 18 e 19.

GUPTA, P. K.; KUMARESAN, R. Binary multiplication with PN sequences. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, v. 36, n. 4, p. 603–606, April 1988. ISSN 0096-3518. Citado na página 24.

HAZUCHA, P.; SVENSSON, C. Impact of CMOS technology scaling on the atmospheric neutron soft error rate. *IEEE Transactions on Nuclear Science*, v. 47, n. 6, p. 2586–2594, Dec 2000. ISSN 0018-9499. Citado na página 14.

HURST, S. L.; MILLER, D. M.; MUZIO, J. C. Spectral techniques in digital logic. 1985. Citado na página 31.

KAHAN, W. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, v. 754, n. 94720-1776, p. 11, 1996. Citado na página 14.

KARPOVSKY, M. G.; STANKOVIC, R. S.; ASTOLA, J. T. *Spectral logic and its applications for the design of digital devices*. [S.l.]: John Wiley & Sons, 2008. Citado na página 31.

KIM, J. et al. 3.5 A 16-to-40Gb/s quarter-rate NRZ/PAM4 dual-mode transmitter in 14nm CMOS. In: IEEE. *Solid-State Circuits Conference-(ISSCC), 2015 IEEE International*. [S.l.], 2015. p. 1–3. Citado na página 14.

KRISHNASWAMY, S. et al. Probabilistic transfer matrices in symbolic reliability analysis of logic circuits. *ACM Trans. Des. Autom. Electron. Syst.*, ACM, New York, NY, USA, v. 13, n. 1, p. 8:1–8:35, fev. 2008. ISSN 1084-4309. Disponível em: <http://doi.acm.org/10.1145/1297666.1297674>. Citado na página 29.

LEDUC-PRIMEAU, F. et al. Stochastic decoding of error-correcting codes. In: *Stochastic Computing: Techniques and Applications*. [S.l.]: Springer, 2019. p. 201–215. Citado 2 vezes nas páginas 15 e 59.

LEE, V. T. et al. Stochastic synthesis for stochastic computing. *CoRR*, abs/1810.04756, 2018. Disponível em: <http://arxiv.org/abs/1810.04756>. Citado na página 43.

LEE, V. T. et al. Energy-efficient hybrid stochastic-binary neural networks for near-sensor computing. *CoRR*, abs/1706.02344, 2017. Disponível em: <http://arxiv.org/abs/1706.02344>. Citado 4 vezes nas páginas 15, 30, 44 e 58.

LI, P.; LILJA, D. J. Using stochastic computing to implement digital image processing algorithms. In: *2011 IEEE 29th International Conference on Computer Design (ICCD)*. [S.l.: s.n.], 2011. p. 154–161. ISSN 1063-6404. Citado na página 30.

ONIZAWA, N. et al. Area/energy-efficient gammatone filters based on stochastic computation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 25, n. 10, p. 2724–2735, Oct 2017. ISSN 1063-8210. Citado 2 vezes nas páginas 15 e 58.

QIAN, W. et al. An architecture for fault-tolerant computation with stochastic logic. *IEEE Transactions on Computers*, v. 60, n. 1, p. 93–105, Jan 2011. ISSN 0018-9340. Citado 2 vezes nas páginas 22 e 31.

QIAN, W. et al. An architecture for fault-tolerant computation with stochastic logic. *IEEE Transactions on Computers*, v. 60, n. 1, p. 93–105, Jan 2011. ISSN 0018-9340. Citado na página 30.

RANJBAR, M.; SALEHI, M. E.; NAJAFI, M. H. Using stochastic architectures for edge detection algorithms. In: IEEE. *Electrical Engineering (ICEE), 2015 23rd Iranian Conference on*. [S.l.], 2015. p. 723–728. Citado 7 vezes nas páginas 15, 30, 36, 39, 40, 42 e 58.

ROCHELLE, R. Pulse-frequency modulation. *IRE Transactions on space electronics and telemetry*, IEEE, n. 2, p. 107–111, 1962. Citado na página 20.

SEDRA, A. S. et al. *Microelectronic circuits*. [S.l.]: Oxford University Press, 2016. Citado na página 39.

Shannon, C. E. Probability of error for optimal codes in a gaussian channel. *The Bell System Technical Journal*, v. 38, n. 3, p. 611–656, May 1959. ISSN 0005-8580. Citado na página 15.

SOBEL, I. An isotropic 3x3 image gradient operator. 02 2014. Citado na página 19.

SZUDZIK, M. An elegant pairing function. In: *Wolfram Research (ed.) Special NKS 2006 Wolfram Science Conference*. [S.l.: s.n.], 2006. p. 1–12. Citado 2 vezes nas páginas 38 e 52.

VERDÚ, S. et al. A general formula for channel capacity. *IEEE Transactions on Information Theory*, IEEE, v. 40, n. 4, p. 1147–1157, 1994. Citado na página 15.

VERMA, R.; ALI, J. A comparative study of various types of image noise and efficient noise removal techniques. *International Journal of advanced research in computer science and software engineering*, v. 3, n. 10, 2013. Citado na página 45.

Apêndices

APÊNDICE A – ALGORITMO DE SOBEL DETERMINÍSTICO EM PYTHON

Arquivo detSobel.py

```
# Author: Danilo Cavalcanti
# Importing System Modules
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt
from bitarray import bitarray
from scipy.stats import bernoulli

# Making constants the Python way
# "Making them in a separate file and importing
# them on a main.py" thing
sKernelX=np.array([[-1,0,1],
                                                [-2,0,2],
                                                [-1,0,1]],np.float64)

sKernelY=np.array([[-1,-2,-1],
                                                [0,0,0],
                                                [1,2,1]],np.float64)

def sobelFilter(img=-1,errRate=0.0):
        '''Alternate function that calculates Gx and Gy of a 3x3
        img in numpy matrix form to compare with Stochastic version
        Arguments:
        - img: 3x3 region to process Gx and Gy
        '''
        if(type(img) != np.ndarray):
                print("Invalid 'img' parameter, returning default (0)")
                return 0
        elif(img.shape!=(3,3)):
                print("Invalid 'img' shape (not 3x3), returning default (0)")
                return 0
```

```
        elif(img.dtype != np.float64):
                print("Invalid 'img' dtype (not float64), returning default (0)")
                return 0
        elif(errRate<0.0 or errRate>1.0):
                print("Invalid error rate, must be between 0.0 and 1.0")
                return 0
        else:
                Gx = np.float64(0)
                Gy = np.float64(0)

                # Do the convolution in one of NumPy's way
                Gx = np.sum(sKernelX*img)
                Gy = np.sum(sKernelY*img)

                ans = np.uint8((0.25*np.abs(Gx)+0.25*np.abs(Gy))/2.0)
                if(errRate!=0):
                        ansBin = bitarray('{0:08b}'.format(ans))
                        for i in range(len(ansBin)):
                                errBit = bernoulli.rvs(errRate,size=1)
                                ansBin[i] = ansBin[i]^errBit
                        ans = int(ansBin.to01(),2)

                return np.uint8(ans)

def createEdgeImage(img=-1,errRate=0.0):
        ''' Applies Sobel filter on a NxM image "img" loaded via OpenCV (cv2 packa
        returns three (N-2)x(M-2) images with sobelX, sobelY and both filters app
        2 rows and 2 columns removed to simplify boundary conditions.
        Arguments:
        - img: Region in Grayscale color scheme and np.uint8 format
        '''
        if(type(img) != np.ndarray):
                print("Invalid 'img' parameter, returning empty matrix")
                return np.array([0],np.float64)
        elif(img.dtype != np.float64):
                print("Invalid 'img' dtype (not float64), returning empty matrix")
                return np.array([0],np.float64)
        elif(errRate<0.0 or errRate>1.0):
                print("Invalid error rate, must be between 0.0 and 1.0")
```

```
        else:
                # Create images ignoring last row and column for simplicity in
                # convolution operation

                xy_image = np.zeros([img.shape[0]-2,img.shape[1]-2])
                for i in range(1,img.shape[0]-1):
                        for j in range(1,img.shape[1]-1):
                                # Reset Gx & Gy for each pixel
                                Gx, Gy = 0, 0
                                # Get 3x3 submatrixes with np.ix_
                                # [i-1,i,i+1] = range(i-1,i+2)
                                # kept explicit for clarity
                                ixgrid = np.ix_([i-1,i,i+1],[j-1,j,j+1])
                                workingArea = img[ixgrid]
                                # Call the convolution function
                                xy_image[i-1][j-1] = sobelFilter(workingArea,errRat

                return xy_image

def detectAndShow(imgpath=0,errRate=0.0,show=True):
        # Load image from path
        # Basic validness check before operating
        if(isinstance(imgpath,str)):
                img = cv.imread(imgpath,cv.IMREAD_GRAYSCALE)
                if(isinstance(img,type(None))):
                        print("Image could not be loaded")
                        return -1,-1
        else:
                print("Invalid image path")
                return -1,-1

        # This is where the processing begins
        xy_img = createEdgeImage(np.array(img,np.float64),errRate)

        # Plot side by side for comparison
        #plt.subplot(1,2,1), plt.imshow(img,cmap='gray')
        #plt.title('Original'), plt.xticks([]), plt.yticks([])
        #plt.subplot(1,2,2), plt.imshow(xy_img,cmap='gray')
        #plt.title('Sobel XY'), plt.xticks([]), plt.yticks([])
```

```
        # Plot only results
        if(show):
                plt.imshow(xy_img,cmap='gray')
                plt.show()

        return np.uint8(img),np.uint8(xy_img)

def detectAndWritePGM(imgpath=0):
        # Load image from path
        # Basic validness check before operating
        if(isinstance(imgpath,str)):
                img = cv.imread(imgpath,cv.IMREAD_GRAYSCALE)
                if(isinstance(img,type(None))):
                        print("Image could not be loaded")
                        return -1,-1
        else:
                print("Invalid image path")
                return -1,-1


        # This is where the processing begins
        xy_img = createEdgeImage(np.array(img,np.float64))

        # Convert back to Grayscale
        xy_img = np.uint8(xy_img)

        # Write back into PGM image files
        flag1 = cv.imwrite("src.pgm",img)
        flag2 = cv.imwrite("edges.pgm",xy_img)
        if(flag1 and flag2):
                print('''Files "src.pgm" and "edges.pgm" successfully created''')
        else:
                print('''Something went wrong during the saving process''')

        return img,xy_img

def opencvSobelFilter(img=-1):
        '''Function that calculates Gx and Gy of a 3x3 img in numpy matrix form
```

```
        Arguments:
        - img: 3x3 region to process Gx and Gy
        '''
        if(type(img) != np.ndarray):
                print("Invalid 'img' parameter, returning default (0, 0)")
                return 0, 0
        elif(img.shape!=(3,3)):
                print("Invalid 'img' shape (not 3x3), returning default (0, 0)")
                return 0, 0
        elif(img.dtype != np.float64):
                print("Invalid 'img' dtype (not float64), returning default (0, 0)"
                return 0, 0
        else:
                Gx = np.float64(0.0)
                Gy = np.float64(0.0)

                # Do the convolution in one of NumPy's way
                Gx = np.sum(sKernelX*img)
                Gy = np.sum(sKernelY*img)

                return Gx,Gy

def opencvCreateEdgeImage(img=-1):
        ''' Applies Sobel filter on a NxM image "img" loaded via OpenCV (cv2 packa
        returns three (N-2)x(M-2) images with sobelX, sobelY and both filters app
        2 rows and 2 columns removed to simplify boundary conditions.
        Arguments:
        - img: Region in Grayscale color scheme and np.float64 format
        '''
        if(type(img) != np.ndarray):
                print("Invalid 'img' parameter, returning empty matrix")
                return np.array([0],np.float64)
        elif(img.dtype != np.float64):
                print("Invalid 'img' dtype (not float64), returning empty matrix")
                return np.array([0],np.float64)
        else:
                # Create images ignoring last row and column for simplicity in
                # convolution operation
                x_image = np.zeros([img.shape[0]-2,img.shape[1]-2])
```

```
                y_image = np.zeros([img.shape[0]-2,img.shape[1]-2])

                xy_image = np.zeros([img.shape[0]-2,img.shape[1]-2])
                for i in range(1,img.shape[0]-1):
                        for j in range(1,img.shape[1]-1):
                                # Reset Gx & Gy for each pixel
                                Gx, Gy = 0, 0
                                # Get 3x3 submatrixes with np.ix_
                                # [i-1,i,i+1] = range(i-1,i+2)
                                # kept explicit for clarity
                                ixgrid = np.ix_([i-1,i,i+1],[j-1,j,j+1])
                                workingArea = img[ixgrid]
                                # Call the convolution function
                                Gx, Gy = sobelFilter(workingArea)
                                x_image[i-1][j-1] = Gx
                                y_image[i-1][j-1] = Gy

                # Take absolute value and
                x_image = np.abs(x_image)
                y_image = np.abs(y_image)
                # Saturate x and y_image to fit 8-bit
                x_image[x_image>255] = 255
                y_image[y_image>255] = 255

                # Sum halves to keep results in [0-255]
                xy_image = 0.5*x_image+0.5*y_image
                return xy_image

def opencvDetectAndShow(imgpath=0):
        # Load image from path
        # Basic validness check before operating
        if(isinstance(imgpath,str)):
                img = cv.imread(imgpath,cv.IMREAD_GRAYSCALE)
                if(isinstance(img,type(None))):
                        print("Image could not be loaded")
                        return -1,-1
        else:
                print("Invalid image path")
                return -1,-1
```

```
        # This is where the processing begins
        xy_img = opencvCreateEdgeImage(np.array(img,np.float64))

        # Convert back to Grayscale
        xy_img = np.uint8(xy_img)

        # Plot side by side for comparison
        #plt.subplot(1,2,1), plt.imshow(img,cmap='gray')
        #plt.title('Original'), plt.xticks([]), plt.yticks([])
        #plt.subplot(1,2,2), plt.imshow(xy_img,cmap='gray')
        #plt.title('Sobel XY'), plt.xticks([]), plt.yticks([])

        # Plot only results
        plt.imshow(xy_img,cmap='gray')
        plt.show()

        return img,xy_img

def saveHex(filename=None,numpyarray=np.zeros(1)):
        '''Saves image in memory in NumPy's uint8 format
        into a text file with two hexadecimal characters
        txt format is preferrable.
        '''
        if(not filename):
                print("Invalid filename")
                return 0
        np.savetxt(filename,numpyarray,"%.2x")

def loadHex(filename=None):
        '''Reads a txt file containing a NumPy image in uint8
        format saved in pairs of hexadecimal characters.
        Returns the image as a NumPy uint8 array.
        '''
        if(not filename):
                print("Invalid hex filename")
                return None
        elif(isinstance(type(filename),str)):
                print("Invalid hex filename")
```

```
                return None
        elif(filename[-3:]!='txt'):
                print("Invalid hex filename")
                return None
        else:
                #Load images stored in hex txt files
                input_img = open(filename)

                #Put the contents into list of strings
                input_str = input_img.readlines()

                #Close file after reading
                input_img.close()

                #Prepare matrices
                decoded_mat = []

                #Take away trailing new line('\n') and convert to
                #List of byte arrays
                for i in range(len(input_str)):
                        input_str[i] = input_str[i].rstrip('\n')
                        decoded_mat.append(bytearray.fromhex(input_str[i]))

                #Convert bytes into unsigned 8-bit integers
                #This is one of the compatible image formats
                decoded_mat = np.array(decoded_mat,np.uint8)
                return decoded_mat

def showHex(filename=None):
        '''Reads a txt file containing a NumPy image in uint8
        format saved in pairs of hexadecimal characters.
        Shows the image on the screen and returns the
        image as a NumPy uint8 array.
        '''
        if(not filename):
                print("Invalid hex filename")
                return 0
        elif(isinstance(type(filename),str)):
                print("Invalid hex filename")
```

```
                return None
        elif(filename[-3:]!='txt'):
                print("Invalid hex filename")
                return 0
        else:
                #Load image stored in hex txt files
                input_img = open(filename)

                #Put the contents into list of strings
                input_str = input_img.readlines()

                #Close file after reading
                input_img.close()

                #Prepare matrices
                decoded_mat = []

                #Take away trailing new line('\n') and convert to
                #List of byte arrays
                for i in range(len(input_str)):
                        input_str[i] = input_str[i].rstrip('\n')
                        decoded_mat.append(bytearray.fromhex(input_str[i]))

                #Convert bytes into unsigned 8-bit integers
                #This is one of the compatible image formats
                decoded_mat = np.array(decoded_mat,np.uint8)

                #Plot the image
                plt.imshow(decoded_mat,cmap='gray')
                plt.show()
                return decoded_mat


def hexToPGM(filename=None,outputfile=None):
        '''Reads a txt file containing a NumPy image in uint8
        format saved in pairs of hexadecimal characters.
        Shows the image on the screen and returns the
        image as a NumPy uint8 array.
        '''
```

```
if(not filename or not outputfile):
        print("Invalid filename")
        return 0
elif(isinstance(type(filename),str) and isinstance(type(outputfile),str)):
        print("Invalid filename")
        return None
elif(filename[-3:]!='txt'):
        print("Invalid hex filename")
        return 0
elif(outputfile[-3:]!='pgm'):
        print("Invalid PGM filename")
        return 0
else:
        #Load image stored in hex txt files
        input_img = open(filename)

        #Put the contents into list of strings
        input_str = input_img.readlines()

        #Close file after reading
        input_img.close()

        #Prepare matrices
        decoded_mat = []

        #Take away trailing new line('\n') and convert to
        #List of byte arrays
        for i in range(len(input_str)):
                input_str[i] = input_str[i].rstrip('\n')
                decoded_mat.append(bytearray.fromhex(input_str[i]))

        #Convert bytes into unsigned 8-bit integers
        #This is one of the compatible image formats
        decoded_mat = np.array(decoded_mat,np.uint8)

        flag1 = cv.imwrite(outputfile,decoded_mat)
        if(flag1):
                print('''File''',outputfile,''' successfully created''')
        else:
```

```
                print('''Something went wrong during the saving process''')

    return decoded_mat
```

APÊNDICE B – ALGORITMO DE SOBEL ESTOCÁSTICO EM PYTHON

Arquivo stochSobel.py

```
# Author: Danilo Cavalcanti
# Importing System Modules
from bitarray import bitarray

# Importing auxiliary modules

def mux2(i1,i2,s1):
        '''Yields input 1 (i1) or i2 depending on switch 1 (s1)'''
        if(not s1):
                return i1
        else:
                return i2

def mux4(i1,i2,i3,i4,s1,s2):
        if(not s1 and not s2):
                return i1
        elif (not s1 and s2):
                return i2
        elif (s1 and not s2):
                return i3
        else:
                return i4

def sobel(z1_1,
                        z2_1,
                        z3_1,
                        z4_1,
                        z6_1,
                        z7_1,
                        z8_1,
                        z9_1,
```

```
                        z1_2,
                        z2_2,
                        z3_2,
                        z4_2,
                        z6_2,
                        z7_2,
                        z8_2,
                        z9_2,
                        r0,r1,r2,r3,r4):
        '''Implements stochastic sobel filter based on Ranjbar et. al 2015
        All inputs uncorrelated'''
        # Different from the article because it implements diagonal filters
        # Each term equals one of the input muxes
        term1 = mux4(z1_1,z2_1,z2_2,z3_1,r0,r1)
        term2 = mux4(z7_1,z8_1,z8_2,z9_1,r0,r1)
        term3 = mux4(z1_2,z4_1,z4_2,z7_2,r2,r3)
        term4 = mux4(z3_2,z6_1,z6_2,z9_2,r2,r3)

        # Absolute value function done by XOR ports
        abs1 = term1^term2
        abs2 = term3^term4

        # Output pixel, z5
        return mux2(abs1,abs2,r4)

def ranjSobel(z1_1,
                    z2_1,
                    z3_1,
                    z4_1,
                    z6_1,
                    z7_1,
                    z8_1,
                    z9_1,
                    z2_2,
                    z4_2,
                    z6_2,
                    z8_2,
                    r0,r1,r2,r3,r4):
        '''Implements stochastic sobel filter based on Ranjbar et. al 2015'''
```

```
        # Different from the article because it implements diagonal filters
        # Each term equals one of the input muxes
        term1 = mux4(z1_1,z2_1,z2_2,z3_1,r0,r1)
        term2 = mux4(z7_1,z8_1,z8_2,z9_1,r0,r1)
        term3 = mux4(z1_1,z4_1,z4_2,z7_1,r2,r3)
        term4 = mux4(z3_1,z6_1,z6_2,z9_1,r2,r3)

        # Absolute value function done by XOR ports
        abs1 = term1^term2
        abs2 = term3^term4

        # Output pixel, z5
        return mux2(abs1,abs2,r4)

def diagSobel(z1_1,
                z2_1,
                z3_1,
                z4_1,
                z6_1,
                z7_1,
                z8_1,
                z9_1,
                z1_2,
                z3_2,
                z7_2,
                z9_2,
                r0,r1,r2,r3,r4):
        '''Implements stochastic sobel filter by Ranjbar et. al 2015'''
        # Each term equals one of the input muxes
        term1 = mux4(z2_1,z3_1,z3_2,z6_1,r0,r1) #<- different from article
        term2 = mux4(z4_1,z7_1,z7_2,z8_1,r0,r1)
        term3 = mux4(z6_1,z9_1,z9_2,z8_1,r2,r3)
        term4 = mux4(z2_1,z1_1,z1_2,z4_1,r2,r3)

        # Absolute value function done by XOR ports
        abs1 = term1^term2
        abs2 = term3^term4

        # Output pixel, z5
```

```
        return mux2(abs1,abs2,r4)
```

APÊNDICE C – TESTBENCH PARA IMPLEMENTAÇÃO DETERMINÍSTICA EM *PYTHON*

Arquivo interactableDetSobel.py

```
# Author: Danilo Cavalcanti
# Importing System Modules
import cv2 as cv
import matplotlib.pyplot as plt
import numpy as np

# Import personal implementation library
from detSobel import *
print('''----------------------------------------------------------------------
----------------------------------------------------------------
Sucessfully loaded personal Sobel Filter implementation
How to demo:
Use functions 'detectAndShow(image)' or 'detectAndWritePGM(image)'
Example: src,edges = detectAndShow('./images/aerial/school.pgm')
Result: photo printed on screen with edges detected,
base image returned as 'src' and
edge image returned into 'edges' variable as numpy matrix
----------------------------------------------------------------
----------------------------------------------------------------''')
import code
code.interact(local=locals())
```

APÊNDICE D – TESTBENCH PARA IMPLEMENTAÇÃO ESTOCÁSTICA *AD HOC* EM *PYTHON*

Arquivo interactableStochSobel.py

```
# Author: Danilo Cavalcanti
# Importing System Modules
import cv2 as cv
import matplotlib.pyplot as plt
import numpy as np

# Import personal implementation library
from stochWrapper import *

ray.init()

print('''------------------------------------------------------------------
-----------------------------------------------------------------
Sucessfully loaded personal Sobel Filter implementation
How to demo:
Use functions 'detectAndShow(image)' or 'detectAndWritePGM(image)'
Example: src,edges = rayDetectAndShow('./images/aerial/school.pgm')
Result: photo printed on screen with edges detected,
base image returned as 'src' and
edge image returned into 'edges' variable as numpy matrix
-----------------------------------------------------------------
-----------------------------------------------------------------''')

import code
code.interact(local=locals())
```

APÊNDICE E – TESTBENCH PARA IMPLEMENTAÇÃO DETERMINÍSTICA EM *VERILOG*

Arquivo Testbench_to_file.v

```
module Testbench_to_file();
        parameter src_rows = 436;
        parameter src_cols = 576;
        parameter src_size = src_rows*src_cols;
        parameter edge_size = (src_rows-1)*(src_cols-1);
        reg [7:0] src [0:src_size-1];
        reg [7:0] edges [0:edge_size-1];

        // Variable to save file handle
        integer f;
        // Auxiliary variables for counting in loops
        integer i,j;

        // Registers and wires for the Sobel module
        reg [7:0] z1;
        reg [7:0] z2;
        reg [7:0] z3;
        reg [7:0] z4;
        reg [7:0] z6;
        reg [7:0] z7;
        reg [7:0] z8;
        reg [7:0] z9;

        wire [7:0] z_out;

        // Auxiliary system signals
        // May be used in the future
        reg clk;
        reg reset;
```

```
// Instantiate device under test (DUT)
sobel3x3det DUT(.z1(z1),
                .z2(z2),
                .z3(z3),
                .z4(z4),
                .z6(z6),
                .z7(z7),
                .z8(z8),
                .z9(z9),
                .clk(clk),
                .reset(reset),
                .z_out(z_out)
               );

// Clock may be used later for stochastic
always begin
        #1 clk <= ~clk;
end

initial begin
        // Default inputs for unit tests
        clk = 0;
        reset = 0;
        z1 = 8'h00;
        z2 = 8'h00;
        z3 = 8'h00;
        z4 = 8'h00;
        z6 = 8'h00;
        z7 = 8'h00;
        z8 = 8'h00;
        z9 = 8'h00;
        #2
        reset = 1;
        #2
        reset = 0;

        #2
        //Load image coded in hexadecimal as memory
```

```
                $dumpfile("test.vcd");
                $dumpvars(0,Testbench_to_file);
                $display("Loading image into memory");
                $readmemh("src.txt",src);
                f = $fopen("edges_hw_det.txt","w");
                // 'Crop' area around central pixel z5
                for (i=1;i<src_rows-1;i=i+1) begin
                        for(j=1;j<src_cols-1;j=j+1) begin
                                // Row above z5
                                z1 <= src[src_cols*(i-1)+j-1];
                                z2 <= src[src_cols*(i-1)+j];
                                z3 <= src[src_cols*(i-1)+j+1];
                                // Row of z5
                                z4 <= src[src_cols*(i)+j-1];
                                //unused middle pixel
                                z6 <= src[src_cols*(i)+j+1];
                                // Row below z5
                                z7 <= src[src_cols*(i+1)+j-1];
                                z8 <= src[src_cols*(i+1)+j];
                                z9 <= src[src_cols*(i+1)+j+1];

                                #2 // give an instant for z_out to update
                                //save 'edges' array for possible future use
                                edges[src_cols*(i-1)+j-1]=z_out;
                                //Write on file
                                $fwrite(f,"%x ",z_out);
                        end
                $fwrite(f,"\n");
                end
                $fclose(f);

                $finish;
        end

endmodule
```

APÊNDICE F – TESTBENCH PARA IMPLEMENTAÇÃO ESTOCÁSTICA *AD HOC* EM *VERILOG*

Arquivo Testbench_to_file.v

```
module Testbench_to_file();
        //parameter src_rows = 436;
        //parameter src_cols = 576;
        //parameter src_rows = 50;
        //parameter src_cols = 50;
        parameter src_rows = 3;
        parameter src_cols = 3;
        parameter src_size = src_rows*src_cols;
        parameter edge_size = (src_rows-2)*(src_cols-2);
        reg [7:0] src [0:src_size-1];
        reg [7:0] edges [0:edge_size-1];

        // Variable to save file handle
        integer f;
        // Auxiliary variables for counting in loops
        integer i,j;

        //Registers for Sobel x
        reg [7:0] pixel_1_bin;
        reg [7:0] pixel_2_bin;
        reg [7:0] pixel_3_bin;
        reg [7:0] pixel_4_bin;
        reg [7:0] pixel_6_bin;
        reg [7:0] pixel_7_bin;
        reg [7:0] pixel_8_bin;
        reg [7:0] pixel_9_bin;

        wire [7:0] x_bin; //binary value of output
```

```
reg start;
wire done;
reg clk;
reg reset;

stochWrapper SobelX (
        .pixel_1_bin(pixel_1_bin),
        .pixel_2_bin(pixel_2_bin),
        .pixel_3_bin(pixel_3_bin),
        .pixel_4_bin(pixel_4_bin),
        .pixel_6_bin(pixel_6_bin),
        .pixel_7_bin(pixel_7_bin),
        .pixel_8_bin(pixel_8_bin),
        .pixel_9_bin(pixel_9_bin),
        .start (start),
        .done (done),
        .z_bin (x_bin),
        .clk (clk),
        .reset (reset)
);


always begin
        #1 clk <= ~clk;
end

initial begin
        //Load image coded in hexadecimal as memory
        $dumpfile("test.vcd");
        $dumpvars(0,Testbench_to_file);
        $display("Loading image into memory");
        //$readmemh("srcFull.txt",src);
        $readmemh("square3.txt",src);
        clk = 0;
        reset = 1;
        start = 0;
        #5 reset = 0;
        start = 1;
```

```
        #2

        // 'Crop' area around central pixel z5
        for (i=1;i<src_rows-1;i=i+1) begin
                for(j=1;j<src_cols-1;j=j+1) begin

                        //3x3 mask for Sobel
                        pixel_1_bin <= src[src_cols*(i-1)+j-1];
                        pixel_2_bin <= src[src_cols*(i-1)+j];
                        pixel_3_bin <= src[src_cols*(i-1)+j+1];
                        pixel_4_bin <= src[src_cols*(i)+j-1];
                        //Unused central pixel
                        pixel_6_bin <= src[src_cols*(i)+j+1];
                        pixel_7_bin <= src[src_cols*(i+1)+j-1];
                        pixel_8_bin <= src[src_cols*(i+1)+j];
                        pixel_9_bin <= src[src_cols*(i+1)+j+1];

                        start <= 0;
                        #518 ; // give time to calculate


                end
        end

        #10

        f = $fopen("edges_hw.txt","w");
        // Write to file
        for (i=0;i<src_rows-2;i=i+1) begin
                for(j=0;j<src_cols-2;j=j+1) begin
                        $fwrite(f,"%x ",edges[src_cols*(i)+j]);
                end
                $fwrite(f,"\n");
        end
        $fclose(f);

        $finish;
end
```

```
        always @(posedge done) begin
                //$display("x: %b, z: %b, expected_z: %b", x_bin, z_bin, expected_
                //$fwrite(f,"%b,%b\n", z_bin, expected_z);
                start <= 1;
                edges[src_cols*(i-1)+j-1]<=x_bin;
        end
endmodule
```

APÊNDICE G – PRODUÇÃO BIBLIOGRÁFICA

(i) CAVALCANTI, D. B.; LOPES, W. T. A.; SOUTO, C. R., Computação Estocástica Aplicada em Detecção de Bordas em Imagens. *IX Conferência Nacional em Comunicações, Redes e Segurança da Informação (ENCOM'19)*, Petrolina, PE, Outubro 2019. *Submetido para revisão.*