UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Co-projeto de Hardware/Software do Filtro
de Partículas para Localização em Tempo
Real de Robôs Móveis

Bruno Franciscon Mazzotti

São Carlos / SP

SERVIÇO DE PÓS-GRADUAÇÃO DO
ICMC/USP

Data de Depósito: _____

Assinatura: _____

Co-projeto de Hardware/Software do Filtro de Partículas para Localização em Tempo Real de Robôs Móveis

Bruno Franciscon Mazzotti

Orientador: Prof. Dr. Eduardo Marques

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação (ICMC/USP) como parte dos requisitos necessários para obtenção do título de Mestre em Ciências de Computação e Matemática Computacional.

USP - São Carlos
Novembro / 2009

Esse som é sobre a ciência da persistência versus a preguiça e a descrença
Paciência é a sapiência do espírito
Viver no presente é a base, a chave para seguir bem na viagem
Evita o desgaste desnecessário durante o seu itinerário no planeta
Esse som é sobre o processo
O processo é lento

...

O processo é lento, não estou dizendo que é fácil
O processo é lento
Tem que trabalhar, trabalhar feito um operário, só que sem horário
O processo é lento
O desapego do resultado é importante
O processo é lento
O caminhar contínuo nessa vibe deve ser o modus operandi
O processo é lento

...

A parada é essa, não tem outra
O negócio é seguir no melhor estilo conta-gotas
Numa relax, numa tranquila, numa boa
Dentro das possibilidades, procurar a melhor opção
O processo é lento
Realidade não é sempre o que parece
O processo é lento
Aceitação e compreensão da situação baixa consideravelmente a taxa de stress
O processo é lento

...

B Negão e Os Seletores de Frequência
in Enxugando o Gelo (2003)

Para Debora.

AGRADECIMENTOS

Aos meus pais por nunca deixarem de acreditar em mim.

Aos meus verdadeiros amigos, parceiros da república e frequentadores do centro acadêmico, pelos momentos de descontração e companheirismo incondicional.

Aos colegas de laboratório pela ajuda prestada em todos os momentos.

Ao meu orientador e amigo Professor Eduardo Marques pelo incentivo e pelos conselhos para toda a vida.

À Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) pelo auxílio financeiro.

SUMÁRIO

Lista de Figuras	vi
Lista de Tabelas	ix
Lista de Algoritmos	xi
Lista de Siglas	xi
Resumo	xv
<i>Abstract</i>	xvii
1 Introdução	1
1.1 Contextualização e Motivação	1
1.2 Objetivos do Projeto de Pesquisa	5
1.3 Organização da Dissertação	8
2 Fundamentação Teórica	9
2.1 Robótica Probabilística	9
2.1.1 Modelagem de Sistemas Robóticos	9
2.1.2 Estimação Probabilística de Estado	11
2.1.3 Filtros de Partículas	12
2.1.4 Controle Probabilístico	15
2.2 Geração de Números Aleatórios	16
2.2.1 Conceitos Fundamentais	16

2.2.2	Recorrências Módulo m	20
2.2.3	Recorrências Módulo 2	21
2.2.4	<i>Mersenne Twister</i>	22
2.2.5	CrITÉrios de Qualidade e Testes Estatísticos	25
2.3	Computação Reconfigurável	28
2.3.1	<i>Field Programmable Gate Arrays</i>	30
2.3.2	FamÍlia <i>Stratix</i> de FPGAs	33
2.4	Trabalhos Relacionados	33
2.4.1	CES: Uma Ferramenta de Programação para RobÓtica Probabilística	34
2.4.2	FPGAs Aplicados à RobÓtica Móvel	34
2.4.3	FPGAs Aplicados à Geração de Números Aleatórios	36
3	Metodologia de Desenvolvimento	37
3.1	Linguagens de Descrição de Hardware	38
3.1.1	VHDL	39
3.1.2	<i>Verilog</i>	40
3.1.3	<i>Bluespec SystemVerilog</i>	41
3.2	Ferramentas de Projeto da <i>Altera</i>	41
3.2.1	<i>Quartus II</i>	41
3.2.2	<i>SOPC Builder</i>	41
3.2.3	<i>Nios II IDE</i>	42
3.2.4	<i>ModelSim – Altera</i>	42
3.3	<i>Bluespec Workstation</i>	43
3.4	Ambiente <i>R</i> para Computação Estatística	43
3.5	Biblioteca <i>TestU01</i> para Testes EmpÍricos de RNGs	44
3.6	Padrão <i>IP-XACT</i>	44
4	Resultados	47
4.1	Projeto de <i>Hardware</i> para o MT19937	47
4.1.1	Implementação em VHDL	47
4.1.2	Implementação em <i>Bluespec SystemVerilog</i>	51
4.1.3	Validação do Modelo de <i>Hardware</i>	52
4.2	Co-projeto de <i>Hardware/Software</i> para Aritmética de Ponto Fixo	53
4.3	SOPC do Filtro de PartÍculas para Localização de RobÔs Móveis	55

5	Considerações Finais	57
5.1	Conclusão	57
5.2	Trabalhos Futuros	58
	Referências	58
A	Créditos das Figuras	67

LISTA DE FIGURAS

1.1	Projeto de pesquisa visto como intersecção multidisciplinar de diferentes áreas	6
2.1	Funções de distribuição de probabilidade normal (a) e triangular (b) com desvio padrão α	12
2.2	Estado como posição do robô em um sistema de coordenadas 2D	14
2.3	Conjunto de 100.000 pontos em $(0, 1)^3$ gerados pelo RANDU	27
2.4	Computação espacial e temporal para a expressão $y = ax^2 + bx + c$	29
2.5	Relação flexibilidade <i>versus</i> desempenho para diferentes classes de processadores	30
2.6	Arquitetura típica de um bloco lógico de FPGA	31
2.7	Arquitetura típica de um FPGA	31
2.8	Principais ferramentas EDA para FPGA.	32
2.9	Módulo lógico adaptativo da família <i>Stratix</i>	34
3.1	Metodologia de desenvolvimento do projeto de pesquisa.	38
3.2	Fluxo de projeto no <i>Quartus II</i>	42
4.1	Diagrama de blocos do <i>hardware</i> para relação de recorrência do MT19937 .	48
4.2	Diagrama de blocos do <i>hardware</i> para <i>tempering</i> do MT19937	49
4.3	Diagrama de blocos do <i>hardware</i> unificado para computação do MT19937 .	50
4.4	Diagrama de blocos do <i>hardware</i> de inicialização do MT19937	50
4.5	Máquina de estados finitos para controle do circuito de inicialização do MT19937	51
4.6	Diagrama de blocos do MT19937	52

4.7	Alocação dos recursos de <i>hardware</i> da implementação em VHDL do MT19937	53
4.8	Alocação dos recursos de <i>hardware</i> da implementação em <i>Bluespec System-Verilog</i> do MT19937	54
4.9	Estratégia adotada para validar os modelos de <i>hardware</i> do MT19937 contra a implementação original em <i>software</i>	55
4.10	Conjunto de 100.000 pontos em $(0, 1)^3$ gerados pelos modelos de <i>hardware</i> do MT19937	55
4.11	Estratégia adotada para aplicar os testes estatísticos ao modelo <i>Bluesim</i> do MT19937	56
4.12	Rotacionador CORDIC iterativo de ponto fixo	56
4.13	Fluxo de projeto com a ferramenta <i>BlueAvalon</i>	56
5.1	Possível implementação paralela do MT19937	58

LISTA DE TABELAS

2.1	<i>Kits</i> de desenvolvimento com dispositivos da família <i>Stratix</i>	33
4.1	Relatório de compilação da implementação em VHDL do MT19937	51
4.2	Relatório de compilação da implementação em <i>Bluespec SystemVerilog</i> do MT19937	52
4.3	Resultados dos testes estatísticos aplicados ao modelo <i>Bluesim</i> do MT19937	54

LISTA DE ALGORITMOS

2.1	Filtro de Partículas	15
2.2	<i>Mersenne Twister</i>	25

LISTA DE SIGLAS

ALU	<i>Arithmetic Logic Unit</i>
ALUT	<i>Adaptative LUT</i>
ANSI	<i>American National Standards Institute</i>
ASIC	<i>Application Specific Integrated Circuit</i>
BSV	<i>Bluespec System Verilog</i>
CAD	<i>Computer-Aided Design</i>
CES	<i>C++ for Embedded Systems</i>
CNPq	Conselho Nacional de Desenvolvimento Científico e Tecnológico
CORDIC	<i>COordinate Rotation DIgital Computer</i>
CPLD	<i>Complex Programmable Logic Device</i>
DSP	<i>Digital Signal Processor</i>
DUT	<i>Design under Test</i>
EDA	<i>Eletronic Design Automation</i>
EKF	<i>Extended Kalman Filter</i>
FAPESP	Fundação de Amparo à Pesquisa do Estado de São Paulo
FIPS	<i>Federal Information Processing Standard</i>
FPGA	<i>Field Programmable Gate Array</i>

FSM	<i>Finite State Machine</i>
GFSR	<i>Generalized Feedback Shift Register</i>
GNU	<i>GNU's Not Unix</i>
GPP	<i>General Purpose Processor</i>
GPS	<i>Global Positioning System</i>
GRICES	Gabinete de Relações Internacionais da Ciência e do Ensino Superior
HDL	<i>Hardware Description Language</i>
HMM	<i>Hidden Markov Model</i>
ICMC	Instituto de Ciências Matemáticas e de Computação
IDE	<i>Integrated Development Enviroment</i>
IEEE	<i>Institute of Electrical and Electronics Engineers</i>
INESC	Instituto de Engenharia de Sistemas e Computadores
LASER	<i>Light Amplification by Stimulated Emission of Radiation</i>
LCG	<i>Linear Congruential Generator</i>
LCR	Laboratório de Computação Reconfigurável
LFSR	<i>Linear Feedback Shift Register</i>
LUT	<i>Lookup Table</i>
MCL	<i>Monte Carlo Localization</i>
MRG	<i>Multiple Recursive Generator</i>
MT	<i>Mersenne Twister</i>
NIST	National Institute of Standards and Technology
PAE	Programa de Aperfeiçoamento de Ensino
PC	<i>Personal Computer</i>
PLD	<i>Programmable Logic Device</i>
PRNG	<i>Pseudo Random Number Generator</i>

RAM	<i>Random Access Memory</i>
RNG	<i>Random Number Generator</i>
SIMD	<i>Single Instruction Multiple Data</i>
SLAM	<i>Simultaneous Localization and Mapping</i>
SOPC	<i>System on Programmable Chip</i>
SRAM	<i>Static Random Access Memory</i>
VCMU	<i>Vehicle Control and Monitoring Unit</i>
VHDL	<i>VHSIC Hardware Description Language</i>
VHSIC	<i>Very High Speed Integrated Circuit</i>
XML	<i>eXtensible Markup Language</i>

RESUMO

Sofisticadas técnicas para estimação de modelos baseadas em simulação, os filtros de partículas ou métodos de Monte Carlo Seqüenciais, foram empregadas recentemente para solucionar diversos problemas difíceis no campo da robótica móvel. No entanto, o sucesso dos filtros de partículas limitou-se à computação de parâmetros em espaços de baixa dimensionalidade. Os atuais esforços de pesquisa em robótica móvel têm começado a explorar certas propriedades estruturais de seus domínios de aplicação que envolvem a utilização de filtros de partículas em espaços de maior dimensão, aumentando consideravelmente a complexidade da simulação envolvida. Simulações estatísticas dessa natureza requerem uma grande quantidade de números pseudo-aleatórios que possam ser gerados eficientemente e atendam a certos critérios de qualidade. O processo de geração de números pseudo-aleatórios torna-se o ponto crítico de tais aplicações em termos de desempenho. Neste contexto, a computação reconfigurável insere-se como uma tecnologia capaz de satisfazer a demanda por alto desempenho das grandes simulações estatísticas pois sistemas baseados em arquiteturas reconfiguráveis possuem o potencial de mapear computação em *hardware* visando aumento de eficiência sem comprometer seriamente sua flexibilidade. Tecnologias reconfiguráveis também possui o atrativo de um baixo consumo de energia, uma característica essencial para os futuros robôs móveis embarcados. Esta dissertação apresenta a implementação um sistema embarcado baseado em FPGA e projetado para solucionar o problema de localização de robôs por meio de técnicas probabilísticas. A parte fundamental de todo este sistema é um veloz gerador de números aleatórios mapeado ao *hardware* reconfigurável que foi capaz de atender rígidos critérios estatísticos de qualidade.

Palavras chave: robótica móvel, algoritmo de localização, filtro de partículas, geração de números pseudo-aleatórios, sistemas embarcados, computação reconfigurável.

ABSTRACT

Sophisticated techniques for estimation of models based on simulation, particle filters or Sequential Monte Carlo Methods, were recently used to solve many difficult problems in the field of mobile robotics. However, the success of particle filters was limited to the computation of parameters in low dimensionality spaces. The current research efforts in mobile robotics have begun to explore some structural properties of their application's domain involving the use of particle filters in spaces of a higher dimension, greatly increasing the complexity of the involved simulation. Statistical simulations of this nature require a lot of pseudorandom numbers that can be generated efficiently and meet certain quality criteria. The process of generating pseudorandom number becomes the critical point of such applications in terms of performance. In this context, reconfigurable computing is a technology capable of meeting the demand for high performance of large statistical simulations because systems based on reconfigurable architectures have the potential to map computation to hardware aiming to increase efficiency without a serious drawback in flexibility. Reconfigurable technologies are also attractive because of their low energy consume, a essential feature for the future mobile robots. This dissertation presents an implementation of a FPGA based embedded system designed to solve the robot localization problem by the means of probabilistic technics. The fundamental part from the whole system is a fast random number generator mapped to reconfigurable hardware wich attends a rigid quality criteria.

Keywords: mobile robotics, localization algorithm, particle filter, pseudorandom number generation, embedded systems, reconfigurable computing.

CAPÍTULO 1

Introdução

“Robotic control system design is a logical next step for embedded system design teams.”

– Warren Webb, 2005

1.1 Contextualização e Motivação

Nas últimas décadas têm-se observado um rápido crescimento no potencial de aplicação de sistemas robóticos. Os avanços tecnológicos gradativamente proporcionaram subsídios para o surgimento de sensores e outros dispositivos eletrônicos extremamente eficientes e disponíveis a preços acessíveis. Este fato causou um grande impacto nas pesquisas em robótica. Em um primeiro momento, houve o desenvolvimento da robótica industrial. A utilização de robôs no chão de fábrica aumentou vertiginosamente a eficiência dos processos produtivos além de reduzir seus custos [MATARIĆ, 2007]. Atualmente, os pesquisadores em robótica vêm dirigindo seus esforços para a construção de robôs móveis e introduzindo novas capacidades nos robôs a fim de habilitá-los a agir de forma autônoma. A robótica móvel é um vasto campo a ser explorado e possibilita que uma ampla gama de novas aplicações sejam desenvolvidas [GATES, 2007]. Contudo, os desafios enfrentados por robôs móveis encontram-se muito além das dificuldades já superadas pelos primeiros autômatos industriais e a busca por soluções destes problemas concentra a atenção de muitos cientistas.

O robô móvel autônomo, um dos atuais objetos de estudo da robótica, deve ser capaz de perceber o estado de seu ambiente e agir de forma apropriada combinando informações

de diferentes fontes. Os problemas enfrentados no desenvolvimento de robôs móveis geralmente estão relacionados à necessidade de interação adequada com os objetos físicos e entidades de um ambiente desestruturado, desconhecido e dinâmico (pior caso). A capacidade de lidar com a incerteza de ambientes dinâmicos é um ponto chave para os robôs contemporâneos [THRUN, 2000a]. Um robô que carrega consigo a noção de sua própria incerteza e age de acordo com ela, certamente, desempenhará melhor sua função.

Diferentes metodologias foram elaboradas para o desenvolvimento de *software* de controle robótico, indo desde paradigmas baseados em modelos extremamente formais (nos meados da década de 70) até tratamentos puramente reativos, fortemente baseados em sensoramento (nos meados da década de 80). A partir dos anos 90 surgiram modelos híbridos que mesclavam modelagem formal em níveis mais altos com um certo grau de reatividade em níveis mais baixos [MATARIĆ, 2007]. Uma das abordagens desenvolvidas, a robótica probabilística, promoveu recentemente um grande avanço na área. Metodologias de caráter probabilístico, cujas origens pode ser traçada até o início dos anos 60, proporcionaram aos robôs móveis graus de autonomia e robustez nunca antes observados e soluções implementadas segundo seus preceitos demonstram grande escalabilidade em aplicações complexas. Recentemente, a abordagem probabilística tem-se tornado dominante na resolução de vários problemas da robótica móvel [THRUN et al., 2005]. Suas bases remontam solidamente à teoria das probabilidades e à estatística e a idéia central adotada é a de representar a informação utilizada na computação de forma probabilística. Os louváveis méritos da robótica probabilística se devem ao fato da teoria das probabilidades conseguir expressar elegantemente e explicitamente, ou seja; de maneira matemática e formal, a incerteza encontrada nas aplicações da robótica móvel por meio de variáveis aleatórias e distribuições de probabilidade. No entanto, há críticas à abordagem probabilística. Tradicionalmente, as limitações mais freqüentemente citadas a respeito de algoritmos probabilísticos são [THRUN, 2001]: (i) algoritmos probabilísticos geralmente são ineficientes se comparados a algoritmos tradicionais, pois a computação deve ser realizada sobre distribuições de probabilidade inteiras e não sobre variáveis determinísticas convencionais; (ii) algoritmos probabilísticos têm necessidade de realizar aproximações e não trabalham com valores exatos.

As limitações citadas, até certo ponto, não representam obstáculos intransponíveis para os modernos computadores baseados em processadores de propósito geral, apesar de certos algoritmos necessitarem de considerável poder computacional para serem executados em tempo real até mesmo nos *Personal Computers* (PCs) de última geração. Contudo, o custo de se ter um PC em um robô móvel é proibitivo para a maioria das aplicações com potencial de exploração comercial. Além do fator preponderante do custo, é de extrema importância considerar que tais aplicações requerem funcionamento em tempo real e um projeto mecânico compacto que requer baixo consumo, uma vez que robôs são alimentados por baterias. Sistemas embarcados compartilham estes requisitos [WEBB,

2005] e apresentam-se como uma alternativa para atenuar os custos dos robôs móveis valendo-se de projetos otimizados para executar tarefas específicas eficientemente e com baixo consumo de energia.

Controlar robôs por meio de sistemas embarcados envolve algumas questões problemáticas [WOLF et al., 2007; WEBB, 2005]: desenvolver *drivers* de dispositivos específicos, prover comunicação em rede; implementar algoritmos por meio de um *software* enxuto e sob severas limitações de *hardware*. Apesar de acarretarem uma redução do custo dos robôs, os computadores embarcados de propósito específico, dotados de recursos computacionais mais modestos, podem ser incapazes de atender aos requisitos atuais de desempenho dos algoritmos probabilísticos. Obter um desempenho satisfatório para estes complexos algoritmos ao executá-los em dispositivos dedicados ainda é uma tarefa extremamente desafiadora, mas suas conseqüências prometem ser compensadoras. Possíveis soluções para esta questão são propostas de mudanças e personalizações na arquitetura do sistema embarcado que melhorem significativamente o seu desempenho global. As mudanças na arquitetura diferem enormemente de acordo com a aplicação a ser considerada, mas essa técnica mostra-se eficaz ao se levar em conta um sistema de propósito específico, tal como um robô móvel [ARAGÃO et al., 2000].

Tradicionalmente, as otimizações e personalizações na arquitetura de sistemas computacionais dedicados são realizadas por meio da utilização de circuitos digitais integrados especialmente projetados para tal finalidade; os *Application Specific Integrated Circuits* (ASICs). Como qualquer circuito integrado, um ASIC tem sua lógica interna invariável fundida na superfície de um substrato de material semicondutor. Esta tecnologia supera as arquiteturas de propósito geral em desempenho, porém restringe a flexibilidade da arquitetura e exclui totalmente qualquer tipo de otimização ou atualização em tempo de pós-projeto [COMPTON e HAUCK, 2002]. Ademais, o processo de fabricação de ASICs só está acessível às grandes corporações e, na prática, milhares de unidades de um único modelo de *chip* têm que ser fundidas para que sua manufatura seja viável em termos comerciais.

Neste contexto apresentado, a tecnologia de computação reconfigurável mostra-se como opção interessante para preencher a lacuna deixada pelos ASICs e pelos PCs. A estrutura arquitetural da computação reconfigurável mune os desenvolvedores com uma opção intermediária entre os ASICs e as arquiteturas de propósito geral, localizando-se em um ponto intermediário do *continuum hardware – software*. Valendo-se de *hardware* que pode ser programado em campo com o objetivo de promover flexibilidade sem comprometer seriamente o desempenho da aplicação, a computação reconfigurável consegue basicamente combinar a adaptabilidade do *software* com a velocidade do *hardware* convencional [DEHON e WAWRZYNEK, 1999]. Significativos ganhos de desempenho podem ser obtidos com a utilização desta tecnologia, pois o *hardware* utilizado é otimizado para

executar os algoritmos específicos necessários para a aplicação. Em outras palavras, para se obter um melhor desempenho de um algoritmo, este deve ser executado num *hardware* especificamente projetado para ele.

O dispositivo preferencial para o desenvolvimento de arquiteturas reconfiguráveis é o *Field Programmable Gate Array* (FPGA). Basicamente, um FPGA pode ser visto como uma matriz de elementos lógicos e uma rede de interconexão configuráveis via *software*. Desde sua introdução, nos meados dos anos 80, os FPGAs têm recebido grande destaque dentro da área de computação reconfigurável e se tornado uma escolha popular para prototipação de circuitos e produção de produtos em pequenas e médias quantidades [PLAVEC, 2004]. Um FPGA é um tipo de *Programmable Logic Device* (PLD) capaz de implementar circuitos digitais arbitrários e ser reprogramado indefinidamente estando limitado apenas à sua capacidade de conter o projeto de *hardware* em sua estrutura interna.

Graças aos avanços da microeletrônica, os FPGAs estão se tornando cada vez mais densos em termos de elementos lógicos por área de *chip*. A capacidade dos modelos mais modernos de FPGA chega a cerca de 820.000 elementos lógicos [SHAW, 2009]. Essas características tornaram possível a construção, em FPGA, de sistemas complexos formados por um ou mais processadores e blocos de propriedade intelectual e lógica definida pelo usuário. Esses sistemas são denominados *System on Programmable Chip* (SOPC). SOPCs têm se mostrado uma excelente opção de implementação para sistemas embarcados que necessitam de um alto desempenho [COMPTON e HAUCK, 2002]. Deste modo, todo um sistema multiprocessado completo pode estar contido em um único dispositivo eletrônico, tornando o conceito de SOPC cada vez mais viável na prática dos projetistas de sistemas. Componentes padrão comuns a diferentes SOPCs tendem a se tornar núcleos reutilizáveis de *hardware* para sistemas reconfiguráveis e serem fornecidos ou comercializados na forma de blocos de propriedade intelectual. O reuso de componentes torna o ciclo de desenvolvimento e o *time to market* de produtos baseados em FPGAs mais curto do aqueles baseados em ASICs.

Tipicamente, o desenvolvimento de circuitos para FPGAs é realizado por meio de ferramentas *Electronic Design Automation* (EDA) que englobam todas as etapas do processo: projeto, simulação, síntese e configuração do *hardware*. Os modernos *softwares* EDA estão aptos a construir e processar unidades funcionais de sistemas de *hardware* nos mais variados formatos, indo desde diagramas esquemáticos até descrições de circuitos em linguagens de alto nível. Conforme a complexidade dos circuitos cresce, as cada vez mais sofisticadas *Hardware Description Languages* (HDLs) tornam-se a única escolha viável para sua implementação. A utilização de HDLs possibilita que os detalhes de baixo nível fiquem sob responsabilidade das ferramentas de *software* e permite que os projetistas foquem sua atenção no desenvolvimento das funcionalidades do sistema [PLAVEC, 2004],

tonando o processo mais produtivo.

Hoje em dia, a tecnologia do FPGA já é plenamente consolidada e dominada por diversos fabricantes. *Xilinx* e *Altera* são as corporações líderes do mercado, controlando uma fatia de cerca de 80%. Outros competidores neste mercado incluem *Lattice Semiconductor*, *Actel*, *Atmel*, *SiliconBlue Technologies* e *QuickLogic*. A tecnologia já é difundida em diversas áreas e *chips* reconfiguráveis, muito compactos, eficientes e capazes de implementar *hardware* complexo, de baixo consumo, estão disponíveis a preços acessíveis. A versatilidade envolvida no ciclo de desenvolvimento de produtos baseados em FPGA torna a utilização desta tecnologia uma opção interessante para ser aplicada à robótica móvel embarcada. Contudo, há ainda uma lacuna semântica entre o modelo de circuitos lógicos, que é característico da programação para FPGAs e que é usado por projetistas de circuitos, e as expressões algorítmicas de uma linguagem de alto nível, que é utilizada por programadores de aplicações [BONDALAPATI e PRASANNA, 2002]. Essa lacuna tem contribuído, muitas vezes, para que pesquisadores ligados ao desenvolvimento de algoritmos aplicados à robótica não adotem soluções em FPGA e sim no já bastante familiar padrão PC. Mais especificamente, o *notebook* é muitas vezes adotado como plataforma de *hardware* nas atuais pesquisas no campo da robótica móvel. De certo modo, essa postura freia o desenvolvimento de um controle robótico fundado em tecnologias embarcadas de baixo custo e baixa potência. Outras alternativas ao *notebook* bastante utilizadas são as placas padrão *PC/104* [PC/104 Embedded Consortium, 2009].

1.2 Objetivos do Projeto de Pesquisa

Motivado pelos desafios descritos na Seção (1.1), o projeto de pesquisa em desenvolvimento tem como objetivo a construção e validação de um SOPC que seja capaz de realizar computação probabilística adequada às atuais aplicações da robótica móvel. O SOPC em questão deve se basear nas tecnologias, ferramentas e processos criados pela indústria de EDA voltados ao desenvolvimento de *hardware* e sua complexa implementação é de caráter multidisciplinar, como ilustrado pela Figura (1.1).

O SOPC projetado implementará o filtro de partículas com um co-projeto de *hardware/software* no qual toda computação mais pesada e intensiva será efetuada por blocos de *hardware* integrados ao processador *Nios II* como instruções personalizadas. O primeiro passo para tanto é a implementação de instruções para geração de números pseudo-aleatórios, componente fundamental para qualquer computação que envolva variáveis aleatórias. A geração de números pseudo-aleatórios é um processo complexo e, para a aplicação em questão, certamente é um dos pontos críticos em termos de exigências computacionais. Dessa forma, mapear um gerador de números pseudo-aleatórios ao *hardware* da base reconfigurável solucionará grande parte dos gargalos de desempenho

Figura 1.1: Projeto de pesquisa visto como intersecção multidisciplinar de diferentes áreas

encontrados atualmente.

Após um levantamento intensivo acerca dos geradores existentes chegou-se à conclusão que a melhor opção para o projeto é o *Mersenne Twister* (MT) (vide Seção (2.2.4)). Este gerador tem sua implementação baseada em profundos fundamentos matemáticos. Seus criadores o projetaram para superar as deficiências dos seus antecessores e foram bem sucedidos na tentativa: o MT gera números pseudo-aleatórios de alta qualidade muito eficientemente [MATSUMOTO e NISHIMURA, 1998]. Existem diversas variantes do MT capazes de gerar números pseudo-aleatórios de tamanhos distintos em termos de número de *bits*; as mais utilizadas geram números de 32 *bits*. O MT19937, a variante mais popular do MT para números de 32 *bits*, foi escolhida como alvo do projeto. Esta escolha pode ser justificada pelo fato de que o tamanho da palavra do processador *Nios II* também é de 32 *bits*, o que tornará a integração mais facilitada.

Estando o gerador de números pseudo-aleatórios implementado é possível então gerar variáveis de distribuições de probabilidade arbitrárias. Mais detalhes a respeito são encontrados na Seção (2.2). As distribuições de probabilidade mais utilizadas no campo da robótica móvel são a distribuição normal e a distribuição exponencial [THRUN et al., 2005]. Também é objetivo secundário deste projeto implementar mecanismos para gerá-las em *hardware*, visando computação de alto desempenho. Contudo, muitas das transformações matemáticas utilizadas para produzir distribuições de probabilidade contínuas complexas fazem uso de funções transcendentais de difícil mapeamento para *hardware*, tais como senos, cossenos e logaritmos. Implementar este tipo de computação totalmente em *hardware* requer o uso de técnicas específicas como o *COordinate Rotation DIgital Computer* (CORDIC) [VOLDER, 1959; ANDRAKA, 1998]. Pretende-se integrar o módulo do gerador de números pseudo-aleatórios a módulos que implementem o CORDIC para solucionar este problema.

O sistema deve ser implementado em FPGAs de alto desempenho, mais especificamente em dispositivos da família Stratix [ALTERA, 2006]. Esta família de FPGAs é suficientemente densa em termos de elementos lógicos para acomodar *hardware* complexo, e suficientemente veloz para atender minimamente os requisitos da aplicação em questão.

A arquitetura reconfigurável a ser utilizada será baseada no processador embarcado *Nios II*. O *Nios II* é altamente flexível e um dos processadores *softcore* mais utilizados no mundo todo para se implementar SOPCs em FPGAs [ALTERA, 2007b]. A equipe do Laboratório de Computação Reconfigurável do ICMC/USP já desenvolveu diversas aplicações com o *Nios II* e possui todos os recursos necessários para desenvolver mais este projeto. A meta principal do projeto é agregar ao processador *Nios II* capacidades probabilísticas por meio do mecanismo de personalização de instruções. O conjunto de instruções do *Nios II* é extensível e possibilita que os projetistas de sistemas adicionem lógica personalizada à *Arithmetic Logic Unit* (ALU). A utilização de instruções personalizadas visa acelerar trechos críticos de algoritmos implementados em *software* que não atendem os requisitos de desempenho estabelecidos para o sistema. Uma sequência complexa de instruções padrão pode, por exemplo, ser reduzida a uma única instrução personalizada implementada totalmente em *hardware*. O *Nios II* suporta instruções personalizadas de diversos tipos, todas implementadas em HDL. As instruções podem ser executadas em um ou mais ciclos de *clock* e, se for necessário, é possível incluir blocos de *hardware* externos ao *datapath* do processador. A interface de *software* disponível para acesso às instruções personalizadas é de fácil utilização e consegue abstrair adequadamente os detalhes de sua implementação.

O projeto de pesquisa terá como foco uma aplicação muito utilizada na robótica móvel: o filtro de partículas para localização em ambientes já mapeados ou *Monte Carlo Localization* (MCL). O desenvolvimento dos filtros de partículas não é uma contribuição do campo da robótica mas sim uma abordagem genérica de estimação de parâmetros que demonstrou sucesso nesta área de aplicação e foi devidamente apropriada. O objetivo do filtro de partículas é rastrear a evolução de certas variáveis de interesse de um sistema dinâmico em função do tempo [REKLEITIS, 2004]. Tipicamente, estas variáveis são caracterizadas de acordo com uma função de distribuição de probabilidade gaussiana e multimodal. A idéia básica do filtro de partículas é construir uma representação da função de distribuição das variáveis de interesse por meio de um conjunto de amostras aleatórias das mesmas. Múltiplas cópias das variáveis, as partículas, formam o conjunto em questão. O filtro de partículas é recursivo por natureza e opera em duas fases: predição e atualização. Em um primeiro momento, a execução de ações no sistema dinâmico modifica o estado das partículas segundo algum modelo de atuação. Posteriormente o conjunto de partículas é atualizado baseando-se em um peso associado a cada partícula que representa a qualidade e verossimilhança da mesma.

1.3 Organização da Dissertação

As Seções (2.1), (2.2) e (2.3) apresentam o levantamento bibliográfico teórico necessário para execução do projeto de pesquisa, e uma análise comentada dos artigos mais relevantes já publicados na matéria de enfoque da dissertação; contribuindo dessa forma para uma melhor compreensão por parte do leitor. Respectivamente, são descritos os seguintes campos de interesse: robótica probabilística, geração de números aleatórios e computação reconfigurável.

A Seção (2.4) analisa esforços científicos já realizados que serviram como base ou inspiração para a concepção do mestrado.

O Capítulo (3) descreve a metodologia adotada para o desenvolvimento do trabalho, bem como os recursos, técnicas, processos e ferramentas empregados para sua realização.

O Capítulo (4) discorre efetivamente sobre o desenvolvimento da pesquisa e detalha a implementação efetuada.

Finalmente, o Capítulo (5) expõe as conclusões advindas desta pesquisa, analisa a importância das contribuições científicas produzidas, e sugere trabalhos futuros que poderiam aprimorar e dar continuidade à pesquisa apresentada na dissertação de mestrado.

CAPÍTULO 2

Fundamentação Teórica

2.1 Robótica Probabilística

“I personally believe that probability is at the foundation of any intelligence in an uncertain world where you can’t know everything.”

– Eric Horvitz, 2003

2.1.1 Modelagem de Sistemas Robóticos

Atualmente, a modelagem de sistemas dinâmicos é uma importante ferramenta de análise estatística e vem atraindo a atenção de pesquisadores de diversas áreas. Modelos de natureza dinâmica têm sido utilizados nas mais variadas situações, como por exemplo: aprendizado em modelos gráficos, simulação de estruturas de proteínas, sistemas especialistas probabilísticos e otimização combinatória. Apesar de suas complexidades computacionais, esta abordagem é aplicada com bastante sucesso, particularmente no estudo de séries temporais e em sistemas de controle [LIU e CHEN, 1998]. Isto a caracteriza como geral o suficiente para ser aplicada a toda uma vasta gama de situações envolvendo percepção e ação no mundo real [THRUN, 2000a], como é o caso dos robôs móveis.

A modelagem de sistemas robóticos tem o objetivo de arquitetar e expressar formalmente o mecanismo de percepção e controle dos robôs. Prover percepção a um robô consiste em muní-lo dos meios necessários para estimar seu próprio estado a partir da combinação das evidências de seus sensores. Odômetros e instrumentos de medição de

distância baseados em tecnologias como o sonar e o LASER são sensores tipicamente presentes em robôs móveis. Fundamentando-se em seu estado previamente inferido, o robô está apto a desempenhar ações e interagir com o ambiente. A capacidade de decidir realizar ou não estas ações e incorporar seus resultados, tanto perante o ambiente quanto o próprio robô, caracteriza o controle robótico. Podem-se citar como ações características de um robô móvel, a ativação de manipuladores e motores acoplados às rodas.

Tradicionalmente, a cinemática e a dinâmica de dispositivos robóticos são descritas de maneira determinística. Este tipo de modelagem caracteriza robôs idealizados livres de qualquer imprecisão e ruídos, sendo adequada somente aos casos mais simples, tal como um ambiente de manufatura altamente controlado. Ao se valer desta abordagem assume-se que o estado do robô possa ser recuperado a qualquer momento, sem erro algum, a partir das medições de seus sensores, e que uma determinada ação de controle sempre irá conduzir o robô a um determinado estado válido [THRUN, 2001]. Do ponto de vista matemático, o modelo clássico de um sistema robótico pode ser descrito como uma quintupla ordenada (Z, X, U, f, g) na qual:

- Z é o conjunto de todas as possíveis medições captadas pelos sensores do robô;
- X é o conjunto dos possíveis estados válidos que descrevem o robô ou o robô e outros objetos de interesse;
- U é o conjunto das possíveis ações de controle que o robô pode desempenhar;
- $f : X \times U \rightarrow X$ é uma função determinística de transição de estado;
- $g : Z \rightarrow X$ é uma função determinística de recuperação de estado.

Se $x_t \in X$ representar o estado do robô no instante $t \geq 0$, o efeito de se realizar uma ação de controle $u_t \in U$ o conduz ao um novo estado $x_{t+1} \in X$ igual a $f(x_t, u_t)$. O estado do robô é imediatamente recuperado por intermédio das medições de seus sensores em um dado instante, ou seja, $x_t = g(z_t)$ com $z_t \in Z$.

Contrapondo-se à abordagem tradicional, o controle robótico probabilístico tem como fundamento a antecipação de diversas contingências que podem surgir em um ambiente real e incerto por natureza, mesclando assim a captura de informações (*exploration*) com a tomada de decisões orientada a desempenho (*exploitation*). O modelo probabilístico também pode ser encarado como uma generalização do modelo clássico [THRUN, 2001]. Os sensores dos robôs são modelados de acordo com suas limitações e a presença de ruído. Por exemplo, câmeras não podem ver através das paredes e a distância aferida por um sonar nem sempre corresponde à distância verdadeira até um possível obstáculo. As ações desempenhadas pelos robôs nunca são encaradas como produtoras de resultados previamente esperados; pelo contrário, realizar ações aumentam a incerteza do ambiente. Um ambiente modelado de acordo com a abordagem probabilística é geralmente representado através de distribuições de probabilidade que descrevem a dependência de certas variáveis em relação a outras em termos condicionais. O estado do robô também é representado

utilizando-se distribuições de probabilidade que são deduzidas integrando-se as informações dos sensores ao modelo do ambiente.

Matematicamente, o modelo probabilístico para sistemas robóticos é uma quintupla ordenada (Z, X, U, f, g) na qual:

- Z , X e U têm a mesma semântica do modelo clássico;
- $f : X \times U \times X \rightarrow [0, 1]$ é uma função de distribuição de probabilidade que descreve as possibilidades de transição de estado do robô causadas por uma ação de controle;
- $g : X \times Z \rightarrow [0, 1]$ é uma função de distribuição de probabilidade que descreve os possíveis estados do robô inferidos a partir de seu sensoramento.

Dado o estado do robô no instante $t \geq 0$, x_t , a realização de uma ação u_t o conduz a um novo estado x_{t+1} com probabilidade expressa por $f(x_t, u_t, x_{t+1}) = p(x_{t+1}|x_t, u_t)$. Para quaisquer x_t e u_t tem-se que $\int f(x_t, u_t, x_{t+1}) dx_{t+1} = 1$, ou seja, fixada uma ação de controle e um estado, as probabilidades de transição para todos os possíveis estados futuros em X devem perfazer 100%. O estado no instante $t+1$ não é determinístico: para cada elemento de X existe uma probabilidade de transição, mesmo que esta seja igual a zero. Nota-se portanto que a execução de uma série de ações $(u_t, u_{t+1}, u_{t+2} \dots)$ torna a determinação do verdadeiro estado mais difícil. Dadas certas medições dos sensores, z_t , a probabilidade do estado do robô ser x_t é expressa por $g(x_t, z_t) = p(x_t|z_t)$. O sensoramento também pode ser modelado de maneira inversa com $g(x_t, z_t) = p(z_t|x_t)$ dependendo dos detalhes do algoritmo que será utilizado. Geralmente, esta modelagem é utilizada quando o algoritmo vale-se de métodos de inferência bayesiana e aproveita as evidências observadas para atualizar a probabilidade de que uma certa hipótese seja verdadeira.

Os parâmetros que descrevem as distribuições de probabilidade f e g foram omitidos anteriormente com o intuito de contribuir para uma melhor compreensão do modelo matemático. Eles podem ser arbitrários e são especificados de acordo com o problema a ser solucionado. Normalmente, o sensoramento é modelado combinando-se uma distribuição gaussiana para lidar com ruído e uma distribuição exponencial para tratar obstáculos inesperados. A função de transição de estados habitualmente descreve o movimento do robô e tem distribuição normal ou triangular [THRUN et al., 2005], exemplificadas graficamente pela Figura (2.1).

2.1.2 Estimação Probabilística de Estado

A maior parte da pesquisa produzida em robótica móvel está relacionada ao desenvolvimento de algoritmos de localização e mapeamento [THRUN et al., 2005]. Se encarados de forma probabilística, estes algoritmos são reduzidos a problemas de estimação de parâmetros. Estimar parâmetros que descrevem a posição do robô em relação a um sistema de coordenadas externo caracteriza um problema de localização. Estimar parâmetros que

Figura 2.1: Funções de distribuição de probabilidade normal (a) e triangular (b) com desvio padrão α

descrevem a posição de entidades do ambiente define um problema de mapeamento. O mapeamento pode se tornar ainda mais árduo se os objetos alvo mudarem de posição com o passar do tempo.

O procedimento mais utilizado para estimação probabilística de estado em robótica é denominado filtro de Bayes [DIARD et al., 2003]. Por meio deste método, é possível estimar a distribuição de probabilidade sobre o espaço de estados baseando-se nas ações e sensoriamientos do robô. Diversos modelos estatísticos podem ser classificados como instâncias do filtro de Bayes [DIARD et al., 2003].

2.1.3 Filtros de Partículas

Uma implementação popular do filtro de Bayes na área da robótica é o filtro de partículas, um método de Monte Carlo sequencial utilizado para aproximar inferências em sistemas dinâmicos que podem ser modelados por meio de cadeias de Markov parcialmente observáveis [THRUN, 2002a]. Este algoritmo aproxima o estado do robô valendo-se de um conjunto de partículas que representam amostras do espaço de estados distribuídas de acordo com uma função de probabilidade.

A adequação de filtros de partículas como ferramentas de programação para robôs tem sido demonstrada na prática por meio de uma sucessão de experimentos bem sucedidos de crescente complexidade: i) aprendizado de comportamento simples, ii) combinação de comportamentos simples, iii) composição hierárquica de comportamentos e iv) reconhecimento de situações. Séries de experimentos como a citada também podem ser integradas com o intuito de solucionar problemas considerados difíceis no campo da robótica [DIARD et al., 2003], contando ainda com a possibilidade de desenvolvimento incremental. Os êxitos iniciais dos filtros de partículas aplicados à robótica são atribuídos à solução de problemas de localização em ambientes previamente conhecidos [DELLAERT et al., 1999]. Nestes casos, tipicamente a posição do robô deve ser recuperada a partir de inferências realizadas sobre os dados provenientes de seus sensores. Os filtros de partículas também foram capazes de solucionar dois importantes problemas da robótica móvel, am-

bos anteriormente sem solução: o problema de localização global e o problema do “robô seqüestrado”, nos quais o robô deve recuperar sua posição sob condições de incerteza global. Maiores detalhes sobre estes problemas podem ser encontrados em [THRUN et al., 2005]. Tais esforços de pesquisa proporcionaram um relevante aumento na robustez dos robôs móveis e o problema de localização em um ambiente previamente mapeado já é considerado solucionado. Mais recentemente, os filtros de partículas têm sido empregados com o intuito de resolver problemas de dimensionalidade muito maior. Exemplos proeminentes incluem tentativas de *Simultaneous Localization and Mapping* (SLAM). Uma implementação neste sentido já foi realizada por Bonato [BONATO, 2008].

Os filtros de partículas, por meio de inferência bayesiana, estimam *a posteriori* o estado de um sistema dinâmico modelado por meio de uma cadeia de Markov a tempo discreto. Suponha que o estado da cadeia de Markov no instante t seja dado por x_t . Em uma cadeia de Markov controlável [ARAPOSTATHIS et al., 2001; BORKAR, 1991], o estado x_t depende unicamente do estado anterior x_{t-1} e da ação de controle u_t desempenhada no intervalo de tempo $(t-1, t]$ de acordo com a probabilidade $p(x_t|u_t, x_{t-1})$. O estado x_t da cadeia de Markov não é diretamente observável. Contudo, tem-se acesso a z_t , uma projeção estocástica do verdadeiro estado x_t , que é dada de acordo com a probabilidade $p(z_t|x_t)$. A princípio, o estado da cadeia no instante inicial, x_0 , está distribuído no espaço de estados segundo a distribuição $p(x_0)$.

Para a robótica probabilística, o estado x_t da cadeia de Markov normalmente representa a posição de um ou mais entes do ambiente, sejam eles o próprio robô ou outros objetos quaisquer, tal como uma porta ou um obstáculo, por exemplo. Em problemas de localização em ambientes bidimensionais o estado da cadeia representa a posição do robô e é dado pelo vetor $^{(2)}x_t = [x, y, \theta]^T$ que especifica suas coordenadas cartesianas e orientação, vide Figura (2.2). Para a localização em ambientes tridimensionais tem-se de forma análoga $^{(3)}x_t = [x, y, z, \theta, \rho, \phi]^T$. Conforme cresce a complexidade do problema a ser resolvido, a dimensão do vetor estado a ser estimado pelo filtro de partículas também cresce. A projeção z_t da verdadeira posição do robô é representada pelos dados obtidos dos sensores e a ação de controle u_t é disparada pela unidade de controle do robô. A probabilidade $p(x_t|u_t, x_{t-1})$ é denominada modelo de atuação ou modelo de movimento e a probabilidade $p(z_t|x_t)$ é denominada modelo de sensoramento. Estes modelos geralmente são geométricos e generalizam as noções determinísticas da cinemática e da dinâmica definidas pela robótica clássica [THRUN, 2002b,a].

Um problema clássico envolvendo cadeias de Markov parcialmente observáveis é o de recuperar, em qualquer instante t , a distribuição posterior sobre o estado x_t tendo em mãos todas as projeções obtidas, $z^t = z_0, \dots, z_t$, e todas as ações de controle desempenhadas, $u^t = u_0, \dots, u_t$. Uma solução para este problema é dada pelo filtro de Bayes [THRUN, 2002a], que calcula a distribuição posterior recursivamente sob a condição inicial

Figura 2.2: Estado como posição do robô em um sistema de coordenadas 2D

$$p(x_0|z^0, u^0) = p(x_0):$$

$$p(x_t|z^t, u^t) = p(z_t|x_t) \int p(x_t|u_t, x_{t-1})p(x_{t-1}|z^{t-1}, u^{t-1})dx_{t-1} \quad (2.1)$$

Se os estados, controles e sensoriamentos forem todos discretos a cadeia de Markov é equivalente a um *Hidden Markov Model* (HMM) e a expressão (2.1) pode ser implementada de forma que sua computação seja exata. Todavia, em robótica, os filtros de partículas são geralmente aplicados a espaços de estados contínuos. Nestes casos, formas exatas de se calcular (2.1) são conhecidas somente para casos extremamente especializados. Se $p(x_0)$ é gaussiana e $p(x_t|u_t, x_{t-1})$ e $p(z_t|x_t)$ são funções lineares de seus parâmetros com ruído gaussiano independente, tem-se o filtro de Kalman. Uma aproximação comum para sistemas não lineares e não gaussianos é linearizar os modelos de atuação e sensoriamento. Se a linearização for obtida por meio de uma expansão de primeira ordem da série de Taylor, tem-se o *Extended Kalman Filter* (EKF) [THRUN, 2002b].

Os filtros de partículas adequam-se ao caso mais geral, ou seja, cadeias de Markov sem restrições. A idéia básica deste algoritmo é aproximar a distribuição posterior de x_t por meio de um conjunto X_t de M amostras, ou partículas, do espaço de estados. Cada elemento de X_t é uma amostra concreta do espaço de estados. Quando $M \rightarrow \infty$, o procedimento recursivo do algoritmo faz o conjunto de partículas convergir uniformemente para a distribuição desejada $p(x_t|z^t, u^t)$. A versão mais básica do filtro de partículas é dada pelo Algoritmo (2.1). Em uma primeira fase, a fase de predição, o modelo de movimento é utilizado com o intuito de prever o estado do robô. Em uma segunda fase, a fase de atualização, o modelo de sensoriamento é utilizado para incorporar as informações dos sensores.

Filtros de partículas são opções atrativas para a robótica, pois podem ser aplicados a praticamente qualquer problema que seja modelado como uma cadeia de Markov. Sua acurácia aumenta conforme os recursos computacionais tornam-se mais abundantes, pois quanto mais recursos computacionais estiverem disponíveis, mais partículas poderão ser utilizadas simultaneamente. O algoritmo básico do filtro de partículas é de fácil implementação – não há necessidade de linearizar modelos não lineares muito menos preocupar-se

Algoritmo 2.1: Filtro de Partículas

Entrada: Instante t , Conjunto de partículas da iteração anterior X_{t-1} , ação de controle u_t , dados de sensoramento z_t

Saída: Conjunto de partículas da próxima iteração X_t

```

1 início
2   se  $t = 0$  então
3     /* Inicialização */
4     Faça  $X_0$  um conjunto vazio:  $X_0 \leftarrow \emptyset$  ;
5     para  $i \in \{1 \dots M\}$  faça
6       Sorteie  $x_0^{(i)}$  de acordo com distribuição inicial  $p(x_0)$  ;
7       Adicione  $x_0^{(i)}$  ao conjunto  $X_0$ :  $X_0 \leftarrow X_0 \cup \{x_0^{(i)}\}$  ;
8     O conjunto de partículas inicial é  $X_0$ :  $X_t \leftarrow X_0$  ;
9   senão
10    /* Fase de Predição */
11    Faça  $\bar{X}_t$  um conjunto vazio:  $\bar{X}_t \leftarrow \emptyset$  ;
12    para  $i \in \{1 \dots |X_{t-1}|\}$  faça
13      Recupre a  $i$ -ésima partícula do conjunto  $X_{t-1}$ ,  $x_{t-1}^{(i)}$  ;
14      Sorteie  $\bar{x}_t^{(i)}$  de acordo com o modelo de atuação  $p(\bar{x}_t|u_t, x_{t-1}^{(i)})$  ;
15      Adicione  $\bar{x}_t^{(i)}$  ao conjunto  $\bar{X}_t$ :  $\bar{X}_t \leftarrow \bar{X}_t \cup \{\bar{x}_t^{(i)}\}$  ;
16    /* Fase de Atualização */
17    Faça  $X_t$  um conjunto vazio:  $X_t \leftarrow \emptyset$  ;
18    para  $i \in \{1 \dots |\bar{X}_t|\}$  faça
19      Recupre a  $i$ -ésima partícula do conjunto  $\bar{X}_t$ ,  $\bar{x}_t^{(i)}$  ;
20      Sorteie, com reposição,  $x_t^{(i)}$  com probabilidade proporcional ao modelo
21      de sensoramento  $p(z_t|\bar{x}_t^{(i)})$  ;
22      Adicione  $x_t^{(i)}$  ao conjunto  $X_t$ :  $X_t \leftarrow X_t \cup \{x_t^{(i)}\}$  ;
23  retorna  $X_t$  ;
24 fim

```

com a forma do modelo de atuação. Cabe também ressaltar, que para o Filtro de Partículas ter um desempenho satisfatório, é necessário trabalhar com o número de partículas na casa dos milhares.

2.1.4 Controle Probabilístico

Os métodos de controle robótico probabilístico, se comparados aos de estimação de estado, ainda estão pouco desenvolvidos. Uma justificativa plausível para este fato encontra-se na grande complexidade computacional, própria dos processos de tomada de decisão. Entretanto, o controle probabilístico tem recebido considerável atenção recentemente. Os algoritmos existentes podem ser agrupados em duas classes principais: gulosos (*greedy*) e

não-gulosos (*non-greedy*) [THRUN, 2001].

As duas categorias de algoritmos associam uma função de custo às ações de controle. Os algoritmos gulosos maximizam o ganho local levando-se em consideração somente o benefício que pode ser extraído da próxima ação a ser realizada. Os não gulosos atentam a um ganho mais global, considerando-se toda uma seqüência de controle e maximizando o seu ganho cumulativo. A utilização de métodos gulosos é interessante visto que considerar distintas possibilidades, durante o planejamento de ações em longo prazo em um ambiente incerto, ainda é computacionalmente complexo. Desenvolver eficientemente abordagens não-gulosas sobre múltiplos passos ainda é um problema desafiador.

Um algoritmo de planejamento de trajetória que merece destaque é o de navegação costeira (*coastal navigation*) [ROY e THRUN, 1999]. Esta abordagem encontra-se em um ponto intermediário do *continuum* tratabilidade computacional – robustez quando os diversos algoritmos de planejamento de trajetória existentes são levados em consideração. O desenvolvimento deste algoritmo foi originalmente motivado pela observação de navios desprovidos de mecanismos precisos de localização, como o GPS, que se deslocam próximos à costa com o intuito de reduzir o risco de se perderem em alto mar. Navegando desta maneira, os tripulantes são capazes de determinar com maior precisão sua localização já que estão habituados a linha costeira. O sucesso deste método deve-se ao fato da navegação ocorrer em áreas com maior densidade de informação para os marinheiros. O mesmo princípio pode ser aplicado aos robôs móveis: se estes se mantêm suficientemente próximos às áreas do mapa que contém mais informação sobre sua localização, a probabilidade de se perderem é minimizada.

2.2 Geração de Números Aleatórios

2.2.1 Conceitos Fundamentais

Para a maioria dos usuários de computador, o problema de se gerar números aleatórios é considerado solucionado há muito tempo. A maioria dos softwares existentes (bibliotecas padrão de linguagens de programação, compiladores, planilhas etc.), oferece uma ou mais funções capazes de gerar números aleatórios. Supostamente, estas funções produzem números que podem ser utilizados para praticamente todos os propósitos [L'ECUYER,

1989, 1990]. Muitas pessoas utilizam estas funções e ficam satisfeitas com os resultados obtidos. Porém, a pesquisa sobre geradores de números aleatórios está longe de cessar e continua a ser amplamente realizada [PANNETON et al., 2006].

A geração de números aleatórios é um problema fundamental da computação que vem sendo seriamente pesquisado há décadas. Um *Random Number Generator* (RNG) confiável é um ingrediente básico e essencial para qualquer simulação estocástica. A teoria matemática que justifica os métodos de simulação estatística é totalmente baseada nos elegantes conceitos de espaço probabilístico e variável aleatória. Contudo, uma vez que a implementação exata destes conceitos em computadores convencionais é impossível; variáveis aleatórias e outros objetos de natureza probabilística são simulados por meio de algoritmos determinísticos. O objetivo destes algoritmos é produzir seqüências de números cujo comportamento é difícil de ser diferenciado, do ponto de vista da aplicação, daquele exibido por seqüências aleatórias reais [L'ECUYER, 2006]. Um *Pseudo Random Number Generator* (PRNG) é um algoritmo determinístico que emprega métodos aritméticos para gerar uma seqüência de números aproximadamente não correlacionados e que têm propriedades semelhantes a números aleatórios reais [KNUTH, 1981].

A geração de números aleatórios normalmente é decomposta em dois passos distintos, a saber: (1) gerar variáveis aleatórias independentes e identicamente distribuídas (*i.i.d.*) com distribuição uniforme sobre o intervalo $(0, 1)$ e (2) aplicar transformações a estas variáveis *i.i.d.* de distribuição uniforme, com o objetivo de gerar variáveis aleatórias de distribuições arbitrárias [L'ECUYER, 2004]. Os dois passos citados são independentes e os estudiosos destes processos constituem dois diferentes grupos de pesquisa com pouca intersecção entre si. A expressão PRNG, geralmente, refere-se aos algoritmos usados no passo (1).

Do ponto de vista formal, um PRNG pode ser representado por uma quintupla ordenada $G = (\mathcal{S}, \mu, f, \mathcal{U}, g)$ [L'ECUYER, 2006] na qual:

- $\mathcal{S}$ é um conjunto finito que representa o espaço de estados do gerador. A cada iteração, ou seja, cada vez que um número pseudo-aleatório é produzido, o PRNG assume um valor de $\mathcal{S}$. Este valor é suficiente para descrever o gerador em um dado momento.
- μ é uma distribuição de probabilidade sobre $\mathcal{S}$ utilizada para selecionar s_0 , o estado inicial do gerador. O estado s_0 também é denominado semente do gerador.
- $f : \mathcal{S} \rightarrow \mathcal{S}$ é uma função de transição de estados. Para cada número gerado, o estado do gerador é modificado de acordo com f .
- $\mathcal{U}$ é um conjunto que representa o espaço de geração¹. Os números produzidos pertencem necessariamente a $\mathcal{U}$.
- $g : \mathcal{S} \rightarrow \mathcal{U}$ é a função de geração². Esta função mapeia o estado atual do gerador para

¹Tradução livre do original *output space*.

²Tradução livre do original *output function*.

o espaço de geração.

O estado do PRNG evolui de acordo com a recorrência (2.2) e o valor gerado no passo i é dado por (2.3). Os valores $(u_0, u_1, u_2 \dots)$ são denominados números pseudo-aleatórios produzidos pelo PRNG.

$$s_i = f(s_{i-1}) \in \mathcal{S} \quad i \geq 1 \quad (2.2)$$

$$u_i = g(s_i) \in \mathcal{U} \quad i \geq 0 \quad (2.3)$$

Pelo fato do espaço de estados ser finito, existem necessariamente inteiros $l \geq 0$ e $j > 0$ tais que $s_{l+j} = s_l$. Portanto, para todo $i \geq l$ tem-se $s_{i+j} = s_i$ e conseqüentemente $u_{i+j} = u_i$, o que concede caráter periódico aos PRNGs. O menor j para qual esta repetição acontece é chamado período do gerador, denotado por ρ . Isto ocorre essencialmente porque as funções f e g são determinísticas [L'ECUYER, 2006]. Obviamente, o período ρ não pode ultrapassar $|\mathcal{S}|$, a cardinalidade do espaço de estados. Bons PRNGs são projetados de modo que o período esteja o mais próximo possível deste limite. Na maioria das vezes, ρ depende da escolha dos parâmetros da função f e da semente s_0 . Bons geradores conseguem assumir o mesmo período independentemente da escolha da semente s_0 .

Assumindo $\mathcal{U} = (0, 1)$, tem-se um gerador de variáveis aleatórias *i.i.d.* com distribuição uniforme padrão, condição necessária para gerar distribuições de probabilidade arbitrárias. Uma vez que um computador só é capaz de representar números reais com precisão finita, o melhor a ser realizado é representar os elementos de $\mathcal{S}$ por meio de números inteiros num intervalo $[0, m]$, e recorrer à representação real de ponto flutuante somente no cômputo da função $g(s_i) = s_i/m$. Normalmente, o espaço de estados contém 2^w elementos e $m = 2^w - 1$; o maior inteiro que pode ser representado em uma máquina binária com palavras de w bits [KNUTH, 1981].

A princípio, o método mais simples para se gerar uma variável aleatória X , distribuída de acordo com a função de distribuição $F(x) = p(X \leq x)$ a partir de uma variável U de distribuição $U(0, 1)$, é aplicar a função inversa de F à U [KNUTH, 1981; L'ECUYER, 2004]:

$$X = F^{-1}(U) = \min\{x | F(x) \geq U\} \quad (2.4)$$

Este método é conhecido como método da inversão [L'ECUYER, 2004]. X tem a distribuição desejada, pois $p(X \leq x) = p(F^{-1}(U) \leq x) = p(U \leq F(x)) = F(x)$.

Como foi definida anteriormente, a função g produz valores no intervalo fechado $[0, 1]$. Na prática, é importante que os números gerados pelo PRNG estejam estritamente entre 0 e 1, pois algumas das transformações utilizadas para produzir variáveis aleatórias não uniformes podem assumir valores infinitos quando U é 0 ou 1. Por exemplo, uma variável aleatória de distribuição exponencial com média $\lambda = 1$, normalmente é gerada pela inversão $X = -\ln(1 - U) = -\ln(U)$. X resulta em infinito quando $U = 1$. Boas

implementações de PRNGs nunca produzem os valores 0 e 1 e, geralmente, descrevem g por meio das equações (2.5) ou (2.6).

$$u_i = g(s_i) = (s_i + 1)/(m + 1) \quad (2.5)$$

$$u_i = g(s_i) = \begin{cases} s_i/(m + 1) & \text{se } s_i > 0 \\ m/(m + 1) & \text{caso contrário} \end{cases} \quad (2.6)$$

Muitas vezes outros métodos de geração de variáveis aleatórias são preferíveis ao método da inversão, principalmente quando se é difícil ou computacionalmente custoso calcular F^{-1} . Por exemplo, a distribuição contínua não uniforme mais importante, a distribuição normal com média zero e desvio padrão um, tem F definida como:

$$F(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-t^2/2} dt \quad (2.7)$$

Não há uma forma algébrica simples de inverter (2.7), mas diversos algoritmos podem ser utilizados para se obter amostras de uma distribuição normal e de outras distribuições não trivialmente inversíveis como a Cauchy e a Gama. Grande parte destes algoritmos podem ser classificados como métodos de rejeição [DEVROYE, 1986]. Neste caso, os valores de uma função de distribuição de probabilidade arbitrária $F(x)$ são gerados por meio de uma distribuição instrumental $G(x)$, sob a restrição de que $F(x) < mG(x)$ com $m > 1$ sendo um limite apropriado para a razão $F(x)/G(x)$. Ao invés de se obter amostras diretamente da distribuição $F(x)$, estas são obtidas de $mG(x)$. As amostras provenientes $mG(x)$ são probabilisticamente aceitas ou rejeitadas segundo algum critério e outra amostra deve ser sorteada no caso de rejeição. Há ainda outras classes de algoritmos para se gerar variáveis aleatórias não uniformes e uma discussão sobre o assunto foge do escopo deste texto. Uma ampla exposição sobre o tema pode ser encontrada em [DEVROYE, 1986] e em [THOMAS et al., 2007], no caso de interesse específico a respeito de distribuições gaussianas.

Diversos critérios de qualidade devem ser considerados ao se projetar e implementar um PRNG. Um período extremamente longo, assegurado por meio de provas matemáticas, é essencial para garantir que não ocorram repetições na seqüência gerada. O gerador deve ser eficiente, tanto em termos de velocidade de geração quanto em termos de utilização de memória. Também é interessante que uma mesma seqüência de números aleatórios seja exatamente reproduzível quantas vezes forem necessárias, e que o gerador seja portátil e funcione da mesma maneira em diferentes ambientes de *hardware* e *software*. A disponibilidade de métodos eficientes de *jump-ahead*, que computem rapidamente s_{i+v} dado s_i para quaisquer valores grandes de v e i , é extremamente útil. Esta característica permite que a seqüência do PRNG seja particionada em fluxos paralelos disjuntos a fim de se estabelecer um número arbitrário de geradores virtuais a partir de um único gerador real. Os gerado-

res virtuais podem ser usados em processamento paralelo e proverem fontes distintas de aleatoriedade para simulações estatísticas de grande porte [L'ECUYER, 2006].

2.2.2 Recorrências Módulo m

Existem diversos métodos matemáticos disponíveis para se implementar um PRNG, mas o mais conhecido e popular é dado pela recorrência linear (2.8). A teoria que o justifica é de fácil entendimento e sua implementação é realizada sem grandes dificuldades.

$$x_i = (a_1x_{i-1} + a_2x_{i-2} + \dots + a_kx_{i-k}) \bmod m = \left(\sum_{j=1}^k a_jx_{i-j} \right) \bmod m \quad i \geq k \quad (2.8)$$

Os inteiros positivos m e k representam respectivamente o módulo e a ordem da recorrência. Os coeficientes $a_1 \dots a_k$ pertencem a $\mathbb{Z}_m = \{0 \dots m-1\}$, o conjunto dos números inteiros sob o qual todas as operações aritméticas são efetuadas em módulo m . O estado do gerador no passo i é dado por um vetor de k elementos, $s_i = (x_{i+k-1}, x_{i+k-2}, \dots, x_i)^T$. A função de transição de estado f pode ser descrita como uma multiplicação da matriz dos coeficientes A pelo vetor do estado segundo (2.9). A seqüência pseudo-aleatória no intervalo $(0, 1)$ é obtida avaliando-se a função de geração g em x_i , $u_i = g(s_i) = g(x_i)$, conforme descrito em (2.5) ou (2.6).

$$s_i = f(s_{i-1}) = (As_{i-1}) \bmod m = \left(\begin{bmatrix} 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \\ a_k & a_{k-1} & \dots & a_1 \end{bmatrix} \begin{bmatrix} x_{i+k-2} \\ \vdots \\ x_{i-2} \\ x_{i-1} \end{bmatrix} \right) \bmod m \quad (2.9)$$

Um gerador que utiliza a recorrência (2.8), geralmente atribuindo um grande valor para m , é denominado *Multiple Recursive Generator* (MRG). Para ordem $k = 1$, a recorrência é reduzida para a clássica forma (2.10), e trata-se do gerador padrão presente na maioria dos produtos de *software* que também é conhecido como Lehmer PRNG, Park-Miller PRNG ou simplesmente *Linear Congruential Generator* (LCG) [KNUTH, 1981]. Algumas implementações de LCGs efetuam a adição de um termo constante c antes de computar o módulo da recorrência conforme a equação (2.11). Normalmente, as duas formas de LCGs são diferenciadas adotando-se os termos LCG multiplicativo para (2.10) e LCG combinado para (2.11).

$$x_i = (ax_{i-1}) \bmod m \quad (2.10)$$

$$x_i = (ax_{i-1} + c) \bmod m \quad (2.11)$$

2.2.3 Recorrências Módulo 2

Certamente, é uma idéia atraente explorar o fato de que os computadores trabalham com base binária para desenvolver um PRNG. Representar o estado, funções de transição e geração de um PRNG, diretamente em termos de *bits*, cadeias de *bits* e operações *bit a bit* pode ser de veras vantajoso. Com esta finalidade pode-se construir todo um *framework* genérico baseado na teoria dos grupos que dê suporte a esta idéia central.

Em álgebra abstrata, um grupo finito ou grupo de Galois $\mathbb{F}_k$ (batizado em homenagem ao matemático francês Évariste Galois) é uma estrutura algébrica que contém um número finito de elementos, e sua ordem k é sempre um número primo p ou uma potência de um número primo p^n [ROMAN, 1989]. Para ordem $k = 2$, o grupo possui apenas os elementos 0 e 1 e as suas operações fechadas de adição e multiplicação são reduzidas, respectivamente, ao *ou exclusivo* e ao *e lógico*. Há uma equivalência explícita destas operações para com a aritmética modular com módulo igual a 2. As equações (2.12) e (2.13) definem, genericamente, um gerador como uma recorrência linear de matrizes sobre $\mathbb{F}_2$ [L'ECUYER, 2006].

$$\mathbf{x}_i = A\mathbf{x}_{i-1} \quad (2.12)$$

$$\mathbf{y}_i = B\mathbf{x}_i \quad (2.13)$$

O termo $\mathbf{x}_i = (x_0, \dots, x_{k-1})_i^T \in \mathbb{F}_2^k$ representa o estado s_i do PRNG e é um vetor de k *bits*. O termo $\mathbf{y}_i = (y_0, \dots, y_{w-1})_i^T \in \mathbb{F}_2^w$, um vetor de w *bits*, é o valor gerado no passo i da recorrência (2.12). Os valores k e w são constantes inteiras positivas e normalmente w é o comprimento da palavra da máquina que executa o gerador. A e B são matrizes cujos elementos pertencem a $\mathbb{F}_2$. A é denominada matriz de transição e tem dimensão $k \times k$. A multiplicação desta matriz pelo atual vetor de estado conduz o gerador ao estado seguinte da recorrência. B é a matriz de geração³ e tem dimensão $w \times k$. A multiplicação desta matriz pelo vetor de estado é na verdade um mapeamento $\mathbb{F}_2^k \mapsto \mathbb{F}_2^w$.

A saída $\mathbf{y}_i$ pode ser transformada em uma variável aleatória de distribuição uniforme padrão $U(0, 1)$ aplicando-se a fórmula (2.14). A cadeia de *bits* torna-se um valor real, por meio da somatória de seus *bits* ponderados por potências de 2, com expoentes negativos.

$$u_i = g(\mathbf{y}_i) = \sum_{j=1}^w 2^{-j} \cdot y_{j-1} \quad (2.14)$$

Diversos PRNGs populares encaixam-se neste *framework* como casos específicos, por meio da escolha apropriada da forma das matrizes A e B . Entre eles incluem-se o gerador *Tausworthe* [TAUSWORTHE, 1965], os *Generalized Feedback Shift Registers* (GFSRs) [LEWIS e PAYNE, 1973] e *twisted* GFSRs [MATSUMOTO e KURITA, 1992, 1994].

³Tradução livre do original *output transformation matrix*.

2.2.4 *Mersenne Twister*

O *Mersenne Twister* (MT) [MATSUMOTO e NISHIMURA, 1998] é um PRNG desenvolvido no final da década de 90 para ser utilizado em simulações de Monte Carlo. Sua implementação baseia-se em profundos fundamentos teóricos da matemática discreta e da álgebra abstrata. O funcionamento deste gerador se dá por meio de uma recorrência linear matricial sobre o grupo de Galois $\mathbb{F}_2$, conforme descrito na Seção (2.2.3). Os criadores do MT o projetaram para superar as deficiências dos PRNGs existentes, e foram extremamente bem sucedidos na tentativa. O MT gera uma sequência pseudo-aleatória de alta qualidade muito eficientemente. O algoritmo desenvolvido, para uma particular escolha de parâmetros, proporciona um período astronômico de $2^{19937} - 1$, e uma propriedade de equidistribuição em 623, dimensões utilizando-se de uma área de memória de apenas 624 palavras de 32 *bits*. O nome do gerador deriva do fato de seu período ρ ser um primo de *Mersenne*. Um primo de *Mersenne* é um número de *Mersenne* que também é primo. Um número de *Mersenne* M_n , batizado em homenagem ao matemático Marin Mersenne, é sempre uma potência de 2 subtraída de uma unidade: $M_n = 2^n - 1$ [JOHNSONBAUGH, 1990]. Até o presente momento, somente 47 primos de *Mersenne* são conhecidos [GIMPS, 2009]. Existem duas variantes comuns do MT. Estas se diferenciam apenas no tamanho do primo de *Mersenne*, que especifica seu período, e no comprimento, em número de *bits*, dos números gerados. A variante mais popular é o MT19937, um gerador de palavras de 32 *bits*. O MT19937-64, a segunda variante mais popular, gera palavras de 64 *bits*.

O MT é uma variação do algoritmo conhecido como GFSR [LEWIS e PAYNE, 1973]. Os GFSRs são geradores rápidos e suas implementações independem do tamanho da palavra da máquina alvo. Eles são particularmente adequados para serem implementados em *hardware*. Contudo, existem algumas ressalvas quanto a sua utilização. Dentre elas pode-se citar:

- A seleção das sementes é muito crítica e influencia profundamente a aleatoriedade da sequência gerada, tornando uma boa inicialização difícil e custosa.
- O período real da sequência é muito menor que o seu limite superior teórico.
- O algoritmo requer uma grande área de trabalho. Muita memória pode ser consumida, se vários geradores forem implementados simultaneamente.
- A relação de recorrência utilizada baseia-se em um trinômio que deve ser primitivo sobre os inteiros módulo 2; uma condição muito rara.

Por sua vez, o MT apresenta as seguintes propriedades que o qualificam para ser utilizado em grandes simulações estatísticas [MATSUMOTO e NISHIMURA, 1998]:

- Inicialização das sementes livre de restrições severas. Só é necessário que o estado inicial, a semente, seja um vetor não nulo. Na prática, utiliza-se um PRNG mais simples que o

MT, normalmente um LCG, para preencher completamente o seu vetor de estado com valores iniciais.

- O período da sequência gerada é colossal para o MT19937.
- Utiliza muito menos memória do que um GFSR de período equivalente.
- A sequência gerada é estatisticamente equidistribuída da melhor maneira possível.
- O gerador passou por uma bateria de rigorosos testes estatísticos. É importante ressaltar que o MT foi aprovado pelo clássico conjunto de testes *Diehard* [MARSAGLIA, 1995]; uma grande referência na literatura sobre o tema.

Qualquer variante do MT, respeitando-se a condição de que seu período $2^{nw-r} - 1$ seja um primo de Mersenne, pode ser descrito pelas seguintes características [MATSUMOTO e NISHIMURA, 1998]:

- w : Tamanho da palavra a ser gerada em número de *bits*.
- n : Grau de recorrência do gerador.
- m ($1 \leq m \leq n$): Número de seqüências aleatórias que podem ser geradas paralelamente por um único gerador ou posição da “palavra do meio” da recorrência.
- r ($0 \leq r \leq w - 1$): Ponto de separação de uma palavra; divide a palavra gerada em duas partes: inferior e superior. Também pode ser interpretado como o número de *bits* da parte inferior.
- $\mathbf{a}$: Coeficientes da matriz de transição A . A matriz encontra-se na forma racional normal [MATSUMOTO e KURITA, 1992] e apresenta uma disposição conveniente de seus elementos para proporcionar uma multiplicação veloz. Este parâmetro é expresso como um número hexadecimal que compacta os coeficientes $(a_{w-1}, \dots, a_0)$ da matriz e deve ser encarado como um vetor de *bits* de $\mathbb{F}_2^w$.
- $\mathbf{b}$ e $\mathbf{c}$: Máscaras de *bits* para *tempering* [MATSUMOTO e KURITA, 1994]. Estas máscaras são utilizadas para efetuar a multiplicação pela matriz de geração B e também são expressas em hexadecimal representando vetores de $\mathbb{F}_2^w$.
- s e t : *Bit shifts* de *tempering*.
- u e l : *Bit shifts* adicionais.

A relação de recorrência linear matricial do MT é dada por (2.15). $\mathbf{x}_k^u$ denota os $w - r$ *bits* mais significativos de $\mathbf{x}_k \in \mathbb{F}_2^w$ (a parte superior de $\mathbf{x}_k$), $\mathbf{x}_{k+1}^l$ denota os r *bits* menos significativos de $\mathbf{x}_{k+1} \in \mathbb{F}_2^w$ (a parte inferior de $\mathbf{x}_{k+1}$), o operador $|$ representa a concatenação de *bits* que forma uma palavra de comprimento w , e o operador $\oplus$ representa adição módulo 2 *bit a bit* (ou exclusivo) [MATSUMOTO e NISHIMURA, 1998].

$$\mathbf{x}_{k+n} = \mathbf{x}_{k+m} \oplus (\mathbf{x}_k^u | \mathbf{x}_{k+1}^l) A \quad k \geq 0 \quad (2.15)$$

O termo A contido em (2.15) é a matriz de transição; uma constante na relação de recorrência. Esta matriz tem dimensão $w \times w$ e seus elementos pertencem a $\mathbb{F}_2$. A forma

explícita de A é dada por (2.16).

$$A_{w \times w} = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ a_{w-1} & a_{w-2} & a_{w-3} & \dots & a_0 \end{bmatrix} \quad (2.16)$$

O produto do vetor $\mathbf{x}_k^u | \mathbf{x}_{k+1}^l$ pela matriz A pode ser computado de maneira extremamente eficiente devido ao fato de A estar na forma racional normal. A multiplicação de A por um vetor $\mathbf{x}$ qualquer de $\mathbb{F}_2^w$ é efetuada segundo a equação (2.17). O operador binário $\gg$ representa *bit shift* à direita.

$$\mathbf{x}A = \begin{cases} \mathbf{x} \gg 1 & \text{se } x_0 = 0 \\ (\mathbf{x} \gg 1) \oplus a & \text{se } x_0 = 1 \end{cases} \quad (2.17)$$

Antes de serem efetivamente considerados como saída do gerador, os números produzidos pela recorrência (2.15) são submetidos a um processo denominado *tempering* [MATSUMOTO e KURITA, 1994] que nada mais é do que a multiplicação pela matriz de geração B . Com o intuito de aprimorar a propriedade de equidistribuição da sequência pseudo-aleatória, cada palavra gerada $\mathbf{x}$ é multiplicada a direita pela matriz T , a matriz da transformada de *tempering*: $\mathbf{x} \mapsto \mathbf{z} = \mathbf{x}T$. Só então $\mathbf{z}$ pode ser considerado um valor de saída do PRNG. A computação desta multiplicação é eficientemente efetuada pela aplicação sequencial das operações *bit a bit* apresentadas em (2.18). Os operadores $\ll$ e $\wedge$ indicam respectivamente *bit shift* à esquerda e *e lógico bit a bit*.

$$\begin{aligned} \mathbf{y} &\leftarrow \mathbf{x} \oplus (\mathbf{x} \gg u) \\ \mathbf{y} &\leftarrow \mathbf{y} \oplus ((\mathbf{y} \ll s) \wedge b) \\ \mathbf{y} &\leftarrow \mathbf{y} \oplus ((\mathbf{y} \ll t) \wedge c) \\ \mathbf{z} &\leftarrow \mathbf{y} \oplus (\mathbf{y} \gg l) \end{aligned} \quad (2.18)$$

O gerador MT19937, a variante do MT alvo do projeto de pesquisa, é especificada pelos seguintes coeficientes da relação de recorrência:

$$(w, n, m, r, \mathbf{a}) = (32, 624, 397, 31, 9908B0DF_{16})$$

Os coeficientes para a operação de *tempering* são:

$$(u, s, \mathbf{b}, t, \mathbf{c}, l) = (11, 7, 9D2C5680_{16}, 15, \text{EFC60000}_{16}, 18)$$

O Algoritmo (2.2) sintetiza todo o processo de geração de números aleatórios que foi descrito nesta Subseção. Graças aos seus louváveis méritos, o MT vem rapidamente se tornando o PRNG padrão para simulações estatísticas. Uma biblioteca de *software* portátil disponível gratuitamente ⁴ e uma geração de números aleatórios eficiente e confiável raramente faz do MT uma má escolha para aplicações de simulação. No entanto é importante ressaltar que os esforços científicos no desenvolvimento de geradores de números aleatórios está longe de cessar [PANNETON et al., 2006] e as contribuições oferecidas pelo MT continuam a ser aprimoradas.

Algoritmo 2.2: *Mersenne Twister*

Entrada: Semente do gerador s_0 (necessário somente para inicialização)

Saída: Número pseudo-aleatório z

```

1 início
  /* Inicialização */
2 se o gerador ainda não foi inicializado então
3   para  $i \in \{1 \dots n - 1\}$  faça
4      $x_i \leftarrow$  qualquer valor diferente de 0 calculado a partir de  $s_0$  ;
5    $k \leftarrow 0$  ;
  /* Recorrência */
6  $y \leftarrow x_k^u \mid x_{(k+1) \bmod n}^l$  ;
7  $x_k \leftarrow x_{(k+m) \bmod n} \oplus (y \gg 1) \oplus \begin{cases} 0 & \text{se } y_0 = 0 \\ a & \text{se } y_0 = 1 \end{cases}$  ;
  /* Tempering */
8  $y \leftarrow x_k \oplus (x_k \gg u)$  ;
9  $y \leftarrow y \oplus ((y \ll s) \wedge b)$  ;
10  $y \leftarrow y \oplus ((y \ll t) \wedge c)$  ;
11  $z \leftarrow y \oplus (y \gg l)$  ;
12  $k \leftarrow (k + 1) \bmod n$  ;
13 retorna  $z$  ;
14 fim

```

2.2.5 Critérios de Qualidade e Testes Estatísticos

É importante ter ciência de que as propriedades elementares apresentadas final da Seção (2.2.1), apesar de boas e desejáveis, não são suficientes para garantir a qualidade de um gerador. Um PRNG pode possuir todas estas características e não ser capaz de apresentar um comportamento que aproxime-se satisfatoriamente do apresentado por uma sequência aleatória real [L'ECUYER, 2006]. Assim o problema de mensurar a qualidade

⁴*Mersenne Twister Home Page*, <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>. Último acesso em 09/02/2009.

de um PRNG reduz-se ao problema de obter uma definição quantitativa e formal de comportamento aleatório [KNUTH, 1981].

Um critério razoável para determinar a aleatoriedade de uma seqüência qualquer é a t -equidistribuição [KNUTH, 1981]. Uma seqüência $(u_0, u_1, u_2 \dots)$ de variáveis aleatórias de distribuição $U(0, 1)$ é dita t -equidistribuída se e somente se, para todos os inteiros $i > 0$ e $t > 0$, o vetor $\mathbf{u}_{i,t} = (u_i, u_{i+1}, \dots, u_{i+t-1})$ é uniformemente distribuído sobre o espaço t -dimensional do hipercubo unitário $(0, 1)^t$ [L'ECUYER, 2006]. Obviamente esta conjectura não é verdadeira para uma seqüência gerada por um PRNG pois qualquer vetor de t valores sucessivos por ele produzido pertence obrigatoriamente a $\Psi_t = \{(u_0, u_1, \dots, u_{t-1}) \mid s_0 \in \mathcal{S}\}$. Ψ_t é o conjunto finito de todos os vetores compostos por t números sucessivos que podem ser produzidos pelo PRNG a partir de todos os seus possíveis estados iniciais. O vetor $\mathbf{u}_{0,t}$ é distribuído uniformemente sobre Ψ_t supondo-se que a semente s_0 seja selecionada aleatoriamente de $\mathcal{S}$ também de maneira uniforme, ou seja, μ é uma distribuição uniforme sobre $\mathcal{S}$. Uma vez que o objetivo do gerador é aproximar uma distribuição uniforme sobre $(0, 1)^t$ fica claro que Ψ_t deve proporcionar uma boa cobertura do hipercubo. Em outras palavras, Ψ_t atua como uma aproximação do espaço amostral teórico $(0, 1)^t$, do qual os vetores $\mathbf{u}_{i,t}$ supostamente são sorteados aleatoriamente. Logo conclui-se que um bom PRNG deve possuir um espaço de estados de grande cardinalidade não só para produzir uma seqüência de período longo, mas principalmente para que a cardinalidade de Ψ_t seja grande, implicando em uma melhor cobertura do hipercubo unitário e conseqüentemente em uma melhor qualidade dos números gerados.

Diversos geradores desenvolvidos apresentam características desastrosamente não aleatórias e sua aproximação produzida para $(0, 1)^t$ só pode ser considerada satisfatória para valores pequenos de t . Em outras palavras, uma aproximação bem preenchida do hipercubo unitário só é obtida em dimensões muito baixas e em dimensões mais altas encontram-se padrões extremamente regulares em hiperplanos eqüidistantes. Normalmente os parâmetros da função de transição de PRNGs ruins são especificados sem nenhum conhecimento da teoria matemática que os fundamenta ou com o único objetivo de construir um gerador computacionalmente eficiente e veloz [PARK e MILLER, 1988] em detrimento da qualidade da seqüência produzida.

Um PRNG muito conhecido na literatura pela má qualidade dos números produzidos é o RANDU – este gerador é o contra-exemplo clássico quando se discute sobre bons procedimentos para gerar números pseudo-aleatórios [PARK e MILLER, 1988]. O RANDU foi introduzido no início dos anos 60 como parte do pacote de rotinas do célebre IBM System/360 [IBM, 1968] e popularizou-se rapidamente servindo de base para a implementação de outros PRNGs. Os parâmetros do RANDU, um LCG multiplicativo com $a = 65539 = 2^{16} + 3$, $m = 2^{31}$ e x_0 ímpar [IBM, 1968], foram escolhidos de modo que sua representação binária fosse conveniente para a arquitetura de 32 *bits* do System/360

executar as operações de módulo e multiplicação rapidamente. Todavia, com tais parâmetros, o RANDU não estava apto a atingir um período longo o bastante e apresentava um comportamento terrível em testes de equidistribuição com dimensão maior do que dois.

Figura 2.3: Conjunto de 100.000 pontos em $(0, 1)^3$ gerados pelo RANDU

Um conjunto de 100.000 pontos tridimensionais $(x = u_i, y = u_{i+1}, z = u_{i+2}) \in (0, 1)^3$ gerados sequencialmente pelo RANDU é ilustrado pela Figura (2.3). Sem valer-se de nenhuma ferramenta de análise formal percebe-se que todos os pontos estão contidos em apenas quinze planos, o que demonstra a má qualidade deste gerador. A assustadora conclusão a respeito do RANDU encontra-se claramente nas palavras de Donald Knuth [KNUTH, 1981]:

“...RANDU is enough to bring dismay into the eyes and stomachs of many computer scientists. ...the generator fails most three dimensional criteria for randomness and it should never have been used.”

Bons PRNGs são projetados com base em análises matemáticas de suas propriedades teóricas, só então implementados e por fim submetidos à baterias de testes estatísticos empíricos. Estes testes tentam detectar evidências empíricas contra a hipótese nula $\mathcal{H}_0$: “os valores u_i da sequência fornecida pelo PRNG são realizações, ou amostras sorteadas, de variáveis aleatórias *i.i.d.* com distribuição $U(0, 1)$ ” [L’ECUYER, 2004]. Um teste pode ser definido como uma função T que mapeia a sequência $(u_0, u_1, u_2 \dots)$ a um número real x para o qual uma aproximação exata está disponível se $\mathcal{H}_0$ for verdadeira. Quanto melhor for a aproximação obtida para x por T maior serão as evidências contra a rejeição de $\mathcal{H}_0$ e maior será confiabilidade do gerador analisado por T . Para o teste T ser passível de implementação, o cálculo para aproximar x deve depender de um número finito, mas possivelmente aleatório, de realizações u_i [L’ECUYER, 2004].

A aprovação em diversos testes empíricos aumenta a confiabilidade de um dado PRNG, porém não há nenhuma garantia de que o gerador é a prova de falhas para todo e qualquer

tipo de simulação em que for utilizado. Construir um gerador que seja aprovado em todos os testes já propostos e que ainda serão criados é uma meta impossível de ser alcançada [KNUTH, 1981]. Assim sendo, o senso comum geralmente adotado diz que geradores ruins são aqueles que falham em testes simples enquanto que bons geradores são aqueles que falham somente em testes rigorosos, complicados, difíceis de se elaborar e executar. A experiência prática mostra que PRNGs com períodos muito longos, uma boa estrutura em seu conjunto Ψ_t e não baseados em relações de recorrência simplistas passam na maioria dos testes não extremos [L'ECUYER e SIMARD, 2007a].

2.3 Computação Reconfigurável

Pesquisas sobre a arquitetura de sistemas computacionais são a muito tempo uma preocupação central da ciência e da engenharia de computação. Tanto os objetivos destes estudos quanto as arquiteturas por eles produzidas diferenciam-se enormemente em suas características de acordo com as aplicações que devem ser executadas pelo sistema computacional em pauta. Questões como o preço final do equipamento, seu o ambiente de utilização, a programabilidade e flexibilidade do sistema, a potência elétrica dissipada e muitas outras fazem parte constantemente da ponderação dos projetistas de arquiteturas. Para máquinas de alto desempenho o foco está em grandes frequências de *clock*, computação paralela e canais de comunicação velozes as custas da energia consumida e da área de silício utilizada pelos *chips*. Em muitos sistemas embarcados o preço final do produto é um fator preponderante e um pequeno e barato microcontrolador capaz de captar dados de sensores, processá-los e prover informações de controle aos atuadores a uma taxa minimamente aceitável muitas vezes é a melhor escolha [BOBDA, 2007].

Tradicionalmente, a implementação de sistemas de computação é realizada por meio de de *hardware* específico ou *software* executado em uma plataforma de *hardware* genérica como um *General Purpose Processor* (GPP). Ao implementar um sistema computacional, os desenvolvedores e projetistas devem optar por uma das alternativas para todo o projeto ou, em alguns casos, deliberar quais subsistemas devem ser implementados em *hardware* ou em *software* ponderando a respeito das vantagens e desvantagens relativas de cada abordagem [BONDALAPATI e PRASANNA, 2002].

Soluções em *hardware* oferecem alto desempenho, pois possuem execução espacial e paralelismo intrínseco e sua lógica interna é especificamente projetada para a aplicação. Soluções em *software* podem ser descritas como engenhos de execução capazes de interpretar um fluxo de instruções flexível porém relativamente ineficientes em relação ao *hardware*. Isto se dá devido à característica temporal e seqüencial da execução do fluxo de instruções e ao fato dos operadores implementados não serem específicos para a aplicação. A título de exemplo, a Figura (2.4) demonstra a diferença entre computação espacial e

temporal para o cálculo da expressão $y = ax^2 + bx + c$. Em implementações espaciais cada operador utilizado existe em um ponto distinto do espaço possibilitando paralelismo real. Em implementações temporais um menor número de recursos computacionais mais genéricos são reutilizados com o passar do tempo [DEHON e WAWRZYNEK, 1999].

Figura 2.4: Computação espacial e temporal para a expressão $y = ax^2 + bx + c$

Contudo, o desenvolvimento de tecnologias de computação reconfigurável introduziu uma nova possibilidade que, até certo nível, mescla as propriedades das alternativas convencionais. A computação reconfigurável destaca-se cada vez mais como uma nova e importante estrutura organizacional para computação, combinando a programabilidade dos processadores convencionais com a capacidade de computação paralela do *hardware* de propósito específico. A Figura (2.5) dispõe os diferentes tipos de elementos processadores relacionando os aspectos flexibilidade e desempenho para cada uma das classes. Sistemas reconfiguráveis enquadram-se em um ponto intermediário entre as arquiteturas tradicionais. A emergência deste novo paradigma de computação tornou as fronteiras tradicionais entre *hardware* e *software* difusas e levou ao surgimento de toda uma nova classe de arquiteturas de computadores: as arquiteturas reconfiguráveis.

As características chave que diferenciam máquinas com arquitetura reconfigurável de máquinas convencionais são o seu alto grau de utilização de computações personalizadas, espaciais e dotadas de controle e recursos distribuídos além da capacidade de realizar por si só mudanças substanciais no *datapath*. Por outro lado, a maior diferença em relação ao *hardware* de uso específico é a possibilidade de adaptação da arquitetura em tempo de execução por meio do carregamento de um novo circuito. Arquiteturas reconfiguráveis têm se mostrado uma alternativa válida graças ao seu bem sucedido uso em diversas aplicações tais como criptografia, reconhecimento de voz, reconhecimento automático de alvos, comparação de padrões e compressão de dados [BONDALAPATI e PRASANNA, 2002].

Figura 2.5: Relação flexibilidade *versus* desempenho para diferentes classes de processadores

2.3.1 *Field Programmable Gate Arrays*

Os recentes avanços da computação reconfigurável são, em grande parte, frutos do desenvolvimento dos FPGAs. Suas raízes históricas se encontram nos *Complex Programmable Logic Devices* (CPLDs) dos meados da década de 80. Em um primeiro momento os FPGAs foram utilizados como mecanismos de prototipação rápida ou como lógica de interconexão de circuitos mais complexos, porém o aumento da velocidade e da capacidade destes dispositivos fez com que eles se tornassem a base da moderna computação reconfigurável. Um FPGA é um dispositivo semiconductor que contém internamente um arranjo de elementos cujas funcionalidades podem ser determinadas através de múltiplos *bits* de configuração. Estes elementos, blocos lógicos no jargão da área, são interconectados por um conjunto de rotas também programável. Desta forma, circuitos arbitrários podem ser implementados em um FPGA mapeando suas funções de computação aos blocos lógicos, utilizando as rotas para conectá-los e então obter o comportamento global esperado [COMPTON e HAUCK, 2002]. A reconfigurabilidade de um FPGA é obtida por meio do uso de antifusíveis ou *bits* de *Static Random Access Memory* (SRAM) para controlar o estado de transistores. A tecnologia dos antifusíveis vale-se de correntes elétricas altas e é tipicamente pouco reprogramável. FPGAs cuja configurabilidade é baseada em memória SRAM podem ser reprogramados ainda ligados através do *download* de diferentes *bits* de configuração para suas células.

O bloco lógico típico de um FPGA, ilustrado pela Figura (2.6), contém uma ou mais *Lookup Tables* (LUTs), *flip-flops* opcionais, alguns elementos extras de lógica combinacional e uma célula de SRAM para controle de configuração. As LUTs funcionam como uma tabela verdade e permitem que qualquer função booleana seja implementada, proporcio-

Figura 2.6: Arquitetura típica de um bloco lógico de FPGA

nando lógica genérica. Os *flip-flops* podem ser utilizados para *pipelining*, armazenamento de estados de autômatos finitos, composição de *bits* de registradores genéricos ou em qualquer outra situação na qual um sinal de *clock* faça-se necessário. Os elementos extras são um recurso especial que proporciona um aumento de velocidade para execução de outras operações como aritmética binária e verificação de paridade [COMPTON e HAUCK, 2002], por exemplo.

Figura 2.7: Arquitetura típica de um FPGA

A maioria das arquiteturas de FPGA organiza suas estruturas de roteamento como um mar de recursos de conexão de modo a permitir uma comunicação rápida e eficiente entre as linhas e colunas dos blocos lógicos ilhados. Como é mostrado na Figura (2.7), os blocos lógicos se encontram imersos em uma estrutura de roteamento geral, com suas portas de entrada e saída anexadas ao meio de roteamento através de blocos de conexão. Os blocos de conexão são implementados com multiplexadores programáveis que selecionam qual sinal de qual canal de roteamento estará conectado aos terminais do bloco lógico. Os sinais gerados nos blocos lógicos seguem até os blocos de conexão e depois até os fios da rede de roteamento. Normalmente também existem comutadores inseridos entre os blocos

de conexão. Os comutadores dispõem de conexões programáveis entre as trilhas verticais e horizontais que permitem que os sinais mudem de direção caso seja necessário [COMPTON e HAUCK, 2002]. Pode-se notar que a arquitetura de roteamento de um FPGA é uma estrutura bastante complexa e projetada para suportar padrões de conexão arbitrários – os blocos de conexão e comutadores ao redor de um único bloco lógico possuem milhares de pontos programáveis. Contudo, os usuários de FPGAs não necessitam se preocupar com profundos detalhes arquiteturais uma vez que *softwares* de projeto realizam o roteamento automaticamente.

A programação de FPGAs é suportada por ferramentas EDA sofisticadas que possibilitam compilação e alocação eficiente dos recursos de *hardware*. As empresas *Altera* e *Xilinx*, as duas principais fabricantes de FPGAs da atualidade, fornecem um conjunto completo de ferramentas EDA para o desenvolvimento de sistemas embarcados em FPGAs. As quatro principais ferramentas da Altera são o *Quartus II*, *SOPC Builder*, *DSP Builder* e *Nios II IDE*, sendo as três primeiras designadas para o desenvolvimento de *hardware* e a última para o *software*. Mais detalhes a respeito dessas ferramentas podem ser encontrados na Seção (3.2). O ambiente de desenvolvimento da *Altera* também suporta uma grande quantidade de ferramentas de terceiros, como o *ModelSim* da *Mentor Graphics* para a simulação do *hardware* e do *software* e o *Handel-C* da *Celoxica* para o desenvolvimento do *hardware*. Todas essas ferramentas providas pela *Altera* também possuem equivalentes fornecidas *Xilinx*, onde se pode dizer que o *Quartus II* corresponde ao *ISE*, o *SOPC Builder* ao *Platform Studio*, o *DSP Builder* ao *System Generator* e o *Nios II IDE* ao *EDK*, conforme visto na Figura (2.8).

Figura 2.8: Principais ferramentas EDA para FPGA.

2.3.2 Família *Stratix* de FPGAs

Os FPGAs da família *Stratix* produzidos pela *Altera* são dispositivos de alta densidade, alto desempenho e baixo consumo de energia [ALTERA, 2006]. Eles FPGAs são utilizados em projetos de SOPCs complexos e para prototipação de ASICs.

A tabela (2.1) aponta algumas características de dois dispositivos *Stratix*, respectivamente de segunda e terceira geração, que estão disponíveis para o desenvolvimento do projeto de pesquisa. A Figura (2.9) mostra o elemento básico da arquitetura de um FPGA *Stratix*, o seu módulo lógico adaptativo. Milhares destes componentes constituem o FPGA e exercer diversas funções de acordo com seu modo de operação e *bits* de configuração [ALTERA, 2006].

Kit de Desenvolvimento Família FPGA FPGA	<i>Nios II Development</i> <i>Kit Stratix II Edition</i> <i>Stratix II</i> EP2S60F672C3	<i>Stratix III FPGA</i> <i>Development Kit</i> <i>Stratix III</i> EP3SL150F1152C3N
Preço <i>Kit</i>¹ (US\$)	995,00	2.495,00
Preço <i>FPGA</i>¹ (US\$)	1.457,50	2.905,00
Elementos Lógicos²	60.440	142.500
ALUTs Combinacionais	48.356	113.600
Registradores Lógicos Dedicados	48.356	113.600
<i>Bits</i> de Memória	2.544.192	5.499.000
Blocos DSP	288 (9 <i>bits</i>)	384 (18 <i>bits</i>)

1. Disponível em <http://www.altera.com>. Último acesso em 26/11/2009.

2. Número equivalente de elementos lógicos em arquiteturas baseadas em LUTs de 4 entradas.

Tabela 2.1: *Kits* de desenvolvimento com dispositivos da família *Stratix*

2.4 Trabalhos Relacionados

Nesta Seção são apresentados trabalhos científicos relacionados ao projeto de mestrado desenvolvido. Estes trabalhos de alguma maneira serviram como inspiração ou fonte de informações e conhecimento específico para a realização da pesquisa. É importante destacar que apesar dos temas de enfoque do trabalho serem recorrentes na literatura especializada, não foi encontrado nenhum esforço que aborde justamente a proposta de tratar de forma conjunta os temas robótica probabilística, geração de números aleatórios e computação reconfigurável.

2005]. Os casos qualificados como mais interessantes para o escopo do presente trabalho são sucintamente descritos a seguir:

- Carro de Fórmula 1: Apesar de não ser exatamente um robô, um carro de Fórmula 1 possui muitas características similares, o que o torna um caso válido para estudo realizado. O carro de 2003 da equipe BMW Williams utilizou uma *Vehicle Control and Monitoring Unit* (VCMU) embarcada fabricada com um *Digital Signal Processor* (DSP) da Texas Instruments e um FPGA da família Virtex da Xilinx. A VCMU mostrou-se apta a controlar em tempo real diversos aspectos do veículo como mudanças de marcha, controle de tração e telemetria.
- Controlador Robótico *Fuzzy*: Um modesto FPGA da Altera, com apenas 24.000 elementos lógicos, foi utilizado com sucesso na implementação de um controlador de comportamento autônomo baseado em lógica *fuzzy* para um robô móvel. Todas as etapas do controle, processos de “fuzzyficação”, inferência e “defuzzyficação” são realizadas por módulos de *hardware* programados no próprio FPGA.
- Sondas de Exploração de Marte: FPGAs foram intensivamente usadas nas sondas robóticas de exploração de Marte *Mars Pathfinder*, *Mars Surveyor’98* e *Mars Surveyor’01*. As sondas eram dotadas de diversas câmeras e os FPGAs se encarregaram de realizar tarefas relacionadas com estes instrumentos incluindo a decodificação de comandos do computador central das sondas, armazenamento das imagens capturadas e controle dos motores de passos para movimentar as câmeras. É importante destacar que os dispositivos reconfiguráveis mostraram-se aptos a resistir à uma jornada interplanetária pelo espaço sideral e funcionar corretamente em um ambiente inóspito como o terreno marciano.

O Laboratório de Computação Reconfigurável (LCR) do ICMC/USP e o Instituto de Engenharia de Sistemas e Computadores (INESC) de Lisboa, por meio de um convênio bilateral de cooperação internacional Brasil – Portugal via CNPq e GRICES, estão desenvolvendo em conjunto um projeto à longo prazo que visa a construção de um robô móvel totalmente baseado em computação reconfigurável. O intuito principal desta parceria é criar primeiramente uma biblioteca de algoritmos de robótica implementados em *hardware* e, em futuro próximo, um núcleo inteligente que controle a execução dos mesmos. O uso da tecnologia de computação reconfigurável tornará o sistema altamente flexível. O robô, utilizando um número fixo de circuitos integrados, estará apto a executar todos os algoritmos disponíveis na biblioteca reprogramando-se automaticamente em tempo de execução sem a necessidade de desligamento. O núcleo de controle, valendo-se de técnicas de inteligência artificial, será responsável por escolher a programação dos dispositivos reconfiguráveis adequada a cada situação enfrentada pelo robô. Em estado mais avançado este projeto viabilizará a construção de robôs móveis capacitados a realizar tarefas de missão crítica em tempo real graças ao desempenho proporcionado pelas otimizações em

hardware.

Outros trabalhos interessantes pode ser encontrados em [WOLF et al., 2007] e [BO-NATO, 2008]. No primeiro caso, um controlador de robôs móveis baseado na plataforma *Player* foi portado para execução em FPGA. No segundo, implementou-se um sistema de SLAM para robôs móveis baseado em câmeras e processamento intensivo de imagens totalmente em *hardware* reconfigurável.

2.4.3 FPGAs Aplicados à Geração de Números Aleatórios

Graças à crescente demanda das grandes simulações estocásticas por PRNGs cada vez mais velozes, as pesquisas sobre geração de números pseudo-aleatórios por meio de *hardware* tem concentrado a atenção de diversos pesquisadores. PRNGs são uma aplicação emergente dentro da área da computação reconfigurável e as tendências apontam para a realização de estudos cada vez mais aprofundados sobre o tema.

Métodos clássicos para geração de números aleatórios por *software* foram adaptados com sucesso ao *hardware* reconfigurável. Pode-se destacar os *Linear Feedback Shift Registers* (LFSRs) e os geradores conhecidos como *Lagged Fibonacci* em [McCOLLUM et al., 2003]. Os tradicionais LCGs são tratados em [CHU e JONES, 1999]. Estes PRNGs simples produzem *hardware* compacto ideal para aplicações embarcadas com restrições severas de custo e que não exijam uma qualidade estatística muito elevada.

Diversos trabalhos [TSOI et al., 2003; KOHLBRENNER e GAJ, 2004; FISCHER e DRUTAROVSKY, 2003] valeram-se de FPGAs para produzir geradores de números aleatórios reais.

Distribuições de probabilidade mais complexas foram produzidas com sucesso por métodos matemáticos robustos em [CUI et al., 2002, 2004] e [LEE et al., 2006, 2005; ZHANG et al., 2005].

CAPÍTULO 3

Metodologia de Desenvolvimento

Esta Seção descreve a metodologia e a plataforma de desenvolvimento adotadas para a realização do projeto de mestrado.

Inicialmente, um sistema *Nios II* básico será especificado com o *SOPC Builder* e programado na FPGA Stratix com o *Quartus II*. Uma versão simples do filtro de partículas para localização *offline* de um único robô móvel será desenvolvida na linguagem *ANSI C* e validada funcionalmente em um PC. Após os requisitos funcionais serem atendidos, o *software* será portado, sem muitas dificuldades, para a plataforma embarcada *Nios II* com ajuda do aplicativo *Nios II IDE*. Em seguida, se dará o desenvolvimento dos blocos de *hardware* dedicados com HDLs e, posteriormente, a validação dos mesmo com *testbenches* executados pelo *ModelSim*. A partir deste momento, o desenvolvimento do sistema passa para um estágio de refinamentos sucessivos e independentes de *hardware* e *software*. Basicamente, a metodologia destes ciclos é resumida pela Figura (3.1).

Todos os recursos do Laboratório de Computação Reconfigurável estão disponíveis para realização do projeto. O laboratório possui os equipamentos essenciais para o desenvolvimento de arquiteturas baseadas em *hardware* reconfigurável e as ferramentas utilizadas representam o estado da arte dentro do mercado internacional de *softwares* EDA. Esta Seção descreve sucintamente as ferramentas de *software* e *hardware* necessárias para realização do mestrado. O objetivo desta Seção é destacar o papel que cada uma destas tecnologias desempenha ou desempenhará no projeto de pesquisa proposto.

Figura 3.1: Metodologia de desenvolvimento do projeto de pesquisa.

3.1 Linguagens de Descrição de Hardware

Uma HDL é uma linguagem formal para descrição de circuitos eletrônicos, mais especificamente para lógica e circuitos digitais. Linguagens desta classe, por meio de suas próprias construções sintáticas, são capazes de descrever o projeto, organização e funcionamento de circuitos, além de também poderem verificar sua operação valendo-se de mecanismos de simulação [HARTENSTEIN, 1987]. As construções das HDLs são expressões de texto que especificam as estruturas espacial e temporal, bem como o comportamento de sistemas eletrônicos.

Como no caso das linguagens de programação concorrente, a sintaxe e a semântica de uma HDL inclui notações explícitas para indicar paralelismo em termos de computação e de acesso aos recursos compartilhados [PEDRONI, 2004]. No entanto, em contraste à maioria das linguagens de programação, também está presente uma noção explícita de temporização; uma característica fundamental do *hardware*. Linguagens formais cuja única funcionalidade é a de expressar a conectividade dos blocos hierárquicos de um

circuito são classificadas apenas como linguagens de *netlist* e não como autênticas HDLs. Todavia, as linguagens de *netlist* têm sua aplicação e nicho específico dentro do mercado de *Computer-Aided Design* (CAD) para eletrônica, sendo amplamente utilizadas.

HDLs são usadas para se produzir especificações executáveis de um componente de *hardware* por meio de simulação. Um simulador é um programa concebido para implementar a semântica subjacente das instruções da HDL e a passagem do tempo, fornecendo ao projetista de *hardware* a habilidade de modelar e testar seus componentes mesmo antes deles serem fisicamente criados. É esta executabilidade real do simulador que dá a ilusão de que as HDLs são linguagens de programação tradicionais, o que na verdade não passa de uma desordem de conceitos. De fato, também é possível representar a semântica do *hardware* com a utilização de linguagens de programação tradicionais, embora o funcionamento de tais aplicativos esteja normalmente ligado à complexas e extensas bibliotecas de *software*. No entanto, as linguagens de programação tradicionais não possuem nenhuma capacidade de expressar explicitamente a temporização intrínseca do *hardware* e é por esta razão que não podem funcionar como uma HDL.

Utilizando-se de um subconjunto de construções apropriado de uma dada HDL, um programa denominado sintetizador (ou ferramenta de síntese) pode inferir operações lógicas de *hardware* a partir das declarações de alto nível, e gerar um *netlist* genérico equivalente que implemente o comportamento especificado. Os sintetizadores de lógica digital normalmente ignoram as expressões de caráter temporal do código fonte e usam o mecanismo de bordas de *clock* para sincronizar e temporizar os circuitos que estão processando.

Ao fixar-se um determinado algoritmo, é viável, com o uso de uma HDL, projetar um *hardware* personalizado para implementá-lo e aproveitar possibilidades de paralelismo para obter um desempenho superior a uma implementação equivalente em *software*, ao custo de um orçamento maior para bancar o processo de desenvolvimento. Projetar sistemas completos em HDLs é, geralmente, muito mais difícil, complexo e dispendioso em termos de tempo do que escrever um programa que tenha a mesma funcionalidade. Conseqüentemente, há muitos trabalhos sendo realizados sobre conversão automática de código fonte de *software* tradicional, normalmente programas em C, para código HDL [CARDOSO e DINIZ, 2008]. Também existem esforços no sentido de desenvolver novas HDLs com um maior poder de abstração, que encapsulem automaticamente os aspectos mais tediosos e sujeitos à erros das linguagens já estabelecidas e se aproximem mais do paradigma dominado há décadas pelos programadores.

3.1.1 VHDL

VHSIC Hardware Description Language (VHSIC é um acrônimo para *Very High Speed Integrated Circuit*) é uma HDL projetada para descrever circuitos digitais [ASHENDEN, 1990]. A linguagem foi originalmente desenvolvida sob as ordens do Departamento de

Defesa dos Estados Unidos, no final da década de 80, para documentar o comportamento de ASICs que compunham os equipamentos vendidos às forças armadas norte-americanas.

A *VHSIC Hardware Description Language* (VHDL) foi desenvolvida com o propósito de substituir os complexos manuais que descreviam o funcionamento dos ASICs, pois até aquele momento, a única metodologia largamente utilizada no projeto de circuitos digitais era a criação de diagramas esquemáticos. O problema com esta abordagem é o fato de que os desenhos têm menor portabilidade, são mais complexos para compreensão e são extremamente dependentes da ferramenta utilizada para produzi-los. O desenvolvimento da VHDL serviu, inicialmente, aos propósitos de documentação dos projetos militares. Contudo, buscava-se também uma linguagem que facilitasse o projeto de um circuito, de modo a permitir que um algoritmo processasse sua descrição textual e o desenvolvesse sem que ocorresse a necessidade de uma especificação explícita das ligações existentes entre os seus diversos componentes. A VHDL presta-se adequadamente a tais propósitos, podendo ser utilizada para as tarefas de documentação, descrição, síntese, simulação, teste e ainda compilação de *software* em alguns casos [ASHENDEN, 1990].

Posteriormente, a linguagem foi adotada como um padrão do *Institute of Electrical and Electronics Engineers* (IEEE) e sua definição foi posta em domínio público. Este fato contribuiu enormemente para a ampliação de sua utilização. Conseqüentemente alterações e novas funcionalidades foram propostas, como é natural num processo de aprimoramento, e a linguagem sofreu algumas revisões, sendo a última em 2001.

3.1.2 Verilog

A primeira linguagem de descrição de *hardware* moderna, *Verilog HDL* ou simplesmente *Verilog*, foi introduzida pela *Gateway Design Automation* em 1985. Inicialmente, a linguagem foi projetada apenas com propósitos de simulação, e as características que permitiriam o uso de *Verilog* para síntese de circuitos foram incorporadas posteriormente. Em 1989 a *Cadence Design Systems* adquiriu a *Gateway* e disponibilizou a linguagem em domínio público em 1990 devido às pressões do mercado e à crescente concorrência imposta por VHDL. *Verilog* se tornou um padrão do IEEE em 1995, que vem sendo revisado, e continua evoluindo até o presente momento com *SystemVerilog*.

Verilog é uma linguagem muito similar a VHDL. Possui capacidades equivalentes em termos de modelagem de *hardware* e foi criada com o mesmo propósito de substituir o projeto de *hardware* com diagramas esquemáticos por uma alternativa mais portátil e de mais alto nível. Os projetistas de *Verilog* focaram-se em uma sintaxe similar à da linguagem de programação *C*, que já era amplamente utilizada no desenvolvimento de *software* de engenharia. Esta medida visava facilitar o treinamento e adaptação da mão-de-obra já qualificada em programação para o desenvolvimento de *hardware* [SMITH, 1996].

Em termos práticos, as duas HDLs padrão no mercado de ASICs e FPGAs são *Verilog* e *VHDL*. Cada ferramenta específica possui sua própria biblioteca de componentes e mega-funções escritas em *Verilog* e/ou *VHDL*. Conseqüentemente, é importante que os profissionais da área de projeto de circuitos digitais conheçam ambas as linguagens e que os fornecedores de software EDA providenciem um ambiente de desenvolvimento que também suporte ambas HDLs de maneira conjunta e em uníssono [SMITH, 1996].

3.1.3 *Bluespec SystemVerilog*

Bluespec SystemVerilog (BSV) [BLUESPEC, 2009a] é uma linguagem de descrição de *hardware* de alto nível concebida no MIT com o objetivo de facilitar e acelerar o projeto de sistemas digitais complexos. Esta linguagem é uma alternativa à *VHDL* e *Verilog*.

3.2 Ferramentas de Projeto da *Altera*

3.2.1 *Quartus II*

O *software* de projeto *Quartus II* é uma ferramenta da *Altera* que oferece um ambiente unificado de desenvolvimento para sistemas baseados em dispositivos lógico programáveis desta empresa. A utilização do *Quartus II* pelos projetistas, assim como de toda moderna ferramenta EDA, visa diminuir o tempo de engenharia investido em um projeto de *hardware* reconfigurável, agilizando o processo de desenvolvimento por meio de automação de tarefas e integração automática de programas de mais baixo nível.

Todo o fluxo de projeto do *hardware* de um SOPC, apresentado na Figura (3.2), pode ser inteiramente realizado com o *Integrated Development Environment* (IDE) do *Quartus II*. O *software* ainda tem a flexibilidade de permitir que o usuário possa, em cada fase do projeto, utilizar uma interface gráfica ou uma interface de linha de comando e *scripts* personalizados caso seja necessário [ALTERA, 2007a].

3.2.2 *SOPC Builder*

O *SOPC Builder*, ferramenta integrada ao *Quartus II*, provê uma interface gráfica padronizada para criação de SOPCs compostos pelo processador *Nios II*, controladores de memórias, periféricos padrão de microcontroladores (portas paralelas e seriais, *timers* etc.) e periféricos definidos pelo próprio usuário. O *SOPC Builder* permite que os projetistas selecionem os módulos de seu sistema a partir de uma extensa biblioteca de componentes e, se necessário, personalizem cada um deles de acordo com suas necessidades. O *software* combina os componentes de propriedade intelectual automaticamente, gerando toda a lógica de barramentos, interconexões e integração e poupando o usuário de uma tarefa

Figura 3.2: Fluxo de projeto no *Quartus II*

complexa. Dessa forma, os desenvolvedores focam-se mais em projetar lógica específica do que em construir a topologia do sistema [ALTERA, 2007d].

3.2.3 *Nios II IDE*

O *Nios II IDE* é a principal ferramenta de desenvolvimento de *software* para toda a família de processadores embarcados *Nios II*. Este aplicativo é baseado no projeto *open source* de IDE *Eclipse* e utiliza a eficiente *tool chain* de compilação GNU. Todas as tarefas necessárias no processo de desenvolvimento de *software* são desempenhadas através do *Nios II IDE*: gerenciamento de projeto, edição de código fonte, compilação e depuração [ALTERA, 2007c].

3.2.4 *ModelSim – Altera*

O *ModelSim* é uma ferramenta de verificação e simulação da *Mentor Graphics* para *hardware* especificado em HDLs que suporta diversas linguagens incluindo VHDL, *Verilog* e *SystemVerilog* [ALTERA, 2008]. O *ModelSim Altera Edition* é uma versão do *ModelSim* especialmente distribuída com bibliotecas de simulação dirigidas para os dispositivos lógicos programáveis da *Altera*. Ele é fornecido juntamente com os demais programas do *design suite* da *Altera* por meio de uma parceria entre as empresas envolvidas. O *software* possibilita a realização de simulações em diversos níveis por meio de uma interface gráfica amigável e intuitiva. A utilização desta ferramenta fez-se necessária, pois o simulador

padrão disponibilizado pela *Altera* com o *Quartus II* não está apto a realizar tarefas complexas e, principalmente, porque não suporta a simulação por meio de *testbenchs* escritos em HDLs. A utilização do ModelSim contribui de forma significativa para a detecção de erros e validação dos blocos de *hardware* desenvolvidos.

3.3 *Bluespec Workstation*

O *Bluespec Workstation* é um *software* IDE enxuto, porém não menos funcional, que pode ser utilizado durante o processo de desenvolvimento com BSV [BLUESPEC, 2009b]. A ferramenta está muito bem integrada com outros produtos de diversos fornecedores, tais como simuladores de HDL, visualizadores de formas de onda, editores de texto etc. Basicamente, o *Bluespec Workstation* auxilia o desenvolvedor a gerenciar seu projeto, utilizar o compilador e simulador da linguagem BSV de modo automático e não sujeito à erros.

3.4 Ambiente *R* para Computação Estatística

R é uma linguagem e um ambiente integrado especialmente projetado para realizar computação estatística e trabalhar com gráficos [VENABLES et al., 2007]. O ambiente também provê eficientes capacidades de recuperação, manipulação e armazenamento de dados. *R* é um projeto de *software* livre similar à linguagem *S* que foi desenvolvida por John Chambers e seus colegas da *Bell Laboratories*. *R* disponibiliza uma enorme variedade de modelos e testes estatísticos, técnicas gráficas para análise de dados e operadores para manipulação de vetores e matrizes.

O principal papel desta ferramenta dentro do trabalho de mestrado é o de testar e validar estatisticamente as computações de caráter probabilístico realizadas pelo *hardware* reconfigurável em desenvolvimento, tanto em sua fase de simulação por modelos executáveis quanto em sua fase de real execução. Notadamente, serão exigidos testes estatísticos para verificar se os números pseudo-aleatórios gerados estão de acordo com as funções de distribuição de probabilidade implementadas. Dentre estes procedimentos pode-se destacar os testes de *Kolmogorov-Smirnov* [KNUTH, 1981] e *Anderson-Darling* [JAYNES, 1995]. As funções de distribuição de probabilidade em questão são a uniforme padrão $U(a = 0, b = 1)$, a normal padrão $N(\mu = 0, \sigma = 1)$ e a exponencial padrão $Exp(\lambda = 1)$. Testes sobre a distribuição uniforme também podem ser realizados com a *TestU01*, conforme exposto na Seção (3.5). O *R* também será utilizado na produção de gráficos e estatísticas resumo que possam de alguma forma auxiliar na compreensão e análise dos conjuntos de dados obtidos durante os experimentos de validação.

3.5 Biblioteca *TestU01* para Testes Empíricos de RNGs

TestU01 é uma biblioteca de *software* implementada em linguagem *ANSI C* que oferece uma grande coleção de ferramentas para testes estatísticos empíricos em geradores de números aleatórios uniformes [L'ECUYER e SIMARD, 2007a]. A terminação *U01* no nome da biblioteca é uma referência explícita ao espaço de geração uniforme padrão $(0, 1)$ dos PRNGs algorítmicos conforme descrito na Seção (2.2.1).

A biblioteca implementa diversos tipos de geradores de números aleatórios de forma genérica e também muitos geradores específicos propostos na literatura ou encontrados em produtos de *software* amplamente utilizados. Ela também fornece implementações maduras e funcionais de vários testes estatísticos clássicos para geradores de números aleatórios, bem como de vários outros testes propostos na literatura e de testes originalmente projetados por seus autores. Os testes podem ser aplicados aos geradores pré-definidos da própria *TestU01*, aos geradores definidos pelo próprio usuário ou a um conjunto de dados numéricos convenientemente formatados no intervalo $(0, 1)$ provenientes de um arquivo. Há ainda a possibilidade de se executar determinados conjuntos de testes específicos pré-definidos, ou baterias de testes.

A *TestU01* possui qualidades que contribuíram decisivamente para sua utilização no projeto: a biblioteca é portátil e veloz devido a sua implementação em *ANSI C*, distribuída gratuitamente ¹, extremamente bem documentada [L'ECUYER e SIMARD, 2007b], de fácil utilização e representa o estado da arte em *software* para testes de PRNGs. Também é importante frisar que as baterias de teste desta biblioteca já foram aplicadas com sucesso ao *software* original do MT19937 [L'ECUYER e SIMARD, 2007a]. Desta forma, é possível efetuar uma comparação com os resultados obtidos por sua versão em *hardware* reconfigurável a fim de se verificar a equivalência funcional.

3.6 Padrão *IP-XACT*

IP-XACT é um padrão de formato *eXtensible Markup Language* (XML) para descrição e representação de componentes de sistemas eletrônicos. O *IP-XACT* foi definido pelo consórcio *Spirit* [SPIRIT, 2008] como uma norma para permitir a configuração e integração automática de diferentes blocos de propriedade intelectual, independentemente de seus fornecedores, por meio de ferramentas EDA. O padrão define uma interface comum para uso nas ferramentas EDA que permite acessar os metadados dos componentes e sistemas de modo independente de implementação. Conseqüentemente, isso possibilita a migração de componentes e projetos entre qualquer ferramenta compatível com o *IP-XACT*.

¹*TestU01*, <http://www.iro.umontreal.ca/~simardr/testu01/tu01.html>. Último acesso em 23/11/2009.

Os módulos de *hardware* que forem desenvolvidos durante a pesquisa serão especificados segundo as normas do *IP-XACT*, se houver tempo hábil para realizar tal tarefa. A padronização de acordo com o *IP-XACT* visa uma possível contribuição à ferramenta de co-projeto de arquiteturas para robótica móvel *RoboArch* [BONATO e MARQUES, 2008]. Estes *cores* seriam um primeiro estudo de caso para a plataforma *RoboArch* e alavancariam de forma significativa seu desenvolvimento.

CAPÍTULO 4

Resultados

4.1 Projeto de *Hardware* para o MT19937

As implementações de *hardware* efetuadas foram fortemente baseadas na arquitetura sugerida em [KONUMA e ICHIKAWA, 2005]. Basicamente, esta implementação consiste de três módulos: um para calcular a recorrência (2.15), um para calcular o *tempering* (2.18) e um para armazenar o estado interno do circuito que representa o estado do gerador de números aleatórios em um dado momento. Inicialmente, o elemento de estado utilizado foi uma *Random Access Memory* (RAM) multi-porta em uma implementação denominada PMT. Porém, dado que o MT é uma extensão dos algoritmos LFSRs, que podem ser naturalmente implementados por registradores encadeados [CHU e JONES, 1999], sugeriu-se também uma implementação designada FMT, na qual o estado é armazenado em um conjunto de registradores e não mais em uma memória. Comparando-se as duas implementações percebe-se que a versão PMT é mais compacta em termos de utilização de recursos de *hardware* e a versão FMT é mais veloz. Neste trabalho optou-se por projetar uma versão aprimorada da variante FMT por meio de uma metodologia *bottom-up*.

4.1.1 Implementação em VHDL

Os primeiros esforços para o desenvolvimento do trabalho visaram obter uma implementação em *hardware* do gerador de números aleatórios MT19937 utilizando VHDL. Mais especificamente, a implementação foi baseada em metodologias puramente estruturais [PEDRONI, 2004], técnicas para produção de especificações genéricas objetivando reuso de componentes [MEIYAPPAN et al., 1999] e em um subconjunto de construções

da linguagem sintetizável *de facto* [IEEE, 1999] e aceito pela maioria das ferramentas EDA do mercado. Essas diretrizes foram seguidas com o propósito de produzir um *core* independente de plataforma de desenvolvimento, veloz e de fácil compreensão.

Foram escritos módulos de *hardware* de caráter básico tais como multiplexadores, circuitos aritméticos para somadores *ripple-carry* e multiplicadores em arranjo (*array multipliers*) [BROWN e VRANESIC, 2000] e também diversos tipos de elementos de estado. Esta etapa caracterizou-se como uma fase de aprendizado e familiarização necessária com a HDL e os artefatos nela produzidos constituem-se de um excelente material didático para a área de computação reconfigurável. O material em questão foi usado na edição do 1º semestre de 2009 da disciplina “Co-projeto de *Hardware/Software* para Sistemas Embarcados”¹ na qual o mestrando atuou como estagiário do Programa de Aperfeiçoamento de Ensino (PAE).

Em fase mais avançada, os módulos de computação propostos em [KONUMA e ICHIKAWA, 2005] foram implementados. As Figuras (4.1) e (4.2) mostram respectivamente os diagramas de bloco dos circuitos combinacionais que computam as equações (2.15) e (2.18).

Figura 4.1: Diagrama de blocos do *hardware* para relação de recorrência do MT19937

O *hardware* descrito por meio do diagrama de blocos da Figura (4.3) foi projetado para agregar os módulos de recorrência e *tempering* e assim facilitar a posterior integração com o restante do projeto. Este módulo é a única unidade que efetivamente realiza a computação especificada pelo algoritmo do MT. Os outros módulos que compõe o restante do projeto

¹SSC 711 – Co-projeto de *Hardware/Software* para Sistemas Embarcados – 1º Semestre de 2009, <http://www.icmc.usp.br/~lcr/en/courses/hwswcodesign/>. Último acesso em 26/11/2009.

Figura 4.2: Diagrama de blocos do *hardware* para *tempering* do MT19937

de *hardware* são responsáveis apenas por simplesmente armazenar o estado dos circuitos seqüenciais ou prover sua lógica de controle.

O passo seguinte foi implementar uma estrutura de registradores encadeados como no FMT. Porém, esta arquitetura foi estendida com a adição de um *hardware* para inicializar o gerador preenchendo seus registradores de estado com valores produzidos a partir em uma semente s fornecida. O esquema de inicialização se dá por meio de um LCG ligeiramente modificado que fora originalmente proposto em [MATSUMOTO e NISHIMURA, 1998]. Sua formulação matemática é apresentada na equação (4.1) e os parâmetros utilizados na implementação são $a = 1812433253$, $w = 32$ e $h = 2$.

$$\begin{aligned} x_0 &= s \\ x_i &= a(x_{i-1} \oplus (x_{i-1} \gg w - h)) + i, \quad n > i > 0 \end{aligned} \quad (4.1)$$

O diagrama de blocos do módulo de inicialização está representado na Figura (4.4). Foi necessário também projetar uma *Finite State Machine* (FSM) para controlar o *datapath* deste circuito. A máquina de estados, uma máquina de *Moore* [KATZ e BORRIELLO, 2004], é mostrada na Figura (4.5).

Figura 4.3: Diagrama de blocos do *hardware* unificado para computação do MT19937

Figura 4.4: Diagrama de blocos do *hardware* de inicialização do MT19937

A última etapa da implementação constituiu-se em integrar todos os módulos de *hardware* até então produzidos e produzir uma máquina de controle semelhante a da (4.5) para ativar os sinais do *datapath* de todo o circuito. O módulo de mais alto nível é retratado na Figura (4.6) na forma de um diagrama de blocos.

É importante ressaltar que para cada módulo de *hardware* especificado em VHDL, também foi escrito um *testbench* cujo objetivo era verificar sua funcionalidade da maneira mais completa possível, visando uma cobertura de 100%. Para os módulos mais simples, como por exemplo circuitos aritméticos e multiplexadores, os estímulos enviados ao *Design under Test* (DUT) foram gerados de forma *on-line* pelo próprio *testbench* e a cobertura de 100% foi atingida com sucesso. Todavia, os módulos mais complexos e que possuíam um domínio de entrada muito grande foram verificados por *testbenches* que realizam entrada e saída com arquivos. Estes arquivos foram gerados por programas externos aos *testbenches* e seu conteúdo segue a estrutura clássica para o teste funcional: possuem um conjunto de dados de entrada e um conjunto de dados para a saída esperada advinda do processamento dos dados de entrada. Todos os *testbenches* foram executados com o *software ModelSim* que foi descrito na Seção (3.2.4).

Figura 4.5: Máquina de estados finitos para controle do circuito de inicialização do MT19937

Finalmente o *hardware* foi sintetizado com o *Quartus II* versão 9.0 SP1 executando em um PC *Intel Quad Core i7* com 12GB de RAM e sistema operacional Ubuntu Linux 9.04 64 *bits*. Os resultados da compilação seguem na tabela (4.1). Infelizmente o *software* de estimação de potência dissipada não conseguiu obter resultados para o *hardware* implementado.

<i>Kit</i> de Desenvolvimento	<i>Nios II Development Kit</i>	<i>Stratix III FPGA Development Kit</i>
Família FPGA	<i>Stratix II</i>	<i>Stratix III</i>
FPGA	EP2S60F672C3	EP3SL150F1152C3N
ALUTs Combinacionais	853 (2%)	1.364 (2%)
Registradores Lógicos Dedicados	627 (1%)	2.266 (3%)
<i>Bits</i> de Memória	18.715 (< 1%)	2.901 (< 1%)
<i>Clock</i> (MHz)	43.20	55.42
Potência (mW)	—	—
Tempo de Compilação (min:s)	7:28	7:39

Tabela 4.1: Relatório de compilação da implementação em VHDL do MT19937

4.1.2 Implementação em *Bluespec SystemVerilog*

A arquitetura apresentada na Seção (4.1.1) também foi implementada com BSV e conceitualmente é idêntica: os mesmos diagramas de blocos e máquinas de estados são capazes de descrevê-la. Um estudo comparativo entre as duas linguagens pode ser encontrado em

Figura 4.6: Diagrama de blocos do MT19937

[GRUIAN e WESTMIJZE, 2008] e as mesmas conclusões se aplicam ao atual esforço de pesquisa.

O *hardware* produzido foi sintetizado no mesmo ambiente de desenvolvimento e os resultados da compilação seguem na tabela (4.2).

Kit de Desenvolvimento	<i>Nios II Development Kit Stratix II Edition</i>	<i>Stratix III FPGA Development Kit</i>
Família FPGA	<i>Stratix II</i>	<i>Stratix III</i>
FPGA	EP2S60F672C3	EP3SL150F1152C3N
ALUTs Combinacionais	132 (< 1%)	132 (< 1%)
Registradores Lógicos Dedicados	19.984 (41%)	19.984 (18%)
Blocos DSP	8 (3%) (9 bits)	4 (3%) (18 bits)
Clock (MHz)	125.63	122.44
Potência (mW)	644.73	656.97
Tempo de Compilação (min:s)	3:57	2:45

Tabela 4.2: Relatório de compilação da implementação em *Bluespec SystemVerilog* do MT19937

4.1.3 Validação do Modelo de *Hardware*

O modelo executável do *hardware* que implementou do gerador de números aleatórios MT19937 passou por uma pré-validação comparativa com sua versão original em *software*. Este processo teve como objetivo verificar a equivalência funcional entre as versões. A

Figura 4.7: Alocação dos recursos de *hardware* da implementação em VHDL do MT19937

Figura (4.9) descreve todo o trabalho realizado que envolveu, entre muitas atividades, o desenvolvimento de programas em *R* e *ANSI C*.

O resultado desta validação funcional pode ser visto na Figura (4.10), que mostra na forma de gráfico de três dimensões a distribuição uniforme dos números produzidos pelo gerador implementado em *hardware*. Formalmente, há pouco valor em uma equidistribuição de tão baixa dimensionalidade. Porém este gráfico pode auxiliar na compreensão em dimensões mais elevadas e tem valor didático. Há também uma versão animada do gráfico publicada *on-line*².

O gerador também foi submetido a uma rigorosa bateria de testes estatísticos implementados com a *TestU01*. A estratégia de aplicação dos testes envolveu a elaboração de estrutura para comunicação inter-processos conforme sintetizado pela Figura (4.11). Os resultados dos testes são apresentados na Tabela (4.3) e são idênticos aos mostrados em [L'ECUYER e SIMARD, 2007a]. Este fato demonstra a inegável qualidade do PRNG desenvolvido neste projeto.

4.2 Co-projeto de *Hardware/Software* para Aritmética de Ponto Fixo

Como uma alternativa à representação de números reais com ponto flutuante (gargalo de desempenho usual em sistemas embarcados) optou-se por converter a aritmética da

²*Bluespec MT19973 3D Point Plot*, <http://www.youtube.com/watch?v=uSAhqcl6E4Y>. Último acesso em 26/11/2009.

Figura 4.8: Alocação dos recursos de *hardware* da implementação em *Bluespec SystemVerilog* do MT19937

Bateria de Testes	Rigor ¹	Número de Testes	Resultados ²	Núm. Aleatórios Utilizados – $\log_2 n$ ³
Padrão FIPS-140-2 NIST	•	16	P 16	9,29
Marsaglia <i>Diehard</i>	•	15	P 15	33,30
<i>TestU01 Small Crush</i>	•	10	P 10	30,25
<i>TestU01 Crush</i>	• •	96	P 94, S 2 (OK)	40,56 (35)
<i>TestU01 Big Crush</i>	• • •	106	P 44, N 66	41,12 (38)

1. •: pouco rigoroso, • •: rigoroso, • • •: muito rigoroso.

2. P: testes bem sucedidos, S: testes com p -valor suspeito, N: testes não executados.

3. Indicado pelos resultados da bateria. Entre parênteses sugerido em [L'ECUYER e SIMARD, 2007b].

Tabela 4.3: Resultados dos testes estatísticos aplicados ao modelo *Bluesim* do MT19937

aplicação totalmente para ponto fixo.

Implementou-se uma biblioteca de *software* para aritmética de ponto fixo utilizando-se a linguagem de programação *C++*. A biblioteca foi projetada para o processador *Nios II* e pode ser facilmente adaptada caso uma mudança de resolução seja necessária em seu tipo de dado numérico.

Um rotacionador CORDIC iterativo que opera sobre números de ponto fixo, no mesmo formato da biblioteca de *software*, foi implementado em *hardware* com a linguagem BSV. Este *core* é capaz de calcular rapidamente funções trigonométricas e afins. A Fugira (4.12) mostra a arquitetura do *hardware* desenvolvido.

Figura 4.9: Estratégia adotada para validar os modelos de *hardware* do MT19937 contra a implementação original em *software*

Figura 4.10: Conjunto de 100.000 pontos em $(0,1)^3$ gerados pelos modelos de *hardware* do MT19937

4.3 SOPC do Filtro de Partículas para Localização de Robôs Móveis

Um dos maiores desafios enfrentados durante o desenvolvimento de sistemas de *hardware* é a fase de integração dos componentes desenvolvidos. Para tanto, almenjando-se um SOPC completo para a aplicação de interesse, faz-se necessário integrar o módulo *Bluespec* produzido ao barramento *Avalon*. Desta forma o processador *Nios II*, mestre do barramento, será capaz de se comunicar com blocos dedicados. No caso específico deste projeto de pesquisa, os blocos dedicados são o rotacionador CORDIC e o gerador MT19937.

A integração será realizada por meio da ferramenta *BlueAvalon*, que está sendo desenvolvida pelo grupo de pesquisa do LCR. A utilização deste software é mostrada na Figura (4.13).

Figura 4.11: Estratégia adotada para aplicar os testes estatísticos ao modelo *Bluesim* do MT19937

Figura 4.12: Rotacionador CORDIC iterativo de ponto fixo

Figura 4.13: Fluxo de projeto com a ferramenta *BlueAvalon*

CAPÍTULO 5

Considerações Finais

“Transportai um punhado de terra todos os dias e fareis uma montanha.”

– Confúcio

5.1 Conclusão

Geradores de números pseudo-aleatórios representam uma classe de algoritmos fundamental para a ciência da computação e são largamente aplicados para solucionar diversos tipos de problemas por meio de técnicas estocásticas de simulação. Um destes métodos de simulação, o filtro de partículas, possui grande aplicação na área da robótica móvel que certamente, em um futuro próximo, estará presente em nossa vida cotidiana graças à pervasividade inerente aos sistemas embarcados.

Nota-se que para se implementar eficientemente o filtro de partículas em um sistema embarcado é fundamental partir de um bom gerador de números pseudo-aleatórios implementado em *hardware* e que as ferramentas de desenvolvimento EDA são essenciais para o desenvolvimento do trabalho. A área de pesquisa que integra sistemas embarcados, robótica móvel e computação reconfigurável mostrou-se significativamente complexa e desafiadora mas de modo algum deixando de ser interessante e motivadora.

REFERÊNCIAS

- ALTERA *Stratix device handbook volume 1*. Altera Corporation, San Jose, California, 2006.
- ALTERA *Introduction to the Quartus II software*. Altera Corporation, San Jose, California, 2007a.
- ALTERA *Nios II processor reference handbook*. Altera Corporation, San Jose, California, 2007b.
- ALTERA *Nios II software developer's handbook*. Altera Corporation, San Jose, California, 2007c.
- ALTERA *Quartus II handbook volume 4: SOPC bulider*. Altera Corporation, San Jose, California, 2007d.
- ALTERA *Mentor Graphics ModelSim support*. Altera Corporation, San Jose, California, 2008.
- ANDRAKA, R. A survey of CORDIC algorithms for FPGA based computers. In: *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays (FPGA'98)*, Monterey, CA, United States, p. 191–200, 1998.
- ARAGÃO, A.; ROMERO, R.; MARQUES, E. *Computação reconfigurável: Experiências e perspectivas*, v. 1, cap. Computação Reconfigurável Aplicada à Robótica (in CORE-2000 Workshop em Computação Reconfigurável) Rio de Janeiro, RJ: Editora Brasport, p. 184–188, 2000.
- ARAPOSTATHIS, A.; KUMAR, R.; TANGIRALA, S. Controlled markov chains and safety criteria. In: *Proceedings of the 40th IEEE Conference on Decision and Control*, p. 1675–1680, 2001.

- ASHENDEN, P. The VHDL cookbook. 1990.
- BLUESPEC *Bluespec System Verilog reference guide*. Bluespec, Inc., Waltham, MA, 2009a.
- BLUESPEC *Bluespec System Verilog user guide*. Bluespec, Inc., Waltham, MA, 2009b.
- BOBDA, C. *Introduction to reconfigurable computing - architectures, algorithms and applications*. Dordrecht, The Netherlands: Springer, 375 p., 2007.
- BONATO, V. *Proposta de uma arquitetura de hardware em FPGA implementada para SLAM com multi-câmeras aplicada à robótica móvel*. Tese de Doutorado, Instituto de Ciências Matemáticas e de Computação (ICMC), 2008.
- BONATO, V.; MARQUES, E. Proposta de uma ferramenta para o co-projeto de arquiteturas de Hw/Sw desenvolvidas a nível de sistema para a robótica embarcada, projeto FAPESP de Pós-Doutorado, 2008.
- BONDALAPATI, K.; PRASANNA, V. Reconfigurable computing systems. In: *Proceedings of the IEEE*, p. 1201–1217, 2002.
- BORKAR, V. S. *Topics in controlled markov chains*. N. 240 in Pitman Research Notes in Mathematics Series. New York: Longman Scientific & Technical, 179 p., 1991.
- BROWN, S.; VRANESIC, Z. *Fundamentals of digital logic with VHDL design*. McGraw Hill, 2000.
- CARDOSO, J. M. P.; DINIZ, P. C. *Compilation techniques for reconfigurable architectures*. Springer, 223 p., 2008.
- CHU, P.; JONES, R. Design techniques of FPGA based random number generator. In: *MAPLD 1999 - 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference*, Laurel, MD, United States: Johns Hopkins University, p. 28–30, 1999.
- COMPTON, K.; HAUCK, S. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, v. 34, n. 2, p. 171–210, 2002.
- CUI, W.; CHEN, H.; HAN, Y. VLSI implementation of universal random number generator. In: *Asia-Pacific Conference on Circuits and Systems (APCCAS'02)*, p. 465–470, 2002.
- CUI, W.; LI, C.; SUN, X. FPGA implementation of universal random number generator. In: *Proceedings of the 7th International Conference on Signal Processing*, p. 495–498, 2004.

- DEHON, A.; WAWRZYNEK, J. Reconfigurable computing: What, why and implications for design automation. In: *Proceedings of the 36th Design Automation Conference*, p. 610–615, 1999.
- DELLAERT, F.; FOX, D.; BURGARD, W.; THRUN, S. Monte carlo localization for mobile robots. In: *Proceedings of IEEE on Robotics and Automation*, p. 1322–1328, 1999.
- DEVROYE, L. *Non-uniform random variate generation*. New York: Springer-Verlag, 1986.
- DIARD, J.; BESSIERE, P.; MAZER, E. A survey of probabilistic models using the bayesian programming methodology as a unifying framework. In: *Proceedings of the 2nd International Conference on Computational Intelligence, Robotics and Autonomous Systems (IEEE-CIRAS)*, Singapore, 2003.
- FISCHER, V.; DRUTAROVSKY, M. True random number generator embedded in reconfigurable hardware. In: *CHES'02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, London, UK: Springer-Verlag, p. 415–430, 2003.
- GATES, B. A robot in every home. *Scientific American Magazine*, 2007.
- GIMPS Great Internet Mersenne Prime Search. 2009.
Disponível em <http://www.mersenne.org/>
- GRUIAN, F.; WESTMIJZE, M. VHDL vs. Bluespec SystemVerilog: A case study on a Java embedded architecture. In: *SAC '08: Proceedings of the 2008 ACM Symposium on Applied Computing*, New York, NY, USA: ACM, p. 1492–1497, 2008.
- HARTENSTEIN, R. W. *Hardware description languages*. New York, NY, USA: Elsevier Science Inc., 1987.
- IBM *System/360 scientific subroutine package version III - Programmer's manual*. International Business Machines Corporation, White Plains, New York, 1968.
- IEEE *IEEE standard for VHDL register transfer level (RTL) synthesis*. Institute of Electrical and Electronics Engineers, 1999.
- JAYNES, E. *Probability theory: The logic of science*. Cambridge University Press, 1995.
- JOHNSONBAUGH, R. *Discrete mathematics*. New York, NY, United States: Collier Macmillan, livro de Matemática Discreta II, disponível na biblioteca., 1990.

- KATZ, R. H.; BORRIELLO, G. *Contemporary logic design*. 2nd ed. Prentice Hall, 608 p., 2004.
- KNUTH, D. *The art of computer programming*, v. 2 Seminumerical Algorithms, cap. Random Numbers. 2nd ed Reading, MA: Addison-Wesley Publishing Company, p. 1–177, 1981.
- KOHLBRENNER, P.; GAJ, K. An embedded true random number generator for FPGAs. In: *Proceedings of the 2004 ACM/SIGDA 12th international Symposium on Field Programmable Gate Arrays*, ACM New York, NY, USA, p. 71–78, 2004.
- KONUMA, S.; ICHIKAWA, S. Design and evaluation of hardware pseudo-random number generator MT19937. *IEICE Transactions on Information and Systems*, v. E88-D, n. 12, p. 2876–2879, 2005.
- L'ECUYER, P. A tutorial on uniform variate generation. In: *Proceedings of the 21st Conference on Winter Simulation*, Washington, DC, United States, p. 40–49, 1989.
- L'ECUYER, P. Random numbers for simulation. *Communications of the ACM*, v. 33, n. 10 Special Issue on Simulation, p. 85–97, 1990.
- L'ECUYER, P. *Handbook of computational statistics - concepts and methods*, cap. Random Number Generation. N. 2 Springer-Verlag, p. 35–70, 2004.
- L'ECUYER, P. *Handbooks in operations research and management science - simulation*, v. 13 de *Handbooks in Operations Research and Management Science*, cap. Random Number Generation Amsterdam, Netherlands: Elsevier Science, p. 55–81, 2006.
- L'ECUYER, P.; SIMARD, R. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, v. 33, n. 4, p. 22, 2007a.
- L'ECUYER, P.; SIMARD, R. *TestU01 user's guide (compact version)*. Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, 2007b.
- LEE, D.; VILLASENOR, J. D.; LUK, W.; W., L. P. H. A hardware gaussian noise generator using the Box-Muller method and its error analysis. *IEEE Transactions on Computers*, v. 55, n. 6, p. 659–671, 2006.
- LEE, D. U.; LUK, W.; VILLASENOR, J.; ZHANG, G.; LEONG, P. H. W. A hardware gaussian noise generator using the Wallace method. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, v. 13, n. 8, p. 911–920, 2005.

- LEONG, P.; TSOI, K. Field programmable gate array technology for robotics applications. In: *IEEE International Conference on Robotics and Biomimetics (ROBIO)*, p. 295–298, 2005.
- LEWIS, T. G.; PAYNE, W. H. Generalized feedback shift register pseudorandom number algorithm. *Journal of the ACM*, v. 20, n. 3, p. 456–468, 1973.
- LIU, J.; CHEN, R. Sequential monte carlo methods for dynamic systems. *Journal of the American Statistical Association*, v. 93, n. 443, p. 1032–1044, 1998.
- MARSAGLIA, G. The Marsaglia random number CD-ROM with the Diehard battery of tests of randomness. CD-ROM, 1995.
Disponível em <http://stat.fsu.edu/pub/diehard/>
- MATARIĆ, M. *The robotics primer*. London, England: MIT Press, 288 p., 2007.
- MATSUMOTO, M.; KURITA, Y. Twisted GFSR generators. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, v. 2, n. 3, p. 179–194, 1992.
- MATSUMOTO, M.; KURITA, Y. Twisted GFSR generators II. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, v. 4, p. 254–266, 1994.
- MATSUMOTO, M.; NISHIMURA, T. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, v. 8, n. 1 (Special Issue on Uniform Random Number Generation), p. 3–30, 1998.
- MCCOLLUM, J. M.; LANCASTER, J. M.; BOULDIN, D. W.; PETERSON, G. D. Hardware acceleration of pseudo-random number generation for simulation applications. In: *Proceedings of the 35th Southeastern Symposium on System Theory*, p. 299–303, 2003.
- MEIYAPPAN, S.; JARAMILLO, K.; CHAMBERS, P. Ten tips for generating reusable VHDL. *Electronics Design Engineers Magazine*, p. 49–62, 1999.
- PANNETON, F.; L'ECUYER, P.; MATSUMOTO, M. Improved long-period generators based on linear recurrences modulo 2. *ACM Transactions on Mathematical Software*, v. 32, n. 1, p. 1–16, 2006.
- PARK, S. K.; MILLER, K. W. Random number generators: Good ones are hard to find. *Communications of the ACM*, v. 31, n. 10, p. 1192–1201, 1988.
- PC/104 EMBEDDED CONSORTIUM Why PC/104? the need for an embedded-PC standard. 2009.
Disponível em <http://www.pc104.org/specifications.php>

- PEDRONI, V. *Circuit design with vhdl*. MIT Press, prof. Eduardo tem uma cópia., 2004.
- PLAVEC, F. *Softcore processor design*. Dissertação de Mestrado, Department of Electrical and Computer Engineering, University of Toronto, 2004.
- REKLEITIS, I. *A particle filter tutorial for mobile robot localization*. Relatório Técnico TR-CIM-04-02, Centre of Intelligent Machines, McGill University, 3480 University St., Montreal, Québec, Canada, 2004.
- ROMAN, S. *An introduction to discrete mathematics*. 2nd ed. Thomson Learning, 469 p., 1989.
- ROY, N.; THRUN, S. Coastal navigation with a mobile robot. In: *Advances in Neural Information Processing Systems (NIPS'99)*, p. 1043–1049, 1999.
- SHAW, C. Industry's highest density FPGA. *Embedded New Electronics Magazine*, 2009.
- SMITH, D. VHDL and Verilog compared and contrasted-plus modeled example written in VHDL, Verilog and C. In: *Design Automation Conference Proceedings 1996, 33rd*, p. 771–776, 1996.
- SPIRIT *IP-XACT v1.4: A specification for XML meta-data and tool interfaces*. Spirit Consortium, 2008.
- STROUSTRUP, B. *The C++ programming language*. Special ed. Reading Mass. USA: Addison-Wesley, 1029 p., 2000.
- TAUSWORTHE, R. C. Random numbers generated by linear recurrence modulo two. *Mathematics of Computation*, v. 19, n. 90, p. 201–209, 1965.
- THOMAS, D.; LUK, W.; LEONG, P.; VILLASENOR, J. Gaussian random number generators. *ACM Computing Surveys*, v. 39, n. 4, p. 38, 2007.
- THRUN, S. *A framework for programming embedded systems: Initial design and results*. Relatório Técnico CMU-CS-98-142, Carnegie Mellon University, Pittsburgh, PA, United States, 1998.
- THRUN, S. Probabilistic algorithms in robotics. *AI Magazine*, v. 21, n. 4, p. 93–109, 2000a.
- THRUN, S. Towards programming tools for robotics that integrate probabilistic computation and learning. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, San Francisco, CA, United States, p. 306–312, 2000b.

- THRUN, S. Is robotics going statistics? The field of probabilistic robotics, 2001.
- THRUN, S. Particle filters in robotics. In: *Proceedings of the 17th Annual Conference on Uncertainty in AI (UAI)*, 2002a.
- THRUN, S. Probabilistic robotics. *Communications of the ACM*, v. 45, n. 3 Special Issue Robots: Intelligence, Versatility, Adaptivity, p. 52 – 57, 2002b.
- THRUN, S.; BURGARD, W.; FOX, D. *Probabilistic robotics*. Cambridge, MA, United States: MIT Press, 2005.
- TSOI, K.; LEUNG, K.; LEONG, P. Compact fpga-based true and pseudo random number generators. In: *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2003)*, p. 51–61, 2003.
- VENABLES, W. N.; SMITH, D. M.; GENTLEMAN, R.; IHAKA, R.; MAECHLER, M.; THE R DEVELOPMENT CORE TEAM *An introduction to R*. 2007.
- VOLDER, J. E. The CORDIC trigonometric computing technique. *IRE Transactions on Electronic Computers*, v. EC-8, n. 3, p. 330–334, 1959.
- WEBB, W. Mobile robots get serious. *Electronics Design Engineers Magazine*, 2005.
- WOLF, D.; HOLANDA, J. A. D.; BONATO, V.; PERON, R.; MARQUES, E. An FPGA based mobile robot controller. In: *Proceedings of the 3rd Southern Conference on Programmable Logic (SPL'07)*, p. 119–124, 2007.
- ZHANG, G.; LEONG, P.; LEE, D.; VILLASENOR, J.; CHEUNG, R.; LUK, W. Ziggurat-based hardware gaussian random number generator. In: *International Conference on Field Programmable Logic and Applications (FPL'05)*, p. 275–280, 2005.

APÊNDICE A

Créditos das Figuras

(2.1) e (2.2): Obtidas de [THRUN et al., 2005]. Disponível em <http://www.probabilistic-robotics.org/>, último acesso em 27/01/2009.

Epígrafe da Seção (2.2): Dilbert by Scott Adams, Copyright ©2001 United Feature Syndicate, Inc.

(2.3): Produzida por Luis Sánchez, estudante de física da Universidade de Guadalajara. Disponível em <http://en.wikipedia.org/wiki/RANDU> por meio da *GNU Free Documentation License*, último acesso em 03/02/2009.

(2.4): Adaptada de [DEHON e WAWRZYNEK, 1999]. Disponível em <http://doi.acm.org/10.1145/309847.310009>, último acesso em 06/02/2009.

(2.5): Obtida de [BOBDA, 2007].

(2.9): Obtida de [ALTERA, 2006].

(3.2): Obtida de [ALTERA, 2007a].