IMPLEMENTAÇÃO COM PRECISÃO VARIÁVEL DE ALGORITMOS DE FILTRAGEM ADAPTATIVA COM RESTRIÇÕES LINEARES

Bernardo Miranda Rodrigues Penna

TESE SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Aprovada por:

Prof. Marcello Luiz Rodrigues de Campos, Ph.D.

Prof. Sérgio Lima Netto, Ph.D.

Prof. José Antônio Apolinário Junior, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

SETEMBRO DE 2008

PENNA, BERNARDO MIRANDA RODRIGUES

Implementação com precisão variável de algoritmos de filtragem adaptativa com restrições lineares [Rio de Janeiro] 2008

XII, 97 p., 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia Elétrica, 2008)

Dissertação - Universidade Federal do Rio de Janeiro, COPPE

1.Processamento de Sinais 2.CORDIC 3. Filtragem adaptativa 4.BEAMFORMING 5.HOUSEHOLDER 6.HCMLS 7.NHCLMS

I.COPPE/UFRJ II.Título (série)

———————————- Página em Branco ———————————

Agradecimentos

À minha avó Anésia e meus tios Jurema e José Vicente, sem o suporte e carinho de vocês não iria conseguir finalizar o meu mestrado. Vocês fazem a minha vida ser muito especial.

À minha namorada Raquel, que mesmo à distância conseguiu me dar conforto e calma para desenvolver este trabalho.

Aos meus companheiros de Embratel, Adriano Alves, Alexandre Antunes, Carlos Cunha, Francisco Gioia, Ricardo Moura, Walderson Vidal e principalmente ao meu chefe José Silva, sem a ajuda e comprometimento de vocês não teria conseguido cursar o mestrado.

Ao meu orientador e amigo Marcello Campos que sempre me apoiou e direcionou em todo o mestrado. Seu incentivo e conhecimento foram de suma importância para que este trabalho pudesse ser desenvolvido.

Ao novo amigo Bruno de Carvalho Costa, que graças a seus conselhos e ajuda, pude desenvolver de forma mais organizada e objetiva esta tese.

A todos colegas que cursaram matérias do mestrado comigo. Com certeza sem a ajuda de vocês tudo seria mais difícil.

Ao meu grande amigo Fabiano Castoldi, que sempre me ajudou nos momentos mais complicados desta tese.

E por último mas não menos importante, à minha querida mãe Juracema. Sem seu esforço e dedicação não estaria aqui hoje. Só lamento você não estar desfrutando este momento comigo, todavia você continua muito presente em meu coração.

Muito obrigado a todos vocês.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

IMPLEMENTAÇÃO COM PRECISÃO VARIÁVEL DE ALGORITMOS DE FILTRAGEM ADAPTATIVA COM RESTRIÇÕES LINEARES.

Bernardo Miranda Rodrigues Penna

Setembro/2008

Orientador: Marcello Luiz Rodrigues de Campos

Programa: Engenharia Elétrica

Desde que surgiram as Telecomunicações, diversos avanços tecnológicos foram conseguidos graças ao desenvolvimento de algoritmos mais eficientes e *hardwares* mais rápidos. A eficiência de um algoritmo é medida pela quantidade de operações necessárias para processar uma determinada aplicação. Quanto menos operações, mais eficiente é o algoritmo. Sendo assim esta tese busca estudar os algoritmos de filtragem adaptativa com precisão variável a fim de verificar sua eficiência em uma aplicação de filtragem espacial adaptativa.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

IMPLEMENTATION WITH VARIABLE PRECISION OF ADAPTATIVE FILTERING ALGORITHMS WITH LINEAR CONSTRAINTS

Bernardo Miranda Rodrigues Penna

September/2008

Advisor: Marcello Luiz Rodrigues de Campos

Department: Electrical Engineering

Since Telecommunications appeared, several technological advances were achieved thanks to the development of more efficient algorithms and faster devices. The efficiency of an algorithm is measured by the amount of operations needed to process a given application. The least operations needed the most efficient the algorithm is. So this thesis shows the study of adaptative filtering algorithms with variable precision in order to verify their efficiency in an application of adaptive spatial filtering.

Sumário

Lista de Figuras

Capítulo 1

Introdução

1.1 Motivação

Desde que surgiram as telecomunicações, diversos avanços tecnológicos foram conseguidos graças ao desenvolvimento de algoritmos mais eficientes e *hardwares* mais rápidos.

A eficiência de um algoritmo é medida pela quantidade de operações necessárias para processar uma determinada aplicação. Quanto menos operações, mais eficiente é o algoritmo. De acordo com [1], uma das formas para medir a complexidade de um algoritmo é através da notação *flop* (*floating point operation*), por exemplo, se um algoritmo possui n multiplicações e n somas, logo sua complexidade é de $2n$ *flops*.

A rapidez do *hardware* é medida pelo tempo de resposta que o mesmo demora para executar todas operações matemáticas de um algoritmo. O tempo de resposta de cada operação matemática está relacionado com a quantidade de *bits* utilizado no cálculo. À medida que essa quantidade de *bits* é reduzida, a resposta do *hardware* é mais rápida, todavia um erro será introduzido devido à diminuição de precisão no cálculo.

Sendo assim, esta tese busca estudar o comportamento dos algoritmos HCLMS (*Householder Constrained Least Mean Square*) e NHCLMS (*Normalized Householder Constrained Least Mean Square*), desenvolvidos por [2], com um *hardware* onde seja possível realizar operações matemáticas com precisão variável (CORDIC).

O HCLMS e NHCLMS são algoritmos derivados do CLMS (*Constrained Least Mean Square*) com a utilização da transformada de *Householder* (HT). Esses algorit-

mos são muito eficientes e pertencem à família de filtros adaptativos LCAF (*Linearly Constrained Adaptive Filter*) com restrições lineares.

CORDIC (*COordinate Rotation DIgital Computer*) é um algoritmo simples e rápido que utiliza somadores e deslocadores para calcular funções aritméticas, trigonométricas e exponenciais com precisão de *bits* variável. A junção dos algoritmos HCLMS e NHCLMS com o algoritmo CORDIC em uma aplicação de antenas inteligentes adaptativas é o foco de estudo desta tese.

1.2 Organização da Tese

A Figura 1.1 mostra como a tese está organizada. Os assuntos apresentados nesta figura estão associados aos principais temas de cada um dos capítulos. A tese possui sete capítulos, onde todos estão representados na Figura 1.1, com exceção deste primeiro capítulo.

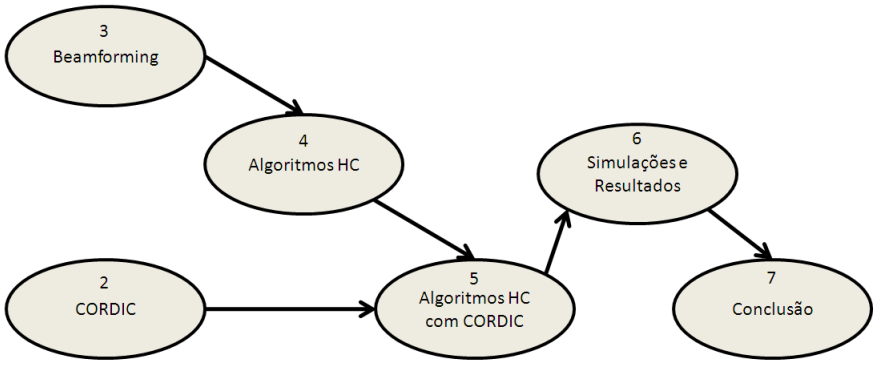


Figura 1.1: Organização da tese.

Conforme mostrado na Figura 1.1, a tese, inicialmente, possui dois desafios, que são: explicar CORDIC e algoritmos HC (*Householder Constraint*). A seguir será explicado como esses assuntos serão abordados em cada um dos capítulos.

No Capítulo 2 , será explicado o funcionamento do algoritmo CORDIC. A implementação de cada função pode ser encontrada com mais detalhes em [3]. Neste capítulo também é proposto uma forma eficiente para resolver às limitações impostas pelo algoritmo CORDIC para as funções aritméticas utilizadas no HCLMS e NHCLMS. No final do capítulo serão realizadas algumas simulações, a fim de verificar eficácia do novo código.

No Capítulo 3, será explicado *beamforming* e os diversos tipo de *beamformers*. Na explicação serão apresentadas as vantagens e desvantagens dos *beamformers*, de modo a mostrar o porquê da escolha do *beamformer* adaptativo para o desenvolvimento da tese.

No Capítulo 4, será explicado o funcionamento dos algoritmos HC (*Householder Constrained*); todavia, previamente, serão abordados os seguintes assuntos: transformada de *Householder*, filtros LC (*Linearly Constrained*) adaptativos e seus diversos tipos de implementações. No fim serão feitos alguns comentários, a fim de reforçar as principais vantagens dos algoritmos HC.

No Capítulo 5, será mostrado como os algoritmos HC com precisão variável foram implementados. Também serão apresentados os pseudocódigos dos algoritmos utilizados na simulação desta tese. Por fim, serão feitos alguns testes a fim de verificar o correto funcionamento das funções CORDIC utilizadas para implementar os algoritmos HC com precisão variável.

No Capítulo 6, serão apresentados os resultados encontrados pela simulação dos algoritmos HC com precisão variável em uma aplicação de antenas inteligentes utilizando um *beamformer* adaptativo.

Já o Capítulo 7 serão apresentados as conclusões do trabalho e serão sugeridos trabalhos futuros para continuação desta tese.

Capítulo 2

CORDIC

2.1 Introdução

Os algoritmos de CORDIC (*COordinate Rotation DIgital Computer*) pertencem à classe de algoritmos iterativos que utilizam apenas deslocamentos e somas para encontrar um resultado.

A grande vantagem de utilizar o CORDIC está na simplicidade do seu *hardware*. Com isso, operações matemáticas mais complexas podem ser realizadas de forma mais eficiente em DSPs (*Digital Signal Processing*), cujas arquiteturas foram projetadas para trabalhar com CORDIC. É importante mencionar que muitos avanços em VLSI (*Very Large Scale Integration*) foram realizados devido à utilização do CORDIC em suas arquiteturas.

O primeiro algoritmo de CORDIC desenvolvido foi implementado por Volder no fim da década de 50. A implementação de Volder utilizava o CORDIC para cálculo da magnitude de um vetor. Os resultados deste desenvolvimento podem ser encontrados em [4].

Após isto, novos trabalhos foram desenvolvidos. O CORDIC passou a ser utilizado para calcular muitas funções transcedentais incluindo funções trigonométricas, exponenciais e aritméticas. Entre os trabalhos mais conhecidos estão os de Walther [5], Meggitt [6] e Daggett [7].

Estes algoritmos trabalham com rotações de vetores realizadas através de operações de deslocamentos e somas. A cada iteração é realizada uma rotação em relação a um determinado eixo, por exemplo, para cálculo da magnitude de um

vetor, deve-se fazer a rotação ao redor do eixo x (eixo das ordenadas). À medida que a quantidade de iterações for aumentando, mais próximo do valor desejado o algoritmo se aproxima, visto que a precisão do algoritmo aumenta em um *bit* por iteração. Vale a pena ressaltar que o ângulo de rotação é reduzido a cada iteração, logo o algoritmo sempre irá convergir para algum valor.

Em relação às dimensões de rotação, é necessário dizer que originalmente o CORDIC foi projetado por Volder para trabalhar com rotações no subespaço euclideano. Quando existe a necessidade de rotacionar um vetor em subespaços cujas dimensões são maiores do que dois, faz-se necessário utilizar um algoritmo que decomponha o vetor original em diversos vetores pertencentes ao subespaço euclideano. Após a rotação ser realizada, é necessário ter um algoritmo de recomposição para o subespaço original. A introdução dos algoritmos de decomposição e recomposição tornam a utilização do CORDIC ineficiente, visto que oneram a velocidade de implementação do algoritmo como um todo. Nesta tese serão utilizadas apenas rotações de vetores bidimensionais.

O subespaço euclidiano é um subconjunto S de $\mathbb{R}^n$ que possui as seguintes propriedades:

1. O vetor $\mathbf{0}$ é um elemento de S.

2. Se dois vetores $\mathbf{u}$ e $\mathbf{v}$ são elementos de S, então a soma desses vetores é um elemento de S.

3. Se o vetor $\mathbf{u}$ for um elemento de S e c for um escalar real, então $c\mathbf{u}$ é um elemento de S.

O estudo do algoritmo CORDIC é de suma importância para a tese, pois as funções CORDIC serão utilizadas para substituir as operações aritméticas dos algoritmos de filtragem adaptativa apresentado no Capítulo 4. A junção desses algoritmos adaptativos com as funções CORDIC está explicado no Capítulo 5. Essa junção faz com que os algoritmos de filtragem adaptativa tenham precisão variável; logo será possível estudar o comportamento desses algoritmos quando utilizados num *hardware* puramente CORDIC.

Inicialmente este capítulo apresentará o algoritmo CORDIC. As variáveis que compoem este algoritmo serão explicadas em seguida. Após a explicação das

variáveis, alguns exemplos do funcionamento do algoritmo CORDIC serão mostrados e por fim serão explicados os algoritmos de adaptação para as funções CORDIC de multiplicação, divisão e raiz quadrada.

2.2 Algoritmo CORDIC

Conforme mencionado anteriormente, o algoritmo CORDIC trabalha com somas e deslocamentos; sendo assim, torna-se necessário apresentar como ocorre a rotação de um vetor $\mathbf{s}_i$ por um ângulo θ_i. Vale a pena ressaltar que o vetor s_i está contido em um espaço bidimensional, ou seja, $\mathbf{s}_i = (x_i, y_i)$.

Matematicamente, esta rotação pode ser expressa por:

$$\begin{cases} x_{i+1} = x_i \cos(\theta_i) - y_i \operatorname{sen}(\theta_i) \\ y_{i+1} = y_i \cos(\theta_i) + x_i \operatorname{sen}(\theta_i) \end{cases} \tag{2.1}$$

O resultado da rotação será representado pelo vetor $\mathbf{s}_{i+1} = (x_{i+1}, y_{i+1})$. Este vetor é idêntico em módulo ao vetor $\mathbf{s}_i$, todavia possui fase diferente. A Figura 2.1 mostra como ocorre a rotação.

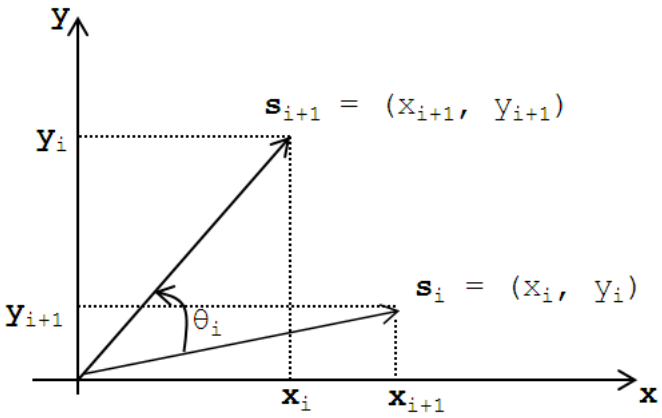


Figura 2.1: Rotacionando o vetor $\mathbf{s}_i$ pelo ângulo θ_i.

Reorganizando a equação (2.1) tem-se:

$$\begin{cases} x_{i+1} = \cos(\theta_i) \cdot [x_i - y_i \operatorname{tg}(\theta_i)] \\ y_{i+1} = \cos(\theta_i) \cdot [y_i + x_i \operatorname{tg}(\theta_i)] \end{cases} \tag{2.2}$$

Na implementação dos algoritmos de CORDIC são sempre feitas rotações de vetores por um ângulo θ_i. À medida que o número de rotações for aumentando, o termo $\cos(\theta_i)$ vai se tornando constante, independente do sentido da rotação. Sendo assim, o termo $\cos(\theta_i)$ poderá ser tratado como uma constante no fim da computação [8].

Caso os ângulos de rotações fossem restringidos a $tg(\theta_i) = \pm 2^{-i}$, então as rotações poderiam ser implementadas utilizando apenas deslocamentos e adições (ou subtrações). Sendo assim, a equação (2.2) pode ser reescrita como:

$$\begin{cases} x_{i+1} = K_i[x_i - y_i \cdot d_i \cdot 2^{-i}] \\ y_{i+1} = K_i[y_i + x_i \cdot d_i \cdot 2^{-i}] \end{cases} \tag{2.3}$$

onde:

$$K_i = \cos(tg^{-1}(2^{-i})) = \frac{1}{\sqrt{1 + 2^{-2i}}} \tag{2.4}$$

$$d_i = \pm 1 \tag{2.5}$$

As variáveis K_i e d_i são conhecidas como fator de correção de escala e função decisão, respectivamente. Estas variáveis serão apresentadas com mais detalhes no decorrer deste capítulo.

O algoritmo de CORDIC faz uso de uma terceira variável responsável por acumular os ângulos rotacionados. Abaixo segue a descrição desta nova variável.

$$z_{i+1} = z_i - d_i \cdot tg^{-1}(2^{-i}) \tag{2.6}$$

Fazendo a junção de (2.3) e (2.6), tem-se:

$$\begin{cases} x_{i+1} = K_i[x_i - y_i \cdot d_i \cdot 2^{-i}] \\ y_{i+1} = K_i[y_i + x_i \cdot d_i \cdot 2^{-i}] \\ z_{i+1} = z_i - d_i \cdot tg^{-1}(2^{-i}) \end{cases} \tag{2.7}$$

A fim de facilitar a compreensão do algoritmo de CORDIC, a equação (2.7) será reescrita de forma mais genérica. Isto torna-se necessário, pois há algumas peculiaridades que não são levadas em consideração por esta equação.

$$\begin{cases} x_{i+1} = K_{m,i}[x_i - m \cdot d_i \cdot y_i \cdot \gamma_{m,i}] \\ y_{i+1} = K_{m,i}[y_i + d_i \cdot x_i \cdot \gamma_{m,i}] \\ z_{i+1} = z_i - d_i \cdot \theta_{m,i} \end{cases} \tag{2.8}$$

A equação (2.8) será responsável por realizar dois papéis: transformação linear do vetor $\mathbf{s}_i$ para $\mathbf{s}_{i+1}$ e armazenamento do somatório dos ângulos de rotação, a cada iteração. Nesta equação foram introduzidas novas variáveis, a fim de modelar de forma mais genérica o funcionamento do algoritmo de CORDIC.

A variável m determina qual sistema de coordenadas será utilizado. Ela pode assumir três valores que são -1 (modo hiperbólico), 0 (modo linear) e 1 (modo circular).

A variável $\gamma_{m,i}$ está substituindo o valor 2^{-i} da equação (2.7). Esta variável é definida como $\gamma_{m,i} = b^{-u_{m,i}}$, onde b é a base do sistema numérico e $u_{m,i}$ é a seqüência de deslocamento de números inteiros não decrescente. Vale a pena ressaltar que os valores atribuídos a $u_{m,i}$ são dependentes do sistema de coordenadas utilizado e a base numérica utilizada na tese é a binária, ou seja, $\gamma_{m,i} = 2^{-u_{m,i}}$.

Como a equação (2.7) não está parametrizada para trabalhar com os três sistemas de coordenadas, foi introduzida em (2.8) a variável $\theta_{m,i}$ para substituir o valor $tg^{-1}(2^{-i})$. Sua função é determinar o ângulo de rotação referente à iteração e sistema de coordenadas desejado. A definição desta variável é dada por:

$$\theta_{m,i} = \frac{1}{\sqrt{m}} \cdot tg^{-1}(\sqrt{m} \cdot \gamma_{m,i}) \qquad \text{sendo i} \in \{1, \ldots, \text{N}\} \tag{2.9}$$

É importante mencionar que a constante N é a quantidade máxima de iterações que será realizada pelo algoritmo.

A variável $K_{m,i}$ está substituindo K_i da equação (2.7). Esta alteração deve ser realizada, pois o fator de correção de escala é uma variável que depende do sistema de coordenadas utilizado. A equação (2.10) define o novo valor a esta variável.

$$K_{m,i} = \frac{1}{\sqrt{1 + m\gamma_{m,i}^2}} \tag{2.10}$$

A seguir serão apresentados de forma mais detalhada todas as variáveis que fazem parte do algoritmo de CORDIC mostrado na equação (2.8).

2.3 Sistema de coordenadas (m)

Conforme citado anteriormente, o sistema de coordenadas pode ser dividido em três modos: hiperbólico, linear ou circular, sendo o valor de m, respectivamente, igual a -1, 0 e +1. A fim de entender o motivo da associação dos valores de m aos três modos mencionados, veja a equação (2.11).

$$\begin{cases} R_{m,i} = \sqrt{x_i^2 + m \cdot y_i^2} \\ \theta_{m,i} = \left(\frac{1}{\sqrt{m}}\right) tg^{-1}\left(\frac{y_i\sqrt{m}}{x_i}\right) \end{cases} \tag{2.11}$$

A equação (2.11) descreve as coordenadas polares de um vetor $\mathbf{t}_i$, sendo $R_{m,i}$ e $\theta_{m,i}$ são, respectivamente, o módulo e a fase deste vetor. Vale a pena ressaltar que o vetor $\mathbf{t}_i$ é formado pelos pontos $T_i(x_i, y_i)$ e a origem, $O(x, y)$. A variável $R_{m,i}$ também pode ser definida como a norma do vetor $\mathbf{t}_i$. A Figura 2.2 ilustra esta situação.

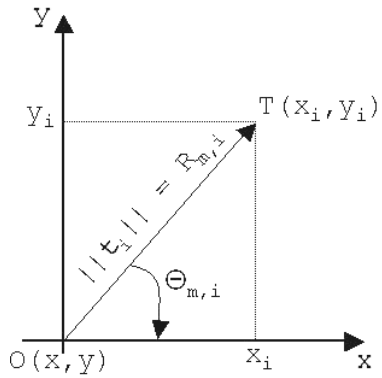


Figura 2.2: Coordenadas polares do vetor $\mathbf{t}_i$.

É importante notar que ao substituir $\frac{y_i}{x_i}$ por $\gamma_{m,i}$ em (2.11) (equação referente a $\theta_{m,i}$), encontra-se a equação (2.9). Esta substituição é extremamente necessária visto que o algoritmo de CORDIC utiliza apenas somas e deslocamentos.

O vetor $\mathbf{t}_i$ é responsável por realizar a transformação linear do vetor $\mathbf{s}_i$ para $\mathbf{s}_{i+1}$ (vide Figura 2.1). O módulo do vetor $\mathbf{t}_i$, ou seja, $R_{m,i}$ será responsável por alterar o módulo do vetor $\mathbf{s}_i$, já a variável $\theta_{m,i}$ alterará a fase do mesmo vetor, originando o vetor $\mathbf{s}_{i+1}$. A equação (2.12) explica como é realizada essa transformação linear.

$$\begin{cases} |\mathbf{s}_{i+1}| = |\mathbf{s}_i| \cdot R_{m,i} \\ \angle\mathbf{s}_{i+1} = \angle\mathbf{s}_i + (d_i \cdot \theta_{m,i}) \end{cases} \tag{2.12}$$

Quando $m = -1$, a trajetória de rotação é uma hipérbole, visto que a norma $R_{-1,i}$ é igual a $\sqrt{x_i^2 - y_i^2}$. Sendo assim a função hiperbólica, que será responsável pela trajetória de rotação, é dada por $f_{-1}(x, y) = x^2 - y^2 = 1$, já os ângulos de rotação $\theta_{-1,i} = tgh^{-1}(\frac{y_i}{x_i})$. A Figura 2.3 ilustra o comportamento da transformação utilizando a função hiperbólica $f_{-1}(x, y)$.

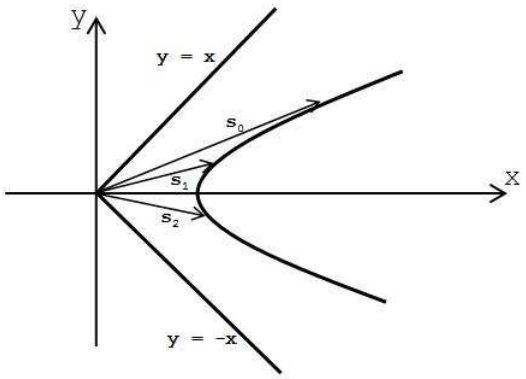


Figura 2.3: Sistema de coordenadas hiperbólico ($m = -1$).

Para $m = 0$, a trajetória de rotação é uma reta, visto que a norma $R_{0,i}$ é igual a x_i. Sendo assim a função linear que será responsável pela trajetória de rotação é dada por $f_0(x) = x = 1$, já os ângulos de rotação $\theta_{0,i} = \frac{y_i}{x_i}$. A Figura 2.4 ilustra o comportamento da transformação utilizando a função linear $f_0(x, y)$.

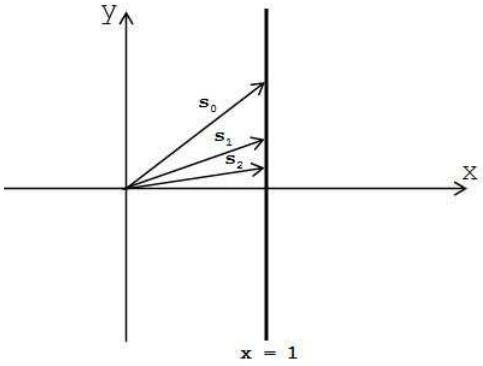


Figura 2.4: Sistema de coordenadas linear ($m = 0$).

Quando $m = +1$, a trajetória de rotação é uma circunferência, visto que a norma $R_{+1,i}$ é igual a $\sqrt{x_i^2 + y_i^2}$. Sendo assim, a função circular, que será responsável pela trajetória de rotação, é dada por $f_{+1}(x, y) = x^2 + y^2 = 1$; já os ângulos de rotação serão dados por $\theta_{+1,i} = tg^{-1}(\frac{y_i}{x_i})$. A Figura 2.5 ilustra o comportamento da transformação utilizando a função circular $f_{+1}(x, y)$.

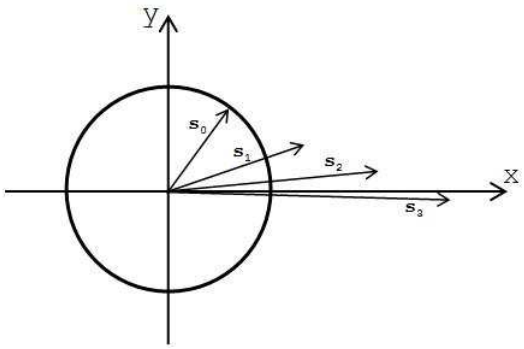


Figura 2.5: Sistema de coordenadas circular ($m = +1$).

Analisando as Figuras 2.3 e 2.5 pode-se notar que os novos vetores originados através das transformações possuem comportamento não linear ao longo das iterações. Estas não uniformidades estão associadas à realização de rotações imperfeitas efetuada pelo algoritmo. Desta forma, faz-se necessária a utilização de uma variável que corrija estas imperfeições. A variável que desempenha este papel é o fator de correção de escala, $K_{m,i}$ (vide a equação (2.8) e (2.10)). No decorrer do capítulo esta variável será estudada com mais detalhes.

2.4 Função decisão (d_i)

A função decisão determina a direção de rotação do algoritmo, ou seja, ela é responsável por definir se o sentido de rotação é horário ou anti-horário. Esta função pode assumir dois valores: -1 (rotação no sentido horário) e $+1$ (rotação no sentido anti-horário).

Esta função pode trabalhar em dois modos de operação: rotação ou vetorização. Utilizando estes dois modos, é possível realizar os mais diversos cálculos matemáticos. Estes modos serão detalhados a seguir.

2.4.1 Modo de rotação

Também conhecido como *Z-Reduction*, este modo de operação distingue-se por reduzir a variável z_i (vide equação(2.8)) a zero, ou seja, o valor de $z_N \rightarrow 0$.

Vale a pena lembrar que a constante N está associada à quantidade máxima de iterações que será realizada pelo algoritmo. Sendo assim z_N pode ser definido como a diferença entre o ângulo inicial, z_0, e o somatório dos ângulos $\theta_{m,i}$ das N iterações. A equação (2.13) representa a explicação acima.

$$z_N = z_0 - \sum_{i=1}^{N} d_i \cdot \theta_{m,i} \tag{2.13}$$

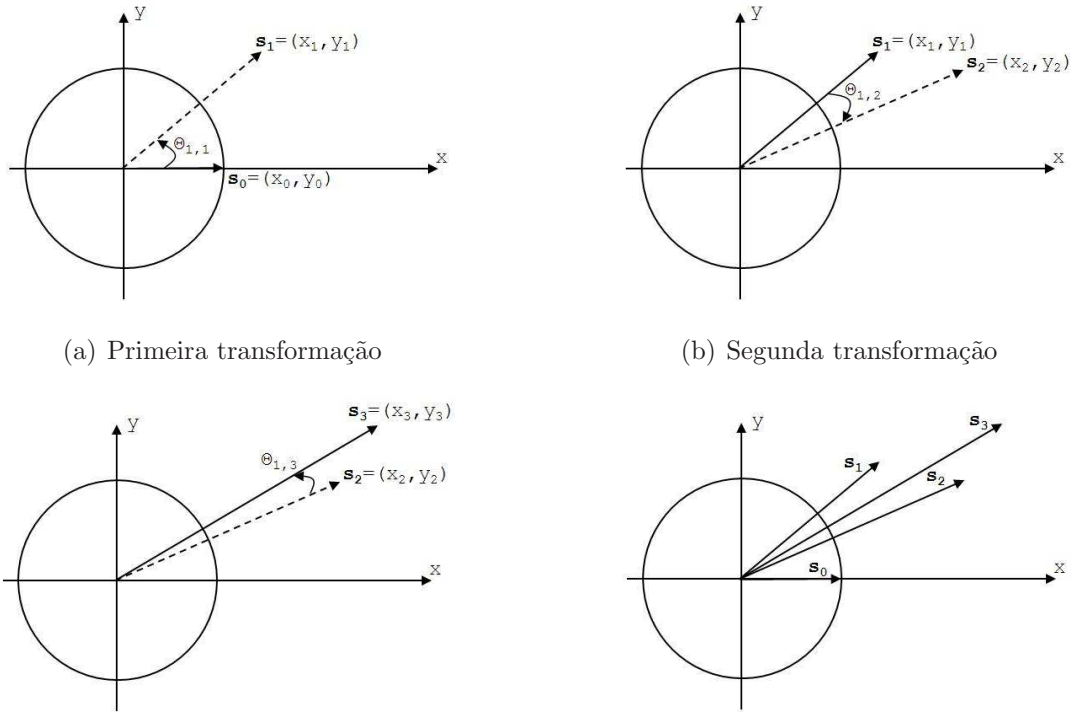
Para que ocorra a convergência do algoritmo, a função decisão, d_i, deve ser implementada de modo a fazer $z_N \rightarrow 0$, logo $z_0 = \sum_{i=1}^{N} d_i \cdot \theta_{m,i}$.

A função decisão, quando utilizada com o modo de rotação, funciona da seguinte forma: sempre que o valor de z_i for maior ou igual a zero, a função decisão será igual a $+1$. Desta forma o vetor s_i será rotacionado de $\theta_{m,i}$ graus no sentido anti-horário para originar o vetor s_{i+1}. Quando o valor de z_i for inferior a zero, a função decisão será igual a -1. Sendo assim o vetor s_i sofrerá uma rotação de $\theta_{m,i}$ graus no sentido horário.

A equação (2.14) resume o funcionamento da função decisão no modo rotação.

$$\begin{cases} se\ z_i \geq 0 \rightarrow d_i = +1 \\ se\ z_i < 0 \rightarrow d_i = -1 \end{cases} \tag{2.14}$$

A seguir será apresentado uma figura que mostra o comportamento do algoritmo de CORDIC quando o modo de rotação está sendo utilizado.



(a) Primeira transformação

(b) Segunda transformação

(c) Terceira transformação

(d) Trajetória completa da transformação

Figura 2.6: Trajetória de transformação utilizando o modo de rotação com o sistema de coordenadas circular.

A partir da Figura 2.6, pode-se verificar que o ângulos de rotação são reduzidos à medida que o número de iterações aumenta. A Tabela 2.1 mostra os valores utilizados em cada iteração para o sistema de coordenadas circular.

Tabela 2.1: Valores para ângulos de rotação para sistema de coordenadas circular.

Nº rotação	**tg($\theta_{m,i}$)**	$\theta_{m,i}$
1	tg $(\theta_{1,1}) = 2^0 = \frac{1}{1}$	$\theta_{1,1} = 45°$
2	tg $(\theta_{1,2}) = 2^{-1} = \frac{1}{2}$	$\theta_{1,2} = 26,5650°$
3	tg $(\theta_{1,3}) = 2^{-2} = \frac{1}{4}$	$\theta_{1,3} = 14,0362°$
4	tg $(\theta_{1,4}) = 2^{-3} = \frac{1}{8}$	$\theta_{1,4} = 7,1250°$
5	tg $(\theta_{1,5}) = 2^{-4} = \frac{1}{16}$	$\theta_{1,5} = 3,5763°$
6	tg $(\theta_{1,6}) = 2^{-5} = \frac{1}{32}$	$\theta_{1,6} = 1,7899°$
7	tg $(\theta_{1,7}) = 2^{-6} = \frac{1}{64}$	$\theta_{1,7} = 0,8638°$
$\vdots$	$\vdots$	$\vdots$

Quando a quantidade de iterações for muito grande, o ângulo de rotação será tão pequeno que sua utilização será desprezível. Desta forma é importante fazer uma escolha otimizada da quantidade de iterações que será utilizada nos algoritmos de CORDIC.

Antes de qualquer análise, é importante ressaltar que, para o exemplo ilustrado pela Figura 2.6, o valor inicial de z_0 é igual a 32° .

A primeira transformação está representada na Figura 2.6(a). Através de deslocamentos e somas (ou substrações) o vetor $\mathbf{s}_1$ é encontrado a partir de $\mathbf{s}_0$. Pode-se perceber que o vetor s_1 está com o seu módulo expandido, devido as não uniformidades do algoritmo, e sua fase está maior $\theta_{1,1}$ graus. De acordo com a Tabela 2.1, verifica-se que o deslocamento foi de 45° no sentido anti-horário ($d_1 = +1$), já que $z_0 = 32° \geq 0°$.

Após a primeira iteração o novo valor para z_1 é $-13°$. Analisando a Figura 2.6(b) percebe-se que a rotação foi sentido horário ($d_2 = -1$), pois $z_1 < 0$. A rotação realizada foi de 26,5° de acordo com a Tabela 2.1. Sendo assim, no fim desta iteração z_2 vale 13,5°.

Como o valor de z_2 é positivo na terceira iteração, a rotação realizada será de 14° no sentido anti-horário ($d_3 = +1$). A Figura 2.6(c) ilustra a expansão e rotação do vetor s_2 para s_3. No fim da iteração, z_3 terá um valor bem próximo a zero, devido a isto não será mostrado mais nenhum iteração.

Vale a pena ressaltar que caso houvesse mais uma iteração, com certeza, o valor de z_4 seria maior que z_3. Isto não quer dizer que sempre irão bastar três iterações para que o resultado desejado seja atingido. Na situação ilustrada pela Figura 2.6, a solução mais eficiente é atingida com três iterações, todavia bastaria uma alteração no valor z_0 para que isto deixasse de ser verdade. A Tabela 2.2 aborda esse assunto.

Tabela 2.2: Exemplo das iterações CORDIC para cálculo do cosseno de 32°.

Nº iteração (i)	z_i	Resultado
1	$-13°$	0,7071
2	$+13,5°$	0,9487
3	$-0,46°$	0,8437
4	$+6,65°$	0,9037
5	$+3,07°$	0,8753
6	$+1,28°$	0,8597
7	$+0,39°$	0,8517
8	$-0,05°$	0,8475
⋮	⋮	⋮

Sabendo que o valor encontrado pela função cos do *Matlab* é de 0,8480, nota-se que o valor encontrado na terceira iteração está muito próximo do valor ideal. Isto ocorre, pois quanto mais próximo de zero estiver a variável z_i, mais próximo do resultado correto estará o algoritmo. À medida que o número de iterações aumenta, cada vez mais a variável z_i tende a zero.

2.4.2 Modo de vetorização

Também conhecido como *Y-Reduction*, este modo de vetorização distingue-se por rotacionar o vetor s_i ao redor do eixo das ordenadas (eixo x). O objetivo deste modo de operação é fazer $y_N \rightarrow 0$, logo a convergência do algoritmo está associada à redução da variável y_i.

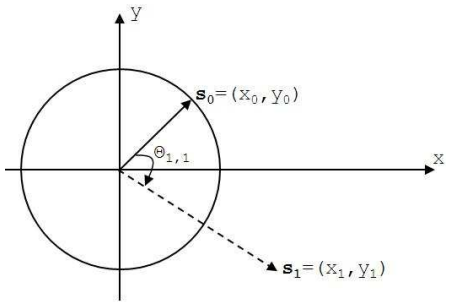
A função decisão quando utilizada com o modo de vetorização funciona da seguinte forma: sempre que o valor de x_i e y_i forem maiores ou iguais a zero, ou, x_i e y_i forem inferiores a zero, a função decisão será igual a -1. Desta forma o vetor s_i será rotacionado de $\theta_{m,i}$ graus no sentido horário para originar o vetor s_{i+1}. Quando o valor de x_i for inferior a zero e y_i for superior a zero, ou vice-versa, a função decisão será igual a $+1$. Sendo assim, o vetor s_i sofrerá uma rotação de $\theta_{m,i}$ graus no sentido anti-horário.

A equação (2.16) resume o funcionamento da função decisão no modo vetorização.

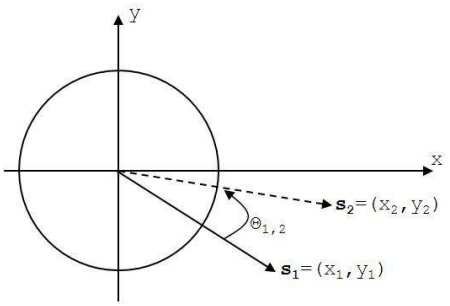
$$\begin{cases} se\ x_i \geq 0\ e\ y_i \geq 0 \rightarrow d_i = -1 \\ se\ x_i < 0\ e\ y_i < 0 \rightarrow d_i = -1 \\ se\ x_i \geq 0\ e\ y_i < 0 \rightarrow d_i = +1 \\ se\ x_i < 0\ e\ y_i \geq 0 \rightarrow d_i = +1 \end{cases} \tag{2.15}$$

Apesar da redução ser realizada com y_i, a variável x_i deve ser levada em consideração para determinação da função decisão, já que o algoritmo possui comportamentos distintos quando esta variável altera de sinal.

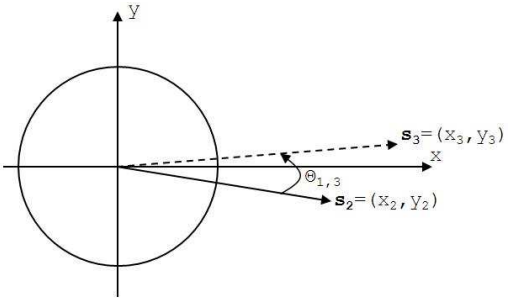
As Figuras 2.7 e 2.8 exemplificam o funcionamento da função decisão quando o modo de vetorização é utilizado em conjunto com o sistema de coordenadas circular e linear, respectivamente.



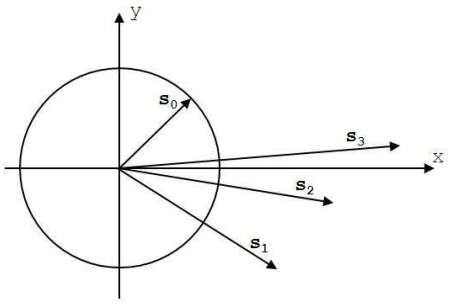
(a) Primeira transformação



(b) Segunda transformação

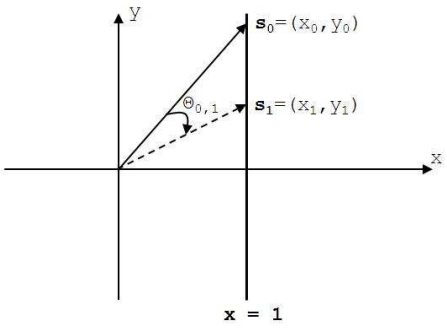


(c) Terceira transformação

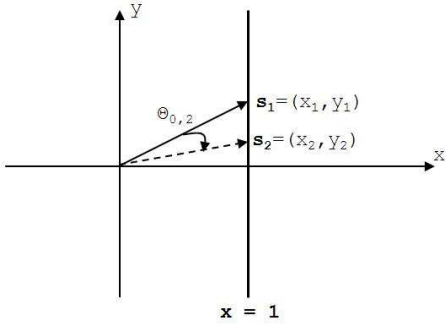


(d) Trajetória completa da transformação

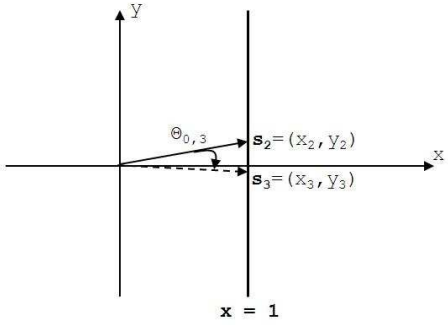
Figura 2.7: Trajetória de transformação utilizando o modo de vetorização com o sistema de coordenadas circular.



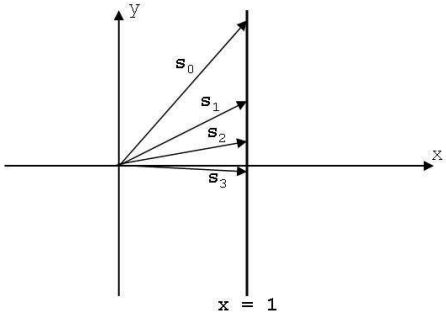
(a) Primeira transformação



(b) Segunda transformação



(c) Terceira transformação



(d) Trajetória completa da transformação

Figura 2.8: Trajetória de transformação utilizando o modo de vetorização com o sistema de coordenadas circular.

2.5 Seqüência de deslocamento ($u_{m,i}$)

A seqüência de deslocamento, $u_{m,i}$, tem o seu valor alterado de acordo com o sistema de coordenadas utilizado. Isto ocorre, pois são necessárias diferentes seqüências por sistemas coordenadas para fazer o algoritmo convergir.

De acordo com a equação (2.8) e (2.9), percebe-se que o valor escolhido para a seqüência de deslocamento está associado diretamente com o ângulo $\theta_{m,i}$. Vale a pena lembrar que este ângulo é responsável pela rotação que será realizada no vetor s_i.

Sendo assim, pode-se concluir que a função decisão não é a única variável responsável por controlar a convergência do algoritmo. É necessário que haja uma seqüência de valores distintos, por sistemas de coordenadas, que determinem os ângulos de rotação responsáveis pela convergência do algoritmo.

A fim de determinar as seqüência de deslocamento, é necessário definir os

critérios de convergência para que $u_{m,i}$ possa ser determinado. De acordo com [4] e [5] dois critérios devem ser seguidos:

$$\begin{cases} \theta_{m,i} - \sum_{j=i+1}^{n-1} \theta_{m,j} \leq \theta_{m,n-1} \\ |D_0| \leq \sum_{i=0}^{N-1} \theta_{m,i} \end{cases} \tag{2.16}$$

onde D_0 é o ângulo desejado para rotação, ou seja, z_0 para o modo de rotação e z_N para o modo de vetorização.

Em relação aos sistemas de coordenadas linear e circular, uma escolha trivial para $u_{m,i}$ é o conjunto de números naturais $\mathbb{N}$, já que a seqüência é inteira e não decrescente e $\gamma_{m,i}$ sempre converjirá.

Já em relação ao sistema de coordenadas hiperbólico, é necessário que alguns valores da seqüência anterior sejam repetidos, a fim da convergência ser atingida. Esta solução foi dada por [5] e consiste em repetir os valores da seqüência anterior, cujos valores são iguais a $(3k+1)$, onde $k \epsilon \mathbb{N}$.

A tabela a seguir apresenta as seqüências de deslocamento, utilizadas nesta tese, para cada um dos sistemas de coordenadas.

Tabela 2.3: Definição das seqüências de deslocamento do CORDIC.

Sistemas de Coordenadas (m)	**Seqüência de Deslocamento** ($u_{m,i}$)	**Região de Convergência** ($\lvert D_{0,m} \rvert$)	**Fator de Correção de Escala Total** ($K_m = \prod_{i=1}^{N} K_{m,i}$)
1	$0, 1, 2, 3, 4, \cdots, N-1$	$\sim 1,74$	$\sim 1,64676$
0	$0, 1, 2, 3, 4, \cdots, N-1$	2,0	1,0
-1	$1, 2, 3, 4, 4, 5, 6, 7, 7, \cdots$	$\sim 1,13$	$\sim 0,82816$

A constante $D_{0,m}$ representa o maior ângulo de rotação para um determinado sistema de coordenadas. Seus valores estão diretamente associado a seqüência de deslocamento usada.

2.6 Ângulo de rotação ($\theta_{m,i}$)

Esta variável foi definida em (2.9); entretanto, um estudo mais detalhado sobre a mesma torna-se necessário, visto que algumas particularidade não podem ser vislumbradas a partir da equação (2.9).

Trabalhando com o sistema de coordenadas linear, verifica-se que ao substituir a variável m por zero, tem-se um valor indeterminado para $\theta_{0,i}$. Aplicando L'Hôpital nesta inderminação, tem-se que o valor de $\theta_{0,i}$ para o sistema de coordenadas linear vale $\gamma_{0,i}$, ou seja, $2^{u_{0,i}}$.

A equação (2.17) define a variável $\theta_{m,i}$ para cada um dos sistema de coordenadas.

$$\begin{cases} \theta_{-1,i} = tgh^{-1}(\gamma_{-1,i}) \\ \theta_{0,i} = \gamma_{0,i} \\ \theta_{+1,i} = tg^{-1}(\gamma_{+1,i}) \end{cases} \tag{2.17}$$

2.7 Fator de correção de escala ($K_{m,i}$)

Assim como o ângulo de rotação, a variável $K_{m,i}$, fator de correção de escala, já havia sido definida na equação (2.10). Esta variável é sempre utilizada nos sistemas de coordenadas hiperbólico ou circular, já que estes introduzem não linearidades às saídas, x_i e y_i, do algoritmo de CORDIC. Estas não uniformidades podem ser verificadas nas Figuras 2.6 e 2.7.

Introduzindo as expansões não uniformes (designadas pela variável $L_{m,i}$) a equação (2.8), tem-se:

$$\begin{cases} x_{i+1} = K_{m,i} \cdot L_{m,i}[x_i - m \cdot d_i \cdot y_i \cdot \gamma_{m,i}] \\ y_{i+1} = K_{m,i} \cdot L_{m,i}[y_i + d_i \cdot x_i \cdot \gamma_{m,i}] \\ z_{i+1} = z_i - d_i \cdot \theta_{m,i} \end{cases} \tag{2.18}$$

É importante frisar que a variável $L_{m,i}$ é sempre introduzida quando os sistemas de coordenadas são hiperbólico ou circular. Devido a isto, essa variável não foi definida em (2.8), já que essa não uniformidade está implícita ao sistema de

coordenadas utilizado. Sendo assim, o fator de correção de escala deve corrigir os erros provenientes das não uniformidades, logo $K_{m,i} = \frac{1}{L_{m,i}}$.

Existem diversos métodos para aplicar o fator de correção de escala, dentre os mais conhecidos estão: multiplicação constante do fator [9], multiplicação de passos de normalização [10] e multiplicação de iterações repetidas e compensadas [11]. Nesta tese, todas as simulações basearam-se no método de multiplicação constante do fator.

2.8 Acurácia do algoritmo

A precisão do algoritmo de CORDIC está associada à quantidade de iterações realizadas até o resultado final. A cada iteração efetuada, a acurácia do algoritmo está sendo aumentada em 1 *bit*, ou seja, quanto mais iterações forem feitas, maiores são as chances de convergir para o valor ideal, vide Tabela 2.2.

Sendo assim, conclui-se que a acurácia é limitada em magnitude pelo ângulo de rotação da última iteração, $\theta_{m,N}$. Uma boa forma de medir a acurácia é verificar se o valor de z_i, para o modo de rotação, ou y_i, para o modo de vetorização, estão próximos de zero.

A acurácia do algoritmo de CORDIC foi fator motivador para a sua empregabilidade nesta tese, pois ela aumenta à medida que o número de iterações cresce. Com isso, é possível controlar a acurácia do algoritmo variando apenas a quantidade de iterações. Sendo assim, o próprio usuário tem a possibilidade de escolher a precisão que deseja para qualquer tipo de cálculo.

Outro fator, que influenciará a acurácia do algoritmo, é a precisão finita das variáveis envolvidas, seja numa implementação de ponto flutuante ou ponto fixo. Quando trabalha-se com o algoritmo CORDIC, o mais adequada é utilizar a representação de ponto fixo. Esta tese faz uso deste tipo de representação.

O formato das variáveis utilizadas no algoritmo CORDIC está representado pela Figura 2.9.

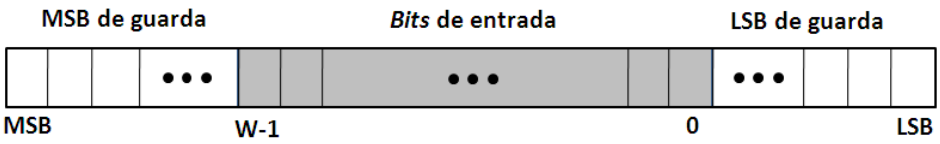


Figura 2.9: Formato das variáveis internas do CORDIC.

O tamanho das variáveis internas do algoritmo CORDIC é composto por W *bits*, adicionado por G *bits* de guarda para o *bit* mais significativo (MSB - *Most Significant Bit*) e C *bits* de guarda para o *bit* menos significativo (LSB - *Least Significant Bit*). Os *bits* de guarda mais significativos são necessários devido às expansões não uniformes geradas pelos sistemas de coordenadas circular e hiperbólico. A fim de exemplificar o que foi dito anteriormente, de acordo com [12], para o sistema de coordenadas circular são necessários dois *bits* de guarda para o lado MSB.

Em implementações CORDIC, é mais interessante utilizar o arredondamento em vez do truncamento, pois o erro introduzido pelo arredondamento é, praticamente, a metade do erro introduzido pelo truncamento. Os *bits* de guarda menos significativos são responsáveis pela acurácia da saída do algoritmo CORDIC, logo influenciam nos erros de arredondamento. Uma escolha errada para o valor de C, fará com que o erro de arredondamento seja muito alto; todavia, uma escolha correta para o mesmo, faz com que esse erro seja mínimo, ou seja, pouco impactante para a implementação CORDIC.

As variáveis internas, utilizadas no algoritmo CORDIC desta tese, foram implementadas utilizando ponto fixo com tratamento de *overflow* e arredondamento. Os valores G e C dos bits de guarda foram escolhidos de modo que, os erros introduzidos devido ao *overflow* e arredodamento fossem minimizados.

2.9 Mapeamento das funções CORDIC

Para utilizar o algoritmo CORDIC no cálculo de funções trigonométricas, exponenciais ou aritméticas, é necessário conhecer o sistema de coordenadas (hiperbólico, linear ou circular) e o modo de operação (*Z-Reduction* ou *Y-Reduction*) relacionados a cada uma das funções. Com esses dois graus de liberdade é possível

calcular diferentes funções alterando apenas o vetor de entrada ($x_0,\ y_0,\ z_0$) do algoritmo CORDIC.

A seguir, será apresentada a Tabela 2.4 que explica como é possível implementar as mais diversas funções utilizando o algoritmo de CORDIC.

Tabela 2.4: Mapeamento das funções CORDIC.

m	**Modo de Operação**	**Entrada**	**Saída**	**Funções**
1	rotação	$x_0 = x$	$x_N = K_1 \cdot (x\cos\phi - ysen\phi)$	tg
		$y_0 = y$	$y_N = K_1 \cdot (y\cos\phi + xsen\phi)$	sen
		$z_0 = \phi$	$z_N = 0$	cos
		$x_0 = 1$	$x_N = \cos\phi$	csc
		$y_0 = 0$	$y_N = sen\phi$	sec
		$z_0 = \phi$	$z_N = 0$	
1	vetorização	$x_0 = x$	$x_N = K_1 \cdot \sqrt{x^2 + y^2}$	
		$y_0 = y$	$y_N = 0$	atg
		$z_0 = \phi$	$z_N = \phi + tg^{-1}(\frac{y}{x})$	
0	rotação	$x_0 = x$	$x_N = x$	soma
		$y_0 = y$	$y_N = y + x \cdot z$	multiplicação
		$z_0 = z$	$z_N = 0$	
0	vetorização	$x_0 = x$	$x_N = x$	
		$y_0 = y$	$y_N = 0$	divisão
		$z_0 = z$	$z_N = z + \frac{y}{x}$	
-1	rotação	$x_0 = x$	$x_n = K_{-1} \cdot (x\cosh\phi + ysenh\phi)$	
		$y_0 = y$	$x_n = K_{-1} \cdot (y\cosh\phi + xsenh\phi)$	tgh
		$z_0 = \phi$	$z_n = 0$	senh
		$x_0 = 1$	$x_n = \cosh\phi$	cosh
		$y_0 = 0$	$y_n = senh\phi$	exp
		$z_0 = \phi$	$z_n = 0$	
-1	vetorização	$x_0 = x$	$x_N = K_{-1} \cdot \sqrt{x^2 - y^2}$	atgh
		$y_0 = y$	$y_N = 0$	sqrt
		$z_0 = \phi$	$z_N = \phi + tgh^{-1}(\frac{y}{x})$	ln

2.10 Exemplos

Nesta seção serão mostrados dois exemplos de cálculo de funções utilizando os algoritmos de CORDIC.

O primeiro exemplo será o cálculo da função cos para $\phi = 56°$. Analisando a Tabela 2.4, verifica-se que a função cos pode ser obtida utilizando o sistema de coordenadas circular, modo de rotação com o vetor de entrada igual a ($x_0 = 1,\ y_0 = 0,\ z_0 = \phi$). A tabela abaixo apresenta a seqüência de rotações realizadas pelo algoritmo para 6 *bits* de precisão, ou seja, serão realizadas 6 iterações.

Tabela 2.5: Exemplo das iterações CORDIC para o cálculo do cosseno de 56°.

Iteração	**Entrada**	**Saída**	**Parâmetros CORDIC**
1	$x_0 = 1$ $y_0 = 0$ $z_0 = +56°$	$x_1 = 0,7071$ $y_1 = 0,7071$ $z_1 = +11°$	$\theta_{1,1} = 45°$ $K_{1,1} = 1,4142$ $d_1 = +1$
2	$x_1 = 0,7071$ $y_1 = 0,7071$ $z_1 = +11°$	$x_2 = 0,3162$ $y_2 = 0,9487$ $z_2 = -15,5°$	$\theta_{1,2} = 26,5°$ $K_{1,2} = 1,1180$ $d_2 = +1$
3	$x_2 = 0,3162$ $y_2 = 0,9487$ $z_2 = -15,5°$	$x_3 = 0,5369$ $y_3 = 0,8437$ $z_3 = -1,52°$	$\theta_{1,3} = 14°$ $K_{1,3} = 1,0308$ $d_3 = -1$
4	$x_3 = 0,5369$ $y_3 = 0,8437$ $z_3 = -1,52°$	$x_4 = 0,6374$ $y_4 = 0,7706$ $z_4 = +5,59°$	$\theta_{1,4} = 7°$ $K_{1,4} = 1,0078$ $d_4 = -1$
5	$x_4 = 0,6374$ $y_4 = 0,7706$ $z_4 = +5,59°$	$x_5 = 0,5881$ $y_5 = 0,8088$ $z_5 = +2,02°$	$\theta_{1,5} = 3,5°$ $K_{1,5} = 1,0020$ $d_5 = +1$
6	$x_5 = 0,5881$ $y_5 = 0,8088$ $z_5 = +2,02°$	$x_6 = \mathbf{0,5625}$ $y_6 = 0,8268$ $z_6 = +0,23°$	$\theta_{1,6} = 1,8°$ $K_{1,6} = 1,0005$ $d_6 = +1$

A Figura 2.10 mostra a trajetória de rotações do cosseno de 56°. Nesta figura não há não uniformidades, pois o fator de potência já corrigiu essas expansões não desejadas. Como está sendo utilizado o modo *Z-Reduction*, espera-se que o valor de $z_N \rightarrow 0$. Isto pode ser verificado com $z_6 = +0,23$, já que para o exemplo $N = 6$.

De acordo com a Tabela 2.4, espera-se que o valor do cos(56°) esteja armazenado na variável x_6. Comparando o valor calculado (V_C) pelo algoritmo, x_6, com o valor esperado (V_E), verifica-se que o erro relativo, E_R, é menor que 1%. Segue abaixo a definição de erro relativo.

$$E_R = \frac{\mid V_C - V_E \mid}{V_E} \tag{2.19}$$

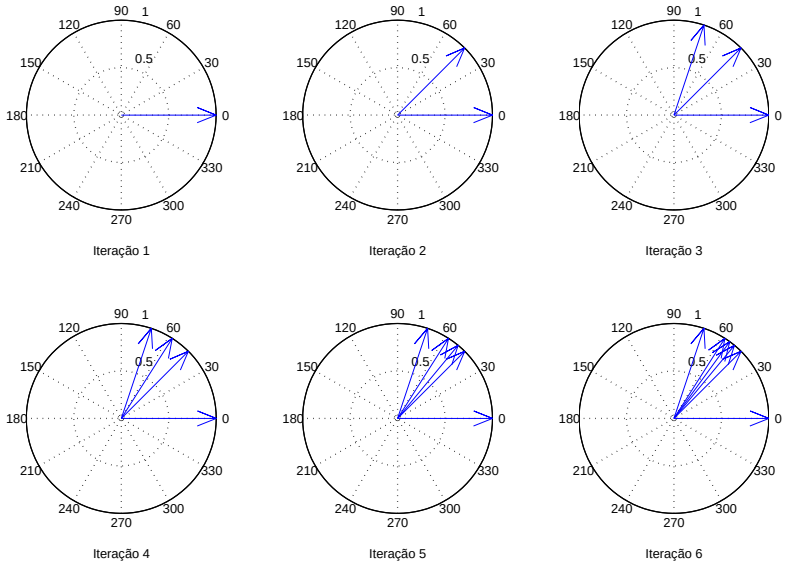


Figura 2.10: Trajetória de rotações para cálculo do cosseno de 56°.

O segundo exemplo será o cálculo da função aritmética de divisão quando o numerador e o denominador valem, respectivamente, $0,5$ e $1,5$. Analisando a Tabela 2.4, verifica-se que a função e divisão pode ser obtida utilizando o sistema de coordenadas linear, modo de vetorização com o vetor de entrada, para o exemplo, igual a $(x_0 = 0,5,\ y_0 = 1,5,\ z_0 = 0)$. A tabela abaixo apresenta a seqüência de rotações realizadas pelo algoritmo para 6 *bits* de precisão.

Tabela 2.6: Exemplo das iterações CORDIC para o cálculo da divisão $\frac{0,5}{1,5}$

Iteração	**Entrada**	**Saída**	**Parâmetros CORDIC**
1	$x_0 = +1,5000$ $y_0 = +0,5000$ $z_0 = 0° = 0\ rad$	$x_1 = +1,5000$ $y_1 = -1,0000$ $z_1 = +57,29° = 1,0000\ rad$	$\theta_{1,1} = 57,29°$ $K_{1,1} = 1$ $d_1 = -1$
2	$x_1 = +1,5000$ $y_1 = -1,0000$ $z_1 = +57,29° = 1,0000\ rad$	$x_2 = +1,5000$ $y_2 = -0,2500$ $z_2 = +28,64° = 0,5000\ rad$	$\theta_{1,2} = 28,6°$ $K_{1,2} = 1$ $d_2 = +1$
3	$x_2 = +1,5000$ $y_2 = -0,2500$ $z_2 == +28,64° = 0,5000\ rad$	$x_3 = +1,5000$ $y_3 = +0,1250$ $z_3 = +14,32° = 0,2500\ rad$	$\theta_{1,3} = 14,3°$ $K_{1,3} = 1$ $d_3 = +1$
4	$x_3 = +1,5000$ $y_3 = +0,1250$ $z_3 = +14,32° = 0,2500\ rad$	$x_4 = +1,5000$ $y_4 = -0,0625$ $z_4 = +21,48° = 0,3750\ rad$	$\theta_{1,4} = 7°$ $K_{1,4} = 1$ $d_4 = -1$
5	$x_4 = +1,5000$ $y_4 = -0,0625$ $z_4 = +21,48° = 0,3750\ rad$	$x_5 = +1,5000$ $y_5 = +0,0313$ $z_5 = +17,90° = 0,3125\ rad$	$\theta_{1,5} = 3,5°$ $K_{1,5} = 1$ $d_5 = -1$
6	$x_5 = +1,5000$ $y_5 = +0,0313$ $z_5 = +17,90° = 0,3125\ rad$	$x_6 = +1,5000$ $y_6 = -0,0156$ $z_6 = +19,69° = \mathbf{0,3438}\ rad$	$\theta_{1,6} = 1,8°$ $K_{1,6} = 1$ $d_6 = -1$

A Figura 2.11 mostra a trajetória de rotações da divisão $\frac{0,5}{1,5}$. É importante notar que nesta figura não há as não uniformidades, pois o sistema de coordenadas utilizado é o linear. Como está sendo usado o modo *Y-Reduction*, espera-se que o valor de $y_N \rightarrow 0$. Isto pode ser verificado com $y_6 = -0,0156$, já que para o exemplo $N = 6$.

De acordo com a Tabela 2.4, espera-se que o resultado da divisão $\frac{0,5}{1,5}$ esteja armazenado na variável z_6. Comparando o valor calculado (V_C) pelo algoritmo, z_6, com o valor esperado (V_E), verifica-se que o erro relativo, E_R, é igual a 3%.

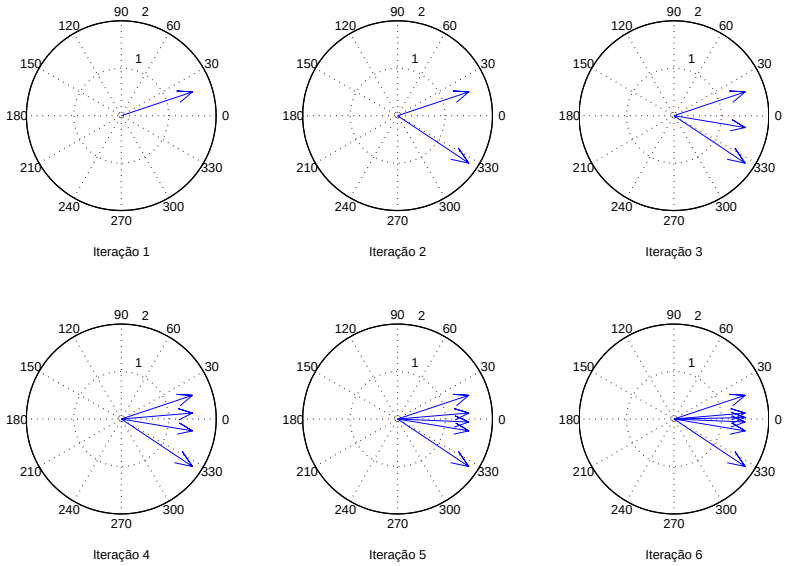


Figura 2.11: Trajetória de rotações para cálculo da divisão $\frac{0,5}{1,5}$.

2.11 Implementação

Todas as funções CORDIC apresentadas na Tabela 2.4 foram implementadas no *Matlab* e resultados de suas simulações podem ser encontradas em [3]. Este trabalho utilizará como base estas funções, aperfeiçoando as mesmas quando houver necessidade. Sempre que alguma alteração significativa for realizada, todos testes feitos em [3] serão repetidos, a fim de verificar-se a capacidade do novo algoritmo.

Tendo como base a implementação realizada em [3], pode-se verificar que algumas funções apresentaram restrições quanto à convergência. Estas restrições já eram esperadas, pois cada sistema de coordenadas possui limitações quanto a região de convergência conforme apresentado na Tabela 2.3. A seguir serão apresentadas as limitações e formas de corrigi-las.

2.11.1 Limitações das funções CORDIC

A tabela abaixo apresenta as funções CORDIC e suas respectivas limitações.

Tabela 2.7: Limitação das funções CORDIC

Função	**Limitação**
$\text{tg}(\phi)$	-
$\text{atg}(\phi)$	-
$\cos(\phi)$	-
$\text{sen}(\phi)$	-
$\sec(\phi)$	-
$\text{csc}(\phi)$	-
adição $(y + x)$	-
multiplicação $(x \cdot z)$	$z \leq 2$
divisão $(\frac{y}{x})$	$y \leq 2 \cdot x$
$\text{tgh}(\phi)$	$\phi < 1,1181\ rad$
$\text{atgh}(arco)$	$arco \leq 0,8069$
$\cosh(\phi)$	$\phi < 1,1181\ rad$
$\text{senh}(\phi)$	$\phi < 1,1181\ rad$
sqrt $(\sqrt{n^{\underline{o}}})$	$0,026 \leq n^{\underline{o}} \leq 2,3816$
exp $(e^{n^{\underline{o}}})$	$n^{\underline{o}} \leq 1,1181$
ln $(\ln n^{\underline{o}})$	$n^{\underline{o}} \leq 9,3$

Analisando a Tabela 2.7, percebe-se que mais da metade das funções CORDIC possui limitações. Como um dos objetivos desta tese é trabalhar com os algoritmos HC (*Householder Constrained*) em uma máquina CORDIC, algumas funções devem ter suas limitações corrigidas.

Os algoritmos HC, que serão vistos adiante, vide equações (4.38) e (4.39), trabalham basicamente com quatro funções matemáticas que são: adição, multiplicação, divisão e raiz quadrada. Estas são as funções que devem ter suas limitações corrigidas.

A seguir será apresentado um método, que consegue se adaptar às limitações apresentadas na Tabela 2.7, utilizando apenas deslocamentos e somas.

2.11.2 Adaptação às limitações das funções CORDIC

O algoritmo utilizado para eliminar às limitações das funções CORDIC é divido em duas partes: pré-CORDIC e pós-CORDIC. O pré-CORDIC e o pós-

CORDIC são algoritmos que foram desenvolvidos para serem executados antes e depois, respectivamente, da função CORDIC. A figura abaixo mostra em que momento estes algoritmos são utilizados.

Figura 2.12: Diagrama de utilização dos algoritmos pré-CORDIC e pós-CORDIC.

Mesmo utilizando estes algoritmos as funções CORDIC continuaram com as suas limitações.

O pré-processamento servirá para adaptar a entrada a um valor que garanta a convergência da função CORDIC. O pós-processamento realizará o papel inverso da função pré-CORDIC. Vale a pena ressaltar que a introdução desses algoritmos à função CORDIC causarão um aumento de processamento à solução completa.

2.11.2.1 Algoritmo pré-CORDIC

A função deste algoritmo é adaptar a entrada às limitações impostas pelas funções CORDIC. Esta adaptação é feita utilizando apenas deslocamentos e somas, logo este algoritmo pode ser utilizado, perfeitamente, em um *hardware* específico para CORDIC.

A idéia básica deste algoritmo consiste em utilizar deslocamentos para fazer a adaptação da entrada. Como a entrada é alterada, é necessário criar mecanismos para que o controle não seja perdido. No algoritmo pré-CORDIC, todo controle é feito por adições de valores inteiros.

Na Tabela 2.8 encontra-se o pseudocódigo deste algoritmo. A letra W é a entrada do algoritmo pré-CORDIC. Dependendo da função CORDIC utilizada, o valor de W pode estar representando um ângulo, um arco ou um número. As variáveis $R_{j,l}$ são as restrições impostas pelas funções CORDIC, onde o índices j e l identificam, respectivamente, a função e a restrição dentro da função. A constante L define a quantidade de restrições que uma função pode ter. A variável *control* é responsável por controlar a quantidade de deslocamentos que foram realizados até a restrição não ser mais verdadeira.

As restrições são mutualmente exclusivas; entretanto, não é possível implementar este algoritmo com um único *loop*, visto que os deslocamento realizados não são iguais para diferentes restrições.

Tabela 2.8: Algoritmo pré-CORDIC.

```
dado  W
control = 0;
if ( ( R_{j,1} for verdade ) ou ( R_{j,2} for verdade ) ou ... ou ( R_{j,L} for verdade ))
        if ( R_{j,1} for verdade )
                while (R_{j,1} for verdade)
                        W = desloca uma vez W;
                        control = control +1;
                end while
        end if
        if ( R_{j,2} for verdade )
                while (R_{j,2} for verdade)
                        W = desloca uma vez W;
                        control = control +1;
                end while
        end if
        ⋮
        if ( R_{j,L} for verdade )
                while (R_{j,L} for verdade)
                        W = desloca uma vez W;
                        control = control +1;
                end while
        end if
end if
```

Em relação aos deslocamentos, estes podem ser realizados tanto para a esquerda quanto para a direita. Esta decisão depende apenas do $R_{j,l}$. Para maiores informações a respeito da variável $R_{j,l}$ e dos deslocamentos, vide Tabela 2.9. A Seção 2.11.3 explica o funcionamento da função pré-CORDIC.

Tabela 2.9: Resumo das restrições com deslocamentos.

Índice (j)	**Função**	**Restrição** ($R_{j,l}$)	**Deslocamento**
1	multiplicação ($x \cdot z$)	$R_{1,1} = (z > 2)$	Direita
2	divisão ($\frac{y}{x}$)	$R_{2,1} = (y > 2 \cdot x)$	Direita
3	tgh(ϕ)	$R_{3,1} = (\phi \geq 1,1181\ rad)$	Direita
4	atgh($arco$)	$R_{4,1} = (arco > 0,8069)$	Direita
5	cosh(ϕ)	$R_{5,1} = (\phi \geq 1,1181\ rad)$	Direita
6	senh(ϕ)	$R_{6,1} = (\phi \geq 1,1181\ rad)$	Direita
7	sqrt ($\sqrt{n^{\underline{o}}}$)	$R_{7,1} = (n^{\underline{o}} < 0,026)$ $R_{7,2} = (n^{\underline{o}} > 2,3816)$	Esquerda Direita
8	exp ($e^{n^{\underline{o}}}$)	$R_{8,1} = (n^{\underline{o}} > 1,1181)$	Direita
9	ln ($\ln n^{\underline{o}}$)	$R_{9,1} = (n^{\underline{o}} > 9,3)$	Direita

2.11.2.2 Algoritmo pós-CORDIC

A função deste algoritmo é fazer o papel inverso da função pré-CORDIC. O algoritmo pós-CORDIC só será utilizado caso o pré-CORDIC tenha sido executado anteriormente.

Caso alguma adaptação tenha sido realizada no algoritmo pré-CORDIC, a variável *control* será diferente de 0. Isto indica que o algoritmo pós-CORDIC deve ser utilizado. A introdução do pré-processamento causa uma falta de consistência ao resultado encontrado pela função CORDIC.

A razão para esta inconsitência são os deslocamentos realizados pela função pré-CORDIC. Sendo assim o objetivo da função pós-CORDIC é realizar deslocamentos no sentido oposto aos realizados pela função pré-CORDIC. A variável *control* servirá para informar a quantidade de deslocamentos que devem ser realizados, a fim de eliminar a adaptação realizada pela função pré-CORDIC.

Na Tabela 2.10 encontra-se o pseudocódigo deste algoritmo. A Seção 2.11.3 explica o funcionamento do algoritmo pós-CORDIC.

Tabela 2.10: Algoritmo pós-CORDIC.

```
dado  W
dado  control
if (control ≠ 0)
        if ( R_{j,1}  for  verdade )
                for  k = 1,2,3,...,control
                        W = desloca  uma  vez  W  no  sentido  oposto;
                end  for
        end  if
        if ( R_{j,2}  for  verdade )
                for  k = 1,2,3,...,control
                        W = desloca  uma  vez  W  no  sentido  oposto;
                end  for
        end  if
        ⋮
        if ( R_{j,L}  for  verdade )
                for  k = 1,2,3,...,control
                        W = desloca  uma  vez  W  no  sentido  oposto;
                end  for
        end  if
end  if
```

2.11.2.3 Considerações parciais

As respostas dos algoritmos adaptativos utilizados nas simulações desta tese podem ser encontrados utilizando apenas as funções *adição*, *multiplicação*, *divisão* e *raiz quadrada* (*sqrt*). Sendo assim os algoritmos de pré-processamento e pós-processamento só serão aplicados, quando necessário, a estas funções.

Em relação aos algoritmos apresentado nas Subseções 2.11.2.1 e 2.11.2.2, algumas observações devem ser feitas:

1. A função *adição* não faz uso destes algoritmos, pois não possui limitações.

2. Os algoritmos apresentados nas Tabelas 2.8 e 2.10 devem ser utilizados no pré-processamento e pós-processamento das funções *multiplicação* e *divisão*.

3. A função *sqrt* utiliza os algoritmos mostrados nas Tabelas 2.8 e 2.10 com algumas modificações. Nas Tabelas 2.11 e 2.12 estão os pseudocódigos de pré-processamento e pós-processamento, respectivamente, desta função. As modificações estão marcadas em negrito.

Tabela 2.11: Algoritmo pré-CORDIC para a função *sqrt*.

```
dado W
control = 0;
if ( ( R7,1 for verdade ) ou ( R7,2 for verdade) )
        if ( R7,1 for verdade )
                while ((R7,1 for verdade) ou (control for impar))
                        W = desloca uma vez W para esquerda ;
                        control = control +1;
                end while
        end if
        if ( R7,2 for verdade )
                while ((R7,2 for verdade) ou (control for impar))
                        W = desloca uma vez W para direita;
                        control = control +1;
                end while
        end if
end if
```

Tabela 2.12: Algoritmo pós-CORDIC para a função *sqrt*.

```
dado  W
dado  control
if (control ≠ 0)
        if ( R7,1 for verdade )
                for k = 1,2,3,...,control/2
                        W = desloca uma vez W para direita;
                end for
        end if
        if ( R7,2 for verdade )
                for k = 1,2,3,...,control/2
                        W = desloca uma vez W para esquerda;
                end for
        end if
end if
```

2.11.3 Simulação das novas funções

As funções CORDIC utilizadas nesta tese foram desenvolvidas em [3], entretanto algumas dessas funções possuem limitações para uso conforme mostrado na Tabela 2.7. Sendo assim, os algoritmos pré-CORDIC e pós-CORDIC serão utilizados, respectivamente, antes e depois a essas funções CORDIC que possuem limitações.

A junção da função CORDIC, desenvolvida em [3], com os algoritmos pré-CORDIC e pós-CORDIC resultam numa nova função CORDIC que não possui limitações de cálculo. A seguir serão mostradas as simulações realizadas no *Matlab* para as novas funções CORDIC (*mult_c_a*, *div_c_a* e *sqrt_c_a*).

2.11.3.1 Função Multiplicação CORDIC Avançada (*mult_c_a*)

A função *mult_c_a* é a evolução da função *mult_c* desenvolvida em [3], pois ela é composta pelos algoritmos de adaptação e pela própria *mult_c*. O objetivo dela é permitir ao usuário realizar a operação de multiplicação sem nenhuma restrição quanto às entradas da função.

A Tabela 2.13 mostra o funcionamento do algoritmo pré-CORDIC quando deseja-se multiplicar 2 unidades (entrada x) por 34 unidades (entrada z) com uma precisão de 20 *bits*.

O usuário deve executar o comando da seguinte forma: mult_c_a(2,34,20).

Tabela 2.13: Funcionamento do pré-processamento de *mult_c_a*.

Valor do *control*	**Valor de x**	**Valor de z (W)**
0	2	34
1	2	17
2	2	8,5000
3	2	4,2500
4	2	2,1250
5	**2**	**1,0625**

Os valores de x e z, que realmente estão entrando na função *mult_c* [3], são 2 e $1,0625$, respectivamente. Todo pré-processamento foi realizado na variável z, pois a função *mult_c* só converge quando $z \leq 2$. Após a função pré-CORDIC o novo valor de z é igual a $1,0625$. Já a variável *control* é igual a 5, pois foram necessários cinco deslocamentos para a direita para que z fosse inferior ou igual a 2.

A saída da função *mult_c*, para os novos valores das entradas x e z, foi igual a $2,125$. Isto comprova o funcionamento correto da função *mult_c*, todavia o resultado final ainda não foi atingido.

De acordo com a Tabela 2.4, o resultado da multiplicação estará em y_{20}. Esta saída terá que passar por um pós-processamento a fim de compensar os deslocamentos realizados no pré-processamento.

A Tabela 2.14 explica o funcionamento do algoritmo pós-CORDIC quando deseja-se multiplicar 2 unidades (entrada x) por 34 unidades (entrada z) com uma precisão de 20 *bits*. Ao todo, no pós-processamento, foram realizados cinco deslocamentos para a esquerda, a fim de compensar os deslocamentos realizados no pré-processamento. A Tabela 2.14 mostra o resultado de cada um dos deslocamentos até atingir o valor da variável *control*. Essa variável determina a quantidade de deslocamentos necessários para ajustar o cálculo da função *mult_c_a*.

Tabela 2.14: Funcionamento do pós-processamento de *mult_c_a*.

Deslocamento	**Valor de y_{20} (W)**
0	2,1250
1	4,2500
2	8,5000
3	17,0000
4	34,0001
5	**68,0001**

A Figura 2.13 explica o funcionamento do algoritmo *mult_c_a*.

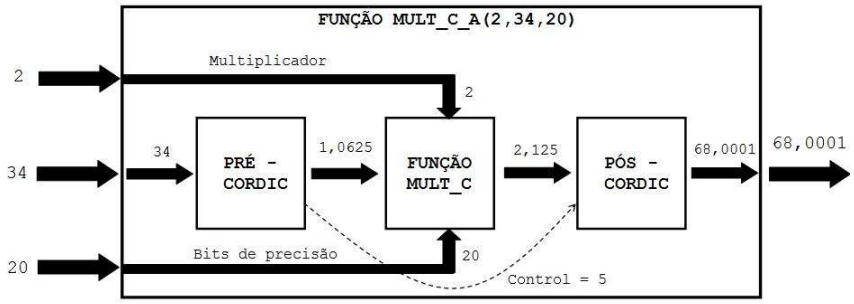


Figura 2.13: Esquemático da função *mult_c_a*.

2.11.3.2 Função Divisão CORDIC Avançada (*div_c_a*)

A função *div_c_a* é a evolução da função *div_c* desenvolvida em [3], pois ela é composta pelos algoritmos de adaptação e pela própria *div_c*. O objetivo dela é permitir ao usuário realizar a operação de divisão sem nenhuma restrição quanto as entradas da função.

A Tabela 2.15 mostra o funcionamento do algoritmo pré-CORDIC quando deseja-se dividir 60 unidades (entrada y) por 3 unidades (entrada x) com uma precisão de 20 *bits*.

O usuário deve executar o comando da seguinte forma: div_c_a(30,3,20).

Tabela 2.15: Funcionamento do pré-processamento de *div_c_a*.

Valor do *control*	**Valor de x**	**Valor de y (W)**	Valor de $\frac{y}{x}$
0	3	60	20
1	3	30	10
2	3	15	5
3	3	7,5000	2,5000
4	**3**	**3,7500**	**1,2500**

Os valores de y e x, que realmente estão entrando na função *div_c* [3], são $3,75$ e 3, respectivamente. Todo pré-processamento foi realizado na variável z, pois a função *div_c* só converge quando $y \leq 2 \cdot x$. Após a função pré-CORDIC o novo valor de y é igual a $3,7500$. Já a variável *control* é igual a 4, pois foram necessários quatro deslocamentos para a direita para que a razão $\frac{y}{x}$ fosse inferior ou igual a 2.

A saída da função *div_c*, para os novos valores das entradas y e x, foi igual a $1,25$. Isto comprova o funcionamento correto da função *div_c*; todavia, o resultado final ainda não foi atingido.

De acordo com a Tabela 2.4, o resultado da divisão estará em z_{20}. Esta saída terá que passar por um pós-processamento a fim de compensar os deslocamentos realizados no pré-processamento.

A Tabela 2.16 explica o funcionamento do algoritmo pós-CORDIC quando deseja-se dividir 60 unidades (entrada y) por 3 unidades (entrada x) com uma precisão de 20 *bits*. Ao todo, no pós-processamento, foram realizados quatro deslocamentos para a esquerda, a fim de compensar os deslocamentos realizados no pré-processamento. A Tabela 2.16 mostra o resultado de cada um dos deslocamentos até atingir o valor da variável *control*.

Tabela 2.16: Funcionamento do pós-processamento de *div_c_a*.

Deslocamento	**Valor de z_{20} (W)**
0	1,2500
1	2,5000
2	5,0000
3	10,0000
4	**20,0000**

A Figura 2.14 explica o funcionamento do algoritmo *div_c_a*.

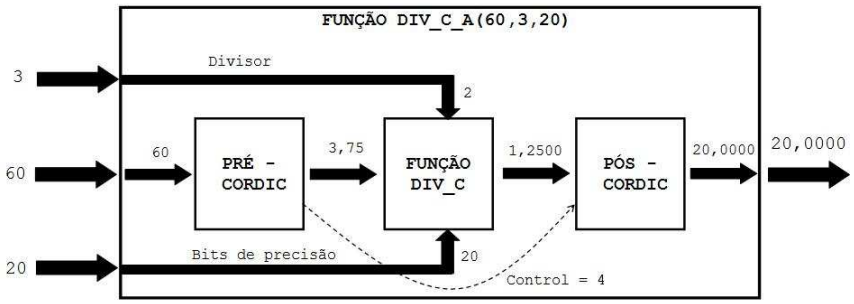


Figura 2.14: Esquemático da função *div_c_a*.

2.11.3.3 Função Sqrt CORDIC Avançada (*sqrt_c_a*)

A função *sqrt_c_a* é a evolução da função *sqrt_c* desenvolvida em [3], pois ela é composta pelos algoritmos de adaptação e pela própria *sqrt_c*. O objetivo dela é permitir ao usuário realizar a operação de raiz quadrada sem nenhuma restrição quanto as entradas da função.

A Tabela 2.17 mostra o funcionamento do algoritmo pré-CORDIC quando deseja-se calcular a raiz quadrada de 169 unidades com uma precisão de 20 *bits*.

O usuário deve executar o comando da seguinte forma: sqrt_c_a(169,20).

Tabela 2.17: Funcionamento do pré-processamento de *sqrt_c_a*.

Valor do *control*	**Valor de W**
0	169,0000
1	84,5000
2	42,2500
3	21,1250
4	10,5625
5	5,2813
6	2,6406
7	1,3203
8	**0,6602**

De acordo com a Tabela 2.7, a função *sqrt_c* só irá convergir quando $0,026 \leq W \leq 2,3816$. Na simulação W é igual a 169. Analisando a Tabela 2.17, percebe-se que esta condição é atingida quando a variável *control* é igual a 7, todavia o pré-processamento ainda realiza mais um deslocamento. Isto ocorre pois a variável *control* deve ser par para que os algoritmos de adaptação funcionem corretamente.

A saída da função *sqrt_c*, para o novo valor de entrada foi igual a $0,8125$. Isto comprova o funcionamento correto da função *sqrt_c*, todavia o resultado final ainda não foi atingido.

De acordo com a Tabela 2.4, o resultado da função raiz quadrada estará em x_{20}. Esta saída terá que passar por um pós-processamento a fim de compensar os deslocamentos realizados no pré-processamento.

A Tabela 2.18 mostra o funcionamento do algoritmo pós-CORDIC quando deseja-se calcular a raiz quadrada de 169 unidades com uma precisão de 20 *bits*. Ao todo, no pós-processamento, foram realizados quatro deslocamentos para a esquerda, a fim de compensar os oito deslocamentos para a direita realizados no pré-processamento. A Tabela 2.18 mostra o resultado de cada um dos deslocamentos até atingir o valor da variável $control/2$.

Tabela 2.18: Funcionamento do pós-processamento de *sqrt_c_a*.

Deslocamento	**Valor de z_{20} (W)**
0	0,8125
1	1,6250
2	3,2500
3	6,5000
4	**13,0000**

É importante notar que, no pós-processamento, não foram realizados oito deslocamentos e sim quatro. Estas diferença de deslocamentos está associado às características da função raiz quadrada.

A Figura 2.15 ilustra o funcionamento do algoritmo *sqrt_c_a*.

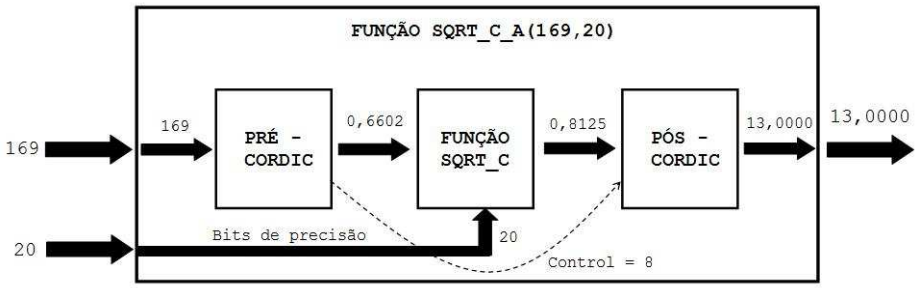


Figura 2.15: Esquemático da função *sqrt_c_a*.

2.12 Resultados

As funções CORDIC implementadas em [3] funcionam corretamente, todavia possuem algumas limitações. As limitações estão associadas diretemente com a escolha da seqüência de deslocamento.

Um método de pós-processamento e outro de pré-processamento foram propostos a fim de adaptar as entradas das funções CORDIC às limitações das funções desenvolvidas em [3]. Sua aplicabilidade só foi testada nas funções CORDIC de multiplicação, divisão e raiz quadrada. Essas funções foram escolhidas, pois, junto com a função adição, formam o conjunto de funções CORDIC necessárias para implementar os algoritmos adaptativos que serão apresentados no Capítulo 4.

De acordo com as simulações realizadas na Seção 2.11.3, verifica-se que os algoritmos de adaptação funcionam perfeitamente. O usuário que estiver utilizando as funções *mult_c_a*, *div_c_a* e *sqrt_c_a* não terá preocupações quanto à convergência dos algoritmos.

As novas funções CORDIC não possuem nenhuma limitação, em contrapartida o processamento de dados será maior. Os algoritmos de adaptação são funções CORDIC, sendo assim podem ser implementados em uma máquina CORDIC.

Capítulo 3

Beamforming

3.1 Introdução

O termo *beamforming* é derivado da palavra *beam*, que em português significa feixe, pois o objetivo das primeiras filtragens espaciais eram criar feixes em uma determinada direção de modo a receber ou transmitir um sinal de forma mais eficiente. O *beamforming* é uma técnica usada num *array* de sensores de modo a direcionar o sinal transmitido ou recebido através de uma forma versátil de filtragem espacial. O papel dos sensores do *array* é coletar amostras espaciais da onda propagada que posteriormente serão processadas pelo *beamformer*. É importante explicar que *beamformer* é um termo dado ao processador que fará a filtragem espacial quando a amostragem espacial é discreta.

Devido ao fato do *beamforming* ser uma técnica que pode ser aplicada tanto na transmissão quanto na recepção de dados, é importante ressaltar que as simulações que serão mostradas nos capítulos seguintes são relacionados à utilização do *beamforming* na recepção de sinais para sistemas de comunicação. Sendo assim, este capítulo dará enfoque as técnicas de *beamforming* para a recepção.

Vale a pena ressaltar que além da aplicação em comunicações, a filtragem espacial pode ser utilizada em outras áreas. A Tabela 3.1 lista outros tipos de aplicação para filtragem espacial.

Tabela 3.1: Lista de aplicações para filtragem espacial.

Aplicação	**Exemplo**
RADAR	Controle de tráfego aéreo
SONAR	Classificação e localização de lugar
COMUNICAÇÕES	Transmissão e recepção direcional
IMAGEM	Tomografia
EXPLORAÇÃO GEOFÍSICA	Exploração de petróleo
EXPLORAÇÃO AEROESPACIAL	Alta resolução de imagem do universo
BIOMEDICINA	Monitoração do coração de um feto

Em sistemas de comunicações, normalmente, o sinal desejado e o interferidor ocupam a mesma banda de freqüência, logo a filtragem temporal não é suficiente para separar o sinal desejado do interferidor. Todavia, o sinal desejado e o interferidor são normalmente originados de diferentes lugares, sendo assim esta separação espacial pode ser explorada a fim de separar o sinal desejado do interferidor.

O principal objetivo do *beamforming* é detectar um determinado sinal desejado a partir de um sinal onde, além do sinal desejado, há diversos outros sinais provenientes de interferidores. Os *beamformers* são os elementos responsáveis por executar a filtragem espacial de modo a separar sinais que possuem freqüência sobrepostas que foram originados em diferentes posições do espaço.

Tipicamente, o *beamformer* faz a combinação linear das amostras espaciais recebidas pelos sensores a fim de obter uma saída escalar. Uma das principais vantagens em se utilizar um *array* de antenas está relacionada à filtragem espacial. Este tipo de filtragem, quando utiliza amostras discretas, permite que mudanças em tempo real possam ser realizadas na combinação linear dos sinais das saídas dos sensores pelo *beamformer*.

Este capítulo tem como objetivo descrever o *beamforming* pela perspectiva do processamento de sinais, provendo um visão completa do projeto de *beamformers*. Neste capítulo, primeiramente, serão apresentados conceitos sobre *beamforming* e filtragem espacial. Posteriormente, será apresentada a classificação dos *beamformers* dando ênfase às suas diferenças, vantagens e desvantagens. Posteriormente, será

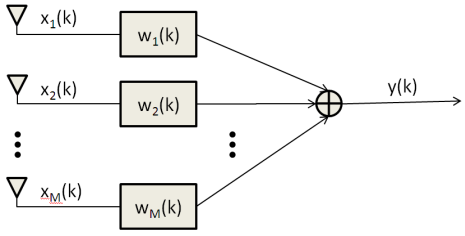
feita uma introdução sobre *beamformer* adaptativo. No final, será apresentado um resumo sobre os assuntos abordados neste capítulo.

3.2 *Beamforming* e filtragem espacial

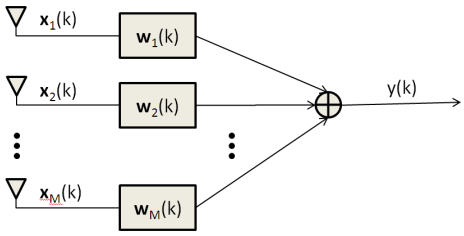
Nesta seção será definido matematicamente o *beamforming* e o assunto filtragem espacial será discutido com mais detalhes. A fim de dar continuidade à seção é necessário fazer a seguinte definição: todos vetores representados daqui em diante são vetores coluna.

Conforme dito anteriormente, o *beamforming* é utilizado nas aplicações descritas pela Tabela 3.1 para distinguir o sinal desejado e interferidor através de suas propriedades espaciais. Já o *beamformer* é o dispositivo utilizado para realizar o *beamforming*.

As Figuras 3.1(a) e 3.1(b) são duas formas de *beamformers*. A Figura 3.1(a) é tipicamente utilizada para processar sinais de banda estreita; já a Figura 3.1(b) trabalha com sinais de banda larga.



(a) *Beamformer* de banda estreita (*narrowband*). Cada sensor tem sua saída multiplica por um escalar.



(b) *Beamformer* de banda larga (*broadband*). Cada sensor tem sua saída multiplicada por um um filtro de comprimento N.

Figura 3.1: Tipos de *beamformer*: (a) banda estreita; (b) banda larga.

A saída do *beamformer* de banda estreita ou larga pode ser representada por

$$y(k) = \mathbf{w}^H(k)\mathbf{x}(k) \tag{3.1}$$

onde, para o *beamformer* de banda estreita,

$$\mathbf{w}(k) = \left[\begin{array}{cccc} w_1(k) & w_2(k) & \dots & w_M(k) \end{array} \right]^T \tag{3.2}$$

$$\mathbf{x}(k) = \left[\begin{array}{cccc} x_1(k) & x_2(k) & \dots & x_M(k) \end{array} \right]^T \tag{3.3}$$

e para o *beamformer* de banda larga:

$$\mathbf{w}(k) = \left[\begin{array}{cccc} \mathbf{w}_1^H(k) & \mathbf{w}_2^H(k) & \ldots & \mathbf{w}_M^H(k) \end{array} \right]^H \tag{3.4}$$

$$\mathbf{x}(k) = \left[\begin{array}{cccc} \mathbf{x}_1^H(k) & \mathbf{x}_2^H(k) & \ldots & \mathbf{x}_M^H(k) \end{array} \right]^H . \tag{3.5}$$

É importante ressaltar que, para o *broadband beamformer*, o comprimento dos vetores $\mathbf{w}(k)$ (pesos do *beamformer* no instante k) e $\mathbf{x}(k)$ (vetor de entrada no instante k) é igual ao número de sensores, ou seja, M, multiplicado pelo comprimento dos vetores $\mathbf{w}_i(k)$ e $\mathbf{x}_i(k)$, ou seja, N, para $i = 1, 2, \ldots, M$. Sendo assim, o comprimento dos vetores $\mathbf{w}(k)$ e $\mathbf{x}(k)$ é igual a MN. Os vetores $\mathbf{w}_i(k)$ e $\mathbf{x}_i(k)$ podem ser representados como

$$\mathbf{w}_i(k) = \left[\begin{array}{cccc} w_i(k) & w_i(k-1) & \ldots & w_i(k-N+1) \end{array} \right]^T \tag{3.6}$$

$$\mathbf{x}_i(k) = \left[\begin{array}{cccc} x_i(k) & x_i(k-1) & \ldots & x_i(k-N+1) \end{array} \right]^T . \tag{3.7}$$

É importante notar que, quando o comprimento dos filtros $\mathbf{w}_i$ for igual a 1, os vetores representados por (3.4) e (3.5) serão, respectivamente, iguais aos vetores (3.2) e (3.3).

A resposta da filtragem espacial realizada pelo *beamformer* visa dar um ganho para a direção dos sinais desejados. Este ganho será representado pelo vetor $\mathbf{f}$. Matematicamente este vetor pode ser expresso por

$$\mathbf{w}^H(k)\mathbf{S}(\phi) = \mathbf{f} \tag{3.8}$$

onde

$\mathbf{w}(k)$: vetor de coeficientes, cuja dimensão é $MN \times 1$;

$\mathbf{S}(\phi)$: matriz de resposta do *array* ou *steering matrix*. Sua dimensão é igual a $MN \times r$, onde r é igual a número de sinais desejados (restrições);

ϕ : ângulo elétrico de chegada do sinal desejado;

$\mathbf{f}$: vetor ganho. Sua dimensão é igual a $r \times 1$.

A partir de agora, todo desenvolvimento será feito para $r = 1$, ou seja, será considerado apenas um sinal desejado. Sendo assim a matriz $\mathbf{S}(\phi)$ passará a ser o vetor $\mathbf{s}(\phi)$ também conhecido como vetor de resposta do *array* ou *steering vector*. A dependência do vetor $\mathbf{s}(\phi)$ pelo ângulo elétrico ϕ para *arrays* lineares é definida por

$$\mathbf{s}(\phi) = \left[\begin{array}{cccc} 1 & e^{-j\phi} & \ldots & e^{-j(M-1)\phi} \end{array} \right]^T . \tag{3.9}$$

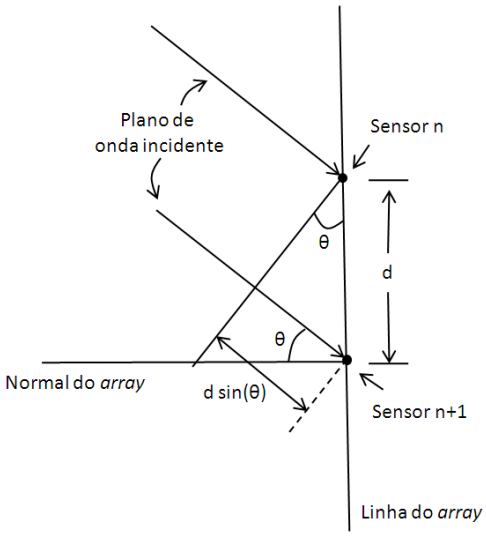


Figura 3.2: Atraso espacial em um *array* linear.

A equação (3.8) passa a ser definida como

$$\mathbf{w}^H(k)\mathbf{s}(\phi) = f \tag{3.10}$$

A equação (3.10) pode ser interpretada como: todo sinal que incide com o ângulo elétrico ϕ é repassado para a saída com o ganho f.

O ângulo elétrico ϕ está associado com o ângulo de incidência θ do sinal desejado cuja medição é feita em relação à normal do *array* de sensores. A Figura 3.2 mostra como o ângulo de incidência θ é medido. Sendo assim a relação desses dois ângulos é expressa por

$$\phi = \frac{2\pi d}{\lambda}\sin(\theta) \tag{3.11}$$

onde:

d : espaçamento entre os sensores adjacentes;

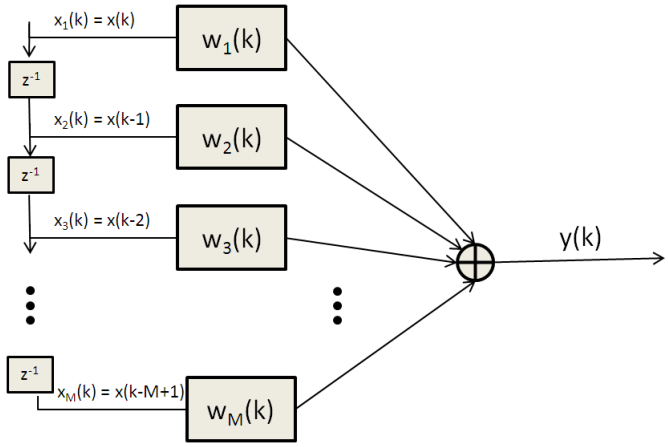
λ : comprimento de onda do sinal desejado.

O ângulo de incidência θ assume valores entre $\frac{-\pi}{2}$ a $\frac{\pi}{2}$, já o ângulo elétrico ϕ depende da razão $\frac{d}{\lambda}$, por exemplo: quando essa razão é igual a $\frac{1}{2}$, o ângulo ϕ assume valor entre $-\pi$ a π. Isto pode ser facilmente provado pela equação (3.11).

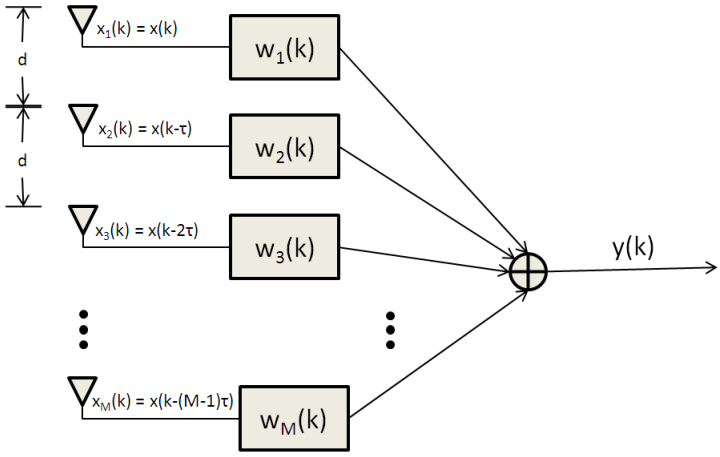
A partir da Figura 3.2, observa-se que a frente de onda incidente tem que percorrer uma distância igual a $d\sin(\theta)$ para chegar ao sensor $n+1$ após ter atingido o sensor n. Essa diferença na distância faz com que a frente de ondas chegue com uma fase diferente em cada um dos sensores. A diferença de fase está representada

pela equação (3.11). Sendo assim, o ângulo elétrico ϕ pode ser interpretado como a diferença de fase entre sensores adjacentes de um *array* linear.

Como o *beamforming* faz uso de filtragem espacial, é importante fazer uma comparação com filtros FIR temporais. A correspondência entre esses tipos de filtro pode ser feita quando o *beamformer* de banda estreita é utilizado, ou seja, o *beamformer* opera com apenas uma freqüência, em um *array* linear que possui sensores igualmente espaçados. Somente para esse caso especial pode-se fazer um mapeamento entre esses tipos de filtro. As Figuras 3.3(a) e 3.3(b) representam, respectivamente, os filtros temporal e espacial.



(a) Filtro FIR temporal.



(b) Filtro FIR espacial.

Figura 3.3: Comparação entre os filtros (a)temporal e (b)espacial.

A constante τ, na Figura 3.3(b), representa o atraso entre os sensores. O valor deste atraso está associado com a distância percorrida pela frente de onda

entre sensores adjacentes e a velocidade da luz ($c = 3 \cdot 10^8$m/s). A equação (3.12) mostra essa relação.

$$\tau = \frac{d \sin(\theta)}{c} = \frac{\phi}{2\pi f_0} \tag{3.12}$$

onde f_0 é a freqüência do sinal desejado.

Assim como na amostragem temporal, pode acorrer *aliasing* caso a freqüência de amostragem espacial seja mal escolhida. Quando há o *aliasing* espacial significa que há ambigüidade na localização das fontes geradoras de sinal. Isso implica que fontes em diferentes localização possuem o mesmo *steering vector*. Isto pode acontecer quando os sensores estão muito afastados. Sendo assim, a escolha do espaçamento dos sensores é muito importante para o correto funcionamento do *beamformer*. A fim de evitar a ambigüidade, a distância entre os sensores deve respeitar a condição $d \leq \frac{\lambda}{2}$.

A seguir será apresentada a classificação dos *beamformers* e suas respectivas explicações.

3.3 Classificação

De acordo com [13], os *beamformers* podem ser classificados como independentes, estatiscamente ótimos e adaptativos, dependendo de como os pesos (coeficientes do vetor $\mathbf{w}$) são escolhidos. No caso dos independentes, os pesos são escolhidos de modo a não dependerem do sinal de entrada $\mathbf{x}(k)$. Vale a pena lembrar que este sinal é formado pelo sinal desejado e por interferidores.

Já para o *beamformer* estatiscamente ótimo, a escolha dos pesos é baseada na estatística do sinal de entrada de modo a otimizar a saída do *array* de sensores, ou seja, $y(k)$. A vantagem deste tipo de *beamformer* em relação ao independe está no fato deste colocar nulos na direção das fontes interferidoras de modo a maximizar a relação sinal desejado/ruído (SNR - *Signal Noise Ratio*) ou minimizar a relação sinal interferidor/ruído (SINR - *Signal Interferer Noise Ratio*) da saída do *beamformer*.

Os *beamformers* adaptativos possuem todas características dos *beamformers* estatisticamente ótimos com uma única diferença: para determinar seus pesos, não é necessário conhecer toda estatística de segunda ordem da entrada. Normalmente, estas estatísticas não são conhecidas, logo, assumindo ergodicidade, é possível es-

timá-las a partir do sinal de entrada.

As técnicas utilizadas nos *beamformers* independentes da entrada são quase sempre utilizadas nos *beamformers* estatisticamente ótimos e adaptativos. Conforme dito anteriormente, as estatísticas da entrada do *array* não são conhecidas, logo faz-se necessário ter algoritmos adaptativos com o intuito de determinar os pesos ótimos. Os algoritmos adaptativos são projetados de modo a convergir para uma solução estatisticamente ótima.

A seguir será explicado o funcionamento dos tipos de *beamformers* listados nesta seção.

3.3.1 *Beamformer* independente

Este tipo de *beamformer* também é conhecido como *beamformer* convencional ou *switched beam smart antenna* [14]. Seus pesos são fixos, e são escolhidos utilizando apenas as informações sobre localização dos sensores no espaço e direção de interesse da frente de onda.

Este tipo de *beamformer* pode ser interessante para algumas aplicações. Por exemplo, é desejável ter como saída do *beamformer* apenas sinais que estejam chegando numa determinada faixa de direção. Sendo assim o *beamformer* deve ser projetado para dar ganho 1 aos sinais que estiverem chegando na faixa de direção desejada. Para esta aplicação não há preocupação com os interferidores, o que está sendo buscado é aumentar o SNR.

Outro bom exemplo de aplicação para *beamformer* independente é descrito a seguir: há um sinal interferidor forte chegando numa certa faixa de direção. Sendo assim, o *beamformer* deve ser projetado para dar ganho 0 aos sinais que estiverem chegando na faixa de direção não desejada. Neste exemplo perceba que a preocupação está apenas no sinal interferidor e não é feito nenhum processamento para o sinal desejado.

A Figura 3.1 ilustra este tipo de *beamformer*. Pode-se perceber que ele é formado por apenas atrasos e adições, por isso, é conhecido também como *delay-and-sum beamformer*.

A seguir serão apresentados alguns tipos de *beamformers* estatisticamente ótimos.

3.3.2 *Beamformer* estatisticamente ótimo

Os pesos deste tipo de *beamformers* são calculados baseados na estatística do sinal recebido pelo *array*, ou seja, $\mathbf{x}(k)$. O objetivo deste é otimizar a resposta do *beamformer* de modo a conter uma contribuição mínina de ruído (interferidores) e máxima do sinal incidente na direção desejada.

O problema destes tipos de *beamformer* está no fato deles necessitarem de informações prévias do sinal de entrada. É necessários que as estatísticas de segunda ordem do sinal de entrada sejam conhecidas previamente para que os algoritmos funcionem corretamente. Nem sempre isto é possível, logo faz-se necessário utilizar algoritmos adaptativos no processamento do *beamformer*. Alguns algoritmos adaptativos serão apresentados com mais detalhes no Capítulo 4.

Há diversos critérios para escolher estatisticamente os pesos de um *beamformer* ótimo. Cada um dos critérios possui um método para encontrar os pesos ótimos do *beamformer*. Entre esse métodos pode-se citar: MSC (*Multiple Sidelobe Canceller*), sinal de referência (*reference signal*), maximização do SNR (max SNR) e LCMV (*Linearly Constrained Minimum Variance*). A seguir cada um desses métodos serão explicados [13].

3.3.2.1 MSC (*Multiple Sidelobe Canceller*)

Este critério foi desenvolvido por Applebaum [14]. O MSC é baseado em dois canais: principal e auxiliar. O canal principal é formado por um *beamformer* independente que dá ganho em uma determinada faixa de direção, onde está o sinal desejado. Já o canal auxiliar recebe os sinais interferidores.

O objetivo deste critério é escolher os pesos do canal auxiliar de modo a cancelar a componente de interferência no canal principal. Sendo assim, é necessário descobrir um vetor de pesos para o canal auxiliar de modo que sua combinação linear com os sinais interferidores deste canal sejam iguais à resposta dos interferidores no canal principal.

A utilização do MSC é muito eficiente em aplicações onde o sinal desejado é muito fraco, quando comparado ao sinal interferidor, ou em momentos em que o sinal desejado não estiver sendo transmitido. Sendo assim nos momentos de falta de transmissão do sinal desejado os pesos são atualizados e na presença seu valor não

será alterado.

3.3.2.2 Sinal de referência

Este critério foi desenvolvido por Widrow, Mantey, Griffiths e Goode [15]. O sinal de referência é um critério que pode ser adotado somente quando o sinal desejado é conhecido. Desta forma, os pesos do *beamformer* serão escolhidos de modo a minimizar o erro entre a saída do *beamformer* e o sinal desejado. Vale a pena ressaltar que o sinal desejado é também conhecido como sinal de referência.

Para cálculo do vetor de pesos é necessário saber a matriz de covariância do sinal de entrada e a matriz de correlação cruzada do sinal de referência com o sinal de entrada do *array* de sensores. Sendo assim, há uma necessidade prévia de conhecer a estatística de segunda ordem dessas entradas.

É importante mencionar que, para esse critério funcionar, é necessário que o sinal de referência seja descorrelacionado com os sinais interferidores pertencentes ao sinal de entrada no *array*. A grande vantagem deste critério está no fato de não necessitar do conhecimento da direção de chegada do sinal desejado, todavia o sinal de referência deve ser conhecido previamente.

3.3.2.3 Maximização do SNR

Este critério está explicado com mais detalhes em [16]. Ele consiste em calcular o vetor de pesos de modo a maximizar a relação sinal/ruído da saída do *beamformer*.

Para que a maximização possa ser feita é necessário que a matriz de autocorrelação do sinal desejado e do sinal inteferidor sejam conhecidas previamente. Sendo assim, antes de aplicar esse critério é necessário ter conhecimento dessas matrizes. A vantagem deste critério está em justamente conseguir maximizar a SNR.

3.3.2.4 LCMV

Normalmente, o sinal desejado é desconhecido e estará sempre presente no sinal de entrada do *array* de sensores para diversas aplicações. Sendo assim nenhum dos critérios apresentados anteriormente poderão ser aplicados. Essas limitações são resolvidas quando o critério LCMV é utilizado para encontrar o vetor de pesos.

Sua idéia básica é colocar restrições na resposta do *beamformer* de modo a permitir que o sinal desejado passe pelo *beamformer* com um determinado ganho e fase. Os pesos são determinados com o objetivo de minimizar a variância ou a potência da saída do *array* sujeito às restrições impostas na saída do *beamformer*.

Sendo assim, o critério LCMV permite a preservação do sinal desejado enquanto as contribuições dos sinais interferidores e ruído, que chegam em direções diferentes da direção do sinal desejado, são minimizadas na saída do *beamformer*.

Este critério será apresentado com mais detalhes na Seção 4.3, quando serão explicados os filtros LC (*Linearly Contrained*). Este critério será de suma importância para a tese, pois os algoritmos LC baseados na transformada de *Householder*, estudados nesta tese, fazem uso deste método.

3.3.3 *Beamformer* adaptativo

A necessidade pelo *beamformer* adaptativo faz necessário, pois os *beamformers* estatisticamente ótimos necessitam do conhecimento das estatísticas de segunda ordem, que na maioria das aplicações não são conhecidas. Assumido que o sistema é ergódico, é possível estimar essas estatísticas a partir do sinal de entrada.

Outra vantagem dos *beamformers* adaptativos está no fato da estatística do sinal de entrada mudar no decorrer do tempo, já que, normalmente, os interferidores estão mudando de lugar. Com os *beamformers* adaptativos é possível detectar essa variação de forma imediata, pois a estimação das estatísticas de segunda ordem são feitas a partir do sinal de entrada.

É importante observar que todos critérios, apresentados na Subseção 3.3.2, podem ser utilizados com algoritmos adaptativos, com isso suas limitações são sanadas. Os algoritmos adaptativos, de interesse para a tese, estão explicados no Capítulo 4.

3.4 Considerações parciais

A fim de avançar nos próximos assuntos que serão abordados na tese, um breve resumo deste capítulo será realizado nesta seção. O resumo torna-se necesário, pois o Capítulo 4 utilizará vários conceitos apresentados até o momento.

O *beamformer* é responsável por gerar uma saída escalar a partir da combinação do vetor de pesos com o sinal recebido pelo *array* de sensores. Os pesos determinam as características do filtro espacial e permite a separação dos sinais quando os mesmos são gerados por origens diferentes.

Os *beamformers* podem ser classificados como independentes, estatisticamente ótimos e adaptativos. A diferença entre esses *beamformers* está no modo como os pesos dos filtros espaciais são calculados.

Os pesos dos *beamformers* independentes são escolhidos de modo a ter uma resposta independente do sinal recebido. Os pesos dos *beamformers* estatisticamente ótimos e adaptativos são determinados com o intuito de otimizar a resposta do *beamformer* baseado nas estatísticas do sinal de entrada.

A diferença entre os *beamformers* estatisticamente ótimos e os adaptativos está no fato do primeiro necessitar da estatística de segunda ordem, enquanto o segundo determina a mesma durante o processo de adaptação fazendo uma estimação através do sinal de entrada.

O *beamformer* adaptativo é sem dúvidas o mais eficiente dos *beamformers*, pois não necessita saber nenhuma estatística do sinal de entrada. Sendo assim ele consegue se adaptar rapidamente a situações onde há uma variação do comportamento dos interferidores, sendo ótimo para aplicações em tempo real.

O próximo capítulo abordará diversos assuntos que visam explicar o funcionamento dos algoritmos LC baseados na transformada de *Householder*. Esses algoritmos serão utilizados na tese para implementar o *beamformer* adaptativo que será utilizado nas simulações. O resultado dessas simulações em uma máquina CORDIC estão apresentadas no Capítulo 6.

Capítulo 4

Algoritmos adaptativos com restrição baseados na transformada de *Householder*

4.1 Introdução

O HCLMS (*Householder Constrained Least Mean Square*) e o NHCLMS (*Normalized Householder Constrained Least Mean Square*) são algoritmos baseados na junção da transformada de *Householder* com o algoritmo LMS (*Least Mean Square*). Esse algoritmos serão o foco de estudo deste capítulo. De acordo com [2] e [17] , tais algoritmos são implementações muito eficientes de filtros LCMV (*Linearly Constrained Minimum Variance*).

Do ponto de vista de adaptação, esses algoritmos, apesar de serem derivados do CLMS (*Constrained Least Mean Square*), assemelham-se a uma estrutura conhecida como GSC (*Generic Sidelobe Canceller*) pois, após a transformação do sinal de entrada, é possível aplicar qualquer algoritmo adaptativo sem restrição, por exemplo, o algoritmo LMS (*Least Mean Square*)[18].

Neste capítulo, serão explicados de forma detalhada, os conceitos necessários para a compreensão dos algoritmos HCLMS e NHCLMS. Inicialmente será explicado o funcionamento da transforma de *Householder*. Depois serão abordados os conceitos sobre filtros LC (*Linearly Constrained*), onde será explicado o algoritmo CLMS e a estrutura GSC.

Após a apresentação desses conceitos, os algoritmos HC (*Householder Constrained*) serão explicados. Por fim será feita uma comparação entre todos esses algoritmos dando ênfase às vantagens dos algoritmos com restrição baseados na transformação de *Householder* em relação aos demais.

4.2 Transformada de *Householder*

Também conhecida como reflexão ou matriz de *Householder*. Nesta tese, a transformada de *Householder* será representada pela matriz $\mathbf{Q}$ cuja dimensão é $M \times M$ e está definida conforme equação (4.1)[1].

$$\mathbf{Q} = \mathbf{I} - \frac{2\mathbf{u}\mathbf{u}^T}{\|\mathbf{u}\|^2} \tag{4.1}$$

o vetor $\mathbf{u}$ possui dimensão $M \times 1$ e é conhecido como vetor de *Householder*. Já $\mathbf{I}$ corresponde à matriz identidade de dimensão $M \times M$.

Quando multiplica-se um vetor $\mathbf{x}$ pela matriz $\mathbf{Q}$, ou seja, $\mathbf{Qx}$, o resultado encontrado para esta operação será o vetor $\mathbf{x}$ refletido em relação ao subespaço ortogonal ao vetor $\mathbf{u}$, ou seja, span $\{\mathbf{u}^{\perp}\}$. Essa projeção está representada na Figura 4.1.

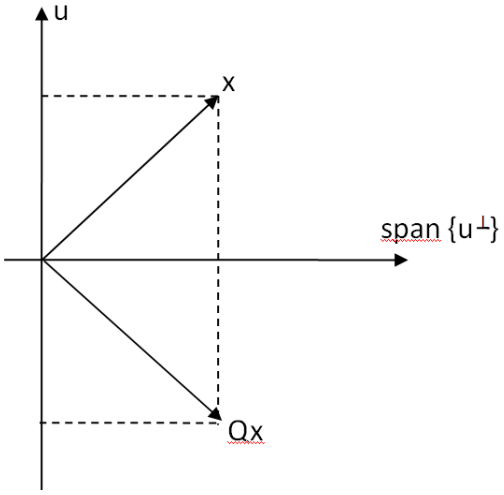


Figura 4.1: Interpretação geométrica da transformada de *Householder*.

4.2.1 Propriedades

A transformada de *Householder* será sempre uma matriz hermitiana ($\mathbf{Q}^H = \mathbf{Q}$), unitária ($\mathbf{Q}^{-1} = \mathbf{Q}^H$) e o comprimento do vetor resultante da operação $\mathbf{Qx}$ é igual ao comprimento do vetor $\mathbf{x}$, ou seja, $\|\mathbf{Qx}\| = \|\mathbf{x}\|$. Esta última característica é ilustrada também na Figura 4.1.

Outra propriedade importante está relacionada à manutenção do produto interno entre dois vetores quando a transformada de *Householder* é aplicada previamente aos mesmos. A equação (4.2) mostra o funcionamento desta propriedade.

$$(\mathbf{Qx})^H(\mathbf{Qy}) = \mathbf{x}^H\mathbf{Q}^H\mathbf{Qy} = \mathbf{x}^H\mathbf{y} \tag{4.2}$$

4.2.2 Aplicação

Uma aplicação para a transformada de *Householder* é a triangularização de matrizes. Essa triangularização é realizada através de transformações ortogonais baseadas em reflexões de vetores em planos convenientes. Para que a triangularização aconteça é necessário que cada vetor coluna possua uma matriz de transformação de *Householder* ($\mathbf{Q}_i$). A equação (4.3) mostra como essas matrizes são geradas.

$$\begin{cases} \mathbf{Q}_i = \begin{bmatrix} \mathbf{I}_{i-1} & \mathbf{0}^T \\ \mathbf{0} & \bar{\mathbf{Q}}_i \end{bmatrix} \\ \bar{\mathbf{Q}}_i = \mathbf{I} - \frac{2\mathbf{u}_i\mathbf{u}_i^T}{\|\mathbf{u}_i\|^2} \end{cases} \tag{4.3}$$

onde i é o índice que determina o número da transformação. Este índice pode assumir valores entre 1 e r, sendo r o número total de tranformações.

A matriz $\mathbf{Q}$, responsável pela triangularização de uma determinada matriz, é encontrada conforme a equação (4.4).

$$\mathbf{Q} = \mathbf{Q}_r \ldots \mathbf{Q}_2\mathbf{Q}_1 \tag{4.4}$$

Cada uma das matrizes de transformação possui um vetor de *Householder* $\mathbf{u}_i$. A escolha desses vetores é feita em função do vetor de entrada $\mathbf{x}_i$ (vetor coluna da matriz que será triangularizada) de acordo com a equação (4.5).

$$\mathbf{u}_i = \mathbf{x}_i \pm \|\mathbf{x}_i\|\mathbf{e}_1 \qquad \therefore \qquad \mathbf{e}_1 = \mathbf{I}(:,1) \tag{4.5}$$

A fim de explicar como os vetores $\mathbf{x}_i$ são encontrados, considere a seguinte situação: deseja-se triangularizar a matriz $\mathbf{A}$ que possui dimensão $r \times r$, sendo r igual a 4. Segue abaixo esta matriz.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \tag{4.6}$$

Os vetores $\mathbf{x}_i$ possuem dimensão $M - i + 1$, conseqüentemente, os vetores $\mathbf{u}_i$ também terão essa dimensão. A equação (4.7) mostra como os valores de $\mathbf{x}_i$ são encontrados em função da matriz $\mathbf{A}$.

$$\mathbf{x}_i = \mathbf{A}(i : M, i) \tag{4.7}$$

Sendo assim os vetores $\mathbf{x}_i$ serão formados por:

$$\begin{cases} \mathbf{x}_1 = \begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} \end{bmatrix}^T \\ \mathbf{x}_2 = \begin{bmatrix} a_{22} & a_{32} & a_{42} \end{bmatrix}^T \\ \mathbf{x}_3 = \begin{bmatrix} a_{33} & a_{43} \end{bmatrix}^T \\ \mathbf{x}_4 = \begin{bmatrix} a_{44} \end{bmatrix}^T \end{cases} \tag{4.8}$$

Os vetores $\mathbf{u}_i$ e as matrizes de *Householder* $\mathbf{Q}_i$ devem ser encontrados, respectivamente, através da substituição dos valores encontrados em (4.8) nas equações (4.5) e (4.3). Para encontrar a matriz de transformação $\mathbf{Q}$ deve-se utilizar a equação (4.4). Sendo assim o produto $\mathbf{QA}$ será igual a uma matriz triangular.

4.3 Filtros LC

4.3.1 Filtros LCMV ótimos

Também conhecidos como filtros de *Wiener* LC [18], estes filtros são utilizados em várias aplicações onde há necessidade de impor restrições aos coeficientes de um filtro. Em outras palavras, os filtros LC são filtros que buscam a solução ótima através do menor MSE (*Minimum Square Error*) ou MOE (*Minimum Output Energy*) satisfazendo a restrição imposta.

Esta tese irá trabalhar com uma aplicação de *array* de antenas, onde deseja-se fazer uma filtragem espacial com restrição de modo a permitir na saída do sistema, somente o sinal posicionado na direção desejada. O *array* de antenas será formado por M sensores em conjunto com M filtros de comprimento um, ou seja, será uma aplicação de *beamformer* de banda estreita. A Figura 4.2 ilustra essa situação.

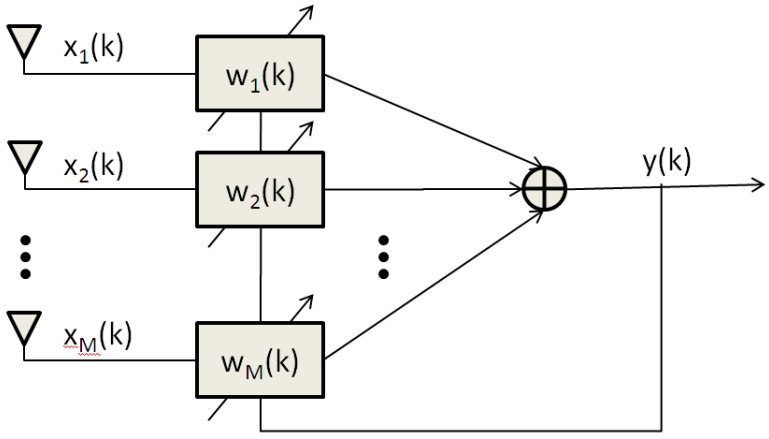


Figura 4.2: *Array* de antenas adaptativas.

A saída do *array* é representado pela vetor y cuja expressão é

$$y(k) = \mathbf{w}^H(k)\mathbf{x}(k) \tag{4.9}$$

onde:

$$\mathbf{w}(k) = \left[\begin{array}{cccc} w_1(k) & w_2(k) & \ldots & w_M(k) \end{array} \right]^T \tag{4.10}$$

$$\mathbf{x}(k) = \left[\begin{array}{cccc} x_1(k) & x_2(k) & \ldots & x_M(k) \end{array} \right]^T \tag{4.11}$$

Devido à escolha deste tipo de aplicação, a função objetivo estará relacionada ao MOE, ou seja, buscar-se-á sempre minimizar a energia média da saída do sistema. A modelagem do sistema está descrita pela equação (4.12).

$$\begin{aligned} &\xi_w = E[|y^2(k)|^2] = E[\mathbf{w}^H\mathbf{x}(k)\mathbf{x}^H(k)\mathbf{w}] = \mathbf{w}^H\mathbf{R}\mathbf{w} \\ &sujeito\ a : \mathbf{C}^H\mathbf{w} = \mathbf{f} \end{aligned} \tag{4.12}$$

onde

$\mathbf{w}$: vetor de coeficientes do filtro adaptativo de tamanho M;

$\mathbf{C}$: matriz de restrição de dimensão $M \times r$;

$\mathbf{f}$: vetor de ganho de dimensão $r \times 1$;

$\mathbf{R}$: matriz de autocorrelação do sinal de entrada x;

r : número de restrições.

Algumas situações são importantes de serem mencionadas: quando o vetor $\mathbf{f}$ é igual a $\mathbf{1}$, o filtro resultante é conhecido como MVDR (*Minimum Variance Distortionless Response*). Quando $\mathbf{C}$ tiver dimensão $M \times 1$ e todos elementos forem iguais a 1, o *array* de antenas irá permitir passar apenas o sinal que estiver incidindo a 0 grau em relação ao plano normal do *array* de antenas.

Utilizando os multiplicadores de Lagrange [18], chega-se à solução ótima ($\mathbf{w}_o$) para esse problema de otimização. A equação (4.13) mostra esse resultado.

$$\mathbf{w}_o = \mathbf{R}^{-1}\mathbf{C}(\mathbf{C}^H\mathbf{R}^{-1}\mathbf{C})^{-1}\mathbf{f} \tag{4.13}$$

A equação (4.13) descreve matematicamente a resposta ótima quando uma ou mais restrições são levadas em consideração no problema de otimização. Caso houvesse apenas uma restrição a equação (4.13) passa a ser representada por (4.14).

$$\mathbf{w}_o = \mathbf{R}^{-1}\mathbf{c}(\mathbf{c}^H\mathbf{R}^{-1}\mathbf{c})^{-1}f \tag{4.14}$$

onde

$\mathbf{w}_o$: vetor de coeficientes do filtro adaptativo de tamanho M ótimo;

$\mathbf{c}$: vetor de restrição de dimensão $M \times 1$;

f : valor escalar do ganho da restrição;

$\mathbf{R}$: matriz de autocorrelação do sinal de entrada x.

Apesar de saber como encontrar os coeficientes ótimos do filtro a partir de (4.13) e (4.14), a implementação prática dessas equações não são eficientes. Para encontrar o valor de $\mathbf{w}_o$ é necessário saber o valor de $\mathbf{R}^{-1}$, o que não é nada trivial de ser encontrado, já que é necessário estimar a matriz $\mathbf{R}$ a cada iteração e aplicar a sua inversa, tornando o algoritmo ineficiente.

A estimação da matriz $\mathbf{R}$ irá melhorar à medida que o número de amostras for aumentando. Isto resultará numa melhor estimativa, fazendo com que a convergência seja atingida pelo algoritmo implementado.

Devido à ineficiência para estimar e aplicar a inversa à matriz de autocorrelação da entrada, foram desenvolvidos os algoritmos LC adaptativos. Suas van-

tagens em relação aos filtros ótimos foram apresentados no Capítulo 3. A seguir alguns desses algoritmos serão apresentados.

4.3.2 Filtros LC adaptativos

No início da década de 70, os primeiros resultados relacionados a filtros LC adaptativos começaram a ser divulgados. Nesta época foi proposto por Frost [19] um algoritmo que estimava de forma eficiente o vetor de coeficientes $\mathbf{w}_o$.

O algoritmo projetado faz uso do método gradiente, mais especificamente, baseia-se no LMS, algoritmo amplamente utilizado em filtragem adaptativa. Maiores informações a respeito desse último algoritmo podem ser encontradas em [18].

O algoritmo desenvolvido por Frost [19] é conhecido por CLMS. Em comparação a (4.13) e (4.14), o CLMS é extremamente mais simples de ser implementado e seu custo computacional é bem mais baixo. Sua quantidade de multiplicações e armazenamentos são diretamente proporcionais ao número de coeficientes do vetor $\mathbf{w}$, enquanto o algoritmo descrito pelas equações (4.13) e (4.14) necessitam de uma quantidade de multiplicações por iteração proporcional ao cubo do número de coeficientes de $\mathbf{w}$.

De acordo com [19], o algoritmo CLMS é definido por

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \mu[\mathbf{I} - \mathbf{C}(\mathbf{C}^H\mathbf{C})^{-1}\mathbf{C}^H]\mathbf{R}_{xx}\mathbf{w}(k) + \mathbf{C}(\mathbf{C}^H\mathbf{C})^{-1}[\mathbf{f} - \mathbf{C}^H\mathbf{w}(k)] \quad (4.15)$$

Definindo o vetor $\mathbf{F}$ de dimensão $M \times 1$ como

$$\mathbf{F} \triangleq \mathbf{C}(\mathbf{C}^H\mathbf{C})^{-1}\mathbf{f} \quad (4.16)$$

e a matriz P de dimensão $M \times M$ como

$$\mathbf{P} \triangleq \mathbf{I} - \mathbf{C}(\mathbf{C}^H\mathbf{C})^{-1}\mathbf{C}^H \quad (4.17)$$

o algoritmo (4.15) pode ser reescrito como

$$\mathbf{w}(k+1) = \mathbf{P}[\mathbf{w}(k) - \mu\mathbf{R}_{xx}\mathbf{w}(k)] + \mathbf{F} \quad (4.18)$$

Analisando a equação (4.18), nota-se que para o cálculo do CLMS é necessário saber o valor da matriz de autocorrelação ($\mathbf{R}_{xx}$) do sinal de entrada $\mathbf{x}$, todavia o valor dessa matriz não é sabido *a priori*. Uma forma simples e eficiente de estimar a

matriz $\mathbf{R}_{xx}$ na k-ésina iteração é fazer o produto do k-ésimo vetor de entrada por ele mesmo, ou seja, $\mathbf{x}(k)\mathbf{x}^H(k)$. Sendo assim a equação (4.18) pode ser reescrita como

$$\mathbf{w}(k+1) = \mathbf{P}[\mathbf{w}(k) - \mu y^*(k)\mathbf{x}(k)] + \mathbf{F} \tag{4.19}$$

$$\mathbf{w}(0) = \mathbf{F} \tag{4.20}$$

onde

$\mathbf{w}$(k) : vetor de coeficientes do filtro adaptativo;

μ : fator de convergência;

y(k) : saída do filtro adaptativo;

$\mathbf{x}$(k) : entrada do filtro adaptativo;

$\mathbf{F}$: vetor responsável por manter os coeficientes do filtro adaptativo no hiperplano da restrição;

$\mathbf{P}$: matriz de projeção para o subespaço ortogonal ao subespaço da restrição.

As Figuras 4.3 e 4.4, ilustram o comportamento do vetor $\mathbf{F}$ e da matriz $\mathbf{P}$, respectivamente.

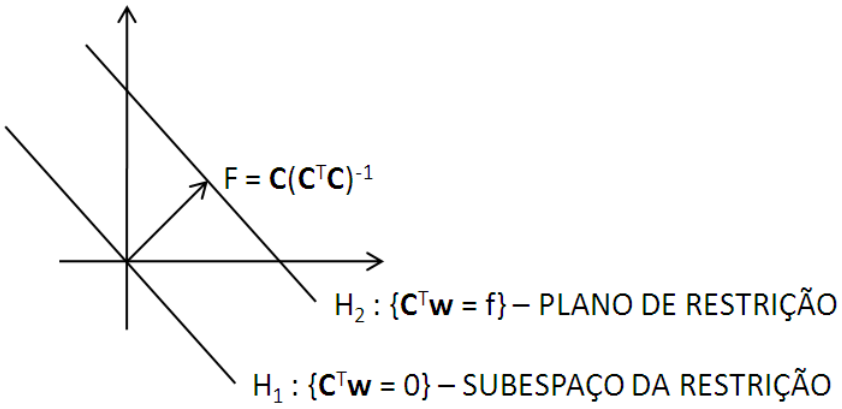


Figura 4.3: Função do vetor $\mathbf{F}$.

A Figura 4.3 possui dois hiperplanos, H_1 e H_2. O hiperplano H_1 está contido no subespaço da restrição, já o hiperplano H_2 é o plano de restrição. O papel desempenhado pelo vetor $\mathbf{F}$ é de deslocar o hiperplano H_1 de modo a este ficar igual ao hiperplano H_2.

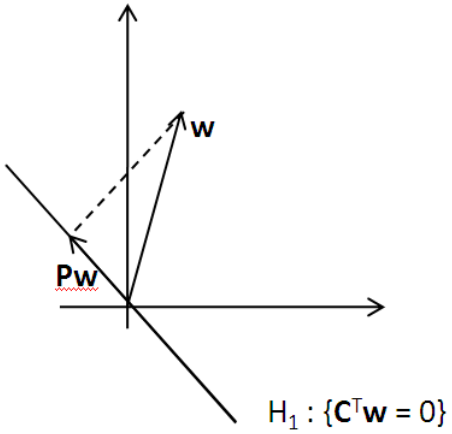


Figura 4.4: Função da matriz de projeção $\mathbf{P}$.

A Figura 4.4 mostra graficamente a transfomação que o vetor $\mathbf{w}$ sofre quando multiplicado pela matriz de projeção $\mathbf{P}$. O vetor $\mathbf{w}$ é projetado no hiperplano H_1, logo o vetor $\mathbf{Pw}$ é a projeção do vetor $\mathbf{w}$ no subespaço ortogonal a matriz de restrição $\mathbf{C}$.

A versão normalizada do algoritmo CLMS é conhecido como NCLMS. A equação (4.21) mostra como é feita a atualização dos coeficientes desse algoritmo.

$$\mathbf{w}(k+1) = \mathbf{P}[\mathbf{w}(k) - \mu \frac{y^*(k)}{\mathbf{x}^H(k)\mathbf{P}\mathbf{x}(k)}\mathbf{x}(k)] + \mathbf{F} \tag{4.21}$$

A Figura 4.5 [2] oferece uma interpretação geométrica com dois coeficientes dos algoritmos adaptativos com restrição vistos até agora (CLMS e NCLMS). A explicação dos índices da figura encontra-se a seguir.

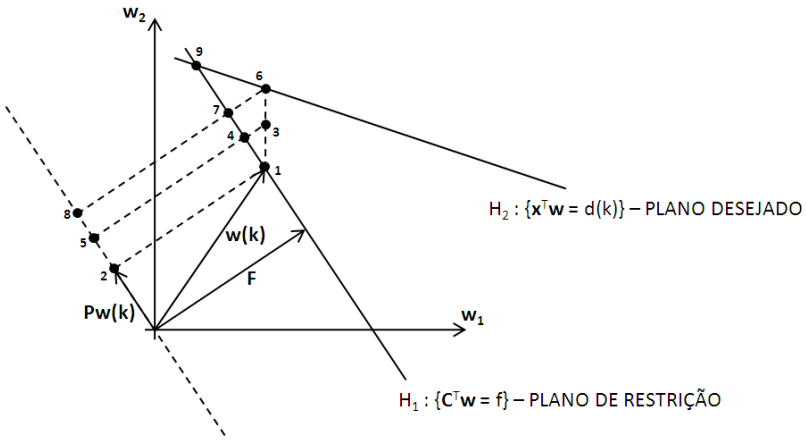


Figura 4.5: Interpretação geométrica do CLMS e NCLMS.

1. Representa o vetor $\mathbf{w}(k) = \mathbf{F} + \mathbf{P}\mathbf{w}(k)$. Neste momento o algoritmo está na k-ésima iteração e o vetor $\mathbf{w}(k)$ representa o valor encontrado pelo algoritmo CLMS ou NCLMS.

2. Representa a projeção do vetor $\mathbf{w}(k)$ no subespaço ortogonal à matriz de restrição $\mathbf{C}$.

3. Representa o vetor $\mathbf{w}(k+1)$ para o algoritmo LMS.

4. Representa o vetor $\mathbf{w}(k+1)$ para o algoritmo CLMS. Vide equação (4.20) para verificar como esse vetor é calculado.

5. Representa a projeção do vetor $\mathbf{w}(k+1)$ encontrado pelo algoritmo CLMS no subespaço ortogonal à matriz de restrição $\mathbf{C}$.

6. Representa o vetor $\mathbf{w}(k+1)$ para o algoritmo NLMS.

7. Representa a projeção em H_1 do vetor $\mathbf{w}(k+1)$ encontrado pelo algoritmo NLMS;

8. Representa a projeção do vetor $\mathbf{w}(k+1)$ encontrado pelo algoritmo NCLMS no subespaço ortogonal à matriz de restrição $\mathbf{C}$.

9. Valor desejado para os coeficiente de $\mathbf{w}$. Neste momento há o cruzamento dos hiperplanos, logo o resultado desejado é atingido e simultaneamente a restrição é satisfeita. Esse valor desejado é apenas atingido quando não houver erro de observação e nem de *undermodeling*.

A Figura 4.5 mostra como os algoritmos sem restrição, com restrição e suas projeções estão relacionados. Analisando esta figura percebe-se que os valores encontrados para $\mathbf{w}(k)$, utilizando os algoritmos CLMS e NCLMS, sempre pertencem ao hiperplano H_1, conseqüentemente podem ser decompostos utilizando os vetores ortogonais $\mathbf{F}$ e $\mathbf{P}\mathbf{w}$(k). Outra análise que pode ser feita está relacionada com o algoritmo NLMS: os valores de $\mathbf{w}(k)$ deste algoritmo estão sempre contidos no hiperplano H_2 desde que o fator de convergência seja igual a 1.

4.3.3 GSC (*Generalized Sidelobe Canceler*)

O termo GSC foi proposto por Griffiths e Jim [**?**] no início da década de 80. Este algoritmo é uma implementação alternativa para filtros LC adaptativos, explicados na Subseção 4.3.2. A Figura 4.6 ilustra o modelo utilizado no GSC.

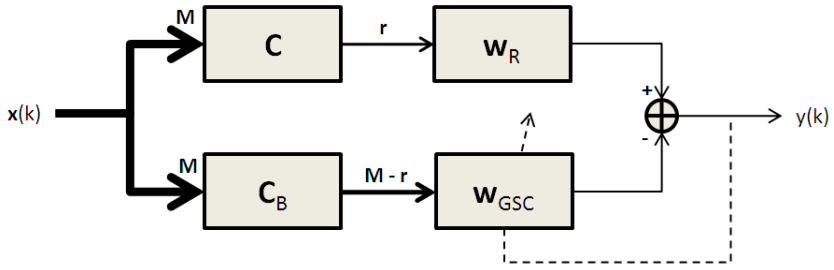


Figura 4.6: Modelo utilizado no GSC.

No modelo GSC, a adaptação do algoritmo é feita utilizando algoritmos adaptativos sem restrição, enquanto os filtros LC adaptativos levam em consideração as restrições para fazer a adaptação. A idéia do GSC consiste, inicialmente, em multiplicar o vetor de entrada $\mathbf{x}(k)$ por uma matriz de transformação $\mathbf{T}$ de dimensão $M \times M$. Abaixo segue a definição da matriz $\mathbf{T}$.

$$\mathbf{T} = \left[\begin{array}{cc} \mathbf{C} & \mathbf{C}_B \end{array} \right] \tag{4.22}$$

onde

$\mathbf{C}$: matriz de restrição de dimensão $M \times r$;

$\mathbf{C}_B$: matriz de bloqueio de dimensão $M \times M - r$.

A matriz de bloqueio $\mathbf{C}_B$ deve pertencer ao subespaço ortogonal ao subespaço da matriz de restrição $\mathbf{C}$, ou seja,

$$\mathbf{C}_B^H \mathbf{C} = \mathbf{0}. \tag{4.23}$$

A matriz $\mathbf{0}$ em (4.23) possui dimensão $(M - r) \times r$. É importante ressaltar também que sempre o valor de M será sempre maior que r, ou seja, $M > r$. Em relação à matriz $\mathbf{T}$, é importante frisar que o posto dessa matriz será sempre igual a M e a sua matriz inversa, ou seja, $\mathbf{T}^{-1}$ sempre existirá pois seu determinante é diferente de zero.

O resultado da multiplicação entre $\mathbf{x}(k)$ e $\mathbf{T}$ será representado por

$$\bar{\mathbf{x}}(k) = \begin{bmatrix} \bar{\mathbf{x}}_R(k) \\ \bar{\mathbf{x}}_B(k) \end{bmatrix} = \begin{bmatrix} \mathbf{C}^H\mathbf{x}(k) \\ \mathbf{C}_B^H\mathbf{x}(k) \end{bmatrix} = \mathbf{T}^H\mathbf{x}(k). \tag{4.24}$$

Os comprimentos dos vetores $\bar{\mathbf{x}}_R(k)$ e $\bar{\mathbf{x}}_B(k)$ são, respectivamente, r e $(M-r)$. O vetor $\bar{\mathbf{x}}_R$ está contido no subespaço da matriz restrição, enquanto $\bar{\mathbf{x}}_B(k)$ encontra-se no subespaço ortogonal. Esses vetores são utilizados como entrada dos filtros $\mathbf{w}_R$ e $\mathbf{w}_{GSC}$, vide Figura 4.6. Desta forma, a saída do sistema, $y(k)$, é igual a

$$\begin{aligned} y(k) &= \mathbf{w}_R^H\bar{\mathbf{x}}_R(k) - \mathbf{w}_{GSC}^H\bar{\mathbf{x}}_B(k) \\ &= \mathbf{w}_R^H\mathbf{C}^H\mathbf{x}(k) - \mathbf{w}_{GSC}^H\mathbf{C}_B^H\mathbf{x}(k) \\ &= (\mathbf{C}\mathbf{w}_R - \mathbf{C}_B\mathbf{w}_{GSC})^H\mathbf{x}(k) \\ &= (\mathbf{T}\mathbf{w})^H\mathbf{x}(k) \\ &= \bar{\mathbf{w}}^H\mathbf{x}(k) \end{aligned} \tag{4.25}$$

onde

$$\mathbf{w} = \begin{bmatrix} \mathbf{w}_R \\ -\mathbf{w}_{GSC} \end{bmatrix} \tag{4.26}$$

$$\bar{\mathbf{w}} = \mathbf{C}\mathbf{w}_R - \mathbf{C}_B\mathbf{w}_{GSC} = \mathbf{T}\mathbf{w} \tag{4.27}$$

A restrição linear será satisfeita se $\mathbf{C}^H\bar{\mathbf{w}} = \mathbf{f}$, todavia $\mathbf{C}^H\mathbf{C}_B = 0$, logo a condição para que a restrição seja satisfeita pode ser expressa por

$$\mathbf{C}^H\bar{\mathbf{w}} = \mathbf{C}^H\mathbf{C}\mathbf{w}_R = \mathbf{f}. \tag{4.28}$$

A partir da equação (4.28) pode-se concluir que o vetor $\mathbf{w}_R$ é fixo e dado por (4.29).

$$\mathbf{w}_R = (\mathbf{C}^H\mathbf{C})^{-1}\mathbf{f} \tag{4.29}$$

Substituindo a equação (4.29) em (4.27) tem-se

$$\mathbf{w} = \mathbf{C}^H(\mathbf{C}^H\mathbf{C})^{-1}\mathbf{f} - \mathbf{C}_B^H\mathbf{w}_{GSC} = \mathbf{F} - \mathbf{C}_B^H\mathbf{w}_{GSC} \tag{4.30}$$

De acordo com o resultado encontrado em (4.30), o diagrama de blocos da Figura 4.6 pode ser representado de forma mais compacta conforme a Figura 4.7.

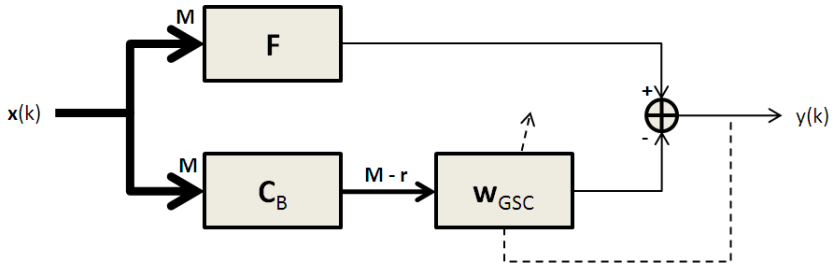


Figura 4.7: Modelo utilizado no GSC mais compacto.

É importante ressaltar que o vetor $\mathbf{w}_{GSC}$ não é afetado pelas restrições, logo sua adaptação poderá ser realizada por algoritmos sem restrição, com o objetivo de reduzir as interferências que chegam no subespaço nulo a matriz $\mathbf{C}$.

Conforme dito anteriormente, o GSC é uma forma alternativa de implementação dos filtros LC adaptativos desenvolvidos por Frost. Sua complexidade está associada diretamente com a forma com que a matriz $\mathbf{C}_B$ é encontrada. Esta matriz pode ser encontrada através do mais diversos tipos de decomposição, como por exemplo, SVD (*Singular Value Decomposition*), todavia essas decomposições nem sempre são eficientes [2]. Devido a isto, os algoritmos adaptativos, que utilizam a transformada de *Householder*, são formas eficientes de resolver problemas de otimização com restrição. A seguir esses algoritmos serão apresentados.

4.3.4 Algoritmos com restrição baseados na transformada de *Householder*

Os algoritmos com restrição baseados na transformada de Householder foram desenvolvidos no final da década de 90 [17]. Estes algoritmos fazem uso da transformada de Householder em conjunto com algoritmos adaptativos como LMS [2] e QN (*Quasi-Newton*) [20] de modo a resolverem problemas de otimização com restrição.

O objetivo desta tese não é verificar as vantagens e desvantagens dos algoritmos com restrição baseados na transformada de *Householder* e sim verificar o comportamento do mesmo quando for utilizado em um *hardware* ou *software* puramente CORDIC.

No decorrer desta seção será explicado como os algoritmos HCLMS e NHCLMS funcionam. Todo estudo realizado nesta tese será baseado nestes dois algoritmos,

devido a isto apenas estes serão apresentados.

O HCLMS, primeiro algoritmo apresentado nesta seção, é deduzido a partir do CLMS (4.20). A idéia central para sua dedução é realizar uma rotação no vetor de coeficientes $\mathbf{w}(k)$ de modo a permitir que o vetor transformado ($\bar{\mathbf{w}}$) não seja perturbado numa direção que não seja excitada por $\mathbf{Px}(k)$. Essa rotação é realizada pela matriz ortogonal $\mathbf{Q}$, cuja função está descrita em (4.31).

$$\bar{\mathbf{w}}(k) = \mathbf{Q}\mathbf{w}(k) \tag{4.31}$$

A Figura 4.8 [2] ilustra a transformação do vetor $\mathbf{w}$ para uma situação bidimensional. Perceba que o vetor transformado é rotacionado por um ângulo α, de modo que o eixo $\bar{w}_1$ fique num subespaço ortogonal a $\mathbf{Px}(k)$, enquanto $\bar{w}_2$ pertence ao mesmo subespaço. Toda adaptação será realizada em $\bar{w}_2$, já que o valor desejado para $\bar{w}_1$ foi atingido com a rotação realizada pela matriz $\mathbf{Q}$. Essa figura ilustra uma situação bidimensional, onde há apenas uma restrição. Isto pode ser comprovado pelo fato de $\bar{w}_1$ não sofrer nenhuma alteração em relação ao seu valor original. Sendo assim a adaptação será feita apenas em $\bar{w}_2$, ou seja, apenas um coeficiente do vetor $\bar{\mathbf{w}}$ necessitará ser atualizado. Expandindo esta análise para dimensões mais elevadas, pode-se afirmar que o número de coeficientes do vetor $\bar{\mathbf{w}}$ que necessitam ser atualizados é igual a comprimento do vetor $\bar{\mathbf{w}}$ menos o número de restrições.

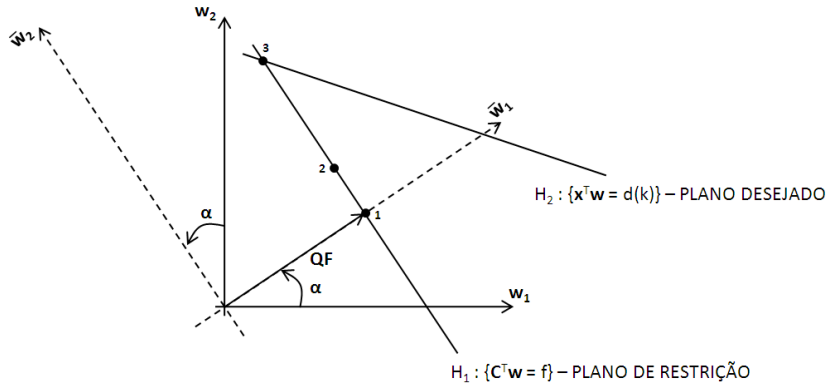


Figura 4.8: Interpretação geométrica do HCLMS e NHCLMS.

A seguir estão as explicações do números representados na Figura 4.8:

1. Representa o vetor inicial $\bar{\mathbf{w}}(0) = \mathbf{QF}$.

2. Representa o vetor $\mathbf{w}(k+1)$ para o algoritmo HCLMS.

3. Representa o vetor $\mathbf{w}(k+1)$ para o algoritmo NHCLMS se o fator de convergência for igual a 1.

A matriz $\mathbf{Q}$ é formada por transformações de *Householder* sucessivas aplicadas às r colunas da matriz $\mathbf{CL}$, onde $\mathbf{L}$ é igual a raiz quadrada da matriz $(\mathbf{C}^H\mathbf{C})^{-1}$, ou seja, $\mathbf{L}\mathbf{L}^H = (\mathbf{C}^H\mathbf{C})^{-1}$.

As equações (4.3) e (4.4) explicam como a matriz $\mathbf{Q}$ é calculada. É importante ressaltar que os valores para os vetores de *Householder* ($\mathbf{u}_i$) são calculados utilizando as equações (4.5) e (4.7) para $\mathbf{A} = \mathbf{CL}$.

Para os algoritmos com restrição baseados na transformada de *Householder* serem utilizados, é necessário que a matriz $\mathbf{Q}$ satisfaça algumas condições que estão listadas a seguir:

$$\mathbf{Q}\mathbf{Q}^H = \mathbf{Q}^H\mathbf{Q} = \mathbf{I} \tag{4.32}$$

$$\bar{\mathbf{C}}(\bar{\mathbf{C}}^H\bar{\mathbf{C}})^{-1}\bar{\mathbf{C}}^H = \begin{bmatrix} \mathbf{I}_{r\times r} & \mathbf{0}_{r\times M-r} \\ \mathbf{0}_{M-r\times r} & \mathbf{0}_{M-r\times M-r} \end{bmatrix} \tag{4.33}$$

Conforme visto na Seção 4.2, a matriz $\mathbf{Q}$ é uma matriz unitária, logo a condição descrita pela equação (4.32) é satisfeita. Já a condição (4.33) é satisfeita devido à forma como a matriz $\mathbf{Q}$ é formada (equação (4.4)).

A matriz $\bar{\mathbf{C}} = \mathbf{QC}$ quando multiplicada pelo vetor $\bar{\mathbf{w}}$ dá como resultado o vetor $\mathbf{f}$, ou seja, a restrição é mantida mesmo com a transformação. A equação (4.34) comprova essa afirmação.

$$\bar{\mathbf{C}}^H\bar{\mathbf{w}}(k) = \mathbf{C}^H\mathbf{Q}^H\mathbf{Q}\mathbf{w}(k) = \mathbf{C}^H\mathbf{I}\mathbf{w}(k) = \mathbf{C}^H\mathbf{w}(k) = \mathbf{f}. \tag{4.34}$$

Já a matriz de projeção transformada ($\bar{\mathbf{P}}$) é igual a

$$\bar{\mathbf{P}} = \mathbf{QPQ}^H = \mathbf{I} - \bar{\mathbf{C}}(\bar{\mathbf{C}}^H\bar{\mathbf{C}})^{-1}\bar{\mathbf{C}}^H = \begin{bmatrix} \mathbf{0}_{r\times r} & \mathbf{0}_{r\times M-r} \\ \mathbf{0}_{M-r\times r} & \mathbf{I}_{M-r\times M-r} \end{bmatrix}. \tag{4.35}$$

Caso o vetor $\bar{\mathbf{w}}(0)$ seja inicializado como

$$\bar{\mathbf{w}}(0) = \bar{\mathbf{C}}(\bar{\mathbf{C}}^H\bar{\mathbf{C}})^{-1}\mathbf{f} = \mathbf{QF} \tag{4.36}$$

os primeiros r elementos de $\bar{\mathbf{w}}$ não necessitarão ser atualizados no processo de adaptação. Esses r elementos a partir de agora serão denominados $\bar{\mathbf{w}}_r(0)$.

Multiplicando a equação (4.20) por $\mathbf{Q}$ é possível deduzir o algoritmo HCLMS. Segue abaixo essa dedução.

$$
\begin{aligned}
\bar{\mathbf{w}}(k+1) = & \\
&= \mathbf{Q}\mathbf{w}(k+1) \\
&= \mathbf{Q}\{\mathbf{P}[\mathbf{w}(k) - \mu y^*(k)\mathbf{x}(k)] + \mathbf{F}\} \\
&= [\mathbf{Q}\mathbf{P}\mathbf{Q}^H][\mathbf{Q}\mathbf{w}(k)] - \mu y^*(k)[\mathbf{Q}\mathbf{P}\mathbf{Q}^H][\mathbf{Q}\mathbf{x}(k)] + \mathbf{Q}\mathbf{F} \\
&= \begin{bmatrix} \mathbf{0}_{r\times r} & \mathbf{0}_{r\times M-r} \\ \mathbf{0}_{M-r\times r} & \mathbf{I}_{M-r\times M-r} \end{bmatrix} \bar{\mathbf{w}}(k) - \mu y^*(k) \begin{bmatrix} \mathbf{0}_{r\times r} & \mathbf{0}_{r\times M-r} \\ \mathbf{0}_{M-r\times r} & \mathbf{I}_{M-r\times M-r} \end{bmatrix} \bar{\mathbf{x}}(k) + \\
&+ \begin{bmatrix} \bar{\mathbf{w}}_r(0) \\ \mathbf{0}_{M-r\times 1} \end{bmatrix} \\
&= \begin{bmatrix} \bar{\mathbf{w}}_r(0) \\ \bar{\mathbf{w}}_L(k) \end{bmatrix} - \mu y^*(k) \begin{bmatrix} \mathbf{0}_{r\times 1} \\ \bar{\mathbf{x}}_L(k) \end{bmatrix}.
\end{aligned}
\tag{4.37}
$$

Sendo assim o algoritmo HCLMS é representado por

$$
\begin{bmatrix} \bar{\mathbf{w}}_r(0) \\ \bar{\mathbf{w}}_L(k+1) \end{bmatrix} = \begin{bmatrix} \bar{\mathbf{w}}_r(0) \\ \bar{\mathbf{w}}_L(k) \end{bmatrix} - \mu y^*(k) \begin{bmatrix} \mathbf{0}_{r\times 1} \\ \bar{\mathbf{x}}_L(k) \end{bmatrix} \tag{4.38}
$$

onde

$\bar{\mathbf{w}}_L(k)$: parte do vetor $\bar{\mathbf{w}}$ que será adaptado. Este vetor é formado pelos $M-r$ últimos elementos de $\bar{\mathbf{w}}$;

$\bar{\mathbf{w}}_r(0)$: parte do vetor $\bar{\mathbf{w}}$ que não será adaptado. Este vetor é formado pelos r primeiros elementos de $\bar{\mathbf{w}}$.

Já o algoritmo NHCLMS é derivado do NCLMS, que está descrito pela equação (4.21). A equação (4.39) descreve o algoritmo NHCLMS.

$$
\begin{bmatrix} \bar{\mathbf{w}}_r(0) \\ \bar{\mathbf{w}}_L(k+1) \end{bmatrix} = \begin{bmatrix} \bar{\mathbf{w}}_r(0) \\ \bar{\mathbf{w}}_L(k) \end{bmatrix} - \mu \frac{y^*(k)}{\bar{\mathbf{x}}_L^H(k)\bar{\mathbf{x}}_L(k)} \begin{bmatrix} \mathbf{0}_{r\times 1} \\ \bar{\mathbf{x}}_L(k) \end{bmatrix} \tag{4.39}
$$

4.4 Considerações parciais

O objetivo desta tese é estudar o comportamento dos algoritmos HCLMS e NHCLMS em uma máquina de CORDIC. Esses algoritmos foram escolhidos, pois são implementações muito eficientes de filtros LC adaptativos. Em [2], é apresentado um estudo sobre complexidade que evidencia a eficiência desses algoritmos. Esse tipo de estudo não foi realizado nesta tese, pois não é o foco da mesma.

Vale a pena ressaltar que o principal fator para redução de complexidade está no fato deles utilizarem a transformada de *Householder*, que permite a redução das dimensões do subespaço onde a adaptação será realizada.

Os algoritmos adaptativos baseados na transformada de *Householder* podem ser comparados ao GSC. A Figura 4.9 mostra como os algoritmos HC (*Householder Constrained*) podem ser colocados sob a perspectiva do modelo GSC (vide Figura 4.7).

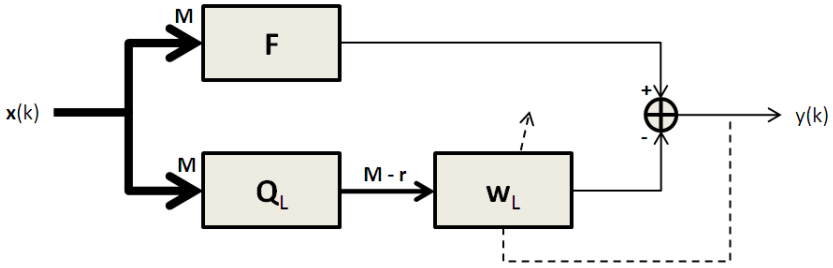


Figura 4.9: Algoritmos HC em modelo GSC.

É importante frisar que a matriz $\mathbf{Q}_L$ é unitária, o que diferencia os algoritmos HC dos demais no cálculo da matriz bloqueio [**?**]. Essa característica faz com que o algoritmo seja mais robusto [21], o que o torna mais interessante para implementações em tempo real.

O próximo capítulo apresentará a junção dos algoritmos HC com o CORDIC.

Capítulo 5

Algoritmos HC com CORDIC

5.1 Introdução

Uma vez apresentados os assuntos CORDIC e algoritmos HC, torna-se necessário mostrar como estes algoritmos serão unidos. O objetivo deste capítulo é implementar um algoritmo capaz de utilizar as vantagens de cada um dos algoritmos apresentados nos Capítulos 2, 3 e 4.

Conforme mencionado no Capítulo 2, os algoritmos HC são implementados utilizando apenas as operações de adição, multiplicação, divisão e raiz quadrada. Devido a isso, as funções CORDIC, desenvolvidas no Capítulo 2, serão utilizadas substituindo as operações matemáticas existentes nos algoritmos HC.

A junção desses algoritmos é bem trivial, já que não há necessidade de analisar a região de convergência de cada operação CORDIC. Isto só foi possível devido a introdução dos algoritmos de adaptação às funções CORDIC apresentados no Capítulo 2.

Este capítulo será dividido da seguinte maneira: primeiramente, será explicado como os algoritmos HC com precisão variável foram implementados. Todas funções CORDIC utilizadas nos algoritmos HC com precisão variável serão apresentadas em seguida. Depois, alguns testes serão realizados a fim de comprovar o correto funcionamento das funções CORDIC. E, por fim, os resultados serão apresentados.

5.2 Implementação

A construção deste algoritmo será feita da seguinte forma: todas operações matemáticas realizadas nos algoritmos HC serão substituídas pelas suas funções CORDIC correspondentes, por exemplo, as operações de multiplicação devem ser substituídas pela função *mult_c_a* (vide a Subseção 2.11.3.1).

A fim de entender como foi feita a implementação dos algoritmos HC por funções CORDIC, é necessário saber os tipos de operações (matricial, vetorial ou escalar) que são realizadas nestes algoritmos. Na Tabela 5.1 encontra-se o pseudocódigo para implementação do algoritmo HCLMS sem utilizar as funções CORDIC desenvolvido em [2]. Este código está escrito de modo a considerar os valores de $\bar{\mathbf{w}}$, $\bar{\mathbf{x}}$ e y complexos.

Tabela 5.1: Algoritmo HCLMS.

dado $\mathbf{x}$(k), $\mathbf{C}$, $\mathbf{f}$, $\mathbf{Q}$ e μ (passo)

Inicializar:

$\bar{\mathbf{w}}(0) = \mathbf{QC}(\mathbf{C}^H\mathbf{C})^{-1}\mathbf{f}$;

for k $= 0,\ 1,\ 2,\ \ldots$

$\bar{\mathbf{x}}(k) = \mathbf{Q}\mathbf{x}(k)$;

$\bar{\mathbf{x}}_L = MN - r$ últimos elementos de $\bar{\mathbf{x}}(k)$;

$$\bar{\mathbf{w}}(k) = \begin{bmatrix} \bar{\mathbf{w}}_r(0) \\ \bar{\mathbf{w}}_L(k) \end{bmatrix};$$

$y(k) = \bar{\mathbf{w}}^H(k)\bar{\mathbf{x}}(k)$;

$\bar{\mathbf{w}}_L(k+1) = \bar{\mathbf{w}}_L(k) - \mu y^*(k)\bar{\mathbf{x}}(k)$;

end for

Todavia, para executar o algoritmo mostrado na Tabela 5.1, é necessário que a matriz $\mathbf{Q}$ seja conhecida, já que esta matriz faz parte dos cálculos de $\bar{\mathbf{w}}(0)$ e $\bar{\mathbf{x}}(k)$. Calcular a matriz $\mathbf{Q}$ e depois multiplicá-la por $\mathbf{C}(\mathbf{C}^H\mathbf{C})^{-1}\mathbf{f}$ ou $\mathbf{x}(k)$ não é uma forma eficiente de implementar o algoritmo HCLMS.

A melhor forma de implementar as transformações do algoritmo descrito pela Tabela 5.1 é utilizando os vetores de *Householder*, $\mathbf{u}_i$(vide equação (4.5)). A seguir estão os pseudocódigos que foram utilizados na implementação dos algoritmos HC. O algoritmo descrito pela Tabela 5.2 mostra como os vetores de *Householder* são

encontrados, já o algoritmo descrito pela Tabela 5.3 mostra como a transformação do vetor $\mathbf{x}(k)$ é realizada a partir dos vetores de *Householder*. Os pseudocódigos apresentados pelas Tabelas 5.1, 5.2 e 5.3 foram baseados nos pseudocódigos de [2].

Tabela 5.2: Algoritmo para determinação dos vetores de *Householder*.

dado $\mathbf{A} = \mathbf{CL}$;

Inicializar:

$\mathbf{U} = \mathbf{0}_{MN \times r}$;

for i = 1:r

$\mathbf{x} = \mathbf{A}(i : MN, i)$;

$\mathbf{e}_1 = \begin{bmatrix} \mathbf{1}_{i \times 1} \\ \mathbf{0}_{(MN-i) \times 1} \end{bmatrix}$;

$\mathbf{u}_i = sign(\mathbf{x}(1))\|\mathbf{x}\|\mathbf{e}_1 + \mathbf{x}$;

$\mathbf{u}_i = \mathbf{u}_i / \|\mathbf{u}_i\|$;

$\mathbf{A}(i : MN, i : r) = \mathbf{A}(i : MN, i : r) - 2\mathbf{u}_i(\mathbf{u}_i^T \mathbf{A}(i : MN, i : r))$;

$\mathbf{U}(i : MN, i) = \mathbf{u}_i$;

end for

Tabela 5.3: Algoritmo para transformação do vetor de entrada $\mathbf{x}$(k), ou seja, $\bar{x}(k) = Qx(k)$.

dado $\mathbf{x}_j = \mathbf{x}(k)$ e $\mathbf{U}$;

for i = 1:r

$\mathbf{x}_j(i : MN) = \mathbf{x}_j(i : MN) - 2\mathbf{U}(i : MN, i)(\mathbf{U}^T(i : MN, i)\mathbf{x}_j(i : MN))$;

end for

$\mathbf{x}(\bar{k}) = \mathbf{x}_j$;

end for

O algoritmo de determinação dos vetores de *Householder* deve ser executado antes do algoritmo HCLMS, já o algoritmo de transformação do vetor de entrada deve ser executado durante o algoritmo HCLMS.

Em relação ao algoritmo de determinação dos vetores de *Householder*, vide Tabela 5.2, é importante fazer as seguintes observações:

- a matriz $\mathbf{U}$ será responsável por armazenar todos vetores de *Householder*;

- o valor da matriz $\mathbf{A}$ no final do algoritmo será igual $\mathbf{QCL}$, logo $\bar{\mathbf{w}}(0) = \mathbf{AL}^T\mathbf{f}$.

Na Tabela 5.4 está o algoritmo NHCLMS. Sua única diferença para o HCLMS está na normalização que ocorre no processo de adaptação do vetor $\bar{\mathbf{w}}_L(k)$.

Tabela 5.4: Algoritmo NHCLMS.

dado $\mathbf{x}$(k), $\mathbf{C}$, $\mathbf{f}$, $\mathbf{Q}$ e μ (passo)

Inicializar:

$\bar{\mathbf{w}}(0) = \mathbf{QC}(\mathbf{C}^H\mathbf{C})^{-1}\mathbf{f}$;

for k $= 0,\ 1,\ 2,\ \ldots$

$\bar{\mathbf{x}}(k) = \mathbf{Q}\mathbf{x}(k)$;

$\bar{\mathbf{x}}_L = MN - r$ últimos elementos de $\bar{\mathbf{x}}(k)$;

$$\bar{\mathbf{w}}(k) = \begin{bmatrix} \bar{\mathbf{w}}_r(0) \\ \bar{\mathbf{w}}_L(k) \end{bmatrix};$$

$y(k) = \bar{\mathbf{w}}^H(k)\bar{\mathbf{x}}(k)$;

$\bar{\mathbf{w}}_L(k+1) = \bar{\mathbf{w}}_L(k) - \mu\frac{y^*(k)}{\bar{\mathbf{x}}_L^H(k)\bar{\mathbf{x}}_L(k)}\bar{\mathbf{x}}(k)$;

end for

Todos algoritmos mostrados até agora neste capítulo terão suas operações matemáticas substituídas pelas funções CORDIC desenvolvidas no Capítulo 2 em uma aplicação de *beamforming* adaptativo. Com isso será possível verificar o comportamento dos algoritmos HC quando trabalham com um *hardware* de precisão variável para este tipo de aplicação. Os resultados desta simulação podem ser vistos no Capítulo 6.

Este capítulo está mais preocupado em verificar o correto funcionamento das funções CORDIC quando aplicadas aos algoritmos apresentados na Tabelas 5.1 a 5.4 antes de utilizar os mesmos numa aplicação de *beamformer* adaptativo.

Analisando os algoritmos das Tabelas 5.1 a 5.4, nota-se que há uma predominância de operações de adição e multiplicação, já que há diversas situações de multiplicação entre vetores, ou entre vetor e escalar. Como as funções desenvolvidas no Capítulo 2 operam apenas com escalares, foi necessário adaptá-las para os tipos de operação utilizados nos algoritmos das Tabelas 5.1 a 5.4.

Foram criadas 9 funções no *Matlab* a fim de utilizar os algoritmos HC com precisão variável. Cada uma dessas funções está descrita abaixo:

- **mat_mult_c** ⇒ função que realiza a multiplicação de matrizes ou vetores por matrizes ou vetores. Esta função faz uso das funções *adicao_c* e *mult_c_a*, que foram apresentadas no Capítulo 2. É importante ressaltar que a multiplicação de matrizes ou vetores complexos pode ser realizada com essa função, pois a mesma faz todo tratamento de números complexos logo não faz uso da função de multiplicação de números complexos do *Matlab*;

- **mat_adicao_c** ⇒ função responsável por somar matrizes ou vetores complexos por matrizes ou vetores complexos. Faz uso apenas da função *adicao_c*;

- **mat_div_c** ⇒ função que realiza a divisão de matrizes ou vetores complexos por um escalar real. Faz uso da função *div_c_a*.

- **escalar_mult_vec_c** ⇒ função que realizada a multiplicação de matrizes ou vetores complexos por escalares reais. Faz uso apenas da função *mult_c_a*;

- **norm_c** ⇒ função responsável por encontrar a norma de um vetor. Esta função faz uso das funções *sqrt_c_a* e *mat_mult_c*.

- **house_vectors_c** ⇒ função que utiliza operações com precisão variável (funções CORDIC) para descobrir os vetores de *Householder*. A função *house_vectors_c* desempenha o mesmo papel que o pseudocódigo da Tabela 5.2. Esta função faz uso das funções *mult_c_a*, *escalar_mult_vec_c*, *mat_adicao_c*, *mat_div_c* e *norm_c*.

- **transf_x_c** ⇒ função responsável por multiplicar o vetor $\mathbf{x}$ pela matriz de transformação $\mathbf{Q}$ utilizando operações com precisão variável. A função *transf_x_c* desempenha o mesmo papel que o pseudocódigo da Tabela 5.3. Esta função faz uso das funções *mult_c_a*, *escalar_mult_vec_c* e *mat_adicao_c*.

- **HCLMS_C** ⇒ função responsável por implementar o algoritmo HCLMS com precisão variável. A função *hclms_c* desempenha o mesmo papel que o pseudocódigo da Tabela 5.1. Esta função faz uso de todas funções descritas acima.

- **NHCLMS_C** ⇒ função responsável por implementar o algoritmo NHCLMS com precisão variável. A função *nhclms_c* desempenha o mesmo papel que o

pseudocódigo da Tabela 5.4. Esta função faz uso de todas funções descritas acima com exceção da *hclms_c*.

Para todas funções acima, é possível escolher a quantidade de *bits* que será utilizado para calcular o resultado da função. A escolha da precisão em *bits* vai determinar o número de rotações CORDIC que será realizado em cada uma das função citadas acima.

5.2.1 Região de convergência

Conforme dito anteriomente, em relação às funções CORDIC, não será necessário se preocupar com a convergência; todavia as funções HC podem não convergir. Para que as funções HC possam ser utilizadas, é necessário que as condições (4.32) e (4.33) sejam satisfeitas.

A seguir serão feitos alguns testes a fim de verificar o correto funcionamento das funções CORDIC. Esses testes servirão para mostrar que as funções CORDIC estão desempenhando o mesmo papel das funções do *Matlab*.

5.3 Testes das funções CORDIC

Os testes que serão realizados nesta seção estão associados às funções CORDIC *transf_x_c* e *house_vectors_c*. Essa funções foram escolhidas, pois, para encontrar seus resultados, é necessário que as funções *mult_c_a*, *escalar_mult_vec_c*, *mat_adicao_c*, *mat_div_c* e *norm_c* estejam funcionando corretamente.

Caso as funções *transf_x_c* e *house_vectors_c* estejam realizando seus cálculos corretamente, conseqüentemente os algoritmos HC com precisão variável estarão funcionando.

5.3.1 Função *transf_x_c*

Para verificar o funcionamento da função *transf_x_c*, o seu resultado será comparado com a função *transf_x*. Essa função foi criada exclusivamente para este teste. Esta possui o mesmo pseudocódigo que a função *transf_x_c*, todavia não utiliza as funções CORDIC para realizar as operações aritméticas. Todas operações aritméticas realizadas por esta função serão feitas utilizando as operações do *Matlab*.

A fim de comprovar o funcionamento do algoritmo *transf_x_c*, considere a seguinte situação: o vetor de entrada $\mathbf{x}$ é igual a

$$\mathbf{x} = \begin{bmatrix} 0,4103 \\ 0,8936 \end{bmatrix} \tag{5.1}$$

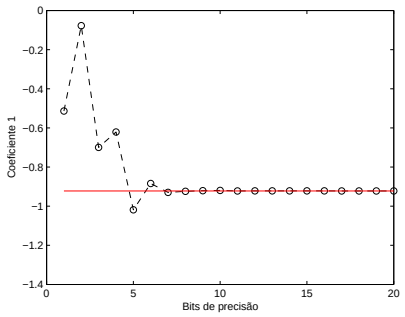
e o vetor de entrada $\mathbf{U}$ (para o exemplo $\mathbf{U}$ será um vetor, todavia este pode ser uma matriz) é igual a

$$\mathbf{U} = \begin{bmatrix} 0,9239 \\ 0,3827 \end{bmatrix}. \tag{5.2}$$

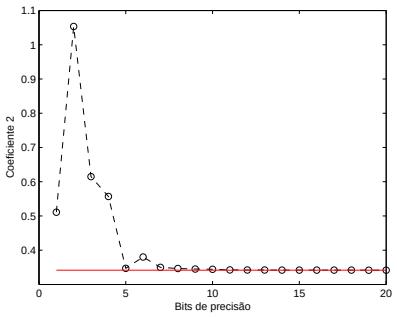
A resposta encontrada pelo algoritmo *transf_x* é igual a

$$\bar{\mathbf{x}} = \begin{bmatrix} -0,9220 \\ 0,3417 \end{bmatrix}. \tag{5.3}$$

As Figuras 5.1(a) e 5.1(b) mostram o comportamento dos coeficientes de $\mathbf{x}$ transformados. A linha contínua mostra o valor encontrado pelo algoritmo *transf_x*, já a linha pontilhada indica o comportamento do algoritmo *transf_x_c* em função do número de *bits* de precisão.



(a) Comportamento do coeficiente 1 após a transformação.



(b) Comportamento do coeficiente 2 após a transformação.

Figura 5.1: Comportamento dos coeficientes de $\mathbf{x}$ após a transformação.

Analisando as Figuras 5.1(a) e 5.1(b), percebe-se que a função *transf_x_c* está funcionanado corretamente, pois, à medida que o número de *bits* de precisão aumenta, mais próximo do valor desejado à resposta desse algoritmo se aproxima. De acordo com as figuras acima, percebe-se que, com 12 *bits*, a função *transf_x_c* está funcionando de maneira similar à função *transf_x*.

É importante ressaltar que, o resultado mostrado pelas Figuras 5.1(a) e 5.1(b) é válido apenas quando as entradas da função *transf_x_c* são (5.1) e (5.2). A fim

de fazer uma análise minuciosa no algoritmo *transf_x_c*, foram gerados 1000 vetores de comprimento igual a 4 que foram transformados pela função *transf_x_c*. Os *bits* de precisão da função *transf_x_c* foram variados de 1 a 64. A Figura 5.2 mostra o comportamento do erro médio, em dB, em função da quantidade de *bits* de precisão. O erro médio é calculado através de um média aritmética da subtração do valor transformado encontrado pelo *Matlab* com o valor transformado encontrado pela função *transf_x_c*.

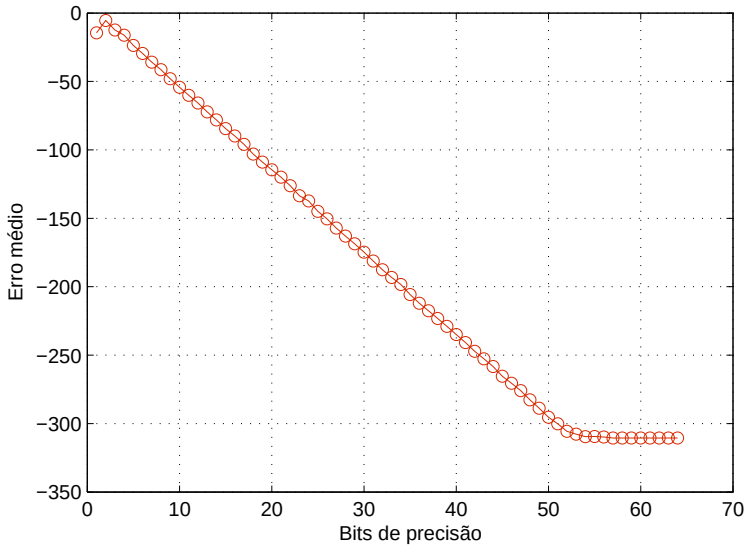


Figura 5.2: Erro médio do algoritmo *transf_x_c* em função do número de *bits* de precisão.

Analisando a Figura 5.2 pode-se notar que, com aproximadamente 18 *bits* o erro médio é igual a 10^{-5}. Sendo assim, conclui-se que aumentando o número de *bits* de precisão, o erro médio é reduzido e é desejável que a função *transf_x_c* seja utilizada com pelo menos 15 *bits* de precisão, pois o erro médio para essa precisão está em torno de 10^{-4}.

5.3.2 Função *house_vectors_c*

Para verificar o funcionamento da função *house_vectors_c*, o seu resultado será comparado com a função *house_vectors*. Essa função possui o mesmo pseudocódigo da função *house_vectors_c*; entretanto, não utiliza as funções CORDIC para realizar as operações aritméticas.

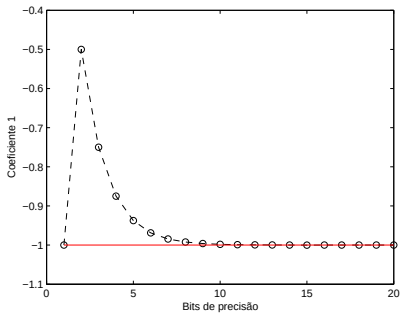
A fim de comprovar o funcionamento da função *house_vectors_c*, considere o vetor de entrada **A** (para o exemplo **A** será um vetor, todavia este pode ser uma matriz) igual a

$$\mathbf{A} = \begin{bmatrix} 0,7071 \\ 0,7071 \end{bmatrix} \tag{5.4}$$

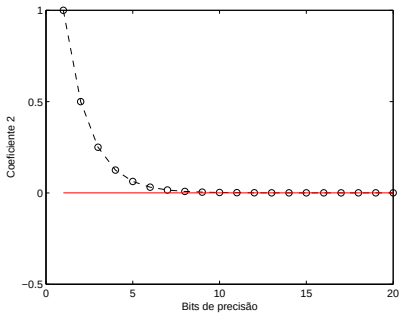
e o vetor de saída **U** (para o exemplo **U** será um vetor, todavia este pode ser uma matriz) do algoritmo *house_vectors* é igual a

$$\mathbf{U} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}. \tag{5.5}$$

As Figuras 5.3(a) e 5.3(b) mostram o comportamento dos coeficientes do vetor **U** encontrados pelos algoritmos *house_vectors_c* e *house_vectors*. A linha contínua mostra o valor encontrado pelo algoritmo *house_vectors*, já a linha pontilhada indica o comportamento do algoritmo *house_vectors_c* em função do número de *bits* de precisão.



(a) Comportamento do coeficiente 1 do vetor **U**.



(b) Comportamento do coeficiente 2 do vetor **U**.

Figura 5.3: Comportamento dos coeficientes de **U**.

Analisando a Figura 5.3, percebe-se que, à medida que o número de *bits* de precisão aumenta, mais próximo do valor correto a resposta do algoritmo *house_vectors_c* se aproxima. De acordo com a Figura 5.3, percebe-se que com 12 *bits*, a função *house_vectors_c* está funcionando de maneira similar a função *house_vectors*.

A fim de fazer uma análise mais detalhada com o algoritmo *house_vectors_c*, foram gerados 1000 vetores de comprimento igual a 4 que foram tratados pela função *house_vectors_c*. Os *bits* de precisão da função *house_vectors_c* foram variados de 1 a 64. A Figura 5.4 mostra o comportamento do erro médio, em dB, em função

da quantidade de *bits* de precisão. O erro médio é calculado através de um média aritmética da subtração do valor encontrado pelo *Matlab* com o valor encontrado pela função *house_vectors_c*.

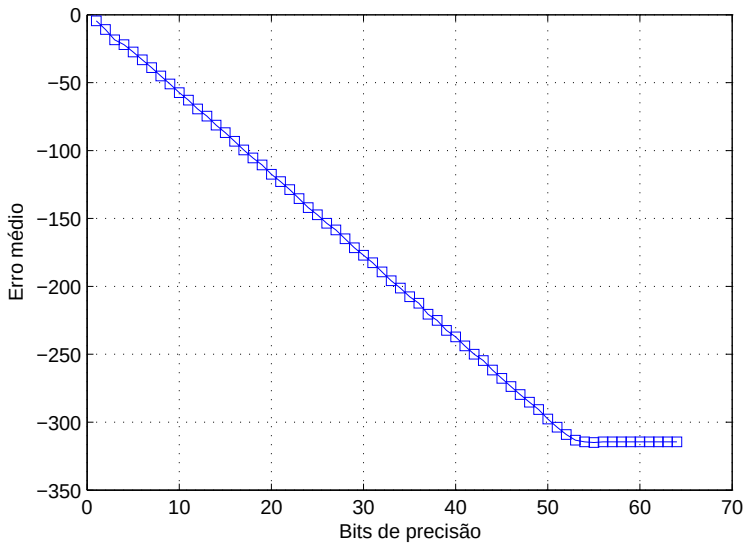


Figura 5.4: Erro médio do algoritmo *house_vectors_c* em função do número de *bits* de precisão.

Analisando a Figura 5.4 conclui-se que, aumentando o número de *bits* de precisão, o erro médio é reduzido. A seguir será feita uma comparação em relação ao erro médio introduzido pelas funções CORDIC implementadas nessa tese. Essa análise é muito importante pois, através dela, será possível identificar as funções CORDIC mais críticas.

5.3.3 Erros introduzidos pelas funções CORDIC

As figuras que serão apresentadas a seguir, mostram o comportamento das funções CORDIC, desenvolvidas na tese, quando são submetidas à variação de *bits* de precisão. A resposta do erro médio de cada função foi calculada através de uma média de 1000 valores. Para simular diversos sinais nas entradas das funções CORDIC, foi utilizada a função *rand* do *Matlab*. Essa função foi configurada para gerar sinais com distribuição normal e média 0.

As figuras serão apresentadas da seguinte forma: cada função CORDIC possui uma figura específica, que enfatiza o comportamento dessa função quando a

mesma é submetida a variação de *bits* de precisão. Em todas as figuras há retas que representam todas as funções CORDIC, entretanto sempre haverá uma reta identificada por um sinalizador, que representa a curva da função apresentada no tópico. Abaixo estão os resultados das simulações:

- ***Mat_adicao_c***

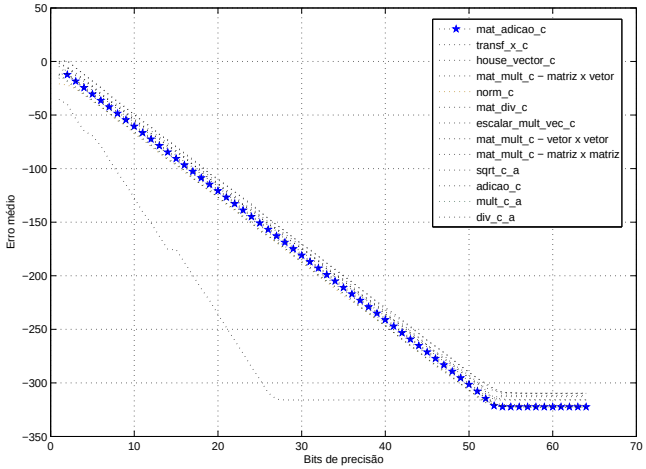


Figura 5.5: Erro médio do algoritmo *mat_adicao_c* em função do número de *bits* de precisão.

- ***Transf_x_c***

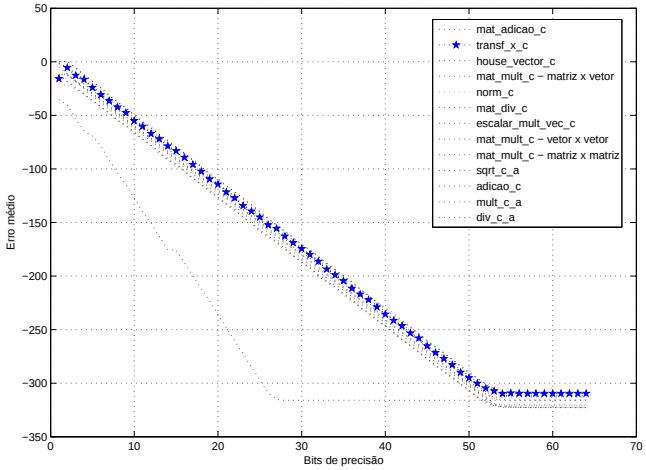


Figura 5.6: Erro médio do algoritmo *transf_x_c* em função do número de *bits* de precisão.

- ***House_vectors_c***

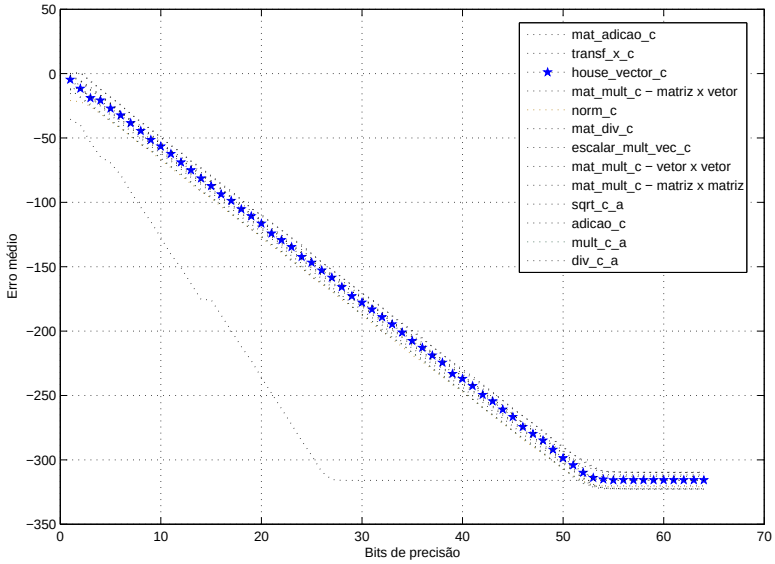


Figura 5.7: Erro médio do algoritmo *house_vectors_c* em função do número de *bits* de precisão.

- ***Mat_multi_c - matriz × vetor***

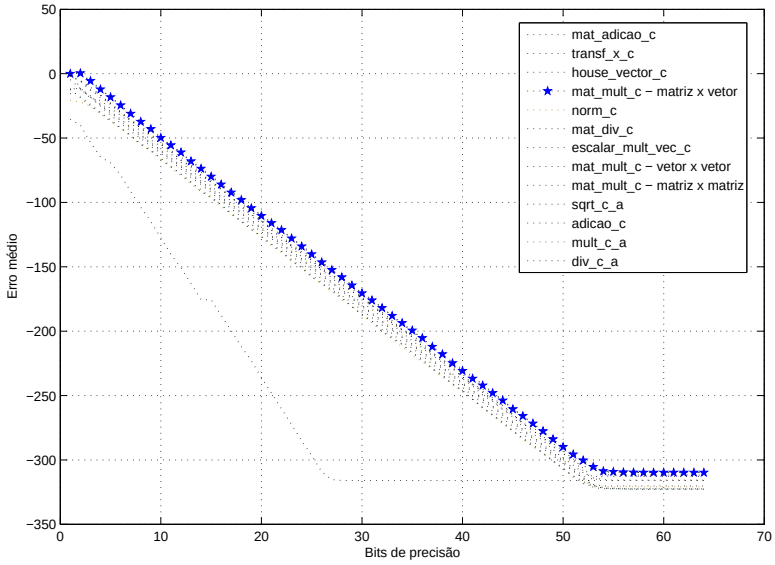


Figura 5.8: Erro médio do algoritmo *mat_mult_c* (matriz × vetor) em função do número de *bits* de precisão.

- ***Norm_c***

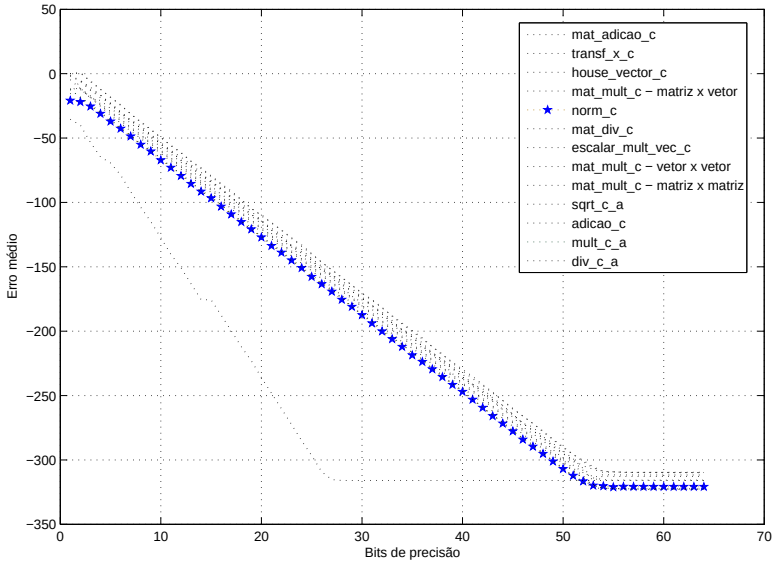


Figura 5.9: Erro médio do algoritmo *norm_c* em função do número de *bits* de precisão.

- ***Mat_div_c***

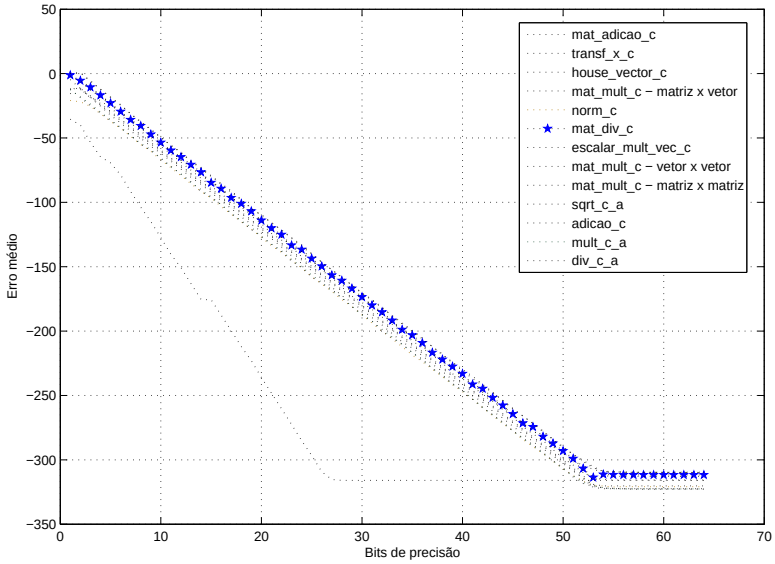


Figura 5.10: Erro médio do algoritmo *mat_div_c* em função do número de *bits* de precisão.

- ***Escalar_mult_vec_c***

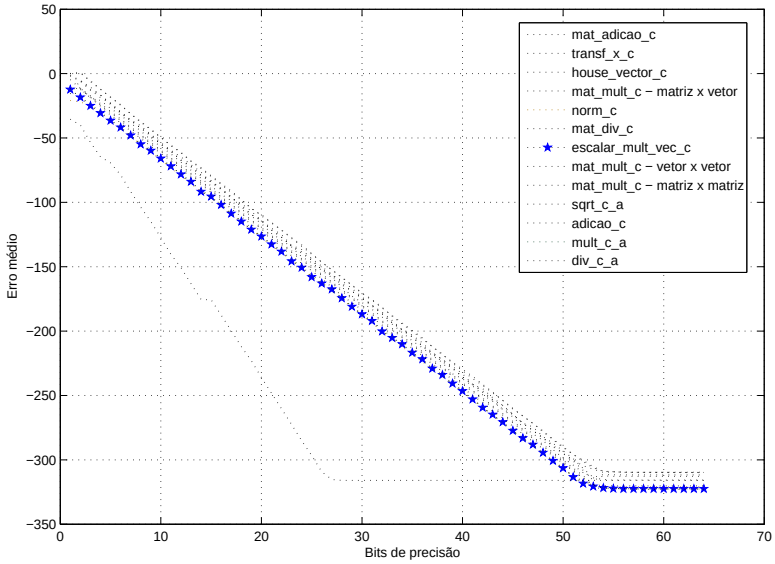


Figura 5.11: Erro médio do algoritmo *escalar_mult_vec_c* em função do número de *bits* de precisão.

- ***Mat_multi_c - vetor* × *vetor***

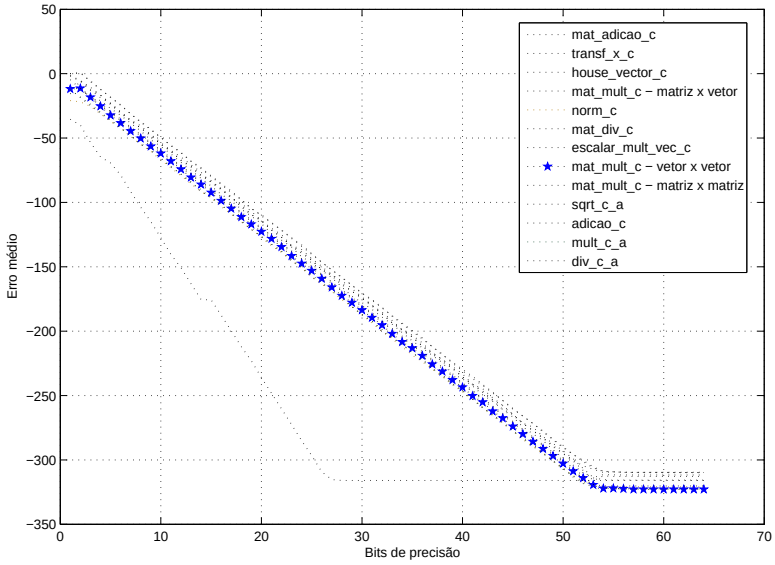


Figura 5.12: Erro médio do algoritmo *mat_mult_c* (vetor × vetor) em função do número de *bits* de precisão.

- ***Mat_multi_c - matriz × matriz***

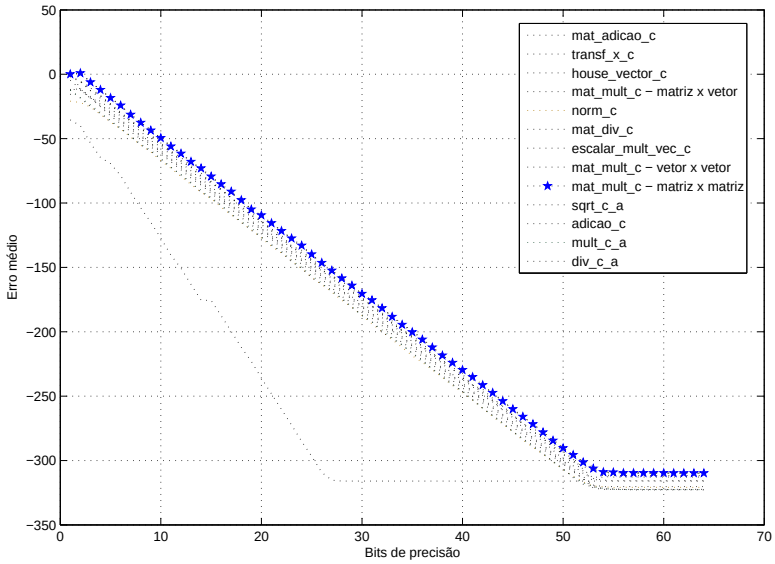


Figura 5.13: Erro médio do algoritmo *mat_mult_c* (matriz × matriz) em função do número de *bits* de precisão.

- ***Sqrt_c_a***

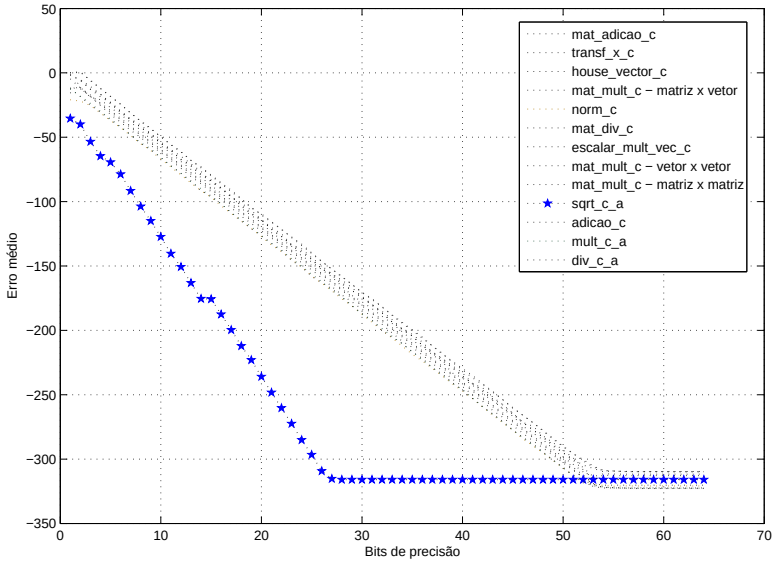


Figura 5.14: Erro médio do algoritmo *sqrt_c* em função do número de *bits* de precisão.

- ***Adicao_c***

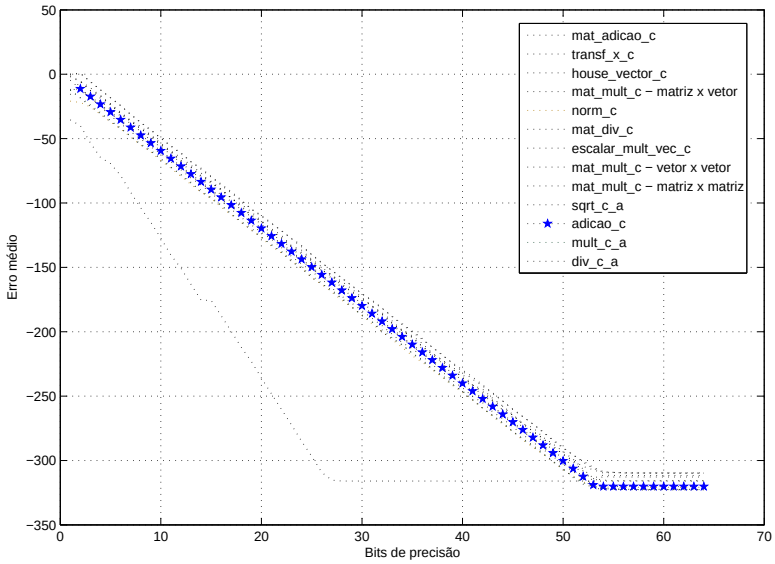


Figura 5.15: Erro médio do algoritmo *adicao_c* em função do número de *bits* de precisão.

- ***Mult_c_a***

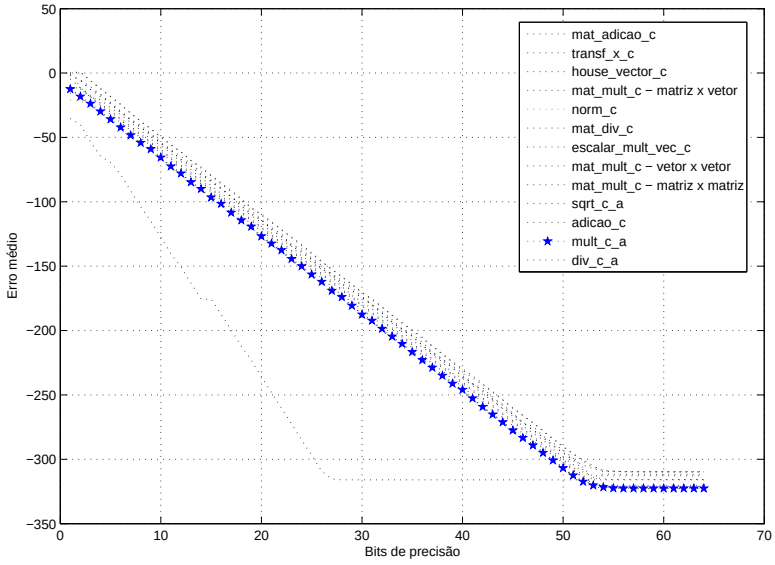


Figura 5.16: Erro médio do algoritmo *mult_c_a* em função do número de *bits* de precisão.

- ***Div_c_a***

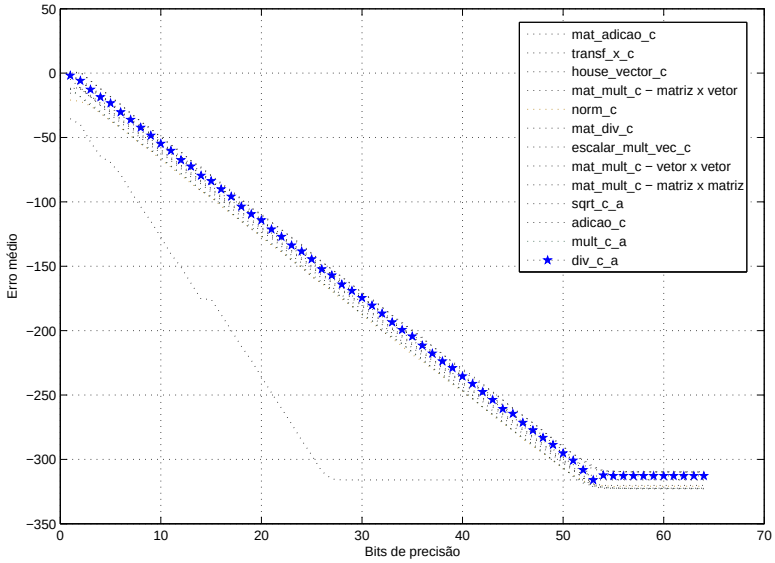


Figura 5.17: Erro médio do algoritmo *div_c_a* em função do número de *bits* de precisão.

Analisando as Figuras 5.5 a 5.17, pode-se notar que a função *mat_mult_c*, quando está trabalhando com operações de multiplicação de matriz com vetor e matriz com matriz, é a operação mais crítica. A criticidade dessa operação aumenta à medida que a dimensão dos vetores e matrizes aumentam. A Figura 5.18 mostra o comportamento dessa função quando varia-se a dimensão dos vetores e matrizes.

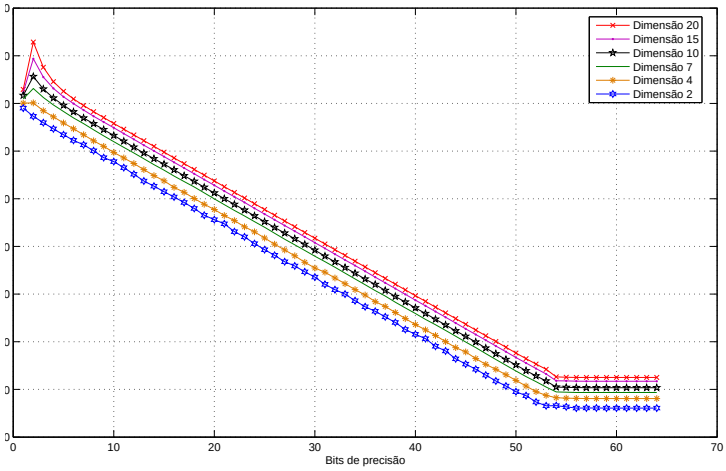


Figura 5.18: Erro médio do algoritmo *mat_mult_c* em função do número de *bits* de precisão com dimensão variável.

De acordo com as simulações mostradas pelas figuras 5.5 a 5.17, a operação mais robusta é a operação de raiz quadrada. Essa afirmação pode ser comprovada pela Figura 5.14.

Algumas funções possuem comportamento similar no que diz respeito a erros introduzidos devido a falta de *bits* de precisão. A função *mat_adicao_c* introduz o mesmo erro que a função *adicao_c*. Esse comentário também é válido para a função *mult_c_a* com a *escalar_mult_vec_c* e *norm_c*, entre as funções *mat_div_c* e *div_c_a* e entre as funções *transf_x_c* e *house_vectors_c*.

5.4 Resultados

Através dos testes realizados na Seção 5.3, pode-se concluir que as funções CORDIC utilizadas nos algoritmos *transf_x_c* e *house_vectors_c* estão funcionando de maneira similar às funções do *Matlab*.

É importante ressaltar que as mesmas funções CORDIC são utilizadas pelos algoritmos HC com precisão variável, ou seja, *hclms_c* e *nhclms_c*. Sendo asim, o correto funcionamento das funções CORDIC nos algoritmos *transf_x_c* e *house_vectors_c*, significam que os algoritmos HC com precisão variável também estão funcionando corretamente. As simulações sobre os algoritmos HC com precisão variável serão

mostrados no Capítulo 6.

A grande vantagem com a substituição das operações do *Matlab* por função CORDIC nos algoritmos das Tabelas 5.1 a 5.4 está no fato do resultado desses algoritmos serem encontrados com menos *bits* de precisão. Todo cálculo do *Matlab* é feito utilizando 64 *bits*, já utilizando as funções CORDIC o número de *bits* de precisão é variável.

Utilizando as Figuras 5.1 e 5.3 como referência, pode-se concluir que é mais vantajoso implementar as funções das Tabelas 5.1 a 5.4 num *hardware* CORDIC, pois com menos de 20 *bits* já é possível encontrar uma resposta bem próxima à resposta de um *hardware* com precisão de 64 *bits*. Desta forma o tempo de resposta do algoritmo será menor, tornando-o mais eficiente. As Figuras 5.2 e 5.4 comprovam essa afirmação.

A função *mat_mult_c*, quando está sendo utilizada para multiplicação entre matrizes e vetores, é a função mais crítica, logo é interessante realizar esse tipo de operação com mais *bits* de precisão, principalmente, quando os vetores e matrizes possuírem dimensões elevadas. A função responsável pelo cálculo de raiz quadrada mostrou ser a função mais robusta.

No próximo capítulo serão mostrados as simulações e resultados quando os algoritmos HC com precisão variável são utilizados em uma aplicação de antenas inteligentes adaptativas.

Capítulo 6

Simulações e resultados

6.1 Introdução

Ao longo dos Capítulos 2, 3, 4 e 5, foi desenvolvida uma ferramenta para implementação de filtros LCMV adaptativos baseados na transformação de *Householder* com precisão variável. Este tipo de filtro pode ser usado em diversas aplicações que tenham a necessidade de atingir a resposta ótima sujeita a restrições. É importante lembrar que o processo de otimização é baseado na minimização do MOE ou MSE.

Neste capítulo serão mostrados as simulações e resultados dos filtros LCMV adaptativos baseados na transformação de *Householder* com precisão variável em uma aplicação de antenas inteligentes adaptativas. Para maiores informações a respeito dos algoritmos utilizados nas simulações, vide o Capítulo 5.

Este capítulo está dividido da seguinte forma: primeiramente, será explicado como as simulações serão realizadas. Após isto, serão mostrados os resultados das simulações para os algoritmos HC com precisão variável. E por fim, será feita uma abordagem final sobre os resultados encontrados neste capítulo.

6.2 Simulações

Inicialmente, nesta seção, será explicado como as simulações serão realizadas. A seguir estão todas informações necessárias para o completo entendimento das simulações e seus resultados.

6.2.1 Considerações iniciais

O desempenho dos algoritmos HC com precisão variável serão comparados com os algoritmos HC implementados através de funções do *Matlab* e também com os algoritmos CLMS e NCLMS com precisão variável.

Todas simulações serão feitas para uma aplicação de antenas inteligentes baseadas em *arrays* adaptativos. Será utilizado um *array* linear, sendo que os sensores estão igualmente espaçados de $\frac{\lambda}{2}$. Todo processamento da saída dos sensores será feito por um *beamformer* MVDR (*Minimum Variance Distortionless Response*) de banda estreita, vide Figura 3.1(a).

No total serão utilizados 10 sensores (M = 10) para receber um sinal desejado (r = 1) e quatro interferidores. A direção de chegada (DOA - *Direction of arrival*) e a relação sinal/ruído (SNR) desses sinais podem ser encontradas na Tabela 6.1.

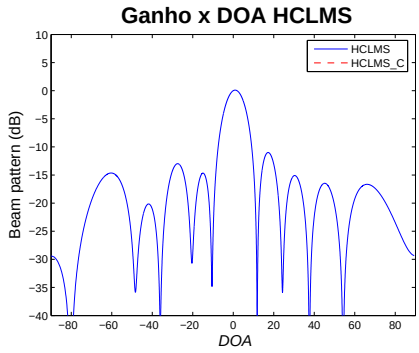
Tabela 6.1: Parâmetros dos sinais.

Sinal	**DOA**	**SNR**
desejado	0^o	20 dB
interferidor 1	24^o	20 dB
interferidor 2	-10^o	20 dB
interferidor 3	-20^o	20 dB
interferidor 4	-50^o	20 dB

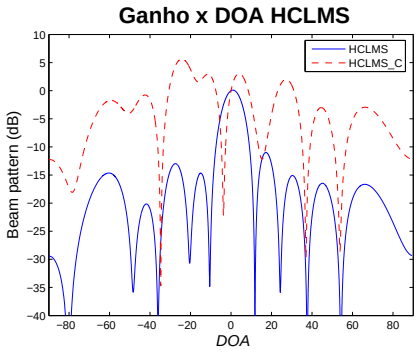
O vetor de entrada $\mathbf{x}$ possuirá comprimento igual a 20000 e o passo de adaptação, μ, será igual a $5 \cdot 10^{-5}$. Estes valores são válidos para todos algoritmos utilizados nas simulações.

6.2.2 HCLMS_C $\times$ HCLMS

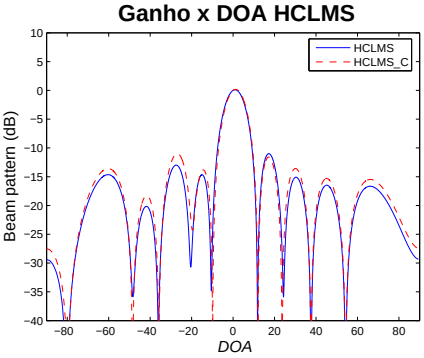
As simulações para o algoritmo HCLMS_C serão realizadas com as seguintes precisões: 10, 13, 15, 18, 20 e 25 *bits*. Essa simulação foi feita utilizando o pseudocódigo da Tabela 5.1 como base para os algoritmos HCLMS e HCLMS_C. O algoritmo HCLMS_C é a versão do algoritmo HCLMS com precisão variável. Abaixo estão os gráficos das simulações realizadas com esses algoritmos.



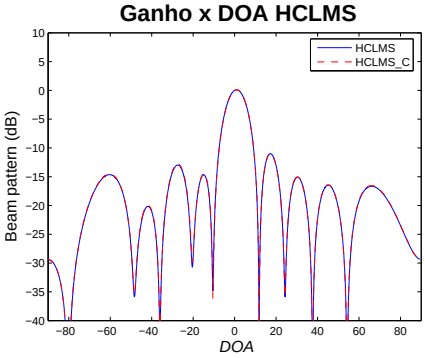
(a) Precisão de 10 *bits*.



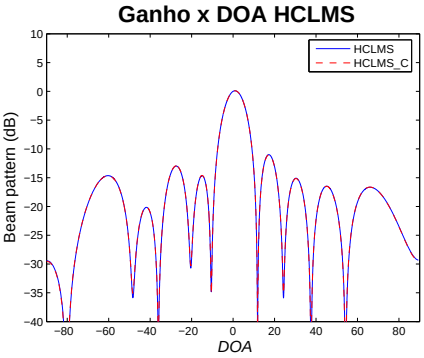
(b) Precisão de 13 *bits*.



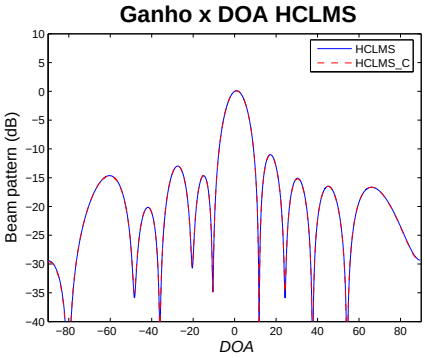
(c) Precisão de 15 *bits*.



(d) Precisão de 18 *bits*.



(e) Precisão de 20 *bits*.



(f) Precisão de 25 *bits*.

Figura 6.1: *Beam Pattern* dos algoritmos HCLMS e HCLMS_C com precisão variável.

Para as figuras acima, o eixo horizontal representa o ângulo de chegada (em graus) do sinal de entrada, $\mathbf{x}$, em relação a reta normal do *array* de sensores (vide Figura 3.2). Já o eixo vertical refere-se ao ganho do *beamformer* adaptativo em decibéis (dB). Nessas figuras as linhas tracejadas mostram o comportamento do algoritmo HCLMS_C, já as linhas contínuas estão relacionadas à resposta do algoritmo *HCLMS*.

Analisando as Figuras 6.1(a) a 6.1(f), percebe-se que, à medida que o número de *bits* de precisão aumenta, o *beam pattern* da função HCLMS_C tende a ficar mais parecido com o *beam pattern* da função HCLMS. Isso acontece porque os erros introduzidos pela imprecisão das funções CORDIC são reduzidos à medida que o número de *bits* de precisão aumenta.

Analisando a Figura 6.1(a), percebe-se que não é possível identificar a resposta do algoritmo HCLMS_C, pois os erros introduzidos devido a falta de *bits* de precisão fizeram com que esse algoritmo divergisse. Devido a isso a curva de resposta do algoritmo HCLMS_C com precisão de 10 *bits* nem aparece na Figura 6.1(a).

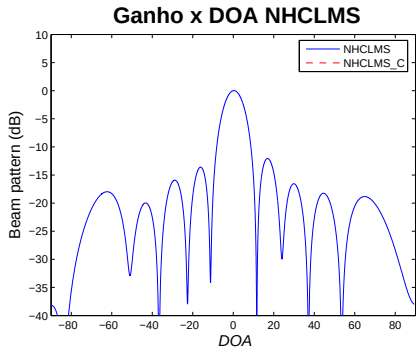
A Figura 6.1(b) ilustra o comportamento da função HCLMS com 13 *bits* de precisão. Nesta figura pode-se notar que o erro, devido à falta de precisão nos cálculos das funções CORDIC, impede com que *beam pattern* desejado seja alcançado. É importante notar que nem a restrição está sendo satisfeita, pois o ganho em 0^o é igual a 0 dB.

No *beam pattern* da Figura 6.1(c), as respostas dos algoritmos HCLMS e HCLMS_C estão um pouco diferentes, todavia os nulos nas direções dos interferidores já podem ser identificados e a restrição já é satisfeita. Isto comprova que o aumento da quantidade de *bits* de precisão faz com que os *beam patterns* do HCLMS e HCLMS_C sejam mais parecidos.

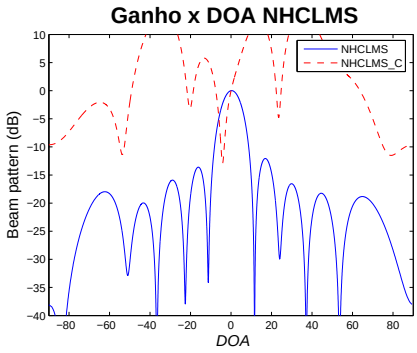
As Figuras 6.1(d), 6.1(e) e 6.1(f) mostram que o comportamento dos *beam patterns* dos algoritmos HCLMS e HCLMS_C são praticamente iguais. Isto comprova que, com, pelo menos, 18 *bits* de precisão para o algoritmo HCLMS_C, já é possível obter uma resposta muito similar à resposta do algoritmo HCLMS.

6.2.3 NHCLMS_C × NHCLMS

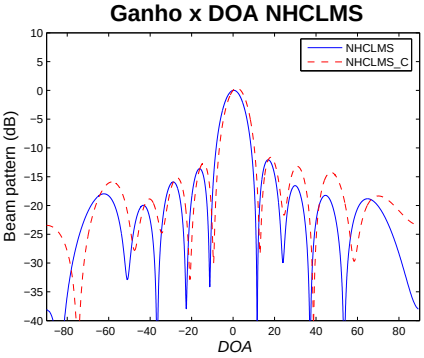
As simulações para o algoritmo NHCLMS_C serão realizadas com as seguintes precisões: 10, 13, 15, 18, 20 e 25 *bits*. Essa simulação foi realizada utilizando o pseudocódigo da Tabela 5.4 como base para os algoritmos NHCLMS e NHCLMS_C. O algoritmo NHCLMS_C é a versão do algoritmo NHCLMS com precisão variável. Abaixo estão os gráficos das simulações realizadas com esses algoritmos.



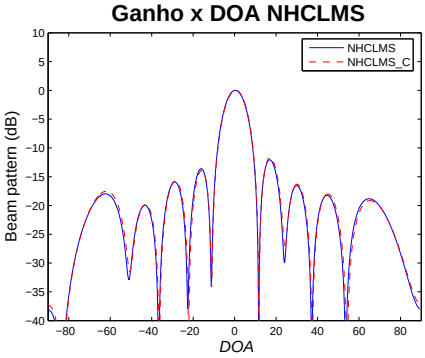
(a) Precisão de 10 *bits*.



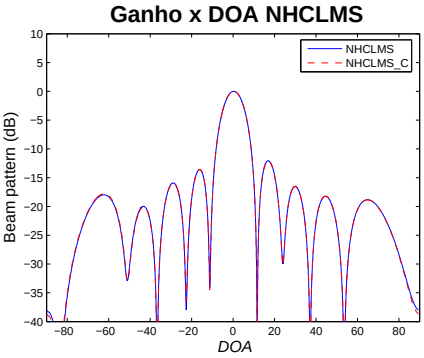
(b) Precisão de 13 *bits*.



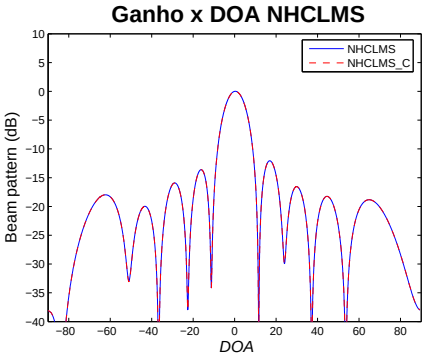
(c) Precisão de 15 *bits*.



(d) Precisão de 18 *bits*.



(e) Precisão de 20 *bits*.



(f) Precisão de 25 *bits*.

Figura 6.2: *Beam Pattern* dos algoritmos NHCLMS e NHCLMS_C com precisão variável.

Todas análises feitas para a Figura 6.1 são válidas para a Figura 6.2. A principal diferença no comportamento entre os algoritmos HC com precisão variável está relacionada à normalização realizada no algoritmo NHCLMS_C. Essa normalização, quando feita por funções CORDIC, gera um erro adicional ao algoritmo HCLMS_C ocasionado pela falta de *bits* de precisão no seu cálculo. Sendo assim, o algoritmo NHCLMS_C necessita de mais *bits* de precisão para chegar à mesma

resposta do algoritmo NHCLMS. Essa afirmação pode ser facilmente comprovada analisando as Figuras 6.1(c) e 6.2(c).

Ao analisar as Figuras 6.1(c) e 6.2(c) pode-se perceber que, para 15 *bits*, o algoritmo HCLMS_C está mais próximo da resposta ideal do que o NHCLMS_C. Essa análise também pode ser feita nas Figuras 6.1(d) e 6.2(d). Repare que a resposta do algoritmo NHCLMS_C ainda não está igual a resposta do NHCLMS, enquanto o algoritmo HCLMS_C já está funcionando de maneira idêntica para todos efeitos práticos ao algoritmo HCLMS.

A análise, que foi feita no parágrafo anterior, pode ser visualizada, de maneira mais clara, pela Figura 6.2(c), já que, para 15 *bits*, o gráfico do HCLMS_C é idêntico ao gráfico do NHCLMS. Sendo assim, pode-se concluir que, para implementar os algoritmos HCLMS_C e NHCLMS_C, necessita-se de um hardware, com pelo menos, 15 e 18 *bits* de precisão, respectivamente.

6.2.4 HCLMS_C $\times$ CLMS_C

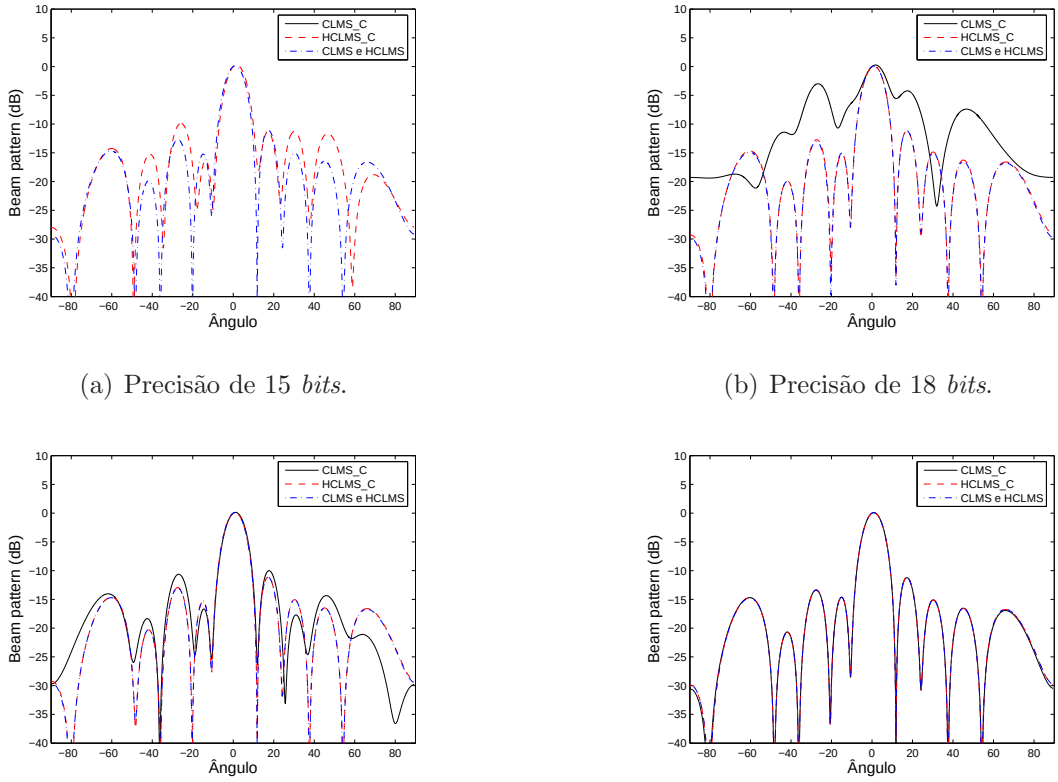
Esta simulação tem como objetivo fazer uma estudo comparativo em relação ao comportamento dos algoritmos HCLMS e CLMS quando implementados num *hardware* com precisão variável. Após a simulação será possível verificar qual dos dois algoritmos é mais eficiente e robusto para trabalhar num sistema que permita a variação na precisão dos cálculos.

As simulações dos algoritmos HCLMS_C e CLMS_C serão realizadas com a seguintes precisões: 15, 18 e 25 *bits*. O algoritmo HCLMS_C foi implementado de acordo com o pseudocódigo da Tabela 5.1, já o algoritmo CLMS_C foi implementado conforme o pseudocódigo apresentado na Tabela 6.2.

Tabela 6.2: Algoritmo CLMS.

dado $\mathbf{x}$(k), $\mathbf{C}$, $\mathbf{f}$ e μ (passo)
Inicializar:
$\mathbf{P} = \mathbf{I} - \mathbf{C}(\mathbf{C}^H\mathbf{C})^{-1}\mathbf{C}^H$;
$\mathbf{F} = \mathbf{C}(\mathbf{C}^H\mathbf{C})^{-1}\mathbf{f}$;
$\mathbf{w}(0) = \mathbf{F}$;
for k $= 0,\ 1,\ 2,\ \ldots$
$y(k) = \mathbf{w}^H(k)\mathbf{x}(k)$;
$\mathbf{w}(k+1) = \mathbf{P}[\mathbf{w}(k) - \mu y^*(k)\mathbf{x}(k)] + \mathbf{F}$;
end for

Abaixo estão os gráficos das simulações realizadas com esses algoritmos.



(a) Precisão de 15 *bits*.

(b) Precisão de 18 *bits*.

(c) Precisão de 20 *bits*.

(d) Precisão de 25 *bits*.

Figura 6.3: *Beam Pattern* dos algoritmos CLMS_C e HCLMS_C com precisão variável.

Para as figuras acima, as linhas tracejadas mostram o comportamento do algoritmo HCLMS_C, já as linhas continuas estão relacionadas a resposta do algo-

ritmo CLMS_C. As linhas tracejadas e pontilhadas são as respostas dos algoritmos HCLMS e CLMS utilizando as funções do *Matlab*.

Analisando as Figuras 6.3(a) a 6.3(d), percebe-se que, à medida que o número de *bits* de precisão aumenta, os *beam patterns* das funções CLMS_C e HCLMS_C tendem a ficar idênticos ao *beam pattern* das funções HCLMS e CLMS . Isso acontece porque os erros introduzidos pela imprecisão das funções CORDIC são reduzidos à medida que o número de *bits* de precisão aumenta.

Analisando a Figura 6.3(a), nota-se que a resposta da função CLMS_C não aparece na figura. Isto acontece porque esse algoritmo é mais sensível à falta de *bits* de precisão do que o HCLMS_C. O algoritmo CLMS_C necessita que seus cálculos sejam realizados com mais de 15 *bits* de precisão para que haja o correto funcionamento do *beamformer*. Já o *beam pattern* do algoritmo HCLMS_C , quando comparado ao *beam pattern* dos algoritmos HCLMS e CLMS, nota-se que há algumas diferenças, todavia os nulos e a restrição já estão sendo satisfeitas.

A diferença de comportamento entre os algoritmos CLMS_C e HCLMS_C, evidenciada pelas Figuras 6.3(a) a 6.3(c), ocorre pois a complexidade computacional do algoritmo HCLMS_C é inferior a complexidade do CLMS_C. A Tabela 6.3 mostra a complexidade computacional desses algoritmos [2].

Tabela 6.3: Complexidade computacional do algoritmos CLMS_C e HCLMS_C.

Algoritmo	**Adições**	**Multiplicações**
CLMS_C	$(2r+2)M-(r-1)$	$(2r+2)M+1$
HCLMS_C	$(2r+2)M-(r^2+2r+1)$	$(2r+2)M-(r^2-1)$

Através da Tabela 6.3 pode-se verificar que o algoritmo HCLMS_C utiliza menos operações de adição e multiplicação para encontrar o mesmo resultado que o algoritmo CLMS_C. Essa característica do algoritmo HCLMS_C faz com que o mesmo seja mais eficiente que o CLMS_C quando utiliza-se poucos *bits* de precisão para as operações aritméticas.

A Tabela 6.4 mostra os erros introduzidos pelas funções CORDIC, que foram utilizadas nos algoritmos CLMS_C e HCLMS_C, quando uma operação aritmética é realizada. Os valores apresentados na Tabela 6.4 foram encontrados através de

simulações no *Matlab*, onde buscou-se comparar os resultados encontrados pelas funções CORDIC de adição e multiplicação com os resultados encontrados pelas funções do próprio *Matlab*. Os gráficos dessas simulações foram apresentados no Capítulo 5.

Tabela 6.4: Erro introduzido pelas funções CORDIC nos algoritmos CLMS_C e HCLMS_C.

Função CORDIC	Erro médio para $15 bits$	Erro médio para 18 *bits*	Erro médio para 20 *bits*	Erro médio para 25 *bits*
adicao_c	$3,0 \cdot 10^{-5}$	$3,8 \cdot 10^{-6}$	$9,5 \cdot 10^{-7}$	$3,0 \cdot 10^{-8}$
mult_c_a	$1,6 \cdot 10^{-5}$	$2,0 \cdot 10^{-6}$	$4,4 \cdot 10^{-7}$	$1,5 \cdot 10^{-8}$
mat_mult_c	$6,9 \cdot 10^{-4}$	$8,5 \cdot 10^{-5}$	$2,1 \cdot 10^{-5}$	$6,8 \cdot 10^{-7}$

Os valores apresentados nas duas primeiras linhas da Tabela 6.4 representam os erros médios de uma única operação aritmética de adição e multiplicação. Já a multiplicação entre vetores ou matrizes é composta por várias operações de multiplicação e adição. A função *mat_mult_c* é responsável por realizar as operações de multiplicação entre vetores e matrizes da tese. Os valores, mostrados pela Tabela 6.4, são dados importantes para que seja entendido o motivo da ineficiência do algoritmo CLMS_C em relação ao HCLMS_C. A justificativa para tal está associada à transformação de *Householder*. Devido a essa tranformação, a adaptação do algoritmo HCLMS_C é realizada numa dimensão r (número de restrições) unidades inferior à dimensão do algoritmo CLMS_C. E como pode ser visto no Capítulo 5, o erro médio aumenta à medida que a dimensão dos vetores e matrizes aumentam, logo o erro médio do CLMS_C é superior ao erro introduzido pelo HCLMS_C.

Na Figura 6.3(d) os *beam pattern* de todos os algoritmos são iguais. Isto comprova que com pelo menos 25 *bits* de precisão o algoritmo CLMS_C se comporta de maneira idêntica aos demais algoritmos. O CLMS_C mostrou ser um algoritmo mais sensível às imprecisões introduzidas pela utilização das funções CORDIC do que o HCLMS_C.

6.2.5 NHCLMS_C × NCLMS_C

Assim como a Subseção 6.2.4, o objetivo desta simulação é fazer uma estudo comparativo entre dois algoritmos com precisão variável, só que nesta subseção os algoritmos são o NHCLMS_C e o NCLMS_C. O algoritmo NHCLMS_C foi implementado de acordo com o pseudocódigo da Tabela 5.4, já o algoritmo NCLMS_C foi implementado conforme o pseudocódigo apresentado na Tabela 6.5.

Tabela 6.5: Algoritmo NCLMS.

dado $\mathbf{x}$(k), $\mathbf{C}$, $\mathbf{f}$ e μ (passo)

Inicializar:

$\mathbf{P} = \mathbf{I} - \mathbf{C}(\mathbf{C}^H\mathbf{C})^{-1}\mathbf{C}^H$;

$\mathbf{F} = \mathbf{C}(\mathbf{C}^H\mathbf{C})^{-1}\mathbf{f}$;

$\mathbf{w}(0) = \mathbf{F}$;

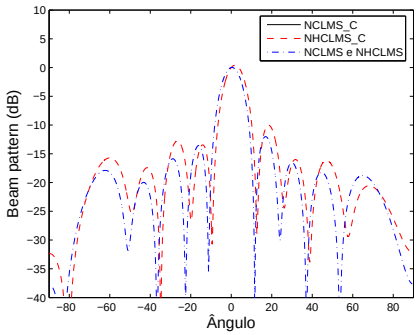
for k $= 0,\ 1,\ 2,\ \ldots$

$y(k) = \mathbf{w}^H(k)\mathbf{x}(k)$;

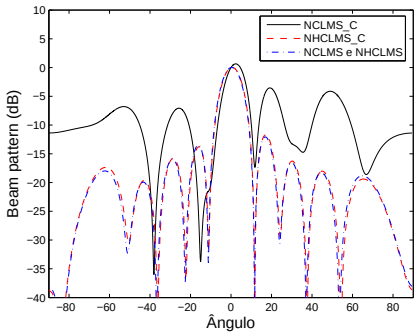
$\mathbf{w}(k+1) = \mathbf{P}[\mathbf{w}(k) - \mu\frac{y^*(k)}{\mathbf{x}^H(k)\mathbf{P}\mathbf{x}(k)}\mathbf{x}(k)] + \mathbf{F}$;

end for

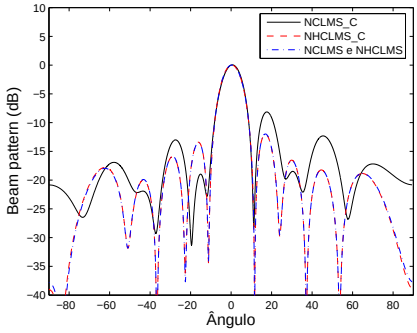
Abaixo estão os gráficos das simulações realizadas com esses algoritmos. As linhas tracejadas mostram o comportamento do algoritmo NHCLMS_C, já as linhas continuas estão relacionadas a resposta do algoritmo NCLMS_C. As linhas tracejadas e pontilhadas são as respostas dos algoritmos NHCLMS e NCLMS utilizando as funções do *Matlab*.



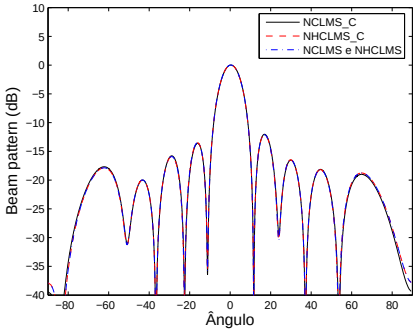
(a) Precisão de 15 *bits*.



(b) Precisão de 18 *bits*.



(c) Precisão de 20 *bits*.



(d) Precisão de 25 *bits*.

Figura 6.4: *Beam Pattern* dos algoritmos NCLMS_C e NHCLMS_C com precisão variável.

As mesmas análises feitas na Subseção 6.2.4 são válidas nesta subseção, por exemplo, analisando a Figura 6.4(a), nota-se que a resposta da função NCLMS_C não aparece na figura. Isto acontece porque esse algoritmo é mais sensível à falta de *bits* de precisão do que o NHCLMS_C.

O algoritmo NHCLMS_C mostrou-se mais robusto do que o NCLMS_C. Esta afirmação pode ser comprovada através da Figura 6.4(b), que mostra além do *beam pattern* dos algoritmos com precisão variável, ilustra também o *beam pattern* dos algoritmos HCLMS e CLMS.

Na Figura 6.3(d) os *beam pattern* de todos os algoritmos são iguais. Isto comprova que com pelo menos 25 *bits* de precisão o algoritmo NCLMS_C se comporta de maneira idêntica aos demais algoritmos. O NCLMS_C mostrou ser um algoritmo mais sensível às imprecisões introduzidas pela utilização das funções CORDIC do que o NHCLMS_C.

6.2.6 Redução de posto

Esta subseção tem como objetivo apresentar as simulações dos algoritmos HCLMS_C e NHCLMS_C, quando submetidos a redução de posto. Após esta subseção será possível verificar se a redução de posto é eficiente ou não quando aplicada a estes algoritmos.

As simulações para redução de posto foram realizadas para as precisões de 18, 25 e 64 *bits*. Em cada uma das figuras, que serão apresentadas a seguir, há 4 gráficos, onde cada um deles está associado a um fator de redução. Os fatores de redução utilizados nas simulações foram de 1, $\frac{1}{2}$, $\frac{1}{4}$ e $\frac{1}{8}$.

A redução de posto foi aplicada à matriz de *Householder* $\mathbf{Q}$. Essa matriz é calculada pela função *house_vectors_c* e é utilizada para transformar o vetor de entrada $\mathbf{x}$ e o vetor de pesos $\mathbf{w}$ em $\bar{\mathbf{x}}$ e $\bar{\mathbf{w}}$, respectivamente. Abaixo segue a tabela que associa o fator de redução ao posto da matriz $\mathbf{Q}$, cuja dimensão é 10×10 nas simulações, vide a subseção 6.2.1.

Tabela 6.6: Relação entre o fator de redução e o posto da matriz $\mathbf{Q}$.

Fator de redução	**Posto**
1	10
$\frac{1}{2}$	5
$\frac{1}{4}$	2
$\frac{1}{8}$	1

A redução de posto foi realizada da seguinte forma: primeiramente, fez-se a decomposição da matriz $\mathbf{Q}$ em valores singulares. Como resposta obteve-se três matrizes, sendo que uma delas é a matriz diogonal que contém os valores singulares. Substituiu-se por zero, os n menores valores singulares. O valor de n está associado ao fator de redução. Todavia, por se tratar de uma matriz de *Householder*, todos os valores singulares são iguais a 1, logo escolheu-se os primeiros elementos da diagonal principal para serem zerados. Esta é a forma mais eficiente de se reduzir o posto de uma matriz de *Householder*, visto que perde-se menos informação desse jeito. Abaixo segue um exemplo que comprova a afirmação acima.

Dada a matriz de *Householder* $\mathbf{Q}$ expressa abaixo.

$$\mathbf{Q} = \begin{bmatrix} \text{-0,50} & \text{-0,50} & \text{-0,50} & \text{-0,50} \\ \text{-0,50} & \text{0,83} & \text{-0,16} & \text{-0,16} \\ \text{-0,50} & \text{-0,16} & \text{0,83} & \text{-0,16} \\ \text{-0,50} & \text{-0,16} & \text{-0,16} & \text{0,83} \end{bmatrix} \tag{6.1}$$

Sua decomposição em valores singulares é igual a

$$\mathbf{Q} = \begin{bmatrix} \text{-0,02} & \text{0,08} & \text{0,86} & \text{-0,50} \\ \text{0,61} & \text{0,52} & \text{-0,33} & \text{-0,50} \\ \text{-0,77} & \text{0,22} & \text{-0,33} & \text{-0,50} \\ \text{0,18} & \text{-0,82} & \text{-0,20} & \text{-0,50} \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 0 & 1 \\ \text{0,62} & \text{0,50} & \text{-0,62} & 0 \\ \text{-0,77} & \text{0,19} & \text{-0,62} & 0 \\ \text{0,19} & \text{-0,85} & \text{-0,50} & 0 \end{bmatrix} \tag{6.2}$$

O fator de redução, no exemplo, será igual a $\frac{1}{2}$. Abaixo estão duas matrizes que representam o resultado da redução de posto, quando são igualados a zero os dois primeiros elementos da diagonal principal ($\mathbf{Q}_{sup}$) e os dois últimos elementos da diagonal principal ($\mathbf{Q}_{inf}$), respectivamente.

$$\mathbf{Q}_{sup} = \begin{bmatrix} \text{-0,50} & \text{-0,53} & \text{-0,53} & \text{-0,42} \\ \text{-0,50} & \text{0,20} & \text{0,20} & \text{0,16} \\ \text{-0,50} & \text{0,20} & \text{0,20} & \text{0,16} \\ \text{-0,50} & \text{0,13} & \text{0,13} & \text{0,10} \end{bmatrix} \tag{6.3}$$

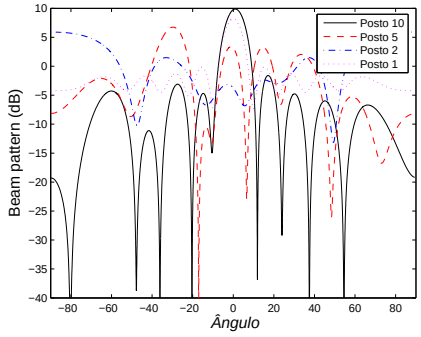
$$\mathbf{Q}_{inf} = \begin{bmatrix} 0 & \text{0,03} & \text{0,03} & \text{-0,07} \\ 0 & \text{0,63} & \text{-0,37} & \text{-0,33} \\ 0 & \text{-0,37} & \text{0,63} & \text{-0,33} \\ 0 & \text{-0,30} & \text{-0,30} & \text{0,73} \end{bmatrix} \tag{6.4}$$

Caso a matriz $\mathbf{Q}_{inf}$ seja utilizada no algoritmo HCLMS_C ou NHCLMS_C, ela fará com que a restrição não seja satisfeita. Isto faz com que esse tipo de redução seja completamente inviável.

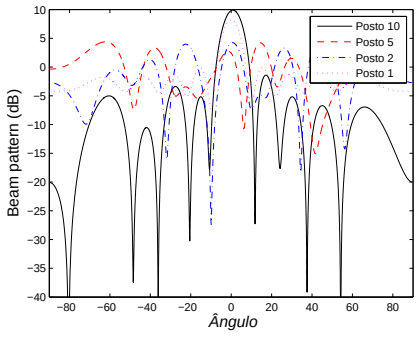
A seguir estão as simulações realizadas para os algoritmos HCLMS_C e NHCLMS_C quando a redução de posto é aplicada.

6.2.6.1 Redução de posto para o algoritmo HCLMS_C

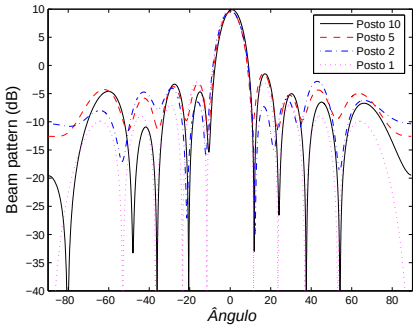
Os gráficos referentes à simulação do algoritmo HCLMS_C com redução de posto encontram-se a seguir.



(a) Redução de posto com precisão de 18 *bits*.



(b) Redução de posto com precisão de 25 *bits*.



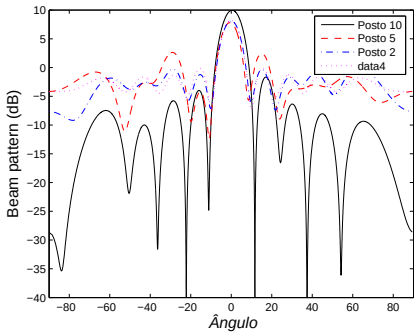
(c) Redução de posto com precisão de 64 *bits*.

Figura 6.5: *Beam Pattern* do algoritmo HCLMS_C utilizando redução de posto com precisão variável.

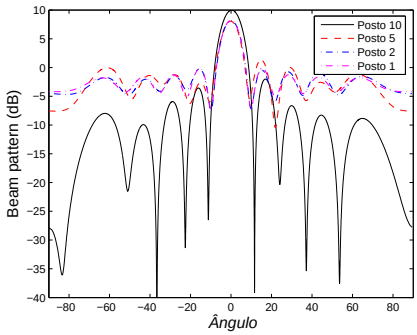
Analisando as Figuras 6.5(a) a 6.5(c), conclui-se que a redução de posto não é eficiente quando aplicada ao algoritmo HCLMS_C. Os nulos na direção dos interferidores não são respeitados, nem mesmo quando utiliza-se a redução de posto com 64 *bits* de precisão.

6.2.6.2 Redução de posto para o algoritmo NHCLMS_C

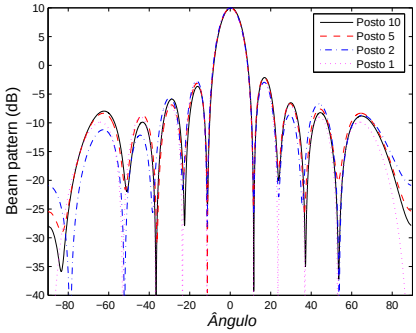
Os gráficos referentes à simulação do algoritmo NHCLMS_C com redução de posto encontram-se a seguir.



(a) Redução de posto com precisão de 18 *bits*.



(b) Redução de posto com precisão de 25 *bits*.



(c) Redução de posto com precisão de 64 *bits*.

Figura 6.6: *Beam Pattern* do algoritmo NHCLMS_C utilizando redução de posto com precisão variável.

Analisando as Figuras 6.6(a) a 6.6(c), conclui-se que a redução de posto é eficiente apenas, quando o algoritmo NHCLMS_C trabalha com precisão elevada. Os nulos na direção dos interferidores e o ganho na direção do sinal desejado só são completamente respeitados quando a precisão de *bits* é bem alta.

6.3 Resultados finais

Neste capítulo, foi visto o comportamento dos algoritmos HC com precisão variável em uma aplicação de antenas inteligentes utilizando um *array* adaptativo. As funções HCLMS_C, NHCLMS_C, CLMS_C, NCLMS_C, CLMS, NCLMS, HCLMS e NHCLMS foram implementadas à fim de estudar o comportamento e as principais caracteríticas dos algoritmos HC.

Através das simulações realizadas nesse capítulo, pode-se concluir que os

algoritmos HC podem ser facilmente implementados por um processador CORDIC. Esse tipo de implementação é muito vantajosa, pois permite que a precisão das operações matemáticas sejam ajustadas de acordo com a aplicação.

De acordo com as simulações, verificou-se que, para a aplicação escolhida nesta tese, é possível implementar com 15 e 18 *bits* de precisão os algoritmos HCLMS_C e NHCLMS_C, respectivamente.

Neste capítulo também foi feita a comparação entre os algoritmos HCLMS_C com CLMS_C e NHCLMS_C com NCLMS_C. Dessas comparações pode-se concluir que o algoritmo HCLMS_C e NHCLMS_C são mais robustos e eficientes que o CLMS_C e NCLMS_C, respectivamente. É importante ressaltar que, os algoritmos HCLMS_C e NHCLMS_C só convergiram com 25 *bits* de precisão.

O algoritmo NHCLMS_C apresentou resultados menos satisfatórios que o algoritmo HCLMS_C quando utiliza-se o posto completo. A diferença nos resultados foi ocasionada pelo erro introduzido com o processo de normalização do algoritmo NHCLMS_C. Todavia, ao implementar a redução de posto nesses algoritmos, o NHCLMS_C mostrou-se mais eficiente que o HCLMS_C, visto que os nulos na direção dos interferidores são respeitados quando trabalha-se com uma precisão elevada de *bits*.

De modo geral, a redução de posto não mostrou ser um método recomendável para ser aplicado em algoritmos HC. Sua implementação com o algortimo HCLMS_C não é recomendável, já que os nulos na direção de chegada dos interferidores não são respeitados. Já para o algortimo NHCLMS_C, a redução de posto não é recomendada pois, para sua utilização de forma satisfatória, é necessário ter uma precisão elevada, sendo assim tornando-o ineficiente.

Capítulo 7

Conclusão

7.1 Contribuição

O principal objetivo e motivação da tese foi verificar o comportamento dos algoritmos HC com precisão variável numa aplicação de antenas inteligentes utilizando um *array* adaptativo.

Dois algoritmos HC foram desenvolvidos, a fim de realizar esse estudo. A seguir estão essas funções:

- **HCLMS_C** ⇒ função responsável por implementar o algoritmo HCLMS com precisão variável.
- **NHCLMS_C** ⇒ função responsável por implementar o algoritmo NHCLMS com precisão variável.

Os pseudocódigos desses algoritmos foram apresentado no Capítulo 5 e em sua implementação todas operações matemáticas foram substituídas pelas funções CORDIC desenvolvidas em [3] que por sua vez foram aperfeiçoadas no Capítulo 2. O aperfeiçoamento à biblioteca criada por [3], está vinculado à criação de três novas funções CORDIC responsáveis pelas operações de multiplicação, divisão e raiz quadrada. A grande vantagem, de utilizar essas funções ao invés das criadas por [3], está no fato do usuário não ter preocupações com relação a região de convergência das funções CORDIC. Caso a máquina CORDIC possua *bits* de precisão suficientes, a resposta das novas funções sempre estará correta.

O Capítulo 2 buscou explicar o funcionamento do CORDIC, destacando como as operações aritméticas, trigonométricas e exponenciais são calculadas. Para cada uma dessas operações foi apresentada a região de convergência. Para as operações que possuem limitações quanto a região de convergência e são utilizadas nos algoritmos HC com precisão variável, foi proposto uma forma de adaptar as entradas das funções CORDIC às limitações das mesmas. Essa adaptação é realizada com algoritmos de pré e pós processamento às funções CORDIC. Esse tipo de adaptação foi aplicado às funções de multiplicação, divisão e raiz quadrada, criando novas funções, que não têm necessidade de fazer análise de convergência, pois estas sempre irão convergir.

Como o objetivo da tese é estudar os algoritmos HC com precisão variável numa aplicação de antenas inteligentes utilizando um *array* adaptativo, fez-se necessário apresentar alguns conceitos relacionados à filtragem espacial e *beamforming*. O Capítulo 3 apresentou diversos conceitos teóricos a respeito de *beamformers*, com o intuito de mostrar as vantagens dos *beamformer* adaptativos em relação aos demais tipos de *beamformers*. Tendo apresentado o que é *beamformer* adaptativo, fica mais fácil entender onde os algoritmos HC, que foram apresentados no Capítulo 4, são aplicados.

O Capítulo 4 mostrou todo o modelamento matemático para os algoritmos HC, apresentando todos conceitos necessários para a compreensão desses algoritmos. Esse capítulo mostra que os algoritmos HC são mais simples quando comparado a outros algoritmos adaptativos, pois ao aplicar a tranformação de *Householder* ao vetor de pesos, reduz-se a dimensão do subespaço onde a adaptação é realizada.

O Capítulo 5 juntou os conceitos apresentados em todos capítulos anteriores, a fim de mostrar como os algoritmos HC com precisão variável foram projetados e implementados. Todas operações matemáticas foram substitídas pelas funções CORDIC, desenvolvidas no Capítulo 2, e pôde-se verificar o correto funcionamento das mesmas através de alguns testes realizados com as funções associadas aos algoritmos HC com precisão variável.

No Capítulo 5, foi também possível identificar a função CORDIC mais crítica. De acordo com as figuras apresentadas neste capítulo, a função *mat_mult_c*, quando utilizada em operações de matriz com vetor ou matriz com matriz, é a função

CORDIC que introduz mais erros à resposta desejada. À medida que a dimensão da matriz ou vetor de entrada aumenta, maior será o erro nesta função.

Através do Capítulo 6, pode-se verificar que, para uma aplicação de antenas inteligentes, os algoritmos HC com precisão variável podem ser implementados em um *hardware* puramente CORDIC, todavia o número de *bits* de precisão deve ser maior do que 15, pois caso contrário as restrições não são satisfeitas e os nulos não são colocados nas direções dos interferidores. Em uma máquina CORDIC, o algoritmo HCLMS_C mostrou-se mais eficiente que o NHCLMS_C, pois consegue atingir a resposta ótima com menos *bits* de precisão. Essa diferença ocorre devido ao erro de cálculo introduzido pela normalização do NHCLMS_C quando poucos *bits* de precisão são utilizados.

O Capítulo 6 também mostrou que os algoritmo HC com precisão variável são mais eficientes que o CLMS_C e NCLMS_C, ou seja, caso deseja-se implementar um desses algoritmos num *hardware* que possua precisão variável, é recomendável utilizar os algoritmos HC. Resumindo, mesmo utilizando um *hardware* de precisão inferior, os algoritmos HC têm a capacidade de encontrar o mesmo resultado que o CLMS_C e NCLMS_C.

Os algoritmos HC são mais eficientes pois sua adaptação é realizada numa dimensão r (número de restrições) unidades inferior à dimensão dos algoritmos não HC, ou seja, CLMS_C e NCLMS_C. E como pode ser visto no Capítulo 5, o erro médio aumenta à medida que a dimensão dos vetores e matrizes aumentam, logo o erro médio dos algortimos não HC é superior ao erro introduzido pelos algoritmos HC.

A redução de posto não mostrou ser um método recomendável para ser aplicado em algoritmos HC. Sua implementação com o algortimo HCLMS_C não é recomendável, já que os nulos na direção de chegada dos interferidores não são respeitados. Já para o algortimo NHCLMS_C, a redução de posto não é recomendada pois, para sua utilização de forma satisfatória, é necessário ter uma precisão elevada, sendo assim tornando-o ineficiente.

Este trabalho deixa como contribuição acadêmica o aperfeiçoamento das funções CORDIC desenvolvidas em [3], a criação de diversas funções CORDIC para simular um *hardware* com precisão variável, uma análise do comportamento dos

algoritmos HC com precisão variável para uma aplicação de antenas inteligentes utilizando um *array* adaptativo e um estudo comparativo entre as funções HCLMS, NHCLMS, CLMS e NCLMS quando implementados num *hardware* com precisão variável. Esta tese comprova as vantagens de se utilizar os algoritmos HC num *hardware* puramente CORDIC.

7.2 Proposta futura

É proposto como um trabalho futuro a implementação da aplicação de *beamforming* numa DSP (*Digital Signal Processor*) CORDIC, a fim de comparar os resultados dessa implementação com os resultados encontrados nesta tese.

Referências Bibliográficas

[1] GOLUB, G. H., VAN LOAN, C. F., *Matrix Computations*. 3^{rd} ed. Prentice Hall, 1983.

[2] CAMPOS, M. L. R., WERNER, S., APOLINARIO JR., J. A., "Constrained adaptation algorithms employing Householder transformation", *IEEE Transactions on Signal Processing.*, v. 50, n. 9, pp. 2187–2195, Setembro 2002.

[3] COSTA, B. C., *Implementação de filtros de* W*iener com posto e precisão reduzidos*. M.Sc. dissertation, Universidade Federal do Rio de Janeiro, Março 2006.

[4] VOLDER, J. E., "The CORDIC trigonometric computing technique", *Signal Processing Series*, , Fevereiro 1999.

[5] WALTHER, J. S., "A unified algorithm for elementary functions", *AFIPS Spring Joint Computer Conference*, v. 38, pp. 379–85, 1971.

[6] MEGGITT, J. E., "Pseudo division and pseudo multiplication processes", *IBM J.*, pp. 210–226, Abril 1962.

[7] DAGGETT, D. H., "Decimal-binary conversions in CORDIC", *IEEE Trans. on Electronic Computers*, v. EC-8, n. 3, pp. 335–39, Setembro 1959.

[8] ANDRAKA, R., "A survey of CORDIC algorithms for FPGA based computers", *Andraka Consulting Magazine*, pp. 191–200, Fevereiro 1998.

[9] HWANG, K., *Computer Arithmetic: Principles, Architectures, and Design*. John Wiley and Sons, 1979.

[10] HAVILAND, G. L., TUSZYNSKI, A. A., "A CORDIC arithmetic processor chip", *IEEE Transactions on Computers*, v. C-29, n. 2, pp. 68–79, Fevereiro 1980.

[11] MEHLING, R., MEYER, R., "CORDIC-AU, a suitable supplementary unit to a general-purpose signal processor", *AEÜ (Archiv für Elektronik und Übertragungstechnik)*, v. 43, n. 6, pp. 394–397, 1989.

[12] MEYR, H., H., D., "CORDIC algorithms and architecture", *IRE Trans. Electronic Computers*, v. EC-8, n. 3, pp. 330–34, Setembro 1959.

[13] VEEN, B. D. V., BUCKLEY, K. M., "Beamforming: A versatile approach to spatial filtering", *IEEE ASSP Magazine*, pp. 4–23, Abril 1988.

[14] APPLEBAUM, S. P., CHAPMAN, D. J., "Adaptive arrays with main beam constraints", *IEEE Transactions of Antennas and Propagation*, v. 24, pp. 650–662, Setembro 1976.

[15] WIDROW, B., MANTEY, P. E., GRIFFITHS, L. J., "Adaptive antenna systems", *Proceedings of the IEEE*, v. 55, pp. 2143–2159, Dezembro 1967.

[16] MONZINGO, R., MILLER, T., *Introduction to Adaptive Arrays*. Wiley and Sons, 1980.

[17] CAMPOS, M. L. R., WERNER, S., APOLINARIO JR., J. A., "Householder-transform constrained LMS algorithms with reduced-rank updating", *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 1857–1860, Março 1999.

[18] DINIZ, P. S. R., *Adaptive Filtering*. 3^{rd} ed. Kluwer Academic Publishers, 2002.

[19] FROST III, O. L., "An algorithm for linearly constrained adaptive array processing", *Proceedings of the IEEE*, v. 60, n. 8, pp. 926–935, Agosto 1972.

[20] CAMPOS, M. L. R., WERNER, S., APOLINARIO JR., J. A., "Constrained quasi-Newton algorithm for CDMA mobile communications", *Proc. Int. Telecommun. Symp*, pp. 371–376, Agosto 1998.

[21] HONIG, M., TSATSANIS, M. K., "Adaptive techniques for multiuser CDMA receivers", *IEEE Transactions of Signal Processing*, pp. 46–61, Maio 2000.