

**UNIVERSIDADE DE BRASÍLIA**  
**FACULDADE DE TECNOLOGIA**  
**DEPARTAMENTO DE ENGENHARIA ELÉTRICA**

**IMPLEMENTAÇÕES EM PROCESSAMENTO DIGITAL DE**  
**SINAIS UTILIZANDO O TMS320C6711**

**KARINA MAHON MATTAR**

**ORIENTADORES: ALEXANDRE ZAGHETTO**

**TRABALHO DE CONCLUSÃO DO CURSO DE GRADUAÇÃO EM**  
**ENGENHARIA ELÉTRICA**

**BRASÍLIA/DF: DEZEMBRO/2006**

**UNIVERSIDADE DE BRASÍLIA**

**FACULDADE DE TECNOLOGIA  
DEPARTAMENTO DE ENGENHARIA ELÉTRICA**

**IMPLEMENTAÇÕES EM PROCESSAMENTO DIGITAL DE  
SINAIS UTILIZANDO O TMS320C6711**

**KARINA MAHON MATTAR**

**TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO AO  
DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE  
DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO  
PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO  
GRAU DE ENGENHEIRO.**

**APROVADA POR:**

---

**Alexandre Zaghetto, *M.Sc.*  
(Orientador)**

---

**Adson Ferreira Da Rocha, *Ph.D.*, ENE/UnB  
(Examinador)**

---

**Pedro de Azevedo Berger, *Dr.*, CIC/UnB  
(Examinador)**

**DATA: BRASÍLIA/DF, 07 DE DEZEMBRO DE 2006**

## **FICHA CATALOGRÀFICA:**

**MATTAR, KARINA MAHON**

Implementações em Processamento Digital de Sinais Utilizando o TMS320C6711  
[Distrito Federal] 2006.

xiv, 151 p., 297 mm (ENE/FT/UnB), Bacharel, Engenharia Elétrica, 2006). Trabalho de conclusão do curso de Graduação em Engenharia Elétrica – Universidade de Brasília. Faculdade de Tecnologia. Departamento de Engenharia Elétrica.

1. Aplicações em DSP utilizando o TMS3206711
2. Filtros com Resposta Impulsional Finita (FIR)
3. Filtros com Resposta Impulsional Infinita (IIR)
4. Transformada Rápida de Fourier (FFT)
5. Filtros Adaptativos

## **REFERÊNCIA BIBLIOGRÁFICA:**

MATTAR, K.M. (2006). Implementações em Processamento Digital de Sinais Utilizando o TMS320C6711. Trabalho de conclusão de curso, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF.

## **CESSÃO DE DIREITOS**

AUTOR: Karina Mahon Mattar

TÍTULO: Implementações em Processamento Digital de Sinais Utilizando o TMS320C6711

GRAU: Bacharel

ANO: 2006

É concedida à Universidade de Brasília permissão para reproduzir cópias desse trabalho de conclusão de curso e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte desse trabalho pode ser reproduzida sem a autorização por escrito do autor.

---

Karina Mahon Mattar  
SHIS QL04 Conjunto 01, Casa 18  
70510-310 – Brasília (DF) - Brasil

## **DEDICATÓRIA**

À minha filha, Isabella,  
que a cada instante enche minha vida de alegria.

# **IMPLEMENTAÇÕES EM PROCESSAMENTO DIGITAL DE SINAIS UTILIZANDO O TMS320C6711**

**Autor: Karina Mahon Mattar**

**Orientador: Adson Ferreira Da Rocha e Alexandre Zaghetto**

**Programa de Graduação em Engenharia Elétrica**

**Brasília, Dezembro de 2006**

Este trabalho apresenta uma série de tutoriais, desenvolvidos para o DSK TMS320C6711 da Texas Instruments, que podem ser utilizados como material de suporte em cursos de Processamento Digital de Sinais (PDS). Tais tutoriais apresentam o *kit* TMS320C6711, bem como o ambiente de desenvolvimento *Code Composer Studio*, e conduzem o leitor na implementação de experimentos que aplicam na prática alguns tópicos em processamento digital de sinais, tais como filtros FIR, filtros IIR, transformada discreta de Fourier (DFT), transformada rápida de Fourier (FFT) e filtros adaptativos. Cada tutorial traz uma discussão teórica inicial seguida de práticas de laboratório que a consolida.

## **ABSTRACT**

### **DIGITAL SIGNAL PROCESSING IMPLEMENTATIONS USING THE TMS320C6711**

**Author: Karina Mahon Mattar**

**Supervisor: Adson Ferreira Da Rocha e Alexandre Zaghetto**

**Programa de Graduação em Engenharia Elétrica**

**Brasília, December of 2006**

This work presents a set of tutorials developed to DSK TMS320C6711, a Texas Instruments digital signal processor starter kit, which can be used as a supplementary material for Digital Signal Processing (DSP) graduated and undergraduate course. Those tutorials presents the TMS320C6711 kit, as well as the integrated development environment *Code Composer Studio*, conducting the reader through the implementation of experiments which applies some topics of digital signal processing theory, such as FIR and IIR filters, Fourier Discrete Transform (FDT), Fourier Fast Transform (FFT) and adaptive filters. Each tutorial also brings a laconic theory discussion followed by practical laboratory experiments.

# ÍNDICE

Capítulo	Página
1 - INTRODUÇÃO.....	3
1.1 OBJETIVO .....	3
1.2 ORGANIZAÇÃO DO TRABALHO.....	4
2 - TUTORIAL 1: APLICAÇÕES EM DSP UTILIZANDO O TMS3206711 .....	5
2.1 OBJETIVO .....	5
2.2 INTRODUÇÃO.....	5
2.2.1 Ferramentas de apoio do DSK.....	6
2.2.2 Placa DSK.....	7
2.2.3 Processador Digital de Sinais TMS320C6711 .....	8
2.2.4 Code Composer Studio .....	9
2.2.5 Instalação do Code Composer Studio v3.1 .....	9
2.2.6 Tipos de arquivos úteis.....	11
2.3 EXEMPLOS PARA TESTE DAS FERRAMENTAS DO DSK.....	11
2.3.1 Exemplo 1: Teste rápido do DSK .....	11
2.3.2 Exemplo 2: Teste de confiança.....	14
2.3.3 Exemplo 3: Geração de onda senoidal .....	16
2.3.4 Exemplo 4: Geração e Simulação de Gráficos de uma Senoide .....	25
2.3.5 Exemplo 5: Produto entre dois Vetores.....	33
3 - TUTORIAL 2: FILTROS COM RESPOSTA IMPULSIONAL FINITA (FIR).....	48
3.1 OBJETIVO .....	48
3.2 INTRODUÇÃO.....	48
3.2.1 Sinais Discretos.....	48
3.2.2 Introdução a Transformada Z.....	50
3.2.3 Transformada Z inversa .....	51
3.2.4 Equações de Diferenças .....	52
3.2.5 Filtros Digitais .....	54
3.2.6 Filtros com Resposta Impulsional Finita (FIR) .....	55
3.3 EXEMPLOS DE FILTROS COM RESPOSTAS IMPULSIONAL FINITA (FIR)	61
3.3.1 Exemplo 1: Filtro Rejeita-Faixa .....	63
3.3.2 Exemplo 2: Filtro Passa-Faixa .....	72
3.3.3 Exemplo 3: Implementação de dois filtros FIR rejeita-faixa para a recuperação de uma entrada de voz corrompida.....	77
4 - TUTORIAL 3: FILTROS COM RESPOSTA IMPULSIONAL INFINITA (IIR) .....	82
4.1 OBJETIVO .....	82
4.2 INTRODUÇÃO.....	82
4.1 Filtros com Resposta Impulsional Infinita (IIR) .....	82
4.2 Transformação Bilinear .....	88
4.3 EXEMPLOS DE FILTROS COM RESPOSTA IMPULSIONAL INFINITA (IIR)	90
4.3.1 Exemplo 1: Filtro Passa-Baixa .....	90

4.3.2	<i>Exemplo 2: Filtro Passa-Faixa</i>	93
5 -	TUTORIAL 4: TRANSFORMADA RÁPIDA DE FOURIER (FFT)	98
5.1	OBJETIVO	98
5.2	INTRODUÇÃO	98
5.2.1	<i>Transformada Discreta de Fourier (DFT)</i>	98
5.2.2	<i>Desenvolvimento do algoritmo raiz-2 com decimação no tempo</i>	101
5.2.3	<i>Desenvolvimento do algoritmo raiz-2 com decimação na frequência</i>	104
5.2.4	<i>Algoritmo de raiz-4</i>	107
5.2.5	<i>Transformada Inversa Discreta de Fourier (IDFT)</i>	108
5.3	EXEMPLOS DE TRANSFORMADA DISCRETA DE FOURIER (DFT) E TRANSFORMADA RÁPIDA DE FOURIER (FFT)	109
5.3.1	<i>Exemplo 1: DFT de uma Seqüência de números reais com saída para a janela do CCS</i>	109
5.3.2	<i>Exemplo 2: FFT formulada por Danielson -Lanczos</i>	119
6 -	TUTORIAL 5: FILTROS ADAPTATIVOS	128
6.1	OBJETIVO	128
6.2	INTRODUÇÃO	128
6.2.1	<i>Filtros adaptativos</i>	128
6.2.2	<i>Algoritmo LMS - Least Mean Square</i>	131
6.2.3	<i>Estruturas Adaptativas</i>	133
6.3	EXEMPLOS DE FILTROS ADAPTATIVOS	136
6.3.1	<i>Exemplo 1: FIR adaptativo para identificação de sistemas</i>	136
7 -	CONCLUSÃO	140
	REFERÊNCIA BIBLIOGRÁFICA	141
	ANEXO I	145



## LISTA DE TABELAS

<b>Tabela</b>	<b>Página</b>
Tabela 3.3.1 – Coeficientes e amostras .....	61
Tabela 3.3.2 – Coeficientes e amostras .....	62
Tabela 5.3.1 – Reordenação utilizando a inversão de Bits .....	122

## LISTA DE FIGURAS

Figura	Página
Figura 2.2.1 – Conversões AD e DA.....	6
Figura 2.2.1.1 – Uma placa com o C6711.....	7
Figura 2.2.5.1 – Habilitando os recursos para o C6711.....	10
Figura 2.3.1.2 – Reiniciando a configuração do CCS.....	12
Figura 2.3.1.3 – Configurando a família e a plataforma.....	12
Figura 2.3.1.4 – Sugestão de execução do CCS.....	13
Figura 2.3.1.5 – Mensagem de conexão.....	13
Figura 2.3.3.1 – Gráfico da função $f(t) = 1000\sin(t)$ .....	17
Figura 2.3.3.2 – Criação do projeto.....	18
Figura 2.3.3.3 – Visualização do projeto.....	19
Figura 2.3.3.4 – Configuração do (a) compilador e do (b) linker.....	21
Figura 2.3.3.5 – Carregando o executável.....	22
Figura 2.3.3.6 – Quick Watch.....	23
Figura 2.3.3.7 – Alterando o valor de variáveis.....	23
Figura 2.3.3.8 – Introdução do arquivo amplitude.gel.....	24
Figura 2.3.3.9 – Ajuste da amplitude do sinal.....	25
Figura 2.3.4.1 – Criação do projeto.....	26
Figura 2.3.4.2 – Visualização do projeto.....	27
Figura 2.3.4.3 – Carregando o executável.....	29
Figura 2.3.4.4 – Janela Graph Property Dialog.....	30
Figura 2.3.4.5 – Janela Graph Property Dialog com modificações.....	31
Figura 2.3.4.6 – Resposta no domínio do tempo.....	31
Figura 2.3.4.7 – Janela Graph Property Dialog com modificações.....	32
Figura 2.3.4.8 – Resposta no domínio da frequência.....	32
Figura 2.3.5.1 – Visualização do projeto.....	35
Figura 2.3.5.2 – Configurações do compilador.....	36
Figura 2.3.5.3 – Configurações do linker.....	36
Figura 2.3.5.4 – Carregando o executável.....	37
Figura 2.3.5.5 – Saída.....	37
Figura 2.3.5.6 – Programa principal.....	38
Figura 2.3.5.7 – Ajuste do Profile.....	39
Figura 2.3.5.8 – Indicação de habilitação do Profile.....	39
Figura 2.3.5.9 – Ajuste do Profile: Ranges.....	40
Figura 2.3.5.10 – Ajuste do Profile: Ranges.....	41
Figura 2.3.5.11 – Ajuste do Profile: Custom/ Cycles.....	41
Figura 2.3.5.12 – Configurações do compilador.....	42
Figura 2.3.5.13 – Visualizador do Profile.....	43
Figura 2.3.5.14 – Visualizador do Profile.....	44
Figura 2.3.5.15 – Janela de conselhos.....	44
Figura 2.3.5.16 – Janela de conselhos.....	45
Figura 2.3.5.17 – Janela de conselhos.....	45
Figura 2.3.5.18 – Configurações do compilador.....	46
Figura 2.3.5.19 – Visualizador do Profile.....	47

Figura 3.2.1.1 – Representação de um sistema no tempo discreto.....	49
Figura 3.2.6.1 – Forma direta para os filtros digitais FIR. ....	57
Figura 3.2.6.2 – Função transferência desejada: (a) passa-baixas; (b) passa-altas; (c) passa-faixa; (d) rejeita-faixa. ....	58
Figura 3.3.1.1 – Visualização do projeto.....	65
Figura 3.3.1.2 – Janela Graph Property Dialog com modificações.....	67
Figura 3.3.1.3 – Resposta impulsional. ....	67
Figura 3.3.1.4 – Janela Graph Property Dialog com modificações para a FFT. ....	68
Figura 3.3.1.5 – FFT do filtro rejeita-faixa. ....	68
Figura 3.3.1.6 – Janela SPTool:startup.stp. ....	69
Figura 3.3.1.7 – Características do filtro FIR rejeita-faixa centrado em 2700Hz. ....	70
Figura 3.3.1.8 – Resposta em frequência da saída do filtro FIR rejeita-faixa centrado em 2700Hz, obtida com um osciloscópio. ....	71
Figura 3.3.1.9 – Resposta em frequência da saída do filtro FIR rejeita-faixa centrado em 2700Hz, obtido por meio da interpolação. ....	71
Figura 3.3.2.1 – Visualização do projeto.....	73
Figura 3.3.2.2 – Resposta impulsional. ....	74
Figura 3.3.2.3 – FFT do filtro passa-faixa centrado em 1750 Hz. ....	74
Figura 3.3.2.4 – Características do filtro FIR passa-faixa centrado em 1750Hz.....	75
Figura 3.3.2.5 – Resposta em frequência da saída do filtro FIR passa-faixa centrado em 1750Hz, obtido com um osciloscópio. ....	76
Figura 3.3.2.6 – Resposta em frequência da saída do filtro FIR passa-faixa centrado em 1750Hz, obtido por meio da interpolação. ....	77
Figura 3.3.3.1 – Visualização do projeto.....	79
Figura 3.3.3.2 – Espectro do sinal de voz corrompido por dois sinais senoidais centrados em 900Hz e 2700Hz. ....	80
Figura 3.3.3.3 – Resposta em frequência da saída dos filtros FIR rejeita-faixas em série centrados em 900Hz e 27000Hz, obtida no osciloscópio.....	80
Figura 3.3.3.4 – Resposta em frequência do sinal recuperado. ....	81
Figura 4.1.1 – Filtro IIR composto por dois filtros FIR. ....	83
Figura 4.1.2 – Forma direta do Tipo I do filtro IIR. ....	84
Figura 4.1.3 – Forma direta do Tipo I de segunda ordem. ....	84
Figura 4.1.4 – Forma direta do Tipo I de segunda ordem $H(z) = H_2(z)H_1(z)$ ....	85
Figura 4.1.5 – Forma direta do Tipo II de segunda ordem.....	85
Figura 4.1.6 – Forma direta do tipo II. ....	86
Figura 4.1.7 – Forma cascata de um filtro IIR.....	87
Figura 4.1.8 – Filtro IIR de quarta ordem com duas seções na forma direta do tipo II em cascata.....	87
Figura 4.1.9 – Forma paralela de um filtro IIR. ....	88
Figura 4.2.1. – Relação entre frequências analógicas e digitais. ....	89
Figura 4.3.1.1 – Visualização do projeto.....	92
Figura 4.3.1.2 – Resposta em frequência da saída do filtro IIR passa-baixa com frequência de corte de 2000Hz, obtida no osciloscópio.....	93
Figura 4.3.1.3 – Resposta em frequência da saída do filtro IIR passa-baixa com frequência de corte de 2000Hz, obtida por meio da interpolação. ....	93
Figura 4.3.2.1 – Visualização do projeto.....	95
Figura 4.3.2.2 – Resposta em frequência da saída do filtro IIR passa-faixa centrado em 2000Hz, obtida no osciloscópio.....	96
Figura 4.3.2.3 – Características do filtro IIR passa-faixa centrado em 2000Hz. ....	97
Figura 5.2.1.1 – Comparação entre as funções $f(N) = N^2$ e $f(N) = N \log_2 N$ .....	100

Figura 5.2.1.2 – Visualização das propriedades de simetria e de periodicidade. ....	101
Figura 5.2.2.1 – Célula Básica do algoritmo de FFT com decimação no tempo. ....	102
Figura 5.2.2.2 – Decomposição de uma DFT com 8 pontos em duas DFTs de 4 pontos com decimação no tempo. ....	103
Figura 5.2.2.3 – Decomposição de duas DFTs de 4 pontos em quatro DFTs de 2 pontos com decimação no tempo. ....	103
Figura 5.2.2.4 – FFT de oito pontos usando decimação no tempo. ....	104
Figura 5.2.3.1 – Célula Básica do algoritmo de FFT com decimação na frequência. ....	105
Figura 5.2.3.2 – Decomposição de uma DFT com 8 pontos em duas DFTs de 4 pontos com decimação na frequência. ....	106
Figura 5.2.3.3 – Decomposição de duas DFTs de 4 pontos em quatro DFTs de 2 pontos com decimação na frequência. ....	106
Figura 5.2.3.4 – FFT de oito pontos usando decimação na frequência. ....	107
Figura 5.3.1.1 – Gráfico da função $x(n) = 1000 * \cos\left(\frac{\pi}{4}n\right)$ ; Frequência do sinal $f = 1000$ Hz e Frequência de amostragem $F_s = 8000$ Hz . ....	111
Figura 5.3.1.2 – Valores da componente real (outRe) da DFT de $N = 8$ gerada no MATLAB. ....	111
Figura 5.3.1.3 – Valores da componente imaginária (outIm) da DFT de $N = 8$ gerada no MATLAB. ....	112
Figura 5.3.1.4 – Gráfico da função $x(n) = 1000 \times \sin\left(\frac{\pi}{5}n\right)$ ; Frequência do sinal $f = 800$ Hz e Frequência de amostragem $F_s = 8000$ Hz . ....	113
Figura 5.3.1.5 – Valores da componente real (outRe) da DFT de $N = 20$ gerada no MATLAB. ....	113
Figura 5.3.1.6 – Valores da componente imaginária (outIm) da DFT de $N = 20$ gerada no MATLAB. ....	113
Figura 5.3.1.7 – Visualização do projeto. ....	114
Figura 5.3.1.8 – Visualização da janela Watch Window para $N = 8$ . ....	115
Figura 5.3.1.9 – Componentes real (outRe) e imaginária (outIm) da DFT de $N = 8$ . ....	115
Figura 5.3.1.10 – Componentes real (outRe) e imaginária (outIm) da DFT de $N = 8$ geradas no MATLAB. ....	116
Figura 5.3.1.11 – Complexidade da função dft.c de uma DFT de comprimento $N = 8$ ...	116
Figura 5.3.1.12 – Visualização da janela Watch Window para $N = 20$ . ....	117
Figura 5.3.1.13 – Valores da componente real (outRe) da DFT de $N = 20$ . ....	117
Figura 5.3.1.14 – Valores da componente imaginária (outIm) da DFT de $N = 20$ . ....	118
Figura 5.3.1.15 – Componentes real (outRe) e imaginária (outIm) da DFT de $N = 20$ geradas no MATLAB. ....	118
Figura 5.3.1.16 – Complexidade da função dft.c de uma DFT de comprimento $N = 20$ . ....	118
Figura 5.3.2.1 –Gráfico da função $x(n) = 1000 * \sin\left(\frac{\pi}{8}n\right)$ ; Frequência do sinal $f = 500$ Hz e Frequência de amostragem $F_s = 8000$ Hz . ....	123
Figura 5.3.2.2 – Componentes real (outRe) e imaginária (outIm) da FFT de $N = 32$ geradas no MATLAB. ....	124
Figura 5.3.2.3 – Componentes real (outRe) e imaginária (outIm) da FFT de $N = 8$ . ....	125
Figura 5.3.2.4 – Complexidade da função fft.c de uma FFT de comprimento $N = 8$ . ....	125
Figura 5.3.2.5 – Valores da componente real (outRe) e da FFT de $N = 32$ . ....	126

Figura 5.3.2.6 – Valores da componente imaginária (outIm) da FFT de $N = 32$ .....	127
Figura 5.3.2.7 – Complexidade da função fft.c de uma FFT de comprimento $N = 32$ ...	127
Figura 6.2.1.1 – Forma geral de um combinador linear adaptativo.....	129
Figura 6.2.1.2 – Diagrama de um filtro adaptativo FIR. ....	129
Figura 6.2.1.3 – Estrutura Básica de um filtro Adaptativo.....	130
Figura 6.2.3.1 – Identificação de Sistema utilizando filtro adaptativo.....	133
Figura 6.2.3.2 – Sistema de modelamento de inversão utilizando um filtro adaptativo. ..	134
Figura 6.2.3.3 – Estrutura de um Cancelamento de Ruído.....	134
Figura 6.2.3.4 – Estrutura de Predição Adaptativa.....	135
Figura 6.3.1.1 – Visualização do projeto.....	137
Figura 6.3.1.2 – Janela Graph Property Dialog com modificações.....	138
Figura 6.3.1.3 – Sinal de cima: Sinal desejado (DESIRED); Sinal de baixo saída do filtro (Y_out). ....	139

## LISTA DE SÍMBOLOS, NOMENCLATURA E ABREVIACÕES

ADC	<i>Analog-to-digital Conversion</i>
CCS	<i>Code Composer Studio</i>
CPU	<i>Central Processing Unit</i>
DSP	<i>Digital Signal Processors</i>
DSK	<i>DSP start kit</i>
DCA	<i>Digital-to-analog Conversion</i>
IDE	<i>Integrated Development Environmet</i>
ISR	<i>Interrupt Service Routine</i>
GEL	<i>General Extension Language</i>
McBSPs	<i>Multichannel Buffered Serial Ports</i>
PCHIP	<i>Piecewise Cubic Interpolating Polynomial</i>
RTDX	<i>Real-time Data Exchange</i>
SDRAM	<i>Synchronous Dynamic RAM</i>
SLR	<i>Source Line Reference</i>
TI	<i>Texas Instruments</i>
VLIW	<i>Very-Long-Instruction-Word</i>

# 1 - INTRODUÇÃO

DSPs (Digital Signal Processors) ou Processadores Digitais de Sinais são utilizados em diversas áreas, tais como telecomunicações, controle, segurança, medicina, processamento de imagens, vídeo e áudio, entre outras. Podem ser encontrados em sistemas de transmissão sem fio, de identificação e classificação biométrica, de navegação, de monitoramento biofísico, em soluções DSL (*Digital Subscriber Line*), VoIP (*Voice over Internet Protocol*), em câmeras de foto e vídeo digitais, ou seja, em uma gama bastante abrangente de aplicações.

Esses processadores também ocupam espaço dentro da sala de aula, nas universidades, oferecendo aos alunos de graduação e pós-graduação uma forma econômica de se estudar, em tempo real, tópicos em processamento digital de sinais.

A *Texas Instruments* disponibiliza como produto a família de processadores TMS320C6x, baseada na arquitetura VLIW (*Very-Long-Instruction-Word*). Essa arquitetura oferece recursos que facilitam o desenvolvimento de compiladores bastante eficientes para linguagem de alto nível. Apesar do *assembly* para o TMS320C6x produzir códigos rápidos, alguns problemas no que diz respeito à documentação e manutenção podem ocorrer. Ao longo deste trabalho, as linguagens C e C++ serão simplesmente referenciadas por C. Com o compilador C que o ambiente de desenvolvimento disponibiliza o programador precisa apenas deixar que a ferramenta faça o trabalho.

## 1.1 OBJETIVO

Este projeto tem por objetivo oferecer a estudantes de engenharia elétrica ou de computação uma série de tutorias que apresentam uma dimensão prática da disciplina de processamento digital de sinais (PDS). É convicção que os princípios abordados em PDS podem mais facilmente ser apreciados por meio de implementações em tempo real. Assume-se que, como pré-requisito, os alunos tenham adquirido conhecimentos em sistemas lineares, processamento digital de sinais e algum conhecimento em linguagem C.

Os tutoriais começam com uma discussão teórica, seguida de um ou mais exemplos práticos que a consolidam. Foram elaboradas treze práticas de laboratório, com a maior parte dos códigos escrita em C e poucas linhas escritas em *assembly*. Os tutoriais abordam tópicos bastante conhecidos em processamento digitais de sinais.

O conteúdo desse projeto pode ser utilizado de diversas formas como, por exemplo, em complemento a um curso de PDS, em um curso específico de tópicos especiais em PDS ou em seminários e oficinas.

## 1.2 ORGANIZAÇÃO DO TRABALHO

O Capítulo 2 apresenta alguns aspectos da placa DSK, bem como cinco exemplos básicos, cujo objetivo é fazer o aluno explorar o *Code Composer Studio* (CCS), ferramenta que acompanha o TMS3206711 *DSP Starter Kit*.

O Capítulo 3 trata de filtros FIR (*Finite Impulse Response*), através da conceituação destes e posterior apresentação de 3 exemplos; filtro rejeita-faixa, filtro passa-faixa; e o último exemplo apresenta a implementação de dois filtros rejeita-faixa para a recuperação de uma entrada de voz corrompida.

O Capítulo 4 introduz os conceitos de filtros IIR (*Infinite Impulse Response*), e apresenta a implementação de dois exemplos; o primeiro constitui um filtro passa-baixa e o segundo apresenta um filtro passa-faixa.

O Capítulo 5 trata sobre FFT (*Fast Fourier Transform*), e é constituído por dois exemplos; Transformada Discreta de Fourier (DFT) e Transformada Rápida de Fourier.

O Capítulo 6 apresenta uma abordagem sobre filtros adaptativos e a implementação de um exemplo sobre filtro FIR adaptativo para sistemas de identificação.

Por fim, o Capítulo 7 traz a conclusão do trabalho.



## 2 - TUTORIAL 1: APLICAÇÕES EM DSP UTILIZANDO O TMS320C6711

### 2.1 OBJETIVO

- Teste do Code Composer Studio™ DSK v3.1 IDE
- Utilização do TMS320C6711 DSK
- Implementação de exemplos para testar as ferramentas de trabalho.

Esse tutorial trata das ferramentas que serão utilizadas ao longo do trabalho. Apresenta o popular *Code Composer Studio* (CCS), que oferece um ambiente de desenvolvimento integrado (*Integrated Development Environment* - IDE); o DSP *starter kit* (DSK), com o processador TMS320C6711 e completo suporte a entrada e saída. Cinco exemplos serão trabalhados com o objetivo de se testar as ferramentas de *software* e *hardware* que o DSK oferece.

### 2.2 INTRODUÇÃO

Processadores digitais de sinais como os da família TMS320C6x são como microprocessadores rápidos, de aplicação específica, com um tipo especializado de arquitetura e conjunto de instruções, apropriados ao processamento de sinais. A notação C6x é utilizada para se designar um membro da família de processadores TMS320C6000 da *Texas Instruments* (TI). A arquitetura do processador C6x é bastante adequada à realização de cálculos numéricos. Baseado na arquitetura que pode ser traduzida como muito-longa-palavra-de-instrução (*very-long-instruction-word* – VLIW), o C6x é considerado o mais poderoso processador da TI.

Processadores digitais de sinais são utilizados em uma gama bastante abrangente de aplicações, que vão de telecomunicações e controle ao processamento de imagem e voz. São encontrados em aparelhos de telefonia celular, fax/modems, *drives* de disco, rádios, DVDs, TVs digitais, e etc. Tais processadores apresentam-se como uma opção interessante para um número considerável de aplicações que atendem ao consumidor final, uma vez que

diminuem o custo de produção de um determinado equipamento, aumentando o lucro obtido na venda do mesmo.

DSPs são mais freqüentemente utilizados no processamento em tempo real. Isso quer dizer que o processamento deve manter o passo com algum evento externo. Tal evento é normalmente uma entrada analógica. Enquanto sistemas analógicos apresentam maior sensibilidade à variação de condições ambientais, como a temperatura, por exemplo, sistemas baseados em DSPs apresentam maior robustez em relação à variação de tais condições. Dessa forma, os DSPs se beneficiam das mesmas vantagens que um microprocessador. São fáceis de usar, flexíveis e econômicos.

As aplicações mais comuns que utilizam esses processadores consideram sinais que estão entre as freqüências 0 e 20kHz. A voz pode ser amostrada a 8 kHz, o que implica em um período de amostragem de 1/8kHz ou 0.125 ms. Normalmente, a freqüência de amostragem utilizada em um *compact disk* (CD) é de 44.1kHz.

O DSK inclui um *codec*, o TLC320AD535 (AD535), que é ao mesmo tempo um conversor AD (analógico-digital) e um conversor DA (digital-analógico). O conversor AD captura o sinal analógico de entrada e gera uma representação digital desse sinal, que é processada pelo DSP. O resultado do processamento é entregue ao conversor DA, que gera o sinal analógico de saída, conforme pode ser observado na Figura 2.1 [Chassaing & Horning, 1990].

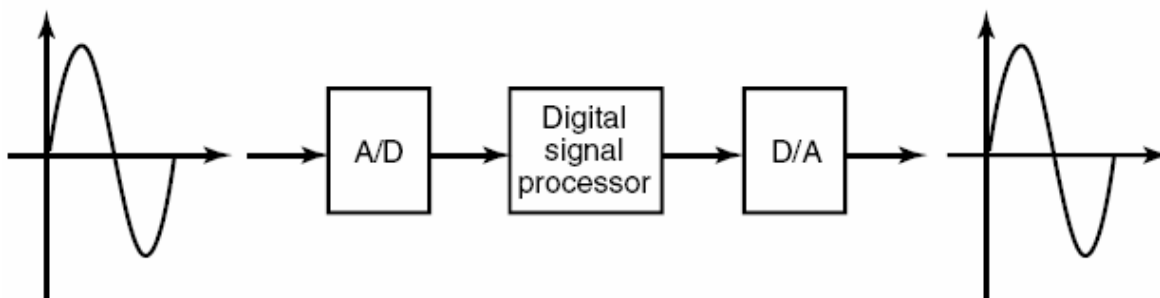


Figura 2.2.1 – Conversões AD e DA.

### 2.2.1 Ferramentas de apoio do DSK

Para que os experimentos possam ser realizados, (em todos os tutoriais), as ferramentas abaixo são necessárias:

1. DSP *starter kit* (DSK) da TI. O pacote DSK inclui:
  - (a) O Code Composer Studio™ DSK v3.1 IDE.
  - (b) Uma placa com o processador TMS320C6711 (C6711), *Figura 2.2.1.1*.
  - (c) Um cabo paralelo (DB25) que conecta a placa ao PC.
  - (d) Uma fonte de tensão que alimenta a placa.
2. Um PC IBM-compatível

Todos os arquivos/ programas discutidos nos tutoriais foram incluídos em um CD que acompanha este volume.

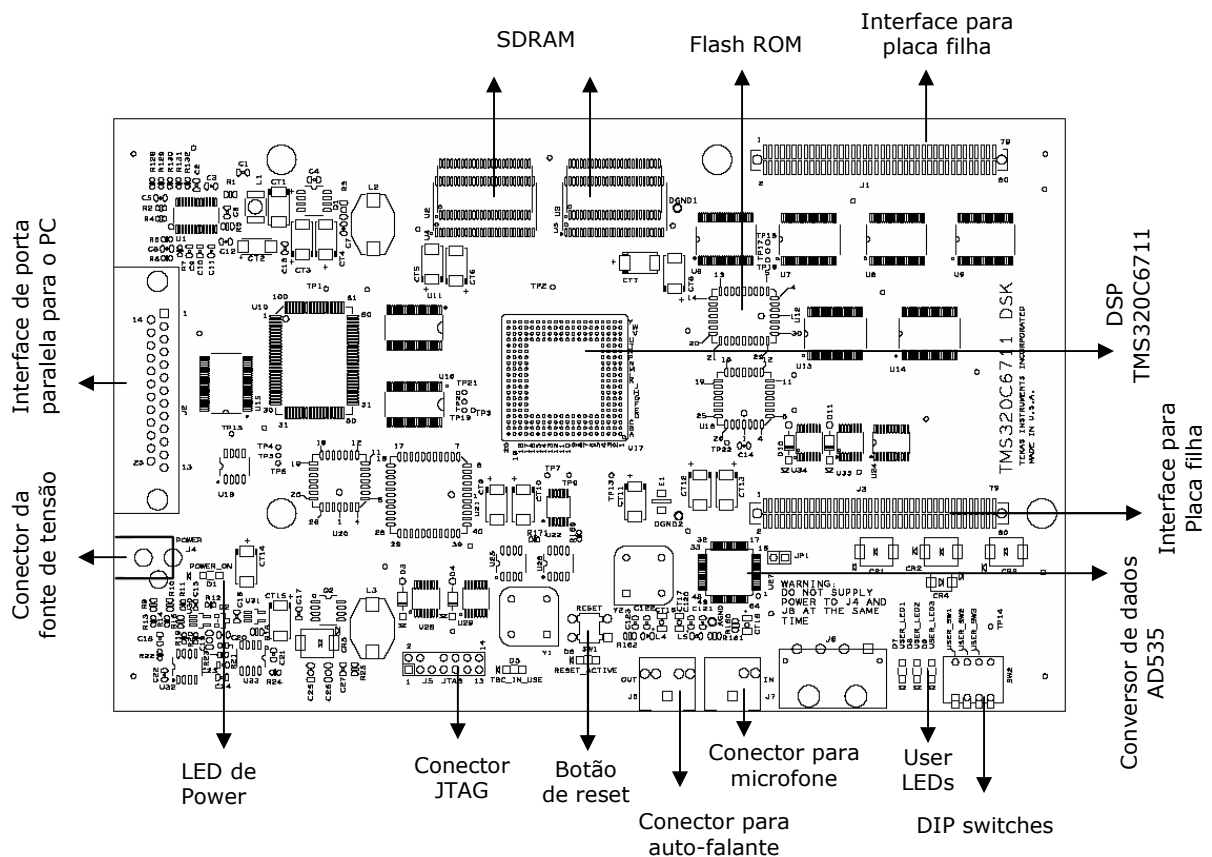


Figura 2.2.1.1 – Uma placa com o C6711.

## 2.2.2 Placa DSK

O pacote DSK, ainda relativamente caro, oferece as ferramentas de suporte de software e hardware necessários ao processamento digital de sinais em tempo real. É um sistema completo de DSP. A placa DSK, com dimensões aproximadas de 5x8 polegadas,

inclui o processador digital de sinais C6711, que trabalha em ponto flutuante, e o *codec* de 16 bits, AD535, para entrada e saída [Kehtarnavaz & Simsek, 2000].

O *codec* AD535 utiliza uma tecnologia sigma-delta [TMS320C6000C, 1998] que realiza as conversões analógico-digital (*analog-to-digital conversion* - ADC) e digital-analógico (*digital-to-analog conversion* - DAC). A taxa de amostragem desse *codec* é fixa em 8 kHz.

A placa DSK também oferece uma expansão para placa filha. Em aplicações que requerem taxas de amostragem até 72 kHz, recomenda-se a utilização de uma placa de expansão baseada no *codec stereo* PCM3003, disponibilizado pela Texas Instruments.

A placa DSK ainda possui 16MB de SDRAM (*synchronous dynamic RAM*) e 128kB de *flash* ROM. Dois conectores possibilitam a entrada e saída dados e são identificados por IN (J7) e OUT (J6), respectivamente. Três dos quatro *dip switches* disponíveis podem ser lidos por um programa. O *clock* da placa é de 150 MHz.

### 2.2.3 Processador Digital de Sinais TMS320C6711

O TMS320C6711 (C6711) é baseado na arquitetura *very-long-instruction-word* (VLIW) [TMS320C6000, 1998], que é bastante apropriada para algoritmos numéricos intensivos. A memória interna de programa é estruturada de maneira a possibilitar a busca de oito instruções por ciclo. Por exemplo, com um *clock* de 150 MHz, o C6711 é capaz de buscar oito instruções de 32 bits a cada 1/150MHz ou 6.66ns [Dahnoum, 2000].

As especificações do C6711 incluem 72KB de memória interna, oito unidades funcionais ou de execução, compostas por seis ALUs (Unidade Lógica e Aritmética), e duas unidades multiplicadoras, um barramento de endereço de 32 bits que pode endereçar 4GB, e dois conjuntos de registradores de 32 bits.

O C67xx (como o C6701 e o C6711) pertencem à família de processadores C6x que operam em ponto flutuante; já o C62xx e o C64xx pertencem à família C6x que operam em ponto fixo. O C6711 é capaz de realizar processamento tanto em ponto fixo como em ponto flutuante.

## 2.2.4 Code Composer Studio

O *Code Composer Studio* (CCS) provê um ambiente de desenvolvimento integrado (*integrated development environment* - IDE) que incorpora os recursos de *software*. O CCS inclui ferramentas para geração de códigos, como um compilador C [TMS320C60000, 1999], um *assembler* [TMS320C6000, 1999], e um *linker*. Apresenta ainda interface gráfica e possibilita o *debug* em tempo real [Chassaing, 1999].

O compilador C compila um programa com a extensão “.c” para produzir um código fonte *assembly* “.asm”. O *assembler* monta o arquivo fonte “.asm” e produz um objeto em linguagem de máquina com a extensão “.obj”. O *linker* combina arquivos de objetos com bibliotecas de objetos para produzir um arquivo executável com a extensão “.out”. Esse arquivo executável pode ser carregado e executado diretamente no processador C6711 [TMS320C6000, 2000].

Para se criar uma aplicação, basta adicionar os arquivos apropriados ao projeto. O compilador e o *linker* oferecem uma série de parâmetros de configuração. Várias opções de *debug* são disponibilizadas, incluindo o uso de pontos de quebra (*breakpoints*), a observação do conteúdo de variáveis, da memória, de registradores, além de resultados gráficos. Pode-se ainda passear pelo programa de diferentes formas (*step into*, *step over* ou *step out*).

A análise em tempo real é facilmente implementada por meio do *real-time data exchange* (RTDX) associado ao DSP/BIOS. O RTDX permite a troca de dados e a análise sem que o DSK precise ser interrompido. Estatísticas e desempenho podem ser monitorados em tempo real [Kehtarnavaz & Simsek, 2000].

## 2.2.5 Instalação do Code Composer Studio v3.1

Para se instalar o CCS, inicialmente deve-se conectar a interface J2 da placa DSK à porta paralela do PC (LTP1 ou LTP2) por meio do cabo paralelo (DB25). A fonte de tensão que acompanha o *kit* deve ser ligada ao conector J4 da placa DSK. Em seguida, deve-se instalar o CCS. Recomenda-se utilizar o diretório padrão “C:\CCStudio\_v3.1”.

É preciso garantir que os recursos para o processador C6711 serão instalados. Para isso, deve-se escolher o modo de instalação customizado (*Custom Install*). Em seguida

seleciona-se a opção “*Entire feature will be installed on local drive*” para a plataforma TMS320C6000, conforme ilustrado na Figura 2.2.5.1.



Figura 2.2.5.1 – Habilitando os recursos para o C6711.

Ao final da instalação, dois ícones são adicionados na área de trabalho do Windows. O ícone “CCStudio 3.1” é utilizado para iniciar o CCS e o ícone “Setup CCStudio v3.1” é utilizado para configurá-lo.

Quando a placa DSK é ligada, os LEDs localizados próximo aos quatro *dip switches* devem contar de 1 a 7 em binário.

Há uma grande quantidade de material de suporte (arquivos pdf) que acompanha o CCS, além de alguns exemplos e tutoriais.

O CCS 3.1 foi utilizado na elaboração e teste dos tutoriais desenvolvidos no presente trabalho. Uma série de arquivos, que se encontram nos subdiretórios dentro do diretório C:\CCStudio\_v3.1, podem ser de grande utilidade. Alguns desses subdiretórios são:

- *docs*: contém documentação e manuais.
- *MyProjects*: todos os programas e projetos podem ser colocados dentro desse subdiretório.
- *bin*: contém vários tutoriais.
- *tutorial*: contém tutoriais incluídos no CCS
- *C6000\cgtools*: contém ferramentas de geração de código.
- *C6000\examples*: contém exemplos incluídos no CCS.
- *C6000\RTDX*: contém arquivos de suporte para transferência de dados em tempo real.

- *C6000bios*: contém arquivos de suporte para DSP/BIOS.

### 2.2.6 Tipos de arquivos úteis

Estar-se-á trabalhando com vários arquivos de extensões diferentes. Entre eles estão:

- *arquivo.pjt*: arquivo de projeto.
- *arquivo.c*: código fonte C.
- *arquivo.asm*: código fonte *assembly* criado pelo usuário, pelo compilador C ou pelo otimizador linear.
- *arquivo.sa*: código fonte *assembly* linear. O otimizador linear utiliza o *arquivo.sa* como entrada para produzir um código *assembly* *arquivo.asm*.
- *arquivo.h*: arquivo de cabeçalho.
- *arquivo.lib*: arquivo de biblioteca.
- *arquivo.cmd*: arquivo de comando do *linker* que mapeia seções na memória.
- *arquivo.obj*: arquivo de objeto criado pelo *assembler*.
- *arquivo.out*: arquivo executável criado pelo *linker* que pode ser carregado e executado no processador.

## 2.3 EXEMPLOS PARA TESTE DAS FERRAMENTAS DO DSK

Com o objetivo de desenvolver um material de suporte para cursos de PDS, foram elaborados cinco exemplos que ilustram algumas das características do CCS e da placa DSK. O foco principal é familiarizar o usuário com as ferramentas de *software* e *hardware* disponíveis. Sugere-se que não se passe aos próximos tutoriais sem antes ter completado os dois primeiros exemplos que compõem este tutorial.

### 2.3.1 Exemplo 1: Teste rápido do DSK

Abra o CCS. Se for a primeira vez que você estiver rodando o programa, uma mensagem é apresentada solicitando a configuração do dispositivo:

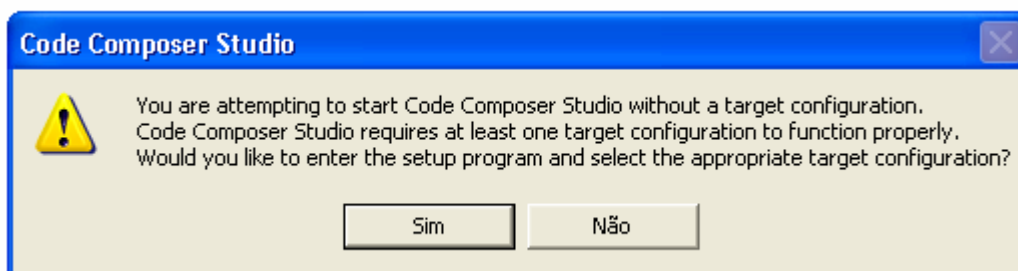


Figura 2.3.1.1 – Mensagem solicitando a configuração do CCS.

Clique em “Sim”. O programa de configuração “*Code Composer Studio Setup*” será aberto. Caso seja necessário rodar o programa de configuração novamente, basta clicar no ícone que a instalação disponibiliza na área de trabalho do Windows:



Figura 2.3.1.2 – Reiniciando a configuração do CCS.

No programa de configuração, para *Family* selecione “C67XX” e para *Platform* selecione “dsk”.

Available Factory Boards	Family	Plat...
	C67xx	dsk
C6711 DSK Port 278 EPP Mode	C67xx	dsk
C6711 DSK Port 278 SPP Mode	C67xx	dsk
C6711 DSK Port 378 EPP Mode	C67xx	dsk
C6711 DSK Port 378 SPP Mode	C67xx	dsk
C6711 DSK Port 3BC EPP Mode	C67xx	dsk
C6711 DSK Port 3BC SPP Mode	C67xx	dsk

Figura 2.3.1.3 – Configurando a família e a plataforma.

Em seguida, escolha a placa de acordo com a configuração da porta paralela na BIOS do computador. Uma configuração comum é “C6711 DSK Port 378 EPP Mode”. Clique no botão “Add” e depois em “Save & Quit”. O “Code Composer Studio Setup” irá sugerir a execução do CCS. Clique em “Sim”.





Figura 2.3.1.4 – Sugestão de execução do CCS.

Antes de se estabelecer à conexão com a placa DSK sugere-se a realização de um *reset* no emulador. Clique em *Debug* ➔ *Reset Emulator* ou pressione as teclas CTRL+Shift+R.

Para conectar-se à placa DSK, dentro do CCS clique em *Debug* ➔ *Connect* ou pressione ALT+C. Se a conexão for bem-sucedida a seguinte mensagem aparecerá no canto inferior esquerdo da tela principal do *Code Composer Studio*:

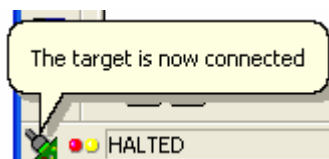


Figura 2.3.1.5 – Mensagem de conexão.

Agora você está pronto(a) para executar o teste rápido.

Clique em *GEL* ➔ *Check DSK* ➔ *Quik Test*. O teste rápido pode ser utilizado para confirmar a instalação e a operação correta da placa DSK. A seguinte mensagem é mostrada:

Switches: 7 Revision: 2  
 \*\*\*\*\*Target is Okay \*\*\*\*\*

Está assumido que os três primeiros *dip switches*, USER\_SW1, USER\_SW2 e USER\_SW3, estão todos levantados (ON). Mude os *switches* para (110x)<sub>2</sub>, mantendo os dois primeiros *switches* levantados e abaixando o terceiro. O quarto *switch* não é utilizado.

Repita o procedimento: clique em *GEL* ➔ *Check DSK* ➔ *Quik Test*. O teste rápido verifica o novo valor dos *dip switches* e mostra “Switches: 3” na tela. Os *dip switches*

podem ser ajustados para qualquer valor inteiro entre 0 e 7. Dentro do seu programa você pode direcionar a execução do código baseando-se nesses oito valores.

### 2.3.2 Exemplo 2: Teste de confiança

Um programa de teste de confiança, incluído no DSK, verifica a operação apropriada da maioria de seus componentes, como interrupções, LEDs, temporizadores etc.

O programa original fornecido pela Texas Instruments teve que ser modificado com o objetivo de se contornar um problema apresentado pelo compilador do CCS. Logo todas as vezes que o código de origem apresentava a seguinte linha,

```
while (flag == 0);
```

Foi modificado para:

```
while (flag == 0) printf("");
```

Assim, para que o teste de confiança possa ser executado sugere-se que, ao invés de se utilizar os arquivos instalados junto com o CCS, deva-se dar preferência aos arquivos contidos no diretório `\Tutorial1\Exemplo2\cnfdsp_nohost\` do CD que acompanha o presente trabalho, copiando-os para um diretório local, como por exemplo:

```
C:\CCStudio_v3.1\MyProjects\Tutorial1\Exemplo2\cnfdsp_nohost.
```

Abra o arquivo de projeto *cnfdsp\_nohost.pjt* a partir do menu *Project* ➔ *Open*. Em seguida compile-o pressione a tecla F7. A seguinte mensagem aparecerá na janela *Build* do CCS:

```
Build Complete,  
0 Errors, 0 Warnings, 0 Remarks.
```

É aconselhável realizar um *reset* na CPU antes de carregar o executável. Tal procedimento é executado pressionando-se as teclas CTRL+R.

Para carregar o programa executável, pressione CTR+L, selecione o diretório *Debug*, clique no arquivo *cnfdsp\_nohost.out* e no botão Abrir. O programa com o teste de confiança será transferido para o DSP. Execute-o pressionando F5.

Se o teste for bem sucedido, a seguinte saída será apresentada na janela *Stdout* do CCS:

```
*TMS320C6211/6711 DSK Confidence Test - NoHost*

USER_SW3=0 USER_SW2=0 USER_SW1=0 Board Rev=2

ISRAM TEST
Write and Read 5's .....
Success.....
Write and Read A's .....
Success.....
SDRAM TEST
Write and Read 5A's (even) A5's (odd).....
Success.....
Write and Read A5's (even) 5A's (odd).....
Success.....
MCBSP TEST
Digital Loopback Mode.....
Success.....
TIMER TEST
Success.....
QDMA TEST
Success.....
LED TEST
CODEC TEST
Play 1KHz tone.....
Success.....
Play CD/MIC input.....
Success.....

That's All Folks!
```

Caso ocorra algum erro, desconecte a alimentação do DSK, feche o CCS e comece novamente: reconecte a alimentação, abra o CCS, pressione ALT+C para estabelecer a conexão com a placa, pressione CTRL+R para reiniciar a CPU, abra o projeto, carregue novamente o executável e execute o programa.

É interessante observar que durante a etapa LED TEST, os *leds* USER\_LED1, USER\_LED2 e USER\_LED3 irão piscar. Durante o CODEC TEST é possível escutar um

tom de 1kHz caso você tenha um fone de ouvido conectado à saída J6 (OUT) da placa DSK.

Verifica-se que a comunicação da placa DSK com o computador é bastante instável. Com frequência a conexão é perdida, fazendo-se necessário reiniciar o processo de execução do programa.

### 2.3.3 Exemplo 3: Geração de onda senoidal

Esse exemplo gera uma senoide utilizando uma *lookup-table*. Além disso, ilustra com maiores detalhes alguns recursos do CCS, tais como a edição de um projeto, o acesso a ferramentas de geração de código e a execução de um programa no processador C6711. O código *seno8\_1.c* implementa a geração da onda senoidal.

```
//seno8_1.c

//Geração de senoide utilizando 8 pontos, f=Fs/Npt

// f -> Frequencia
// Fs -> Frequencia de amostragem
// Npt -> Numero de pontos

short loop = 0;

short sin_table[8] = {0,707,1000,707,0,-707,-1000,-707}; // valores do seno

short amplitude = 10; // fator de ganho

interrupt void c_int11() //rotina do serviço de interrupção (ISR - interrupt
                        service routine)
{
    output_sample(sin_table[loop]*amplitude); // Disponibiliza os valores da
                                                senoide

    if (loop < 7) ++loop; // Incrementa o indice do laço

    else loop = 0; // reinicia o indice para o fim do buffer

    return; // retorna da interrupção
}

void main() // Programa principal
{
```

```

comm_intr();          // Inicializa o DSK, o CODEC e as McBSPs

while(1);             // Laço infinito

}

```

Apesar do objetivo principal desse exemplo ser ilustrar o uso de algumas ferramentas, pode ser útil entender o programa *seno8\_1.c*.

Uma tabela (ou *buffer*) *sin\_table* é criada e preenchida com oito pontos representando  $f(t) = 1000\sin(t)$ , onde  $t = 0, 45, 90, 135, 180, 225, 270$  e  $315$  graus. A Figura 2.3.3.1 mostra o gráfico da função  $f(t)$ .

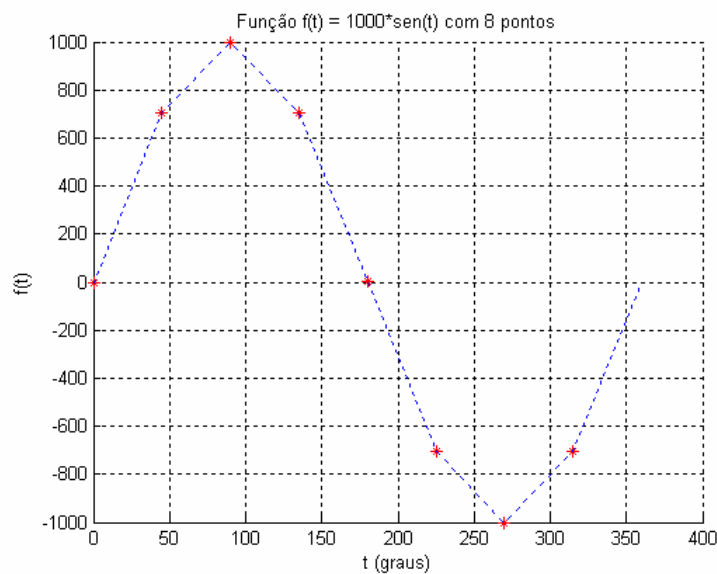


Figura 2.3.3.1 – Gráfico da função  $f(t) = 1000\sin(t)$ .

Dentro da função *main*, uma outra função, *comm\_intr*, é chamada. Tal função está definida em um arquivo de suporte *c6xdskinit.c*. O papel dela é inicializar o DSK, o codec AD535 e as duas McBSPs (*multichannel buffered serial ports*).

O comando *while(1)* dentro da função *main* determina um laço infinito que aguarda a interrupção acontecer. Quando a interrupção ocorrer, a execução prossegue para a rotina do serviço de interrupção (ISR - *interrupt service routine*) *c\_int11*.

Uma vez dentro da ISR a função *output\_sample*, localizada no arquivo de suporte *c6xdskinit.c*, é chamada a disponibilizar o primeiro valor da tabela *sin\_table[0] = 0*. O índice do laço é incrementado até se atingir o final da tabela, após o qual é reiniciado. A

execução, então, retorna da ISR para o comando *while(1)* (laço infinito) até que a nova interrupção ocorra. Uma interrupção ocorre a cada período de amostragem  $T_s$ :

$$T_s = 1 / F_s$$
$$T_s = 1 / 8000 = 0.125ms.$$

Ou seja, a cada período de amostragem de 0.125ms, uma interrupção ocorre, a ISR é acessada e o dado seguinte na tabela *sin\_table* (multiplicado por *amplitude = 10*) é enviado para a saída. Em um período da função *f(t)*, oito valores espaçados de 0.125 ms no tempo são disponibilizados na saída para a geração de um sinal senoidal.

### Criação do projeto

Para criar este projeto no *Code Composer Studio™ IDE*, é preciso adicionar os arquivos necessários à construção do projeto *seno8\_1*:

1. Crie o arquivo denominado *seno8\_1.pjt* clicando em *Project* → *File*. Ao abrir a janela *Project Creation* escreva o nome do projeto em *Project Name*. É importante observar em *Location* que o projeto em questão estará localizado em *C:\CCStudio\_v3.1\MyProjects\Tutorial1\seno8\_1*. Clique no botão *Concluir*.

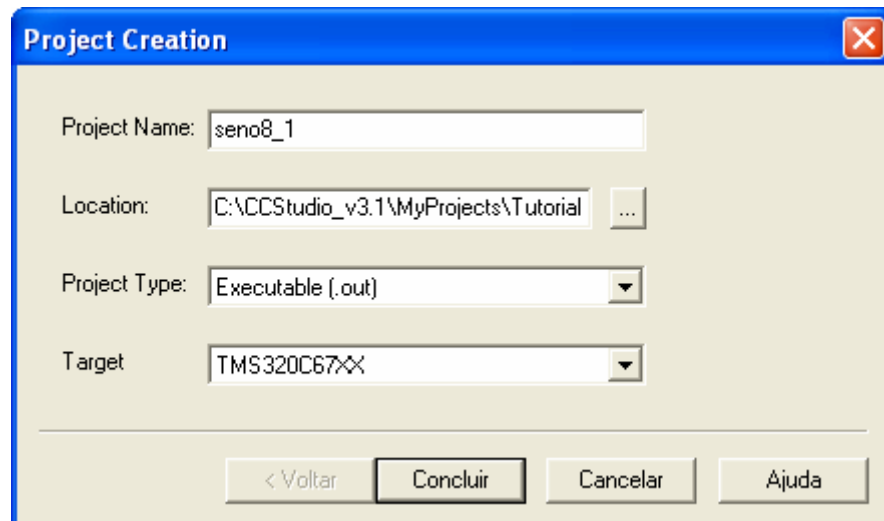


Figura 2.3.3.2 – Criação do projeto.

2. Selecione *Project* ➔ *Add Files to Project* no CCS. Abra a pasta *seno8\_1* do CD que acompanha o presente trabalho, e adicione ao projeto os dois arquivos “.c”, *sine8\_1.c* e *c6xdskinit.c*.
3. Novamente selecione *Project* ➔ *Add Files to Project*. Abra a pasta *sine8\_1* e adicione o arquivo do tipo assembly, *vectors\_11.asm*.
4. Repita o passo 3 e adicione o arquivo *c6xdsk.cmd*.
5. Repita o passo 3 e adicione na pasta *Libraries* do projeto o arquivo *rts6700.lib*, que suporta a arquitetura C67XX. Tal arquivo pode ser encontrado em *c:\CCStudio\_v3.1\c6000\cgtools\lib*.
6. Carregue na pasta *Includes* do projeto os arquivos de cabeçalho. Ou seja, clique em *Project* ➔ *Scan All File Dependencies* e observe a adição dos seguintes arquivos; *c6xdsk.h*, *c6xdskinit.h*, *c6xinterrupts.h*, e *c6x.h*.

O arquivo *Gel* que inicializa o *DSK*, *dsk6221\_6711.gel*, é adicionado automaticamente quando o projeto é criado.

Após a criação do projeto e a adição dos arquivos verifique no canto esquerdo da tela principal do *Code Composer Studio* a janela *Project View*, mostrada na Figura 2.3.3.3.

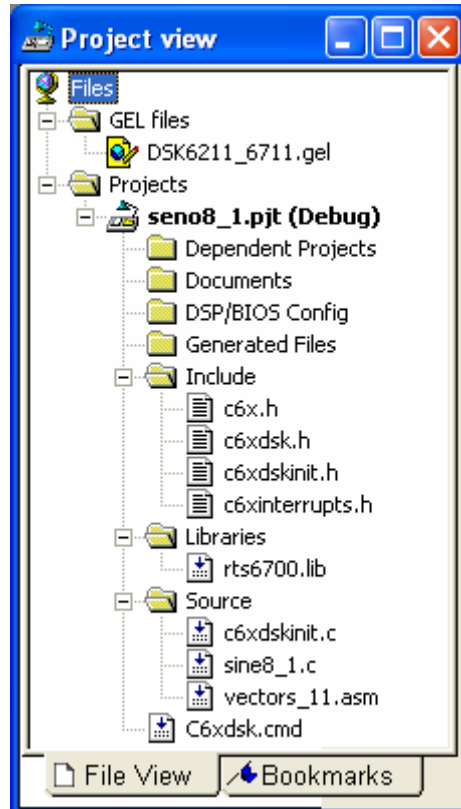


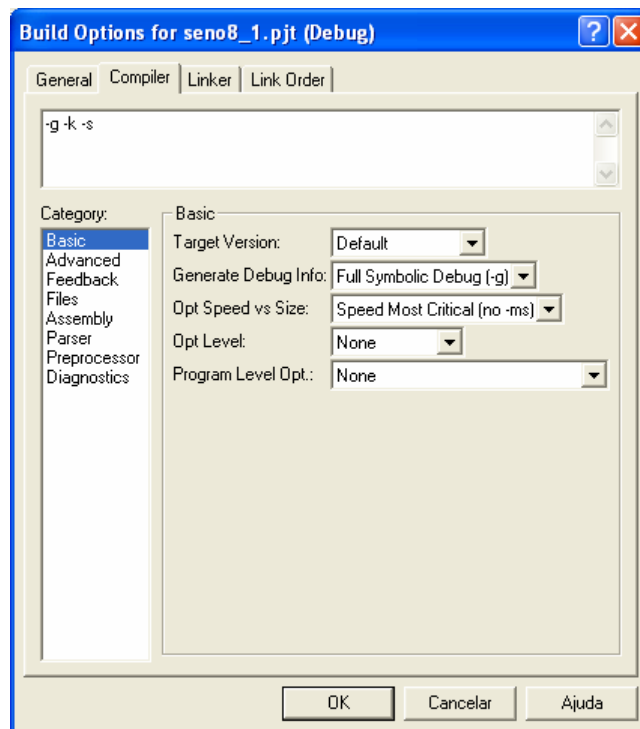
Figura 2.3.3.3 – Visualização do projeto.

## Configuração do compilador e do *linker*

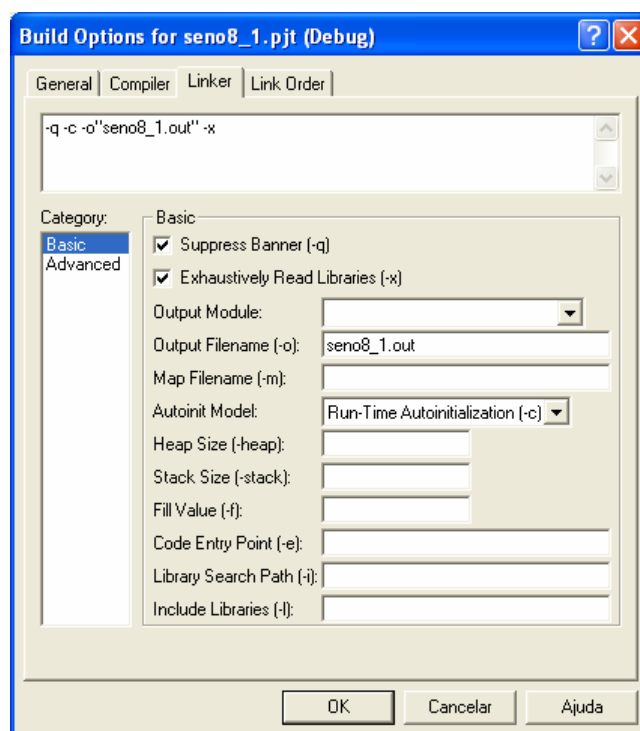
Em seguida é necessário configurar os parâmetros do compilador e do *linker*:

1. Clique em *Project* → *Build Options*.
2. Na janela *Build Options* → *Compiler* (Figura 2.3.3.4(a)).
3. Na lista *Category* → *Basic*.
4. Em *Basic* escolha as seguintes opções:
  - *Target Version* → *Default*, para selecionar a implementação em ponto fixo. No DSK baseado no *C6711* pode-se optar pelo processamento em ponto fixo ou ponto flutuante.
  - *Generate Debug Info* → *Full Symbolic Debug (-g)*. Ao mesmo tempo em que o parâmetro *-g* facilita o processo de debug, diminui a otimização do sistema.
  - *Opt Speed vs Size* → *Speed Most Critical (no ms)*.
  - *Opt Level* → *None*.
  - *Program Level Opt* → *None*.Verifique na janela *Buil Options* a seleção dos parâmetros *-g-k-s*.
5. Clique em *Linker* na janela *Build Options* (Figura 2.3.3.4 (b)).
6. Na lista *Category* → *Basic*.
7. Em *Basic* ajuste as seguintes opções:
  - Habilite *Suppress Banner (-q)*.
  - Habilite *Exhaustively Reas Libraries (-x)*
  - Em *Otput Filename (-o)* coloque o nome do arquivo de saída, *ou seja*, *seno8\_1.out*.
  - Em *Autoint Model* selecione *Run-Time Autoinitialization (-c)*. Este parâmetro inicializa as variáveis de tempo.
8. Clique em *Ok* e feche a janela *Build Options*.






(a)



(b)

Figura 2.3.3.4 – Configuração do (a) compilador e do (b) linker.

## Geração do executável e carregamento do programa

1. Selecione *Project* ➔ *Build*, ou clique no ícone  para que o projeto possa ser construído.
2. Carregue o executável através do comando *File* ➔ *Load Program*, selecionando o arquivo *seno8\_1.out* (Figura 2.3.3.5).

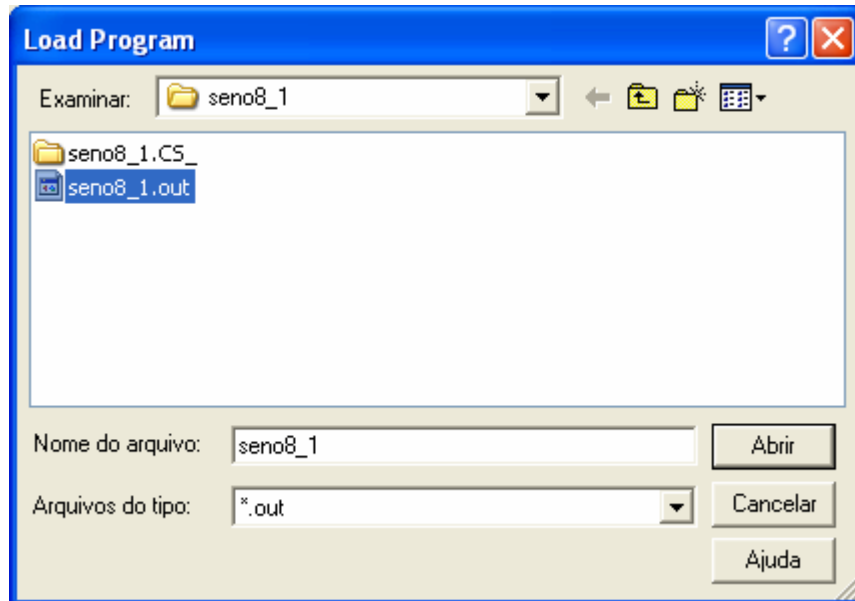




Figura 2.3.3.5 – Carregando o executável.

## Execução do programa

Por fim, execute o programa por meio de *Debug* ➔ *Run*, ou clique no ícone  que se encontra no canto esquerdo do CCS. Conectando-se um fone na saída J6 da placa DSK é possível ouvir um tom.

A frequência de amostragem  $F_s$  do codificador é fixa em 8kHz. A frequência gerada pelo sinal será  $f = F_s / (\text{número\_de\_pontos}) = 8\text{kHz}/8 = 1\text{kHz}$ . Conecte a saída J6 da placa DSK em um osciloscópio e observe a onda senoidal gerada. Sua amplitude deve ser de aproximadamente 0.85V (pico-a-pico).

Para interromper a execução do programa, clique no ícone .

## Watch Window

Certifique-se de que o processador ainda está rodando. A *Watch Window* permite o monitoramento ou a modificação do valor de uma variável:

1. Selecione a janela *View* → *Quick Watch*. Digite *amplitude* e, então, clique em *Add to Watch*. O valor 10 para a amplitude deve aparecer na janela.

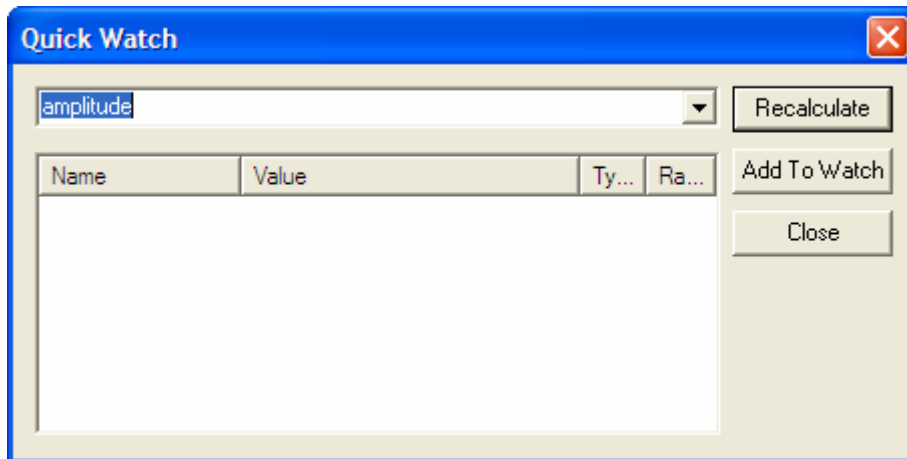


Figura 2.3.3.6 – Quick Watch.

2. Mude a amplitude para um valor (*Value*) entre 10 e 30.

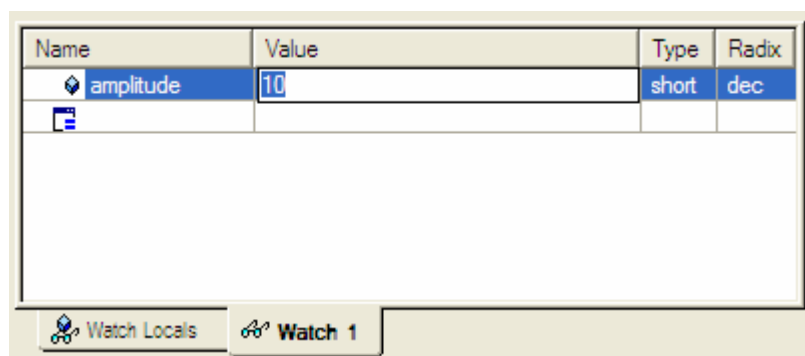


Figura 2.3.3.7 – Alterando o valor de variáveis.

3. Verifique que o volume do tom gerado varia de acordo com o valor ajustado. O valor da amplitude do sinal de saída deve ter variado entre 0.85Vp-p e 2.6Vp-p.
4. Mude a amplitude para 33 (como no item anterior). Verifique a geração de um tom em alta frequência, o que implica no fato de a mudança de frequência ter ocorrido devido a variação da amplitude do sinal. Na verdade, o que ocorreu foi

um *overflow* na capacidade de representação do *codec* de 16-bits AD535. Os valores da tabela são multiplicados por 33, fazendo com que estejam dentro do intervalo de +33.000 a – 33.000. O intervalo permitido para os valores de saída é definido pelos limites  $-2^{15}$  e  $-2^{15} - 1$ , ou seja, -32.768 e +32.767, devido ao *codec* AD535, que utiliza a representação em complemento de 2.

### Aplicando um arquivo GEL

A GEL (*General Extension Language*) é uma linguagem que permite ao projetista modificar o valor de variáveis por meio de uma barra deslizante enquanto o programa está rodando. Para isso, basta que, em primeiro lugar, as variáveis estejam definidas no programa.

No exemplo considerado, para amplificar o sinal de áudio por meio de uma barra deslizante, siga as seguintes instruções:

1. Selecione *File* ➔ *Load Gel* e abra na pasta *seno8\_1* o arquivo *amplitude.gel* (Figura 2.3.3.8).

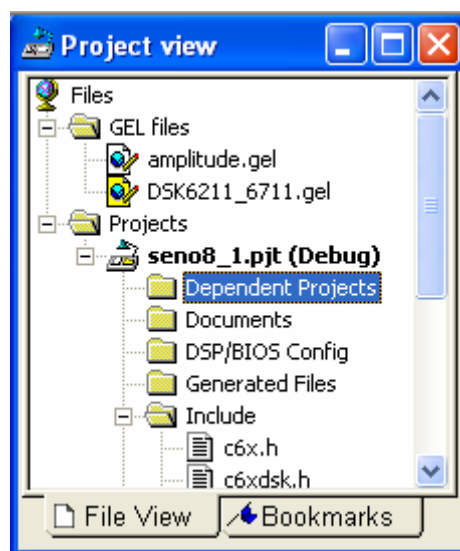


Figura 2.3.3.8 – Introdução do arquivo *amplitude.gel*.

2. Selecione em *Gel* ➔ *Sine Amplitude* ➔ *Amplitude*. Aparecerá a seguinte janela, onde você poderá controlar manualmente o valor da amplitude do sinal de 10 a 35.

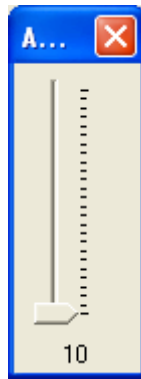


Figura 2.3.3.9 – Ajuste da amplitude do sinal.

3. Faça o debug do programa através de *Debug* → *Run*. Verifique novamente que, para amplitudes acima ou igual a 33, ocorre overflow na capacidade de representação do AD535.

#### 2.3.4 Exemplo 4: Geração e Simulação de Gráficos de uma Senoide

Esse exemplo gera uma senoide com oito pontos e simula no próprio *CCS* os gráficos no domínio do tempo e da frequência. O código *sine8\_buf.c* implementa a geração da onda senoidal.

```
//sine8_buf
//Geração do Seno.

short loop = 0;
short sine_table[8] = {0,707,1000,707,0,-707,-1000,-707}; //Valores do Seno
short out_buffer[256]; // Buffer de saída
const short BUFFERLENGTH = 256; // Tamanho do buffer de saída
short i = 0;

interrupt void c_int11() // Rotina do serviço de interrupção
{
    output_sample(sine_table[loop]); // Disponibiliza os valores da senoide
    out_buffer[i] = sine_table[loop];
    i++;
    if (i == BUFFERLENGTH) i = 0;
    if (loop < 7) ++loop; // Incrementa o indice do laço
    else loop = 0; // Reinicia o indice para o fim do buffer
    return;
}

void main()
```

```

{
    comm_intr();           // Inicializa o DSK, o CODEC e as McBSPs
    while(1);              // Laço Infinito
}

```

Verifica-se que o programa cria um *buffer* para armazenar os dados da saída na memória.

### Criação do projeto

Para criar este projeto no *Code Composer Studio*, você deve adicionar os arquivos necessários para a construção do projeto *seno8\_buffer*, assim:

1. Crie o arquivo denominado *seno8\_buffer.pjt* clicando em *Project* ➔ *File*. Ao abrir a janela *Project Creation* escreva o nome do projeto em *Project Name*. É importante observar em *Location* que o projeto em questão estará localizado em *C:\CCStudio\_v3.1\MyProjects\Tutorial1\seno8\_buffer*. Em seguida clique no botão *Concluir*.

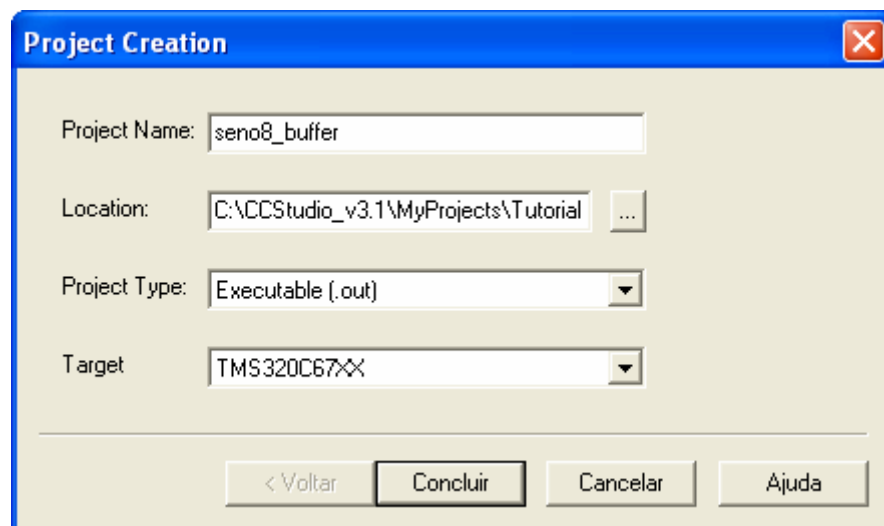


Figura 2.3.4.1 – Criação do projeto.

2. Selecione *Project* ➔ *Add Files to Project* no CCS, abra a pasta *seno8\_buf* do CD que acompanha o presente trabalho e adicione ao projeto os dois arquivos do tipo “.c”, *sine8\_buf.c* e *c6xdskinit.c*.

3. Novamente selecione *Project* → *Add Files to Project*, abra a pasta *sine8\_buf* e adicione o arquivo do tipo assembly, *vectors\_11.asm*.
4. Repita o passo 3 e adicione o arquivo de comando do *linker*, ou seja, *c6xdsk.cmd*.
5. Repita o passo 3 e adicione na pasta *Libraries* o arquivo *rts6700.lib*, o qual suporta a arquitetura C67XX, que se encontra em *c:\CCStudio\_v3.1\c6000\cgttools\lib*.
6. Carregue na pasta *Includes* do projeto os arquivos de cabeçalho. Ou seja, clique em *Project* → *Scan All File Dependencies* e observe a adição dos seguintes arquivos; *c6xdsk.h*, *c6xdskinit.h*, *c6xinterrupts.h*, e *c6x.h*.

Após a criação do projeto e a adição dos arquivos verifique no canto esquerdo da tela principal do *Code Composer Studio* a janela *Project View*, mostrada na Figura 2.3.4.2.

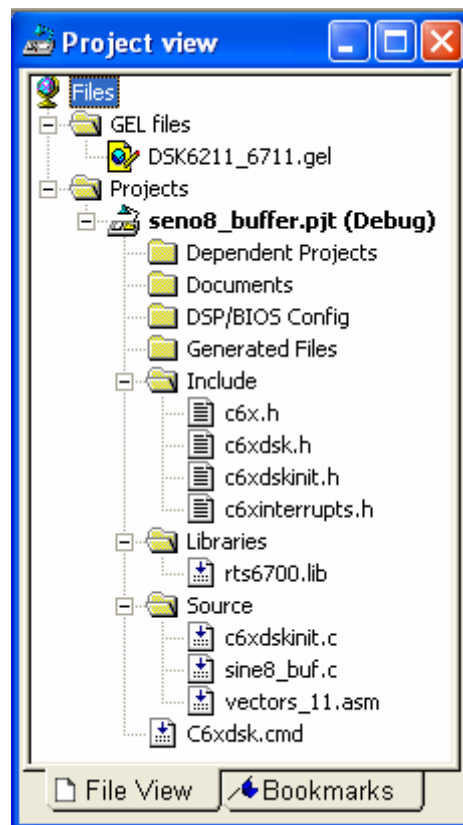


Figura 2.3.4.2 – Visualização do projeto

## Configuração do compilador e do *linker*

Defina para o *seno8\_buffer* os mesmos parâmetros de compilação do exemplo *seno8\_1*, e construa o projeto.

1. Clique em *Project* ➔ *Build Options*.
2. Na janela *Build Options* ➔ *Compiler*.
3. Na lista *Category* ➔ *Basic*.
4. Em *Basic* escolha as seguintes opções:
  - *Target Version* ➔ *Default*
  - *Generate Debug Info* ➔ *Full Symbolic Debug (-g)*.
  - *Opt Speed vs Size* ➔ *Speed Most Critical (no ms)*.
  - *Opt Level* ➔ *None*.
  - *Program Level Opt* ➔ *None*.

Verifique na janela *Buil Options* a seleção dos parâmetros *-g-k-s*.

5. Clique em *Linker* na janela *Build Options*.
6. Na lista *Category* ➔ *Basic*.
7. Em *Basic* ajuste as seguintes opções:
  - Habilite *Suppress Banner (-q)*.
  - Habilite *Exhaustively Reas Libraries (-x)*
  - Em *Otput Filename (-o)* coloque o nome do arquivo de saída, *ou seja*, *seno8\_buffer.out*.
  - Em *Autoint Model* selecione *Run-Time Autoinitialization (-c)*. Clique em *Ok* e feche a janela *Build Options*.
8. Carregue o programa através dos comandos *File* ➔ *Load Program*, abrindo o arquivo *seno8\_buf.out*.



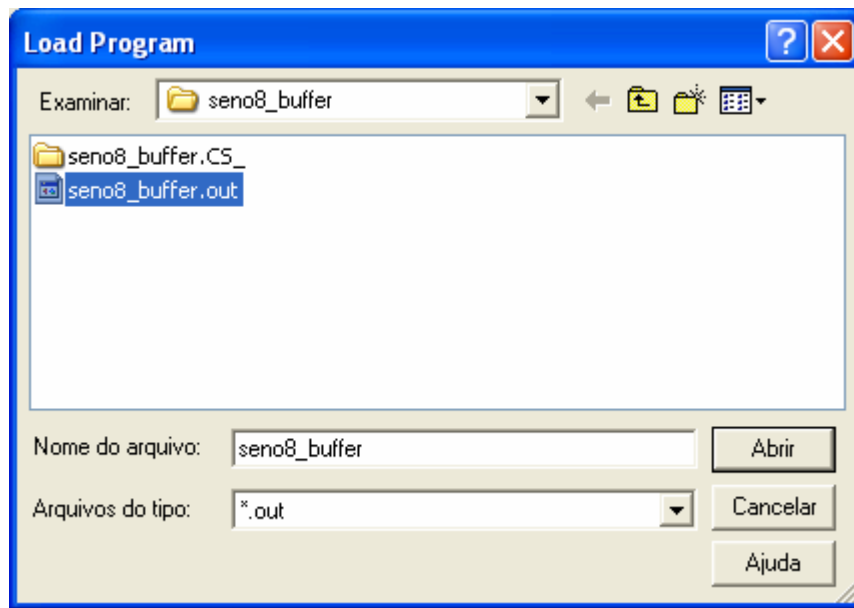


Figura 2.3.4.3 – Carregando o executável.

9. Faça o debug do programa através de *Debug* ➔ *Run*.

Conectando-se um fone no *Conector do Auto Falante J6* da placa do *DSK* você ouvirá um tom, como no exemplo passado.

### Construção gráfica

Para construir os gráficos no domínio do tempo e da frequência siga os passos a seguir:

1. Selecione *View* ➔ *Graph* ➔ *Time/Frequency*. Logo a janela *Graph Property Dialog* abrirá (Figura 2.3.4.4).

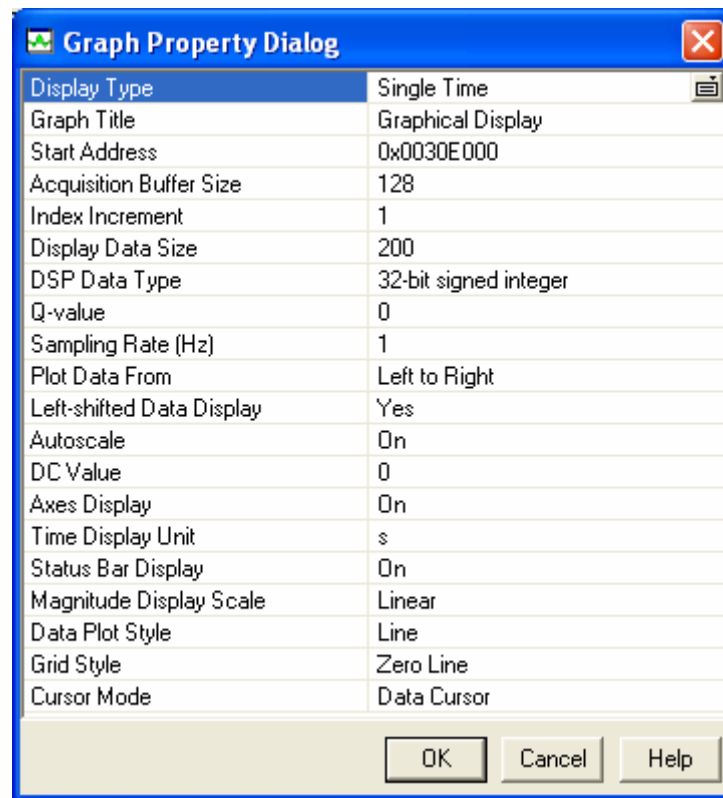


Figura 2.3.4.4 – Janela Graph Property Dialog

É necessário que você faça as seguintes modificações na janela *Graph Property Dialog*, (Figura 2.3.4.5):

- *Start Address* → *out\_buffer*.
- *Acquisition Buffer Size* → 256. Verifique no código fonte que o tamanho do *buffer* é 256.
- *DisplayData Size* → 64.
- *DSP Data Size* → 16-bit signed integer.
- *Sampling Rate (Hz)* → 8000.

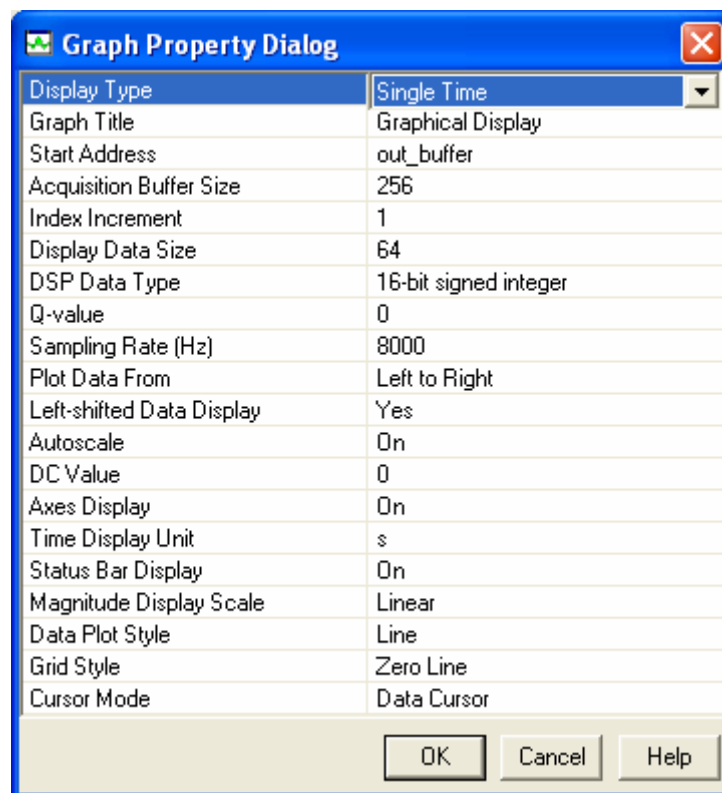


Figura 2.3.4.5 – Janela Graph Property Dialog com modificações.

A Figura 2.3.4.6 mostra a resposta no domínio do tempo.

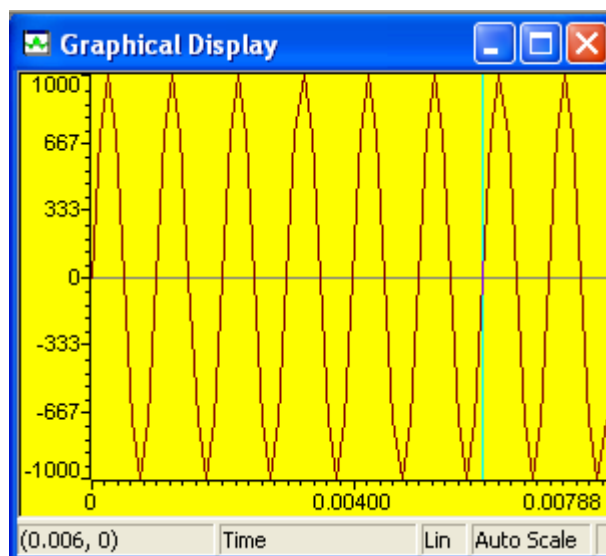


Figura 2.3.4.6 – Resposta no domínio do tempo.

Para o domínio da frequência você deve modificar algumas características na janela *Graph Property Dialog* (Figura 2.3.4.7):

- Selecione em *DisplayType* → *FFT Magnitude*.
- *FFT Framesize* → 256.

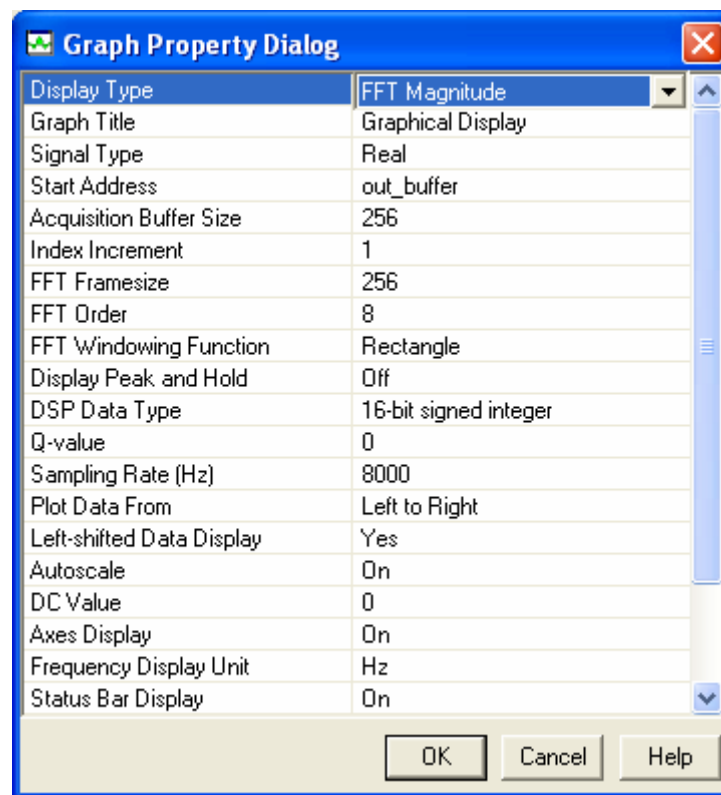


Figura 2.3.4.7 – Janela Graph Property Dialog com modificações

A Figura 2.3.4.8 mostra a resposta no domínio da frequência.

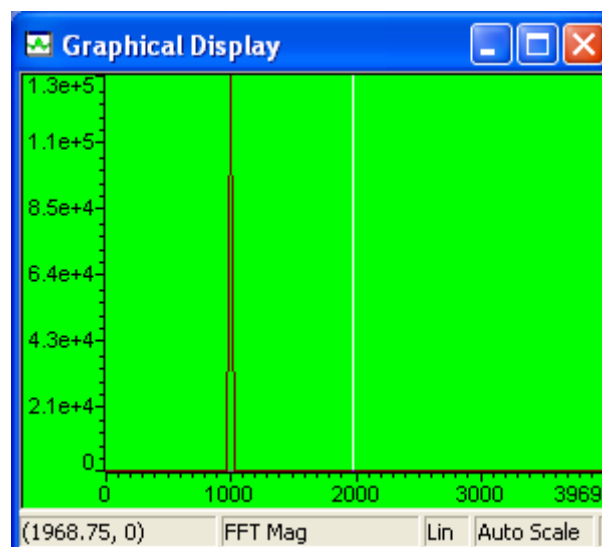


Figura 2.3.4.8 – Resposta no domínio da frequência.

No gráfico você pode verificar que o ponto de 1000Hz representa a frequência da senoide gerada.

### 2.3.5 Exemplo 5: Produto entre dois Vetores

Sabe-se que em um processador de sinais digitais as operações fundamentais são as de adição/subtração e multiplicação. A operação de multiplicação/acúmulo é muito utilizada em aplicações que requerem filtros digitais e análise espectral. Assim, tais operações são essenciais na maioria dos algoritmos de processamento de sinais digitais. Verifica-se que com o C6X pode-se executar duas operações de multiplicação/acúmulo em um único ciclo.

Neste exemplo o arquivo *dotp4.c* apresenta a soma do produto de dois vetores, onde cada vetor é composto por quatro números, como pode ser verificado no arquivo *dotp4.h*.

```
// dotp4.c
// Multiplicação de dois vetores, onde cada vetor possui 4 números

int dotp(short *a, short *b, int ncount);
#include <stdio.h>
#include "dotp4.h"
#define count 4
short x[count] = {x_array}; // Declaração do primeiro vetor
short y[count] = {y_array}; // Declaração do segundo vetor

main()
{
    int resultado = 0; // Resultado inicial igual a zero

    resultado = dotp(x,y,count); // Definindo a função dotp
    printf("resultado = %d (decimal) \n", resultado);
}

int dotp(short *a, short *b, int ncount)
{
    int soma = 0; // Inicializando a soma
    int i;

    for (i = 0; i < ncount; i++)
        soma += a[i] * b[i]; // Soma de produtos
    return(soma); // Retornando o resultado da soma
}
```

```
//dotp4.h
```

```
#define x_array 1,2,3,4
```

```
#define y_array 0,2,4,6
```

Dessa forma, a soma do produto será:

$$(1 \times 0) + (2 \times 2) + (3 \times 4) + (4 \times 6) = 40.$$

### Criação do projeto

1. Crie o arquivo denominado *dotp4.pjt* clicando em *Project ➔ New*.
2. Selecione *Project ➔ Add Files to Project* no CCS, abra a pasta *dotp4* e adicione ao projeto o arquivo *dotp4.c*.
3. Novamente selecione *Project ➔ Add Files to Project*, abra a pasta *dotp4* e adicione o arquivo *vectors.asm*.
4. Repita o passo 3 e adicione o arquivo *c6xdsk.cmd*.
5. Repita o passo 3 e adicione na pasta *Libraries* do projeto o arquivo *rts6700.lib*, que suporta a arquitetura C67XX, e se encontra em *c:\CCStudio\_v3.1\c6000\cgtools\lib*.
6. Carregue na pasta *Includes* do projeto os arquivos de cabeçalho, clicando em *Project ➔ Scan All Dependencies*, observando a adição do arquivo *dotp4.h*.

Após a criação do projeto e a adição dos arquivos, verifique no canto esquerdo da tela principal do *Code Composer Studio* a janela *Project View*, mostrada na Figura 2.3.5.1.

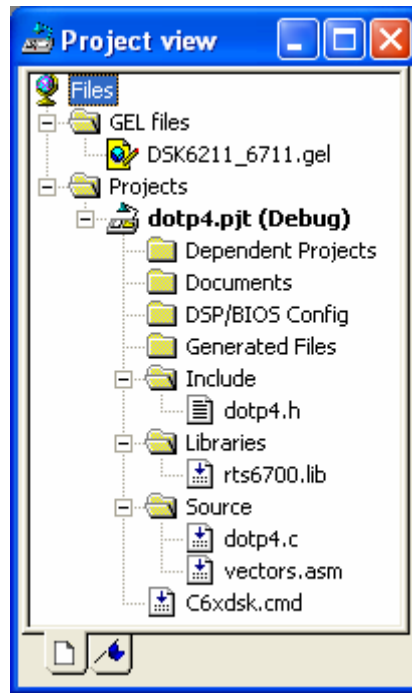


Figura 2.3.5.1 – Visualização do projeto.

Defina os parâmetros de configuração do compilador e do *linker*:

1. Clique em *Project* ➔ *Build Options*.
2. Na janela *Build Options* ➔ *Compiler*.
3. Na lista de *Category* ➔ *Basic*.
4. No checkbox escolha as seguintes opções:

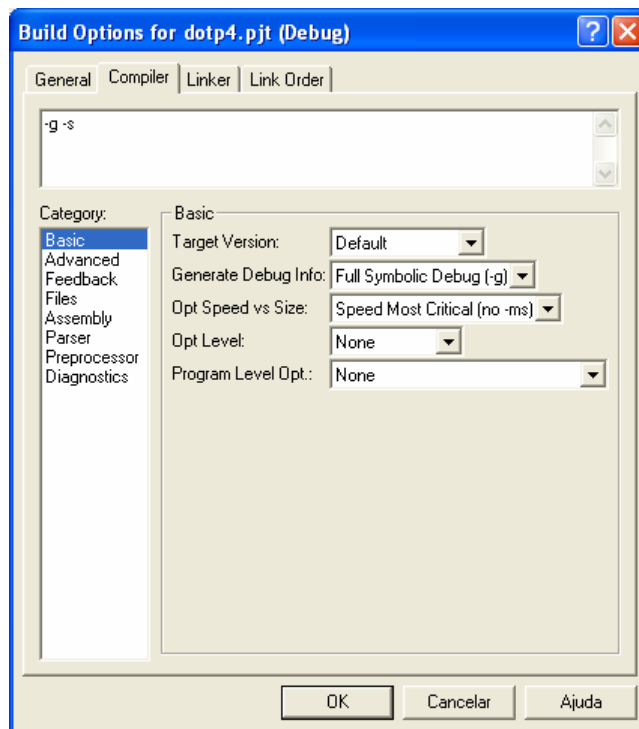


Figura 2.3.5.2 – Configurações do compilador.

5. Clique em *Linker* na janela *Build Options*.
6. Na lista de *Category* → *Basic*.
7. No checkbox escolha as opções apresentadas na Figura 2.3.5.3.

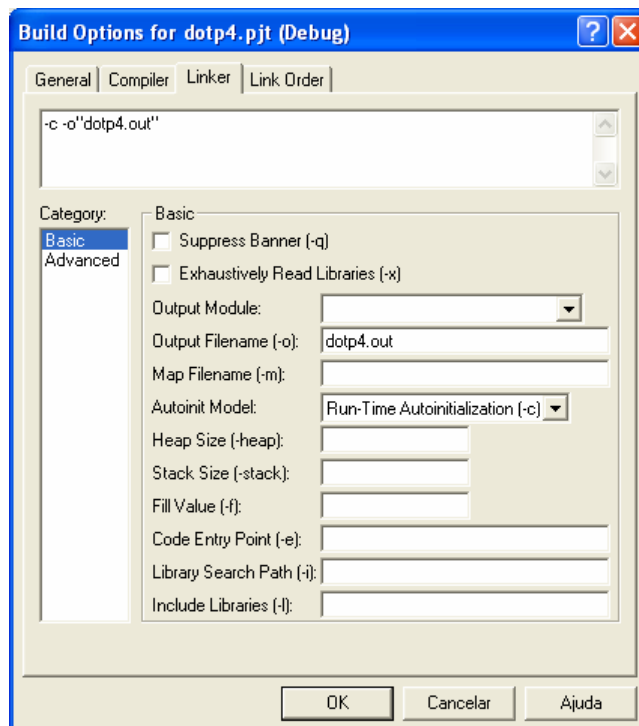



Figura 2.3.5.3 – Configurações do linker.



8. Clique em *Ok* e feche a janela *Build Options*.
9. Selecione em *Project* ➔ *Build*, ou clique no ícone  para que o projeto possa ser construído.
10. Carregue o programa através dos comandos *File* ➔ *Load Program*, abrindo o arquivo *dotp4.out*.

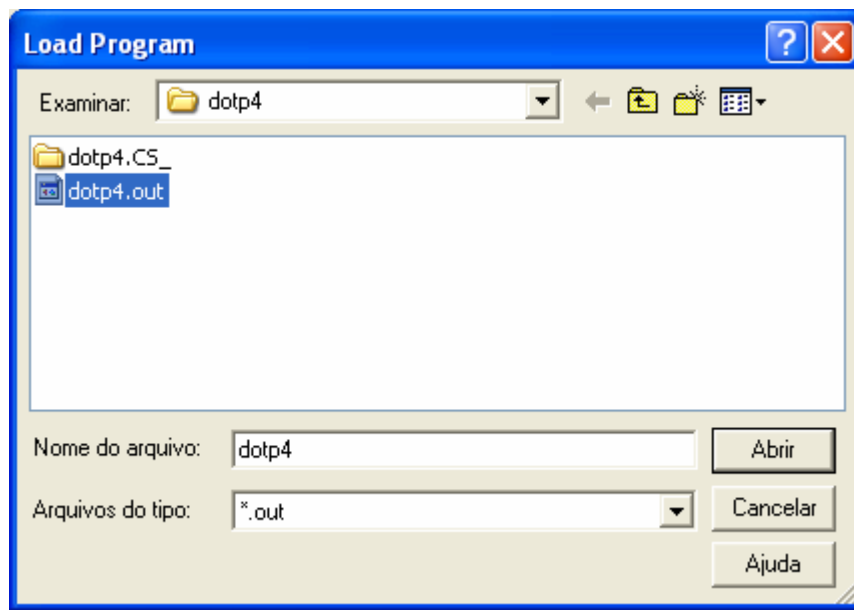


Figura 2.3.5.4 – Carregando o executável.

17. Faça o debug do programa através de *Debug* ➔ *Run*. E o quadro mostrado na Figura 2.3.5.5 aparecerá:

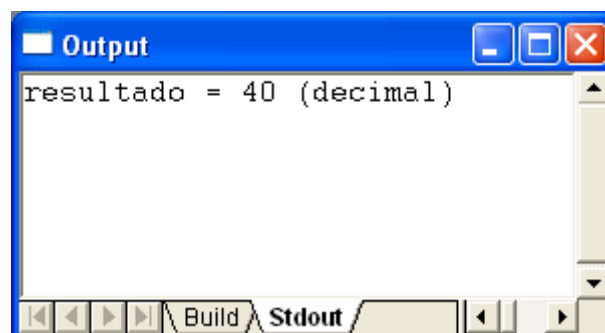
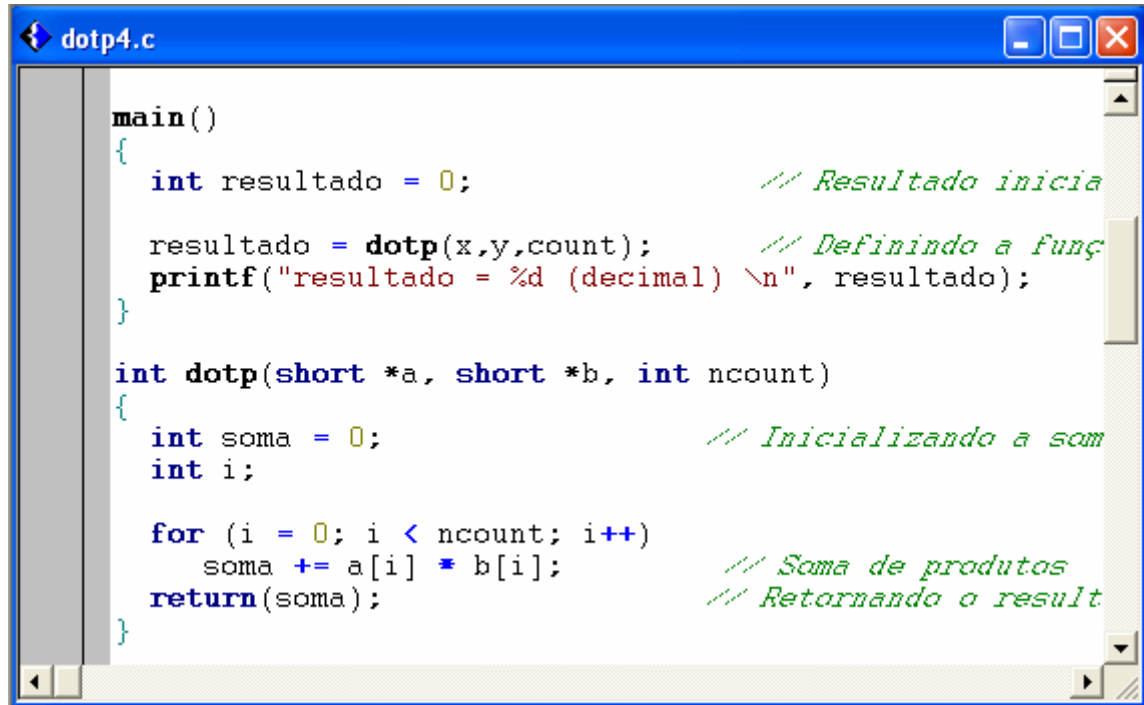


Figura 2.3.5.5 – Saída.

Sabe-se que o ajuste da otimização é um ponto bastante importante quanto ao desenvolvimento do projeto, podendo influenciar positivamente ou negativamente em outros parâmetros do sistema, e até mesmo no custo final do produto. A eficiência é

comumente medida em termos da contagem do número de ciclos e o tamanho do código. O *Code Composer Studio™ IDE* fornece uma série de ferramentas de ajuda para que o programador alcance rapidamente a otimização do projeto.



```
dotp4.c

main()
{
    int resultado = 0;           // Resultado inicia

    resultado = dotp(x,y,count); // Definindo a função
    printf("resultado = %d (decimal) \n", resultado);
}

int dotp(short *a, short *b, int ncount)
{
    int soma = 0;               // Inicializando a soma
    int i;

    for (i = 0; i < ncount; i++)
        soma += a[i] * b[i];    // Soma de produtos
    return(soma);               // Retornando o resultado
}
```

Figura 2.3.5.6 – Programa principal.

Nesta etapa do experimento serão utilizadas as janelas do *Profile Setup* para que o programador selecione a parte do código a ser analisada no *Profile Viewer*, e o *Compiler Consultant Tool* do compilador que apresentará conselhos quanto à otimização do projeto.

1. Clique em *Profile* → *Setup* para que janela *Profile Setup* seja aberta no lado direito da tela principal do *Code Composer Studio*.

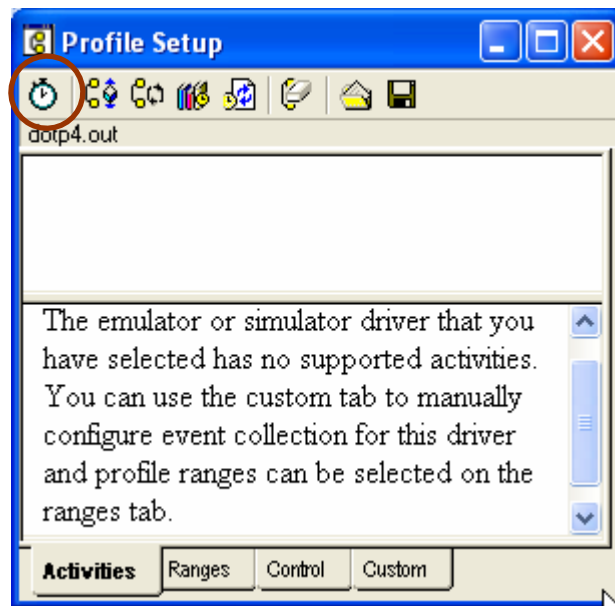


Figura 2.3.5.7 – Ajuste do Profile.

3. Clique no ícone que representa o relógio, o qual significa o comando de *Enable/Disable Profiling*, e observe que ao habilitá-lo será mostrado na parte central inferior da tela principal do *Code Composer Studio* a seguinte mensagem:

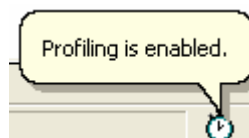


Figura 2.3.5.8 – Indicação de habilitação do Profile.

3. Clique na aba *Ranges* da janela *Profile Setup*:

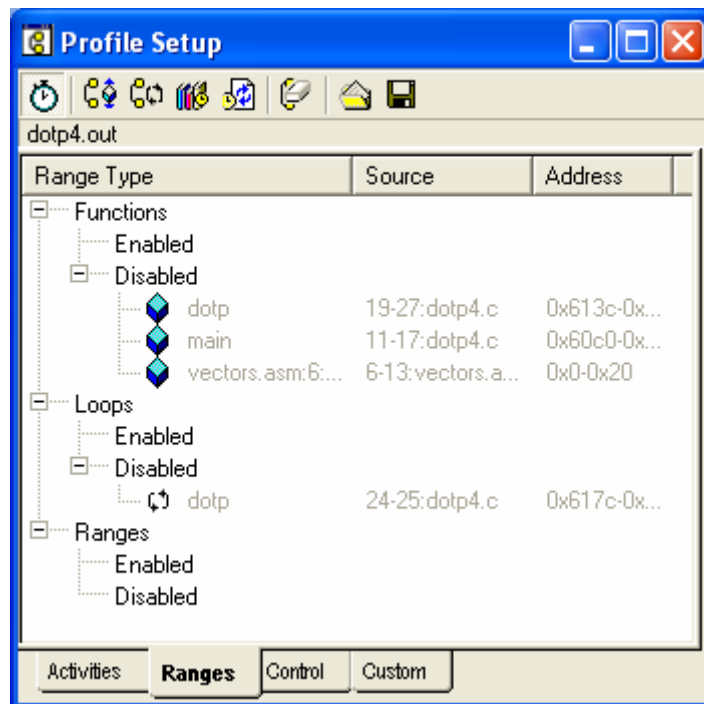




Figura 2.3.5.9 – Ajuste do Profile: Ranges.

4. Habilite as partes desejadas. Para habilitar uma única função basta arrastá-la para *Functions* ➔ *Enable*. Todas as funções podem ser habilitadas simultaneamente clicando-se no ícone . Da mesma maneira, todos os laços (*loops*) podem ser habilitados, clicando-se no ícone .

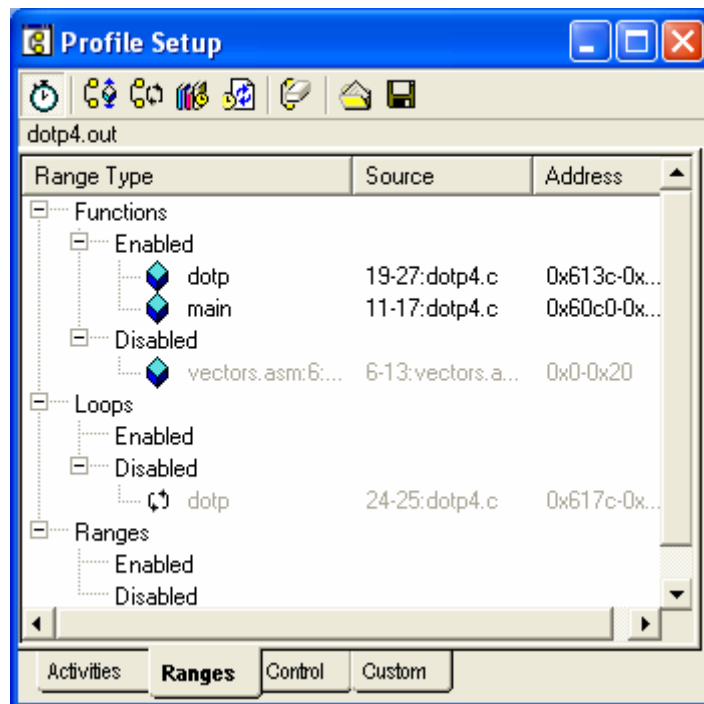


Figura 2.3.5.10 – Ajuste do Profile: Ranges.

Observa-se neste caso que foram habilitados às funções *dotp* e *main*.

5. Clique na aba *Custom* da janela *Profile Setup* e selecione *Cycles*.

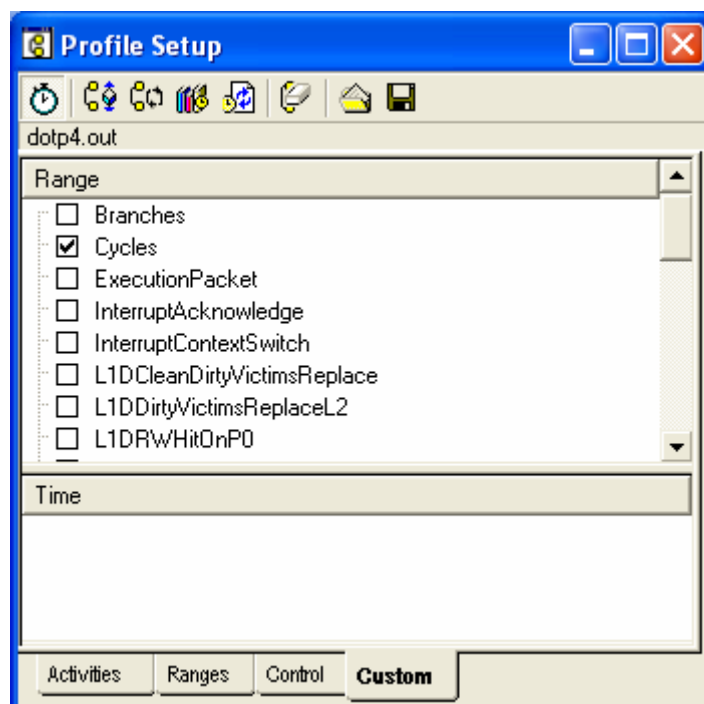


Figura 2.3.5.11 – Ajuste do Profile: Custom/ Cycles.

Após esta primeira parte é necessário que você ajuste alguns parâmetros do compilador.

6. Clique em *Project* ➔ *Build Options*.
7. Na janela *Build Option* ➔ *Compiler*.
8. Na lista de *Category* ➔ *Feedback*.
9. No checkbox clique em *Generate Compiler Consultant*.

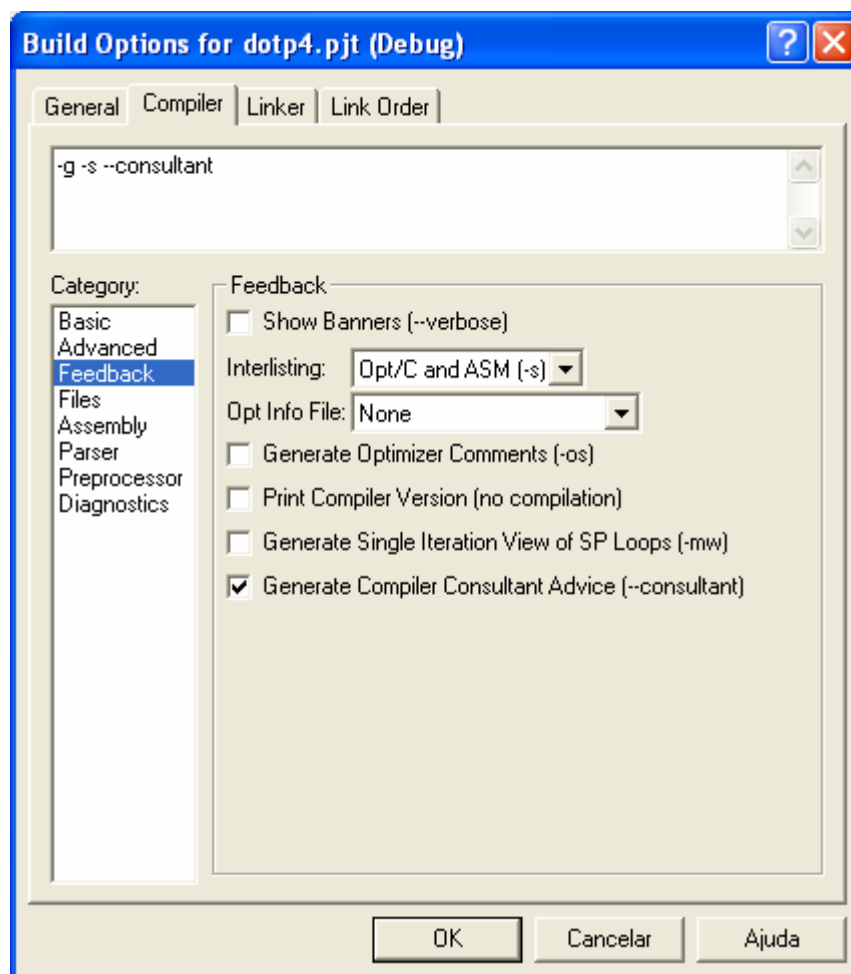


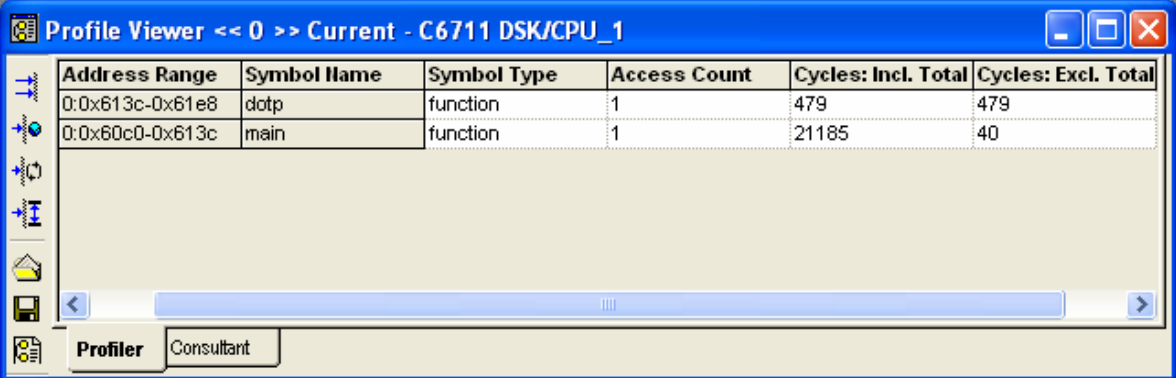
Figura 2.3.5.12 – Configurações do compilador.

10. Clique em OK e feche a janela *Build Options*.
11. No menu *Project*, selecione *Build*, para que as configurações feitas anteriormente sejam consideradas.
12. Clique em *Debug* ➔ *Reset CPU*.
13. Carregue o programa através dos comandos *File* ➔ *Reload Program*.

14. Faça o debug do programa através de *Debug* → *Run*.

Nesta terceira parte verifique as análises das funções por meio da janela *Profile Viewer*.

15. Clique em *Profile* → *Viewer*, e a janela *Profile Viewer* será executado no canto inferior do lado direito da tela principal do *Code Composer Studio* (Figura 2.3.5.13):



Address Range	Symbol Name	Symbol Type	Access Count	Cycles: Incl. Total	Cycles: Excl. Total
0:0x613c-0x61e8	dotp	function	1	479	479
0:0x60c0-0x613c	main	function	1	21185	40

Figura 2.3.5.13 – Visualizador do Profile

Observe que o *Address Range* apresenta o endereço hexadecimal da função selecionada. Em SLR (*Source Line Reference*) verifique as linhas do código que definem o laço, a função ou o intervalo (*range*). *Access Count* significa o número de vezes que a função foi acessada. *Cycles: Incl. Total* indica a quantidade de ciclos total para a execução da função. *Cycles: Excl. Total* indica a quantidade de ciclos para que ocorra a execução da função, excluindo os ciclos das sub-rotinas contidas na mesma [TMS320C6000, 2000].

16. Clique na aba *Consultant*. A linha observada nesta janela representa o único laço presente no código do exemplo proposto.

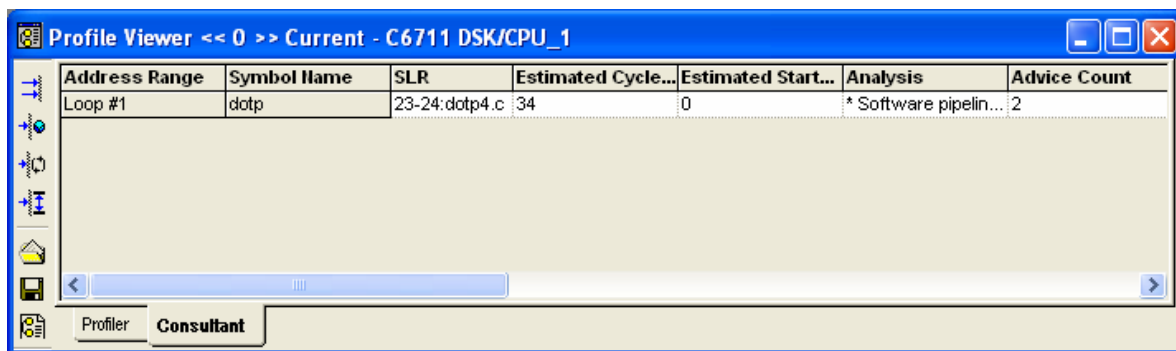


Figura 2.3.5.14 – Visualizador do Profile.

Em *Advice Count* observa-se a quantidade de conselhos oferecidos pelo *Code Composer Studio™ IDE* para a otimização do projeto, os quais podem ser verificados dando um duplo clique na linha correspondente as instruções.

Após o duplo clique, ou através dos comandos *Profile* → *Tuning* → *Advice*, verifica-se a abertura de uma janela de conselhos e análises do *Compiler Consultant* (Figura 2.3.5.15) no lado esquerdo da tela principal do *Code Composer Studio*.

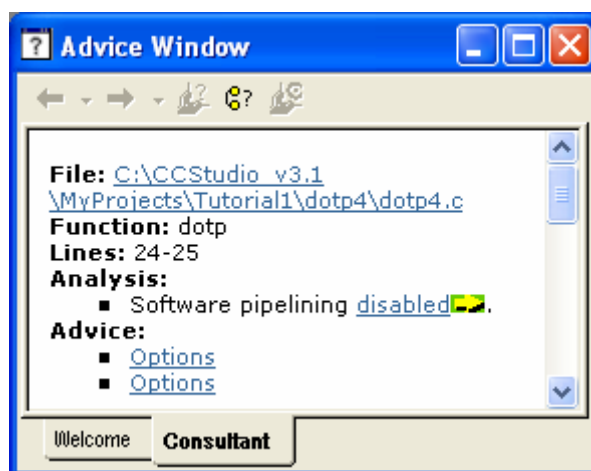


Figura 2.3.5.15 – Janela de conselhos.

Ao aparecer à janela *Compiler Consultant* verifique que no quadro *Consultant* aparece conselhos para a função *dotp4*. Em *Analysis* verifique que a otimização do *Software pipelining* está desabilitada. Em *Advice* aparecem duas opções de conselho. Para visualizá-los basta clicar em *Options*.

O primeiro conselho fornecido em *Options* indica que você está compilando o programa sem otimização.



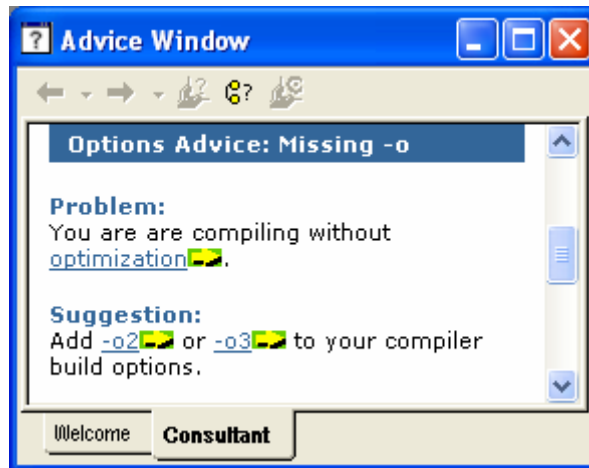



Figura 2.3.5.16 – Janela de conselhos.

Verifique que a palavra *optimization* encontra-se sublinhada, com uma cor diferente do texto inicial, e seguida pelo ícone , o que significa que bastando clicar na mesma aparece uma janela de ajuda explicando os níveis de otimização.

Em *Suggestion*, na janela *Consultant*, verifique que a sugestão oferecida foi que o projetista deveria aplicar `-o2` ou `-o3` na opção do compilador.

Entretanto, antes de qualquer mudança observe na mesma janela o segundo conselho oferecido em *Consultant* → *Advice* → *Options*.

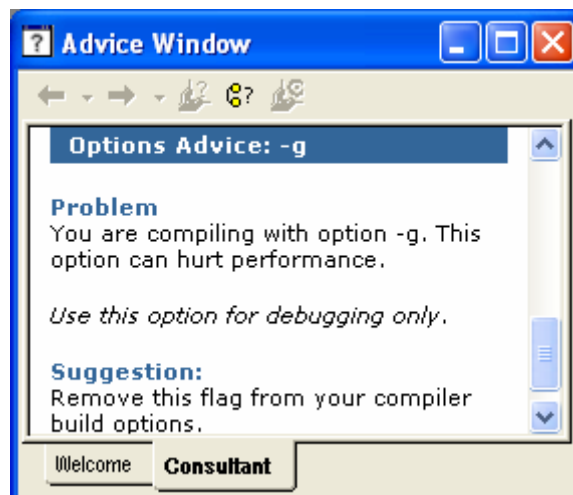


Figura 2.3.5.17 – Janela de conselhos.

Verifique em *Problem* que utilizar a opção `-g` na determinação dos parâmetros do *Debug*, pode interferir no desempenho do projeto. Sugere-se desta forma que o programador retire esta opção do seu *Build Options*.

Seguindo os conselhos sugeridos para a otimização, é preciso configurar os novos parâmetros para o compilador.

1. Clique em *Project* ➔ *Build Options*.
2. Na janela *Build Option* ➔ *Compiler*.
3. Na lista de *Category* ➔ *Basic*.
4. Em *Generate Debug Info* ➔ *No Debug*.
5. Em *Opt Level* ➔ *File (-o3)*.

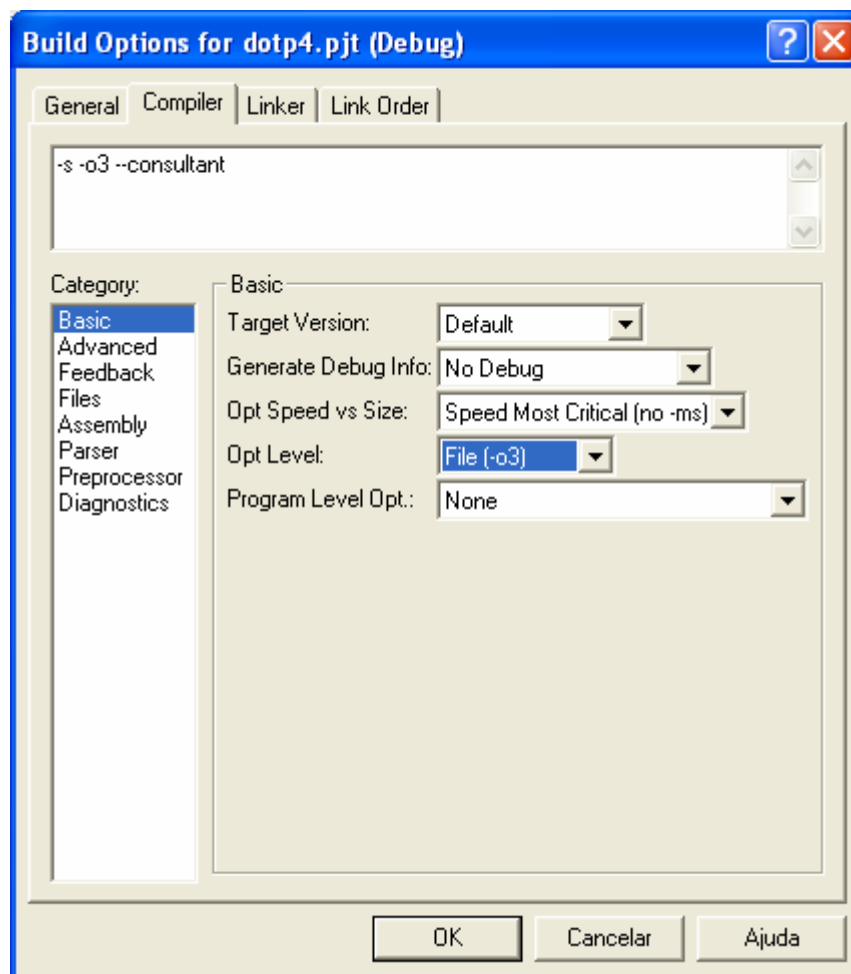
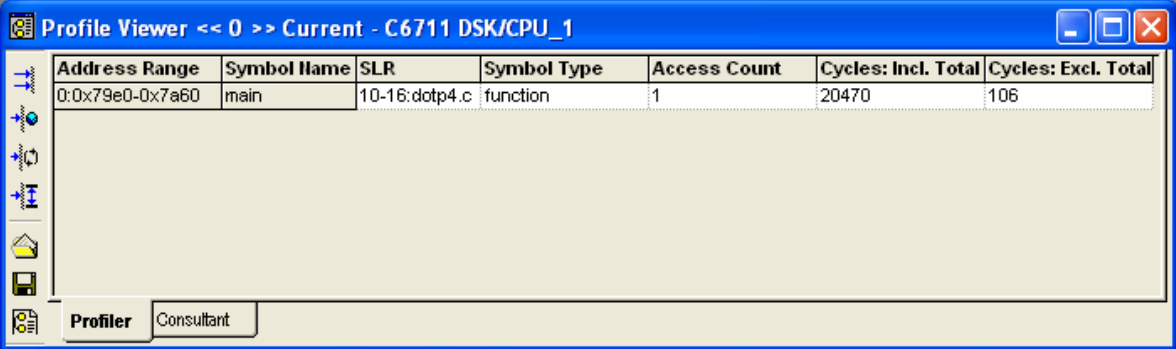


Figura 2.3.5.18 – Configurações do compilador.

6. Clique em *OK* e feche a janela *Build Options*.
7. No menu *Project* selecione *Rebuild All*.
8. Clique em *Debug* ➔ *Reset CPU*.
9. Carregue o programa *File* ➔ *Reload Program*.

10. Execute *Debug* → *Run*.



The screenshot shows a window titled "Profile Viewer << 0 >> Current - C6711 DSK/CPU\_1". It contains a table with the following data:

Address Range	Symbol Name	SLR	Symbol Type	Access Count	Cycles: Incl. Total	Cycles: Excl. Total
0:0x79e0-0x7a60	main	10-16:dotp4.c	function	1	20470	106

Below the table, there are two tabs: "Profiler" and "Consultant".

Figura 2.3.5.19 – Visualizador do Profile.

Observe na janela *Profile Viewer* que, após a determinação dos novos parâmetros para a otimização do projeto, a quantidade de *Cycles: Incl. Total* diminuiu, ou seja, a função *main* e suas sub-rotinas apresentaram um custo menor em termos de ciclos de relógio, após ter sido aplicada à ferramenta de otimização.

## 3 - TUTORIAL 2: FILTROS COM RESPOSTA IMPULSIONAL FINITA (FIR)

### 3.1 OBJETIVO

- Estudo da transformada Z
- Projeto e implementação de filtros com resposta impulsional finita (FIR)
- Programação de exemplos usando C e o TMS320C6711 DSK

Esse tutorial tem por finalidade estudar a transformada Z em conjunto com os sinais discretos. Projetar três exemplos de filtro FIR a partir do método da série de Fourier e implementá-los através da equação de convolução discreta.

### 3.2 INTRODUÇÃO

#### 3.2.1 Sinais Discretos

Um sinal no tempo discreto é aquele que pode ser representado por uma sequência de números, como,

$$x(n) = \sum_{m=-\infty}^{\infty} x(m)\delta(n-m), \quad (3.1)$$

onde  $x(n)$  seria composto pela soma de seqüências de impulsos unitários  $\delta(n)$  atrasados de  $m$  amostras, e multiplicados por uma constante  $x(m)$ . Sabe-se que:

$$\delta(n-m) = \begin{cases} 1, & n = m \\ 0, & n \neq m \end{cases} \quad (3.2)$$

Logo o somatório da Equação 3.1 seria representado por uma seqüência de valores  $x(1), x(2), \dots$ , onde cada valor de amostra da seqüência corresponde a um valor de amostra no tempo ( $n$ ) determinado pelo intervalo de amostragem ou pelo período de amostragem,  $T = 1/F_s$ .

Dessa forma o sistema discreto aceita na sua seqüência de entrada,  $x(n)$ , sinais discretos e fornece na seqüência de saída,  $y(n)$ , também sinais discretos, a partir das relações existentes entre tais seqüências.

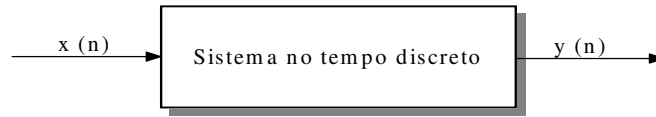


Figura 3.2.1.1 – Representação de um sistema no tempo discreto

Dependendo das relações, o sistema no tempo discreto pode ser classificado de várias formas, linear ou não linear, variante ou invariante no tempo (ao deslocamento) e causal ou não causal. Neste trabalho, porém, serão apenas considerados sistemas discretos lineares e invariantes ao deslocamento (LI).

Um sistema discreto é denominado linear se ao ser aplicada, uma entrada que seja uma combinação linear de duas outras, na saída será encontrada a mesma combinação linear relacionada às saídas correspondentes a essas entradas quando estas foram aplicadas separadamente [Oppenheim & Schaffer, 1989].

Assim se a entrada do sinal  $x(n)$  corresponde à saída  $y(n)$ :

$$x(n) \rightarrow y(n) \quad (3.3)$$

E se  $a_1x_1(n) \rightarrow a_1y_1(n)$  e  $a_2x_2(n) \rightarrow a_2y_2(n)$ , então pela combinação linear:

$$a_1x_1(n) + a_2x_2(n) \rightarrow a_1y_1(n) + a_2y_2(n). \quad (3.4)$$

A equação (3.4) representa o princípio da superposição, onde a seqüência de saída é a soma das respostas de cada entrada.

Se um sistema discreto é dito invariante ao deslocamento isto implica que se a entrada está atrasada de  $m$  amostras a saída também estará, ou seja,

$$x(n) \rightarrow y(n) \therefore x(n-m) \rightarrow y(n-m) \quad (3.5)$$

Caso a resposta de saída ao impulso,  $\delta(n)$ , seja  $h(n)$ , então  $\delta(n) \rightarrow h(n)$ , e este sistema discreto seja considerado invariante no tempo então a seguinte relação torna-se verdadeira;

$$\delta(n-m) \rightarrow h(n-m) \quad (3.6)$$

Multiplicando o impulso por  $x(m)$  e aplicando a linearidade obtém-se:

$$x(m)\delta(n-m) \rightarrow x(m)h(n-m) \quad (3.7)$$

Conseqüentemente a seqüência de saída será:

$$y(n) = \sum_{m=-\infty}^{\infty} x(m)h(n-m) \quad (3.8)$$

Esta operação entre os sinais discretos designa-se por convolução discreta. Fazendo  $k = n - m$ , tem-se:

$$y(n) = \sum_{k=0}^{\infty} h(k)x(n-k) \quad (3.9)$$

### 3.2.2 Introdução a Transformada Z

A transformada Z é um instrumento matemático utilizado para a análise e síntese de sinais discretos no tempo, desempenhando um papel similar à transformada de Laplace para sinais contínuos no tempo [Gold & Radere, 1969].

Considerando um sinal analógico idealmente amostrado  $x(t)$ ,

$$x_s(t) = \sum_{k=0}^{\infty} x(kT)\delta(t-kT) \quad (3.10)$$

A função impulso na equação está atrasada de  $kT$  onde  $T = 1/F_s$ . Verifica-se que a função  $x_s(t)$  será zero para qualquer valor exceto para  $t = kT$ .

A transformada de Laplace de  $x_s(t)$  é:

$$\begin{aligned} X_s(s) &= \int_0^{\infty} x_s(t)e^{-st} dt \\ &= \int_0^{\infty} \{x(t)\delta(t) + x(t)\delta(t-T) + \dots\} e^{-st} dt \end{aligned} \quad (3.11)$$

Pela propriedade da função impulso tem-se:

$$\int_0^{\infty} f(t)\delta(t-kT)dt = f(kT) \quad (3.12)$$

Logo:

$$X_s(s) = x(0) + x(T)e^{-sT} + x(2T)e^{-2sT} + \dots = \sum_{n=0}^{\infty} x(nT)e^{-nsT} \quad (3.13)$$

Fazendo  $z = e^{sT}$  tem-se:

$$X(z) = \sum_{n=0}^{\infty} x(nT)z^{-n} \quad (3.14)$$

Considerando  $x(nT) = x(n)$ , obtém-se:

$$X(z) = \sum_{n=0}^{\infty} x(n)z^{-n} = ZT\{x(n)\} \quad (3.15)$$

O qual representa a transformada Z,  $(ZT)$ , de  $x(n)$ .

Verifica-se que a transformada em Z,  $X(z)$ , de um sinal discreto  $x(n)$  é uma função complexa da variável complexa  $z \in \mathbb{C}$ . A transformada  $X(z)$  só é definida para as regiões do plano complexo em que o somatório converge.

Se, por exemplo,  $x(n) = e^{nk}$ ,  $n \geq 0$  e  $k$  uma constante real, então a transformada  $z$  será:

$$X(z) = \sum_{n=0}^{\infty} e^{nk} z^{-n} = \sum_{n=0}^{\infty} (e^k z^{-1})^n \quad (3.16)$$

Usando séries geométricas,

$$\sum_{n=0}^{\infty} u^n = \frac{1}{1-u} \quad |u| < 1 \quad (3.17)$$

Onde  $n \geq 0$ . Assim,

$$X(z) = \frac{1}{1 - e^k z^{-1}} = \frac{z}{z - e^k} \quad (3.18)$$

Para  $|e^k z^{-1}| < 1$  ou  $|z| > |e^k|$ . Se  $k = 0$ , então  $X(z) = z/(z-1)$ .

### 3.2.3 Transformada Z inversa

A transformada Z inversa é representada pela seguinte equação [Bellanger, 1989];

$$x[n] = \frac{1}{2\pi j} \oint_c X(z) z^{n-1} dz \quad (3.19)$$

Onde a integral é realizada sobre um contorno  $c$  fechado, anti-horário e ao redor da origem do plano  $z$ .

Para se obter este resultado, tem-se a partir da teoria de funções complexas [Rabiner & Gold, 1975],

$$\oint_c z^{-k} dz = \begin{cases} 0; & k \neq 1 \\ 2\pi j; & k=1 \end{cases} \quad (3.20)$$

Onde  $k$  é um inteiro e  $c$  é um contorno fechado, anti-horário e ao redor da origem do plano  $z$ . Logo ao considerar  $c = r \exp(j\theta)$ , onde  $r = \text{constante}$  e  $0 \leq \theta \leq 2\pi$  na equação (3.20), obtém-se:

$$\begin{aligned} \oint_c z^{-k} dz &= \int_0^{2\pi} r^{-k} e^{-jk\theta} j r e^{j\theta} d\theta \\ &= r^{-k+1} j \int_0^{2\pi} e^{-j(k-1)\theta} d\theta \end{aligned} \quad (3.21)$$

Aplicando este resultado, equação (3.21), no cálculo da integral da Equação (3.19), com o contorno  $c$  contido na região de convergência de  $X(z)$ . Usando a transformada Z para substituir  $X(z)$  tem-se:

$$\begin{aligned}\oint_c X(z)z^{n-1}dz &= \sum_{k=-\infty}^{\infty} x[k] \oint_c z^{-(k-n+1)} dz \\ &= \sum_{k=-\infty}^{\infty} x[k]\end{aligned}\quad (3.22)$$

Obtendo desta forma a equação (3.18) que apesar de sua utilização não ser simples, esta equação permite obter a transformada inversa [Oppenheim, 1993].

### 3.2.4 Equações de Diferenças

A transformada de Laplace é utilizada para resolver equações diferenciais, no domínio  $s$ , no plano  $s$ , que representam filtros analógicos. Já a transformada Z resolve equações de diferenças, no domínio  $z$ , no plano  $z$ , representando filtros digitais. O plano  $s$  é um sistema com coordenadas retangulares enquanto que o plano  $z$  utiliza o formato polar [Rabiner & Gold, 1975].

A entrada e a saída de um sistema descrito por uma equação de diferenças linear se relacionam genericamente por:

$$y(n) = bx(n) - ay(n-1) \quad (3.23)$$

Onde cada saída corrente  $y(n)$  está relacionada com o sinal de entrada corrente  $x(n)$ , e com os sinais prévios da saída  $y(n-1)$ . Assumindo o sistema causal, ou seja,  $y(n) = 0$  para  $n < 0$  e  $x(n) = \delta(n)$ , a saída  $y(n)$  pode ser escrita como:

$$\begin{aligned}y(n) &= b_0x(n) + b_1x(n-1) + \dots + b_{L-1}x(n-L+1) - a_1y(n-1) - \dots - a_My(n-M) \\ &= \sum_{k=0}^{N-1} b_kx(n-k) - \sum_{m=1}^M a_my(n-m)\end{aligned}\quad (3.24)$$

O sistema representado pela equação (3.24) é dito recursivo, uma vez que a sua saída depende de suas entradas e das amostras passadas da própria saída. Um sistema discreto é denominado não-recursivo quando a resposta  $y(n)$  do sistema pode ser calculada a partir exclusivamente de sua entrada  $x(n)$ , [Proakis & Manolakis, 1996], assim:

$$y(n) = \sum_{k=0}^{N-1} b_kx(n-k) \quad (3.25)$$



Para resolver a equação de diferença faz-se necessário, como pode ser verificado a partir da equação (3.25), achar a transformada Z para a expressão do tipo  $x(n-k)$ , a qual corresponde a k-ésima derivada  $d^k x(t)/dt^k$  de um sinal analógico  $x(t)$ . A ordem da equação de diferença, que é definida como o número de entradas prévias que devem ser armazenadas a fim de gerar uma dada saída, é determinada pelo maior valor de  $k$ . Assim, por exemplo, se  $k=2$ , o que corresponde à derivada de segunda ordem, a partir da equação (3.15) obtém-se primeiramente a transformada Z para  $x(n)$  [Lyons, 1997]:

$$X(z) = \sum_{n=0}^{\infty} x(n)z^{-n} = x(0) + x(1)z^{-1} + x(2)z^{-2} + \dots \quad (3.26)$$

A transformada Z para  $x(n-1)$ , que corresponde a derivada de primeira ordem  $dx/dt$  é:

$$\begin{aligned} ZT[x(n-1)] &= \sum_{n=0}^{\infty} x(n-1)z^{-n} \\ &= x(-1) + x(0)z^{-1} + x(1)z^{-2} + x(2)z^{-3} + \dots \\ &= x(-1) + z^{-1}[x(0) + x(1)z^{-1} + x(2)z^{-2} + \dots] \\ &= x(-1) + z^{-1}X(z) \end{aligned} \quad (3.27)$$

Verifica-se que  $x(-1)$  representa a condição inicial associada com a equação de diferença de primeira ordem.

A transformada Z para  $x(n-2)$ , equivalente a segunda derivada  $d^2 x/dt^2$  será:

$$\begin{aligned} ZT[x(n-2)] &= \sum_{n=0}^{\infty} x(n-2)z^{-n} \\ &= x(-2) + x(-1)z^{-1} + x(0)z^{-2} + x(1)z^{-3} + \dots \\ &= x(-2) + x(-1)z^{-1} + z^{-2}[x(0) + x(1)z^{-1} + \dots] \\ &= x(-2) + x(-1)z^{-1} + z^{-2}X(z) \end{aligned} \quad (3.28)$$

Verifica-se na equação acima que  $x(-2)$  e  $x(-1)$  representam as condições iniciais necessárias para resolver a equação de diferença de segunda ordem. Assim:

$$ZT[x(n-k)] = z^{-k} \sum_{m=0}^k \{x(-m)z^m + X(z)\} \quad (3.29)$$

Considerando as condições iniciais nulas, tem-se:

$$ZT[x(n-k)] = z^{-k} X(z) \quad (3.30)$$

Se, por exemplo, considerar um sistema descrito pela seguinte equação da diferença:

$$x[n] = y[n] - 0.9y[n-1] \quad (3.31)$$

Tomando a transformada Z em ambos os lados da Equação 3.31, utilizando a propriedade de deslocamento, e considerando a entrada  $x[n] = \mu[n]$  e a condição inicial  $y[-1] = 2$ , obtém a seguinte saída:

$$\begin{aligned} X[z] &= Y[z] - 0.9(y[-1] + z^{-1}Y[z]) \\ X[z] + 0.9y[-1] &= (1 - 0.9z^{-1})Y(z) \\ Y(z) &= \frac{X(z)}{1 - 0.9z^{-1}} + \frac{0.9y[-1]}{1 - 0.9z^{-1}} \end{aligned} \quad (3.32)$$

Logo  $Y(z)$  é representado pela soma de dois termos, um que depende da entrada e outro que depende da condição inicial. O termo dependente da entrada representa a resposta forçada do sistema. O termo dependente da condição inicial representa a resposta natural do sistema.

Fazendo  $x(z) = 1/(1 - z^{-1})$  e  $y[-1] = 2$ ;

$$Y(z) = \frac{1}{(1 - 0.9z^{-1})(1 - 0.9z^{-1})} + \frac{1.8}{1 - 0.9z^{-1}} \quad (3.33)$$

Expandindo em frações parciais obtém-se:

$$Y(z) = \frac{-9}{1 - 0.9z^{-1}} + \frac{10}{1 - z^{-1}} + \frac{1.8}{1 - 0.9z^{-1}} \quad (3.34)$$

Tomando a transformada Z inversa:

$$y[n] = -9(0.9)^n \mu[n] + 10\mu[n] + 1.8(0.9)^n \mu[n] \quad (3.35)$$

### 3.2.5 Filtros Digitais

Filtro digital é um dos grandes campos de aplicação de Processamento Digital de Sinal [Young, 1985]. Enquanto filtros analógicos operam com sinais contínuos e são tipicamente implementados com componentes discretos, ou seja, amplificadores, resistores e capacitores, os filtros digitais são algoritmos matemáticos implementados em *hardware* ou *software* que operam sobre um sinal discreto no tempo para produzir um sinal de saída desejado.

Os filtros digitais freqüentemente operam sobre sinais analógicos digitalizados. Logo sua implementação envolve o uso do conversor *ADC* (*Analog to Digital Converter*) que captura o sinal de entrada analógico amostrado periodicamente, convertendo-o numa série de amostras digitais, e envia o resultado para o conversor *DAC* (*Digital to Analog Converter*). Este converte o sinal de saída filtrado em valores analógicos que são então

filtrados de uma forma analógica para suavizar e remover componentes indesejáveis de alta frequência.

Os filtros digitais quando comparados com os analógicos são preferidos em muitas aplicações tais como compressão de dados, processamento de sinais biomédicos, processamento de imagens, transmissão de dados, áudio digital e cancelamento no eco do telefone. Esta preferência ocorre devido às vantagens oferecidas por estes filtros digitais. Estes podem oferecer uma resposta de fase linear exata, possuem uma maior acurácia; seu desempenho varia muito pouco com a mudança do meio, logo possuem menos sensibilidade à temperatura, o que elimina a necessidade de uma calibragem periódica; a frequência dos filtros pode ser modificada quando são implementados num processador programável; diversos sinais podem ser filtrados por um único filtro sem a necessidade de replicar o *hardware*; os filtros digitais também podem ser utilizados em frequências muito baixas e podem ser elaborados para trabalhar sobre um intervalo grande de frequências por uma simples mudança nas frequências amostradas.

### 3.2.6 Filtros com Resposta Impulsional Finita (FIR)

Um filtro FIR é aquele cujas respostas aos impulsos são de duração finita. Os filtros FIR são considerados ineficientes caso o sistema requeira uma função de transferência de ordem alta, quando comparada à ordem requerida por filtros digitais com resposta ao impulso de duração infinita. Entretanto os filtros FIR possuem vantagens quanto à possibilidade de terem fase linear exata na implementação, e de serem intrinsecamente estáveis quando realizados de forma não-recursiva.

Um sistema é dito estável quando uma entrada limitada em amplitude corresponde sempre a uma saída também limitada em amplitude. Sabe-se também que para um sistema causal estável a região de convergência da transformada Z à resposta impulsional tem que incluir a circunferência unitária [Porat, 1997].

Pode-se demonstrar que a condição necessária e suficiente de estabilidade é:

$$\sum_{k=-\infty}^{\infty} |x(n)| < \infty \quad (3.36)$$

Este resultado implica que  $X(z)$  converge sobre a circunferência unitária, logo a transformada de Fourier de um sinal no tempo discreto existe.

Logo, para o filtro FIR, uma nova amostra de saída  $y(n)$  deve ser gerada a cada  $n$  amostras de entrada. Portanto, para realizar o processamento necessário à implementação do filtro utiliza-se o processador DSP da família TMS320C6x que é projetado para realizar uma operação de multiplicação/acumulação a cada ciclo, leitura de operandos, e escrita na memória de dados. Isto acontece uma vez que o DSP em questão contempla as operações de multiplicação e adição em paralelo (MAC), acesso múltiplo à memória, possui registradores para armazenar dados temporários de instrução, gera de forma eficiente endereços para manipulação dos *arrays*, e possui características especiais como *delays* e modo de endereçamento circular.

Sabe-se que filtros não recursivos são caracterizados por uma equação de diferenças da forma:

$$y(n) = b_0x(n) + b_1x(n-1) + \dots + b_{L-1}x(n-L+1) = \sum_{k=0}^{N-1} b_k x(n-k) \quad (3.37)$$

Verifica-se desta forma que a convolução discreta, equação (3.27), quando constituída por termos finitos é bastante utilizada em projetos de filtros FIR.

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k) \quad (3.38)$$

Os coeficientes  $b_k$  se relacionam diretamente com as respostas ao impulso do sistema, logo  $b_k = h(k)$ . A transformada Z da equação acima, considerando as condições iniciais nulas, é descrita como:

$$Y(z) = h(0)X(z) + h(1)z^{-1}X(z) + h(2)z^{-2}X(z) + \dots + h(N-1)z^{-(N-1)}X(z) \quad (3.39)$$

Esta equação apresenta a multiplicação no domínio da frequência entre os coeficientes e a amostra do sinal de entrada.

$$Y(z) = H(z)X(z) \quad (3.40)$$

Onde  $H(z) = ZT[h(k)]$  é a função de transferência, ou seja, a relação entre a entrada e a saída [Orfanidis, 1996].

$$H(z) = \frac{Y(z)}{X(z)} \quad (3.41)$$

Logo,

$$\begin{aligned} H(z) &= \sum_{k=0}^{N-1} h(k)z^{-k} = h(0) + h(1)z^{-1} + h(2)z^{-2} + \dots + h(N-1)z^{-(N-1)} \\ &= \frac{h(0)z^{(N-1)} + h(1)z^{(N-2)} + h(2)z^{(N-3)} + \dots + h(N-1)}{z^{(N-1)}} \end{aligned} \quad (3.42)$$

Verifica-se que a equação de transferência possui N-1 pólos os quais estão localizados na origem, desta forma dentro da área de convergência, ou seja, dentro da circunferência de raio unitário. Logo, o filtro é considerável estável.

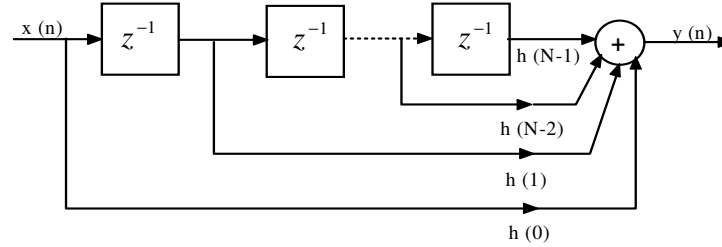


Figura 3.2.6.1 – Forma direta para os filtros digitais FIR.

O projeto de filtros FIR por meio de séries de Fourier é tal que a resposta em magnitude da função de transferência  $H(z)$  é uma aproximação da resposta em magnitude desejada, uma vez que o comportamento de um filtro é melhor caracterizado por sua resposta em frequência  $H(e^{j\omega})$ . A função de transferência desejada é:

$$H_d(\omega) = \sum_{n=-\infty}^{\infty} C_n e^{jn\omega T} \quad |n| < \infty \quad (3.43)$$

Onde  $C_n$  são os coeficientes da série de Fourier. Considerando  $\nu = f / F_N$ , onde  $F_N$  corresponde à frequência de Nyquist, ou seja,  $F_N = F_s / 2$ , e substituindo na equação acima, tem-se:

$$H_d(\nu) = \sum_{n=-\infty}^{\infty} C_n e^{jn\pi\nu} \quad (3.44)$$

Onde  $\omega T = 2\pi f / F_s = \pi\nu$  e  $|\nu| < 1$ . Os coeficientes de Fourier  $C_n$  são definidos como:

$$\begin{aligned} C_n &= \frac{1}{2} \int_{-1}^1 H_d(\nu) e^{-jn\pi\nu} d\nu \\ &= \frac{1}{2} \int_{-1}^1 H_d(\nu) (\cos n\pi\nu - j \sin n\pi\nu) d\nu \end{aligned} \quad (3.45)$$

Considerando que  $H_d(\nu)$  é uma função par, a Equação 3.45 fica reduzida a,

$$C_n = \int_0^1 H_d(\nu) \cos n\pi\nu d\nu, \quad n \geq 0, \quad (3.46)$$

Logo  $H_d(\nu) \sin n\pi\nu$  é uma função ímpar, ou seja,  $\int_{-1}^1 H_d(\nu) \sin n\pi\nu d\nu = 0$ .

Assim  $C_n = C_{-n}$ .

A Figura 3.2.6.2 representa idealmente as funções de transferência desejadas para os filtros seletores em frequência: passa-baixas, passa-altas, passa-faixa, rejeita-faixa, de maneira que seus respectivos coeficientes  $C_n$  são definidos por:

- Filtro passa-baixa ideal é determinado por  $C_0 = \nu_1$  e,

$$C_n = \int_0^1 H_d(\nu) \cos n\pi\nu d\nu = \frac{\sin n\pi\nu_1}{n\pi} \quad (3.47)$$

- Filtro passa-alta ideal é determinado por  $C_0 = 1 - \nu_1$  e,

$$C_n = \sum_{\nu_1}^1 H_d(\nu) \cos n\pi\nu = -\frac{\sin n\pi\nu_1}{n\pi} \quad (3.48)$$

- Filtro passa-faixa ideal é determinado por  $C_0 = \nu_2 - \nu_1$  e,

$$C_n = \int_{\nu_1}^{\nu_2} H_d(\nu) \cos n\pi\nu d\nu = \frac{\sin n\pi\nu_2 - \sin n\pi\nu_1}{n\pi} \quad (3.49)$$

- Filtro rejeita-faixa ideal é determinado por  $C_0 = 1 - (\nu_2 - \nu_1)$  e,

$$C_n = \int_0^{\nu_1} H_d(\nu) \cos n\pi\nu d\nu + \int_{\nu_2}^1 H_d(\nu) \cos n\pi\nu d\nu = \frac{\sin n\pi\nu_2 - \sin n\pi\nu_1}{n\pi} \quad (3.50)$$

As frequências de corte de cada filtro são determinadas pelos coeficientes  $\nu_1$  e  $\nu_2$ .

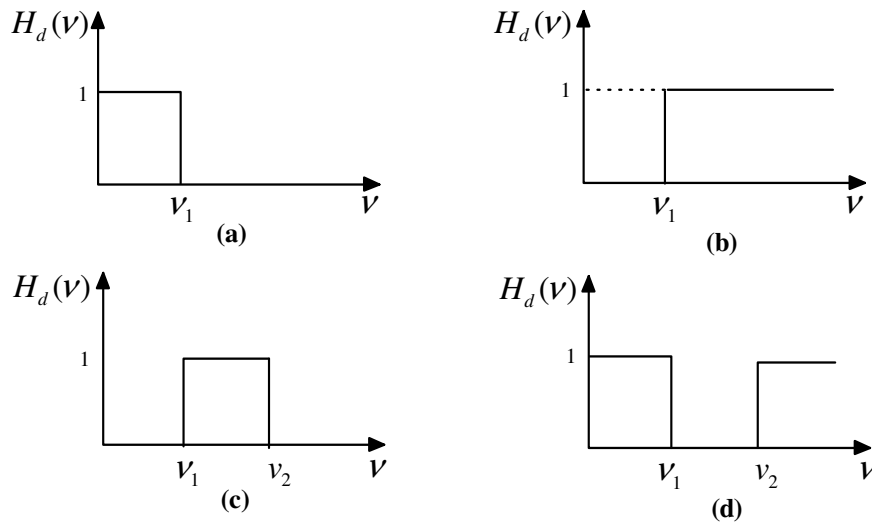


Figura 3.2.6.2 – Função transferência desejada: (a) passa-baixas; (b) passa-altas; (c) passa-faixa; (d) rejeita-faixa.

Verifica-se que todos os filtros apresentados anteriormente possuem uma duração infinita, levando a filtros não-realizáveis. Um filtro causal de duração finita pode ser obtido truncando-se a resposta impulsiva de duração infinita  $C_n$ , isto é multiplicando-a por uma classe de funções do tempo conhecidas como função janela,  $a(n)$ .

$$C'(n) = C(n)a(n) \quad (3.51)$$

E, posteriormente, tornando-a causal através da multiplicação da mesma por um fator de atraso.

Sabe-se que a lenta convergência da série de Fourier  $C_n$  particularmente perto dos pontos de descontinuidade acarreta na introdução de ondulações, ou oscilações de Gibbs. Apesar do aumento do número dos coeficientes  $C_n$  diminuir a amplitude das oscilações que encontram-se fora dos pontos de descontinuidades, este aumento não interfere nas outras amplitudes que permanecem inalteradas. Entretanto através da multiplicação no domínio do tempo, o que implica numa convolução no domínio da frequência, entre os coeficientes e uma função janela finita, Equação 3.52, tal que sua transformada de Fourier tenha baixos níveis em seus lóbulos secundários com respeito ao pico do lóbulo principal, implica na redução da descontinuidade verificada anteriormente. Logo somente uma função janela adequada pode diminuir as ondulações. As funções janelas podem ser do tipo Retangulares, de Hanning, de Hamming, de Blackman e de Kaiser.

Caso a função janela utilizada seja a retangular, tem-se:

$$a_R(n) = \begin{cases} 1, & \text{para } |n| \leq Q \\ 0, & \text{caso contrario} \end{cases} \quad (3.52)$$

O truncamento da série infinita, equação (3.44), e uma janela retangular é:

$$H_a(\nu) = \sum_{n=-Q}^Q C_n e^{jn\pi\nu} \quad (3.53)$$

Onde  $Q$  é positivo, finito e determina a ordem do filtro. Quanto maior o valor de  $Q$ , maior a ordem do filtro e melhor a aproximação da função de transferência desejada.

Substituindo na equação  $e^{jn\pi\nu} = z$  tem-se:

$$H_a(z) = \sum_{n=-Q}^Q C_n z^n \quad (3.54)$$

Ao introduzir uma amostra atrasada na equação anterior, obtém-se:

$$H_a(z) = z^{-Q} H_a(z) = \sum_{n=-Q}^Q C_n z^{n-Q} \quad (3.55)$$

Se  $n - Q = -i$ :

$$H(z) = \sum_{i=0}^{2Q} C_{Q-i} z^{-i} \quad (3.56)$$

Fazendo  $h_i = C_{Q-i}$  e  $N-1 = 2Q$  então:

$$H(z) = \sum_{i=0}^{N-1} h_i z^{-i} \quad (3.57)$$

Assim  $H(z)$  é expressa em termos de coeficientes de respostas impulsiais  $h_i$ , e  $h_0 = C_Q, h_1 = C_{Q-1}, \dots, h_Q = C_0, h_{Q+1} = C_{-1} = C_1, \dots, h_{2Q} = C_{-Q}$ .

Os coeficientes são simétricos com respeito a  $h_Q$ , com  $C_n = C_{-n}$ .

A ordem do filtro é dada por  $N = 2Q + 1$ . Por exemplo, se  $Q = 5$ , o filtro terá 11 coeficientes,

$$\begin{aligned} h_0 &= h_{10} = C_5 \\ h_1 &= h_9 = C_4 \\ h_2 &= h_8 = C_3 \\ h_3 &= h_7 = C_2 \\ h_4 &= h_6 = C_1 \\ h_5 &= C_0 \end{aligned}$$

Por exemplo, vamos calcular os coeficientes de um filtro FIR passa-baixas de ordem,  $N=11$ , com frequência de amostragem  $F_s=10kHz$  e frequência de corte  $f_c=1kHz$ .

Anteriormente foi definido  $F_N$  como,

$$F_N = F_s / 2.$$

Logo,

$$F_N = 10/2 = 5kHz.$$

Considerando  $\nu = f / F_N$ , pode-se calcular  $C_0$ ,

$$C_0 = \nu_1 = \frac{f_c}{F_N} = \frac{1}{5} = 0.2$$

Os outros coeficientes podem ser calculados por meio da Equação 3.47,

$$C_n = \frac{\sin 0.2\pi n}{n\pi}, n = \pm 1, \pm 2, \dots, \pm 5.$$



### 3.3 EXEMPLOS DE FILTROS COM RESPOSTAS IMPULSIONAL FINITA (FIR)

Três exemplos utilizando a equação discreta de convolução (Equação 3.48) serão desenvolvidos para ilustrar a implementação de filtros FIR.

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k) \quad (3.58)$$

No *CCS* os coeficientes são organizados dentro do *buffer* (*array*), onde o primeiro coeficiente,  $h(0)$ , está no começo (na primeira localização) do *buffer*, endereço de memória mais baixo. O último coeficiente,  $h(N-1)$ , encontra-se na última localização do *buffer*, endereço de memória mais alto. As amostras atrasadas são organizadas na memória de modo que as amostras mais novas,  $x(n)$ , encontram-se no começo do *buffer* de amostras, enquanto as amostras antigas,  $x(n-(N-1))$ , estão localizadas no final do *buffer*. Inicialmente todas as amostras são setadas para zero.

Tabela 3.3.1 – Coeficientes e amostras

<b>i</b>	<b>Coeficientes</b>	<b>Amostras</b>
<b>0</b>	$h(0)$	$x(n)$
<b>1</b>	$h(1)$	$x(n-1)$
<b>2</b>	$h(2)$	$x(n-2)$
.	.	.
.	.	.
.	.	.
$N-1$	$h(N-1)$	$x(n-(N-1))$

No tempo  $n$  a amostra mais nova que é adquirida após a geração de uma representação digital do sinal, ou seja, após a passagem do sinal pelo conversor AD, é armazenada no começo do *buffer* de amostras. A saída do filtro no tempo  $n$  é computado na equação de convolução, ou seja:

$$y(n) = h(0)x(n) + h(1)x(n-1) + \dots + h(N-2)x(n-(N-2)) + h(N-1)x(n-(N-1))$$

As amostras com atraso são então atualizadas, logo  $x(n-k) = x(n+1-k)$  podendo ser calculado a saída da unidade de tempo seguinte,  $y(n+1)$ , ou o período de amostragem

$T_s$ . Todas as amostras são armazenadas, exceto a amostra mais nova. Por exemplo,  $x(n-1) = x(n)$ , e  $x(n-(N-1)) = x(n-(N-2))$ . Este processo de armazenamento tem o efeito de mover para baixo os dados na memória como pode ser observado na tabela abaixo:

Tabela 3.3.2 – Coeficientes e amostras			
<b>i</b>	<b>Tempo <math>n</math></b>	<b>Tempo <math>n+1</math></b>	<b>Tempo <math>n+2</math></b>
<b>0</b>	$x(n)$	$x(n+1)$	$x(n+2)$
<b>1</b>	$x(n-1)$	$x(n)$	$x(n+1)$
<b>2</b>	$x(n-2)$	$x(n-1)$	$x(n)$
.	.	.	.
.	.	.	.
.	.	.	.
$N-3$	$x(n-(N-3))$	$x(n-(N-4))$	$x(n-(N-5))$
$N-2$	$x(n-(N-2))$	$x(n-(N-3))$	$x(n-(N-4))$
$N-1$	$x(n-(N-1))$	$x(n-(N-2))$	$x(n-(N-3))$

Verifica-se que no tempo  $n+1$ , a nova amostra  $x(n+1)$  é adquirida e armazenada no topo do *buffer* de amostra, neste caso a saída  $y(n+1)$  pode ser calculada como:

$$y(n+1) = h(0)x(n+1) + h(1)x(n) + \dots + h(N-2)x(n-(N-3)) + h(N-1)x(n-(N-2))$$

As amostras são, então, atualizadas para a unidade de tempo seguinte.

No tempo  $n+2$ , a nova amostra de entrada,  $x(n+2)$ , é adquirida. A saída seria:

$$y(n+2) = h(0)x(n+2) + h(1)x(n+1) + \dots + h(N-1)x(n-(N-3))$$

Este processo continua a calcular a saída do filtro e a atualizar amostras atrasadas em cada unidade de tempo.

### 3.3.1 Exemplo 1: Filtro Rejeita-Faixa

Esse exemplo analisa as características respectivas do filtro FIR do tipo rejeita-faixa através da implementação do código *fir.c* e da utilização de alguns recursos do CCS.

```
//fir.c
//Filtro FIR. Inclui arquivos com coeficiente do tamanho N

#include "bs2700.cof"          // arquivo de coeficiente BS com frequência de 2700Hz
int yn = 0;                   // yn determina a variável da saída do filtro
short dly[N];                 // dly determina o atraso das amostras
interrupt void c_int11()      // ISR
{
    short i;
    dly[0] = input_sample();   // a mais nova entrada no topo do buffer
    yn = 0;                   // inicializando a saída do filtro
    for (i = 0; i < N; i++)
        yn += (h[i] * dly[i]); //  $y(n) += h(i) * x(n-i)$ 
    for (i = N-1; i > 0; i--)   // inicializa o final do buffer
        dly[i] = dly[i-1];    // atualiza atrasos com fluxo dos dados
    output_sample(yn >> 15);   // saída do filtro
    return;
}

void main()
{
    comm_intr();               // inicializa o DSK, codec, McBSP
    while(1);                  // laço infinito
}

//bs2700.cof coeficientes do filtro FIR rejeita- faixa

#define N 89                   // número de coeficientes

short h[N]={-14,23,-9,-6,0,8,16,-58,50,44,-147,119,67,-245,
            200,72,-312,257,53,-299,239,20,-165,88,0,105,
            -236,33,490,-740,158,932,-1380,392,1348,-2070,
            724,1650,-2690,1104,1776,-3122,1458,1704,29491,
            1704,1458,-3122,1776,1104,-2690,1650,724,-2070,
            1348,392,-1380,932,158,-740,490,33,-236,105,0,
            88,-165,20,239,-299,53,257,-312,72,200,-245,67,
            119,-147,44,50,-58,16,8,0,-6,-9,23,-14};
```

#### Considerações sobre o programa

O arquivo *bs2700.cof* incluso no código *fir.c* especifica as características que contêm os coeficientes do filtro FIR rejeita-faixa centrado em 2700Hz.

O *buffer dly[N]*, apresentado no código *fir.c*, é criado para as amostras atrasadas. A mais nova amostra de entrada,  $x(n)$ , é adquirida através de *dly[0]* e armazenada no começo do *buffer*. Os coeficientes são armazenados em outro *buffer*, *h[N]*, com *h[0]* no começo do *buffer* dos coeficientes. As amostras e os coeficientes são organizados nos seus respectivos *buffers*, como foi apresentado anteriormente nas Tabelas 3.3.1 e 3.3.2.

No código *fir.c* pode-se observar a utilização de dois *loops* dentro da rotina de interrupção. O primeiro *loop* implementa a equação de convolução com  $N$  coeficientes e  $N$  amostras atrasadas, num tempo  $n$  específico. Neste tempo  $n$  a saída é:

$$y(n) = h(0)x(n) + h(1)x(n-1) + \dots + h(N-2)x(n-(N-2)) + h(N-1)x(n-(N-1))$$

As amostras atrasadas são atualizadas dentro do segundo *loop* para que sejam utilizadas no cálculo de  $y(n)$ , no tempo de  $n+1$ , ou  $y(n+1)$ . Sabe-se que a amostra de entrada mais recentemente adquirida, neste exemplo, sempre se encontra no começo do *buffer* de amostras. Assim a posição de memória que antes tinha a amostra  $x(n)$  agora contém a amostra mais recentemente adquirida  $x(n+1)$  e conseqüentemente a saída  $y(n+1)$ , no tempo de  $n+1$ .

### Criação do projeto

Para criar este projeto no *Code Composer Studio™ IDE*, é preciso adicionar os arquivos necessários à construção do projeto *Fir*, como fora feito nos exemplos anteriores.

1. Crie o arquivo denominado *fir.pjt* clicando em *Project ➔ File*
2. Selecione *Project ➔ Add Files to Project* no CCS. Abra a pasta *Fir* do CD que acompanha o presente trabalho, e adicione ao projeto os dois arquivos “.c”, *fir.c* e *c6xdskinit.c*.
3. Novamente, selecione *Project ➔ Add Files to Project*. Abra a pasta *Fir* e adicione o arquivo do tipo assembly, *vectors\_11.asm*.
4. Repita o passo 3 e adicione o arquivo *c6xdsk.cmd*.
5. Repita o passo 3 e adicione na pasta *Libraries* do projeto o arquivo *rts6700.lib*.

6. Carregue na pasta *Includes* do projeto os arquivos de cabeçalho. Ou seja, clique em *Project* ➔ *Scan All File Dependencies* e observe a adição dos seguintes arquivos; *bs2700.cof*, *c6x.h*, *c6xdsk.h*, *c6xdskinit.h*. e *c6xinterrupts.h*.

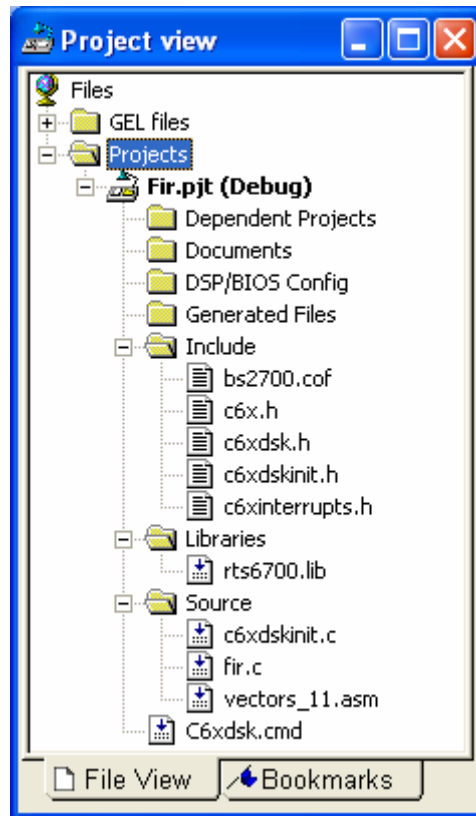


Figura 3.3.1.1 – Visualização do projeto.

### Configuração do compilador e do *linker*

Em seguida é necessário configurar os parâmetros do compilador e do *linker*:

1. Clique em *Project* ➔ *Build Options*.
2. Na janela *Build Options* ➔ *Compiler*.
3. Na lista *Category* ➔ *Basic*.
4. Em *Basic* escolha as seguintes opções:
  - a. *Target Version* ➔ *Default*.
  - b. *Generate Debug Info* ➔ *Full Symbolic Debug (-g)*.
  - c. *Opt Speed vs Size* ➔ *Speed Most Critical (no ms)*.
  - d. *Opt Level* ➔ *None*.

e. *Program Level Opt* → *None*.

Verifique na janela *Buil Options* a seleção dos parâmetros *-g-k-s*.

5. Clique em *Linker* na janela *Build Options*

6. Na lista *Category* → *Basic*.

7. Em *Basic* ajuste as seguintes opções:

f. Habilite *Suppress Banner* (*-q*).

g. Habilite *Exhaustively Reas Libraries* (*-x*)

h. Em *Otput Filename* (*-o*) coloque o nome do arquivo de saída, ou seja, *fir.out*.

i. Em *Autoint Model* selecione *Run-Time Autoinitialization* (*-c*).

8. Clique em *Ok* e feche a janela *Build Options*

### **Geração do executável e carregamento do programa**

1. Selecione *Project* → *Build*, para que o projeto possa ser construído.

2. Carregue o executável através do comando *File* → *Load Program*, selecionando o arquivo *fir.out*.

3. Faça o debug do programa através de *Debug* → *Run*.

### **Construção gráfica no CCS**

Para construir os gráficos no domínio do tempo e da frequência é necessário que você:

1. Selecione *View* → *Graph* → *Time/Frequency*. Ao abrir *Graph Property Dialog* faça as seguintes modificações:

- *Start Adress* → *h*, como pode ser verificado no código *fir.c*.
- *Acquisition Buffer Size* → 89, representando o tamanho do *buffer*.
- *DisplayData Size* → 128.
- *DSP Data Size* → 16-bit signed integer.
- *Sampling Rate (Hz)* → 8000.

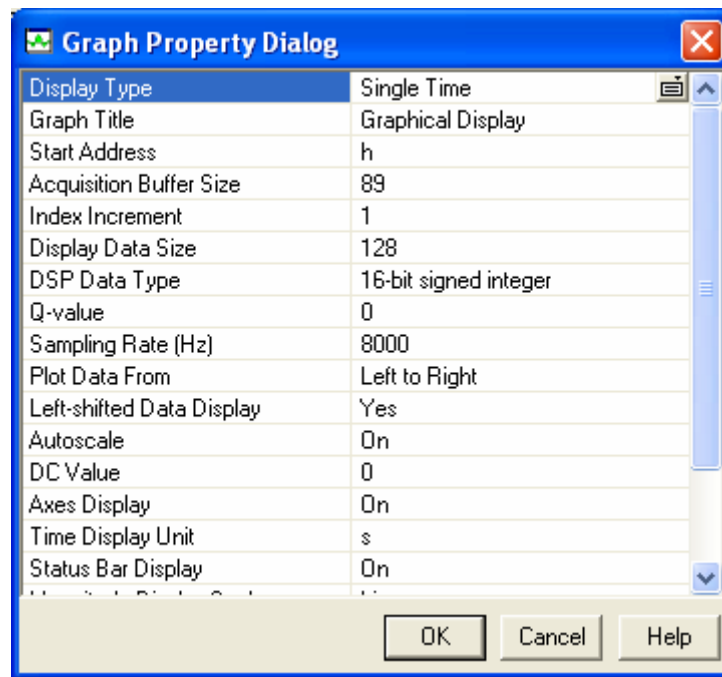


Figura 3.3.1.2 – Janela Graph Property Dialog com modificações.

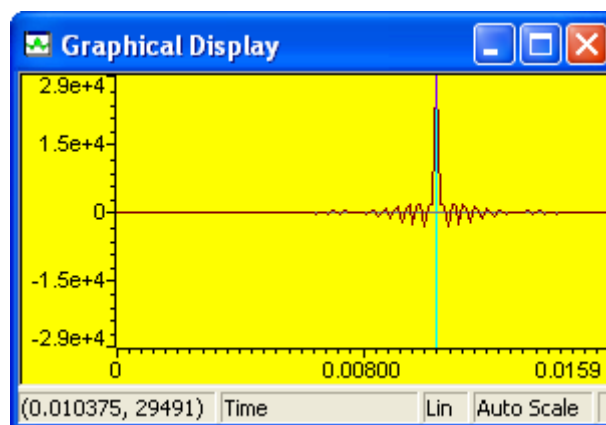


Figura 3.3.1.3 – Resposta impulsional.

Para o domínio da frequência se devem modificar algumas características na janela *Graph Property Dialog*. Logo:

2. Selecione View → Graph → Time/Frequency. Ao abrir *Graph Property Dialog* faça as seguintes modificações:
  - Selecione em *DisplayType* → *FFT Magnitude*.
  - *FFT Framesize* → 89.

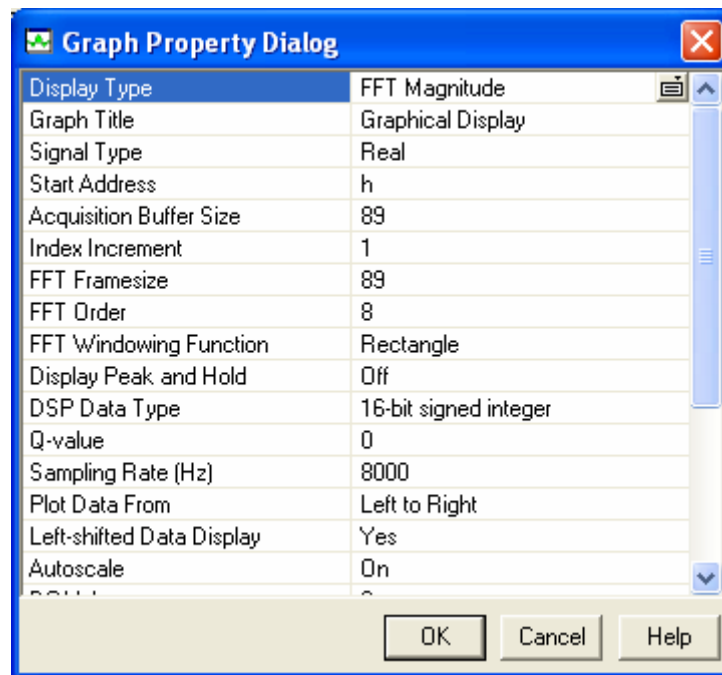


Figura 3.3.1.4 – Janela Graph Property Dialog com modificações para a FFT.

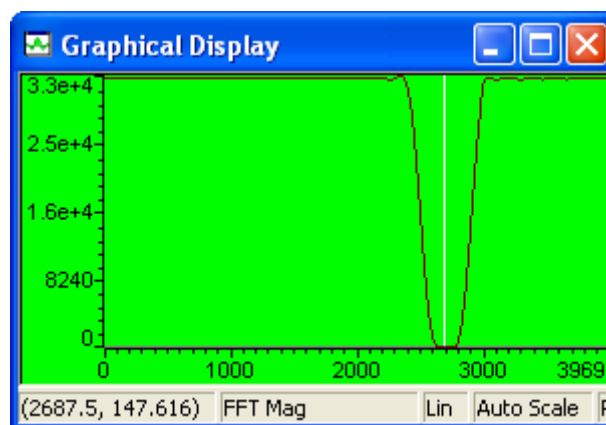


Figura 3.3.1.5 – FFT do filtro rejeita-faixa.

Verifica na Figura 3.3.1.5 que a faixa de rejeição do filtro está centrada em 2700Hz.

### Projeto utilizando o MATLAB

A seguir será explorada a ferramenta de projeto de filtros que o software MATLAB 7.0 disponibiliza.

1. Digite *sptool* na linha de comando do MALAB;



2. Ao abrir a janela *SPTool:startup.stp* selecione *Filter* → *Firbp(design)* → *New*.

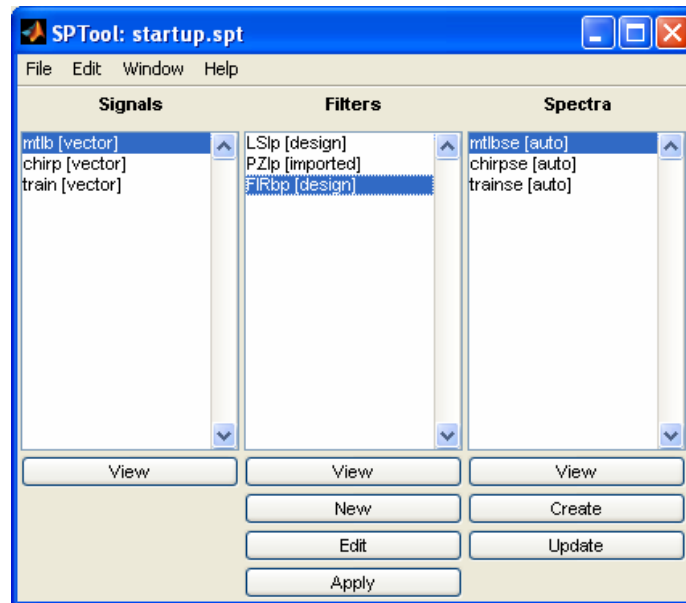


Figura 3.3.1.6 – Janela SPTool:startup.stp.

3. Na janela *Filter Designer* determine as características necessárias para o filtro FIR rejeita-faixa, como pode ser verificado na Figura (3.3.1.6).
- *Sampling frequency* → 8000.
  - *Algorithm* → *Kaiser Window FIR*.
  - *Order* → 88. Observe que o *Algorithm* utilizado é a Kaiser, cujo comprimento é determinado por  $N = M + 1$ , logo o número de *Order* é 88, resultando  $N = 89$ .
  - $Fc1 \rightarrow 2500$  e  $Fc2 \rightarrow 2900$ , determinando a faixa onde existe uma probabilidade do filtro estar centrado.
  - *Beta* → 4. O parâmetro  $\beta$  é utilizado para controlar tanto a largura do lobo principal quanto a razão entre os lobos principal e secundário.
4. Mude o nome do filtro para *bs2700* através dos comandos *Edit* → *Name*. na janela *SPTool:startup.stp*.

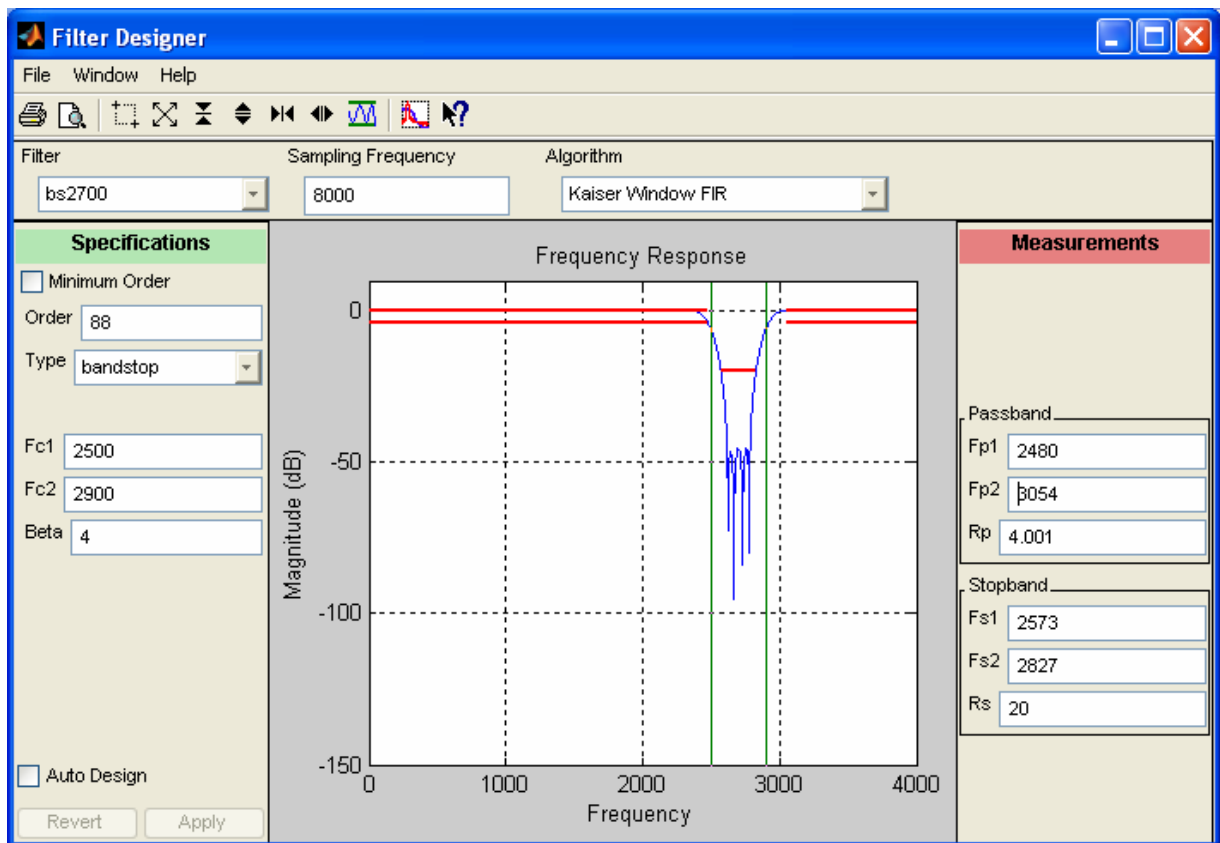


Figura 3.3.1.7 – Características do filtro FIR rejeita-faixa centrado em 2700Hz.

5. Selecione *File* ➔ *Export* ➔ *Filter:bs2700 [design]* ➔ *Export to workspace*
6. Com a finalidade de encontrar os coeficientes da função de transferência acesse a área de trabalho do Matlab e escreva os seguintes comandos:

```
>>bs2700.tf.num;
>>round (bs2700.tf.num*2^15)
```

Logo se observa que os coeficientes escalados perto de  $2^{15}$  do filtro FIR rejeita-faixa, listados dentro da área de trabalho do Matlab, são os mesmos apresentados no arquivo de coeficientes *bs2700.cof*.

## Resultados Experimentais

Ao se injetar na entrada da placa DSK, conector IN (J7), um ruído pseudo-aleatório, gerado a partir do programa que se encontra no anexo I (*ruído.m*), e implementando o filtro em estudo, obteve-se por meio do Osciloscópio – Agilent 5AG21A, conector OUT (J6), uma resposta em frequência muito próxima da que fora obtida teoricamente.

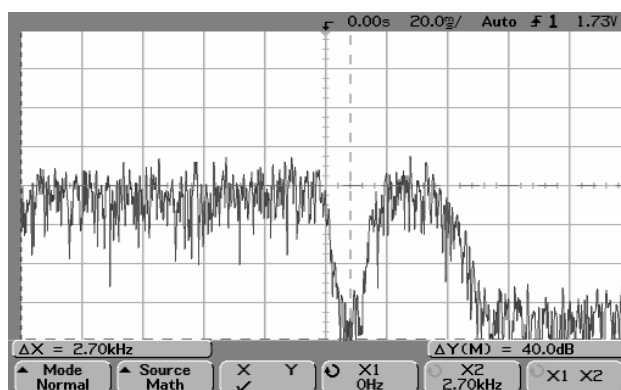


Figura 3.3.1.8 – Resposta em frequência da saída do filtro FIR rejeita-faixa centrado em 2700Hz, obtida com um osciloscópio.

O decaimento da magnitude observado a partir da frequência de aproximadamente 3500Hz, ocorre por conta do próprio *codec* do DSK que é um circuito passa-baixa com frequência de corte de 3500Hz.

A Figura 3.3.1.9 mostra a resposta do filtro construída a partir de quatorze pontos obtidos experimentalmente com o auxílio do osciloscópio. Foi utilizado o método de interpolação PCHIP (*Piecewise Cubic Hermite Interpolating Polynomial*). O código fonte gerador da resposta em frequência observada na Figura 3.3.1.9, implementado na ferramenta MATLAB 7.0, encontra-se no anexo I.

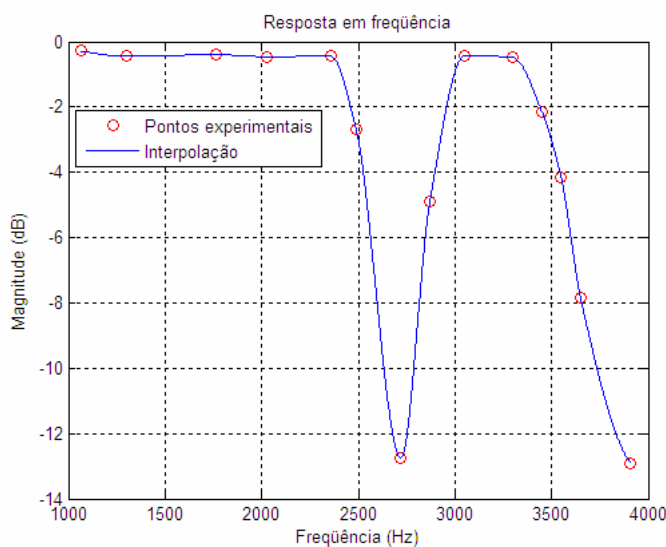


Figura 3.3.1.9 – Resposta em frequência da saída do filtro FIR rejeita-faixa centrado em 2700Hz, obtido por meio da interpolação.

### 3.3.2 Exemplo 2: Filtro Passa-Faixa

Esse exemplo analisa as características do filtro FIR passa-faixa a partir da modificação do arquivo dos coeficientes incluso no código *fir.c*. Logo troca-se *bs2700.cof* por *bp1750.cof* como pode ser verificado abaixo.

```
//fir.c
//Filtro FIR. Inclui arquivos com coeficiente do tamanho N

#include "bp1750.cof"           //arquivo de coeficiente BP com frequência de 1750Hz
int yn = 0;                    //yn determina a variável da saída do filtro
short dly[N];                  //dly determina o atraso das amostras
interrupt void c_int11()       //ISR
{
    short i;
    dly[0] = input_sample();    //mais nova entrada no topo do buffer
    yn = 0;                     //inicializando a saída do filtro
    for (i = 0; i < N; i++)
        yn += (h[i] * dly[i]);  //y(n) += h(i)* x(n-i)
    for (i = N-1; i > 0; i--)    //inicializa o final do buffer
        dly[i] = dly[i-1];     //atualiza atrasos com fluxo dos dados
    output_sample(yn >> 15);    //saída do filtro
    return;
}

void main()
{
    comm_intr();                //inicializa o DSK, codec, McBSP
    while(1);                   //laço infinito
}

// bp1750.cof coeficientes do filtro FIR passa-faixa

#define N 81                    //número de coeficientes

short h[N]= {0,-25,-12,30,28,-21,-29,4,0,-5,57,57,-107,-170,
99,306,0,-392,-162,356,288,-199,-255,31,0,-38,398,389,-708,-1104,
640,1989,0,-2676,-1169,2785,2550,-2119,-3667,793,4096,793,
-3667,-2119,2550,2785,-1169,-2676,0,1989,640,-1104,-708,389,
398,-38,0,31,-255,-199,288,356,-162,-392,0,306,99,-170,-107,57,
57,-5,0,4,-29,-21,28,30,-12,-25,0};
```

O arquivo *bp1750.cof* especifica as características do filtro FIR, onde a quantidade de coeficientes utilizados é  $N = 81$ , para representar o filtro passa-faixa, centrado em 1750Hz.

## Criação do projeto

Após a modificação proposta no código *fir.c*, cria-se o projeto como pode ser visto na Figura 3.3.2.1.

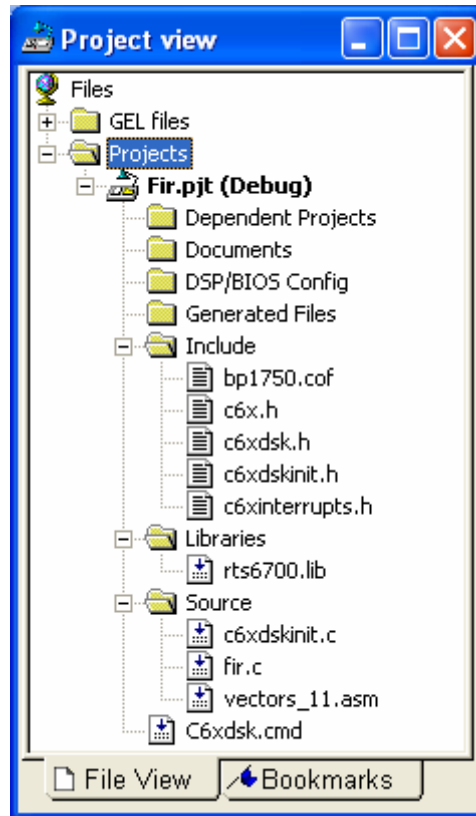


Figura 3.3.2.1 – Visualização do projeto

Para gerar o executável e carregar o programa utilizam-se as mesmas configurações dos parâmetros do compilador e do *linker* do exemplo anterior.

## Construção gráfica no CCS

Para construir os gráficos no domínio do tempo e da frequência é necessário que você:

1. Selecione View → Graph → Time/Frequency.
  - Start Adress → *h*
  - Acquisition Buffer Size → 81.
  - DisplayData Size → 128.

- *DSP Data Size* → 16-bit signed integer.
- *Sampling Rate (Hz)* → 8000.

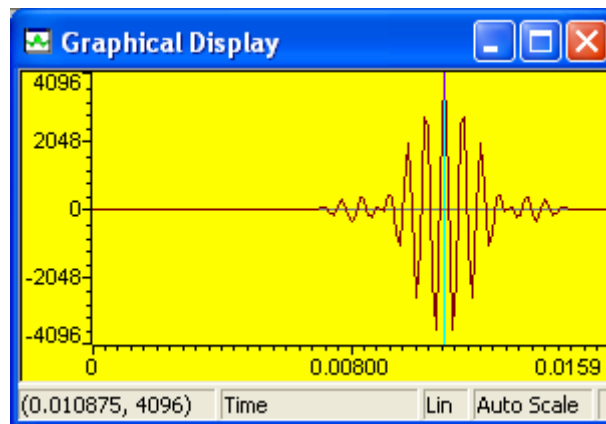


Figura 3.3.2.2 – Resposta impulsional.

Para o domínio da frequência devem-se modificar algumas características na janela *Graph Property Dialog*. Logo:

2. Seleccione View → *Graph* → *Time/Frequency*.
  - Seleccione em *DisplayType* → *FFT Magnitude*.
  - *FFT Framesize* → 81.

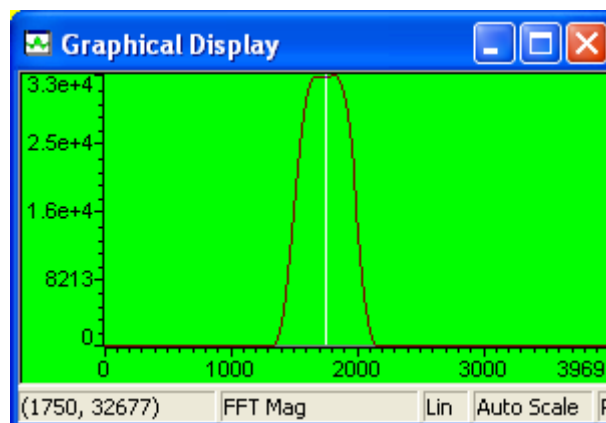


Figura 3.3.2.3 – FFT do filtro passa-faixa centrado em 1750 Hz.

## Projeto utilizando o MATLAB

Utilize a ferramenta *sptool*, como no exemplo anterior, e determine as seguintes características do filtro FIR passa-faixa na janela *Filter Designer*:

- *Sampling frequency* → 8000.
- *Algorithm* → *Kaiser Window FIR*.
- *Order* → 80.
- *Fc1* → 1500 e *Fc2* → 2000, determinando a faixa onde existe uma probabilidade do filtro estar centrado.
- *Beta* → 5.

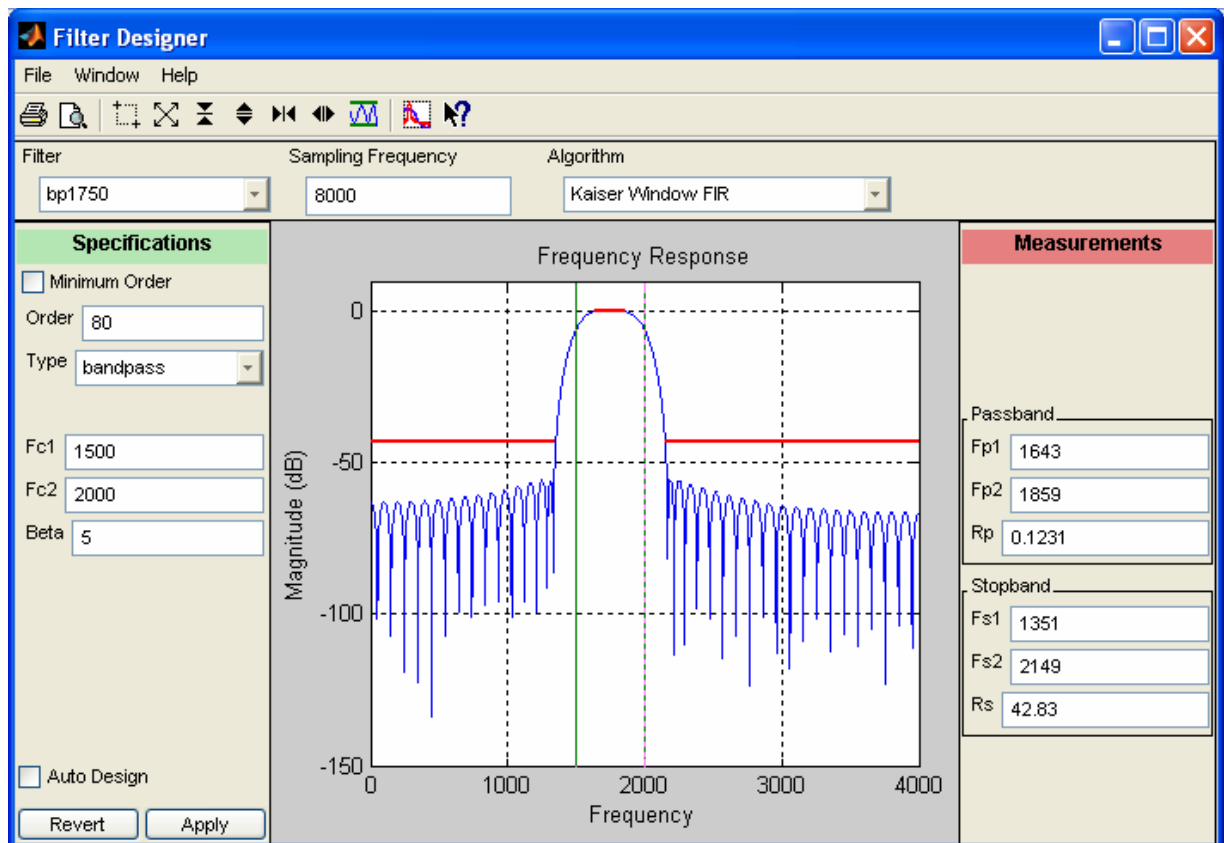


Figura 3.3.2.4 – Características do filtro FIR passa-faixa centrado em 1750Hz.

Exporte o arquivo *Filter:bp1750 [design]* para a área de trabalho do MATLAB e verifique no mesmo, através dos comandos abaixo, a obtenção dos coeficientes contidos no arquivo *bp1750.cof*.

```
>>bp1750.tf.num;  
>>round (bp1750.tf.num*2^15)
```

### Resultados Experimentais

Ao se injetar na entrada do *DSK*, conector IN (J7), o mesmo ruído pseudo-randômico, *ruído.m*, utilizado no exemplo anterior, obteve-se com o auxílio do Osciloscópio – Agilent 5AG21A, que se encontra conectado com a saída da placa DSK, conector OUT (J6), uma resposta em frequência, Figura 3.3.2.5, próxima à obtida teoricamente para o filtro FIR passa-faixa centrada em 1750Hz, Figura 3.3.2.6, que é implementado através da ferramenta do MATLAB 7.0, e o seu código fonte encontra-se no Anexo I.

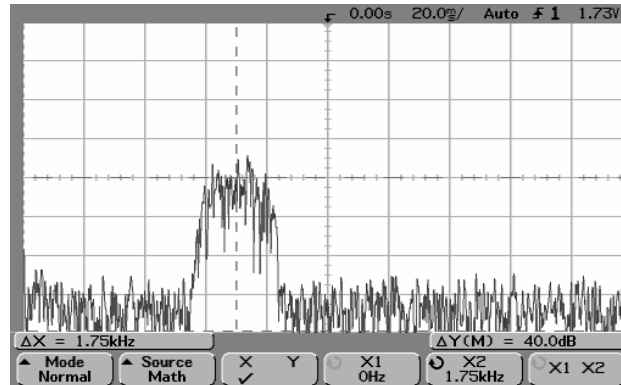


Figura 3.3.2.5 – Resposta em frequência da saída do filtro FIR passa-faixa centrado em 1750Hz, obtido com um osciloscópio.



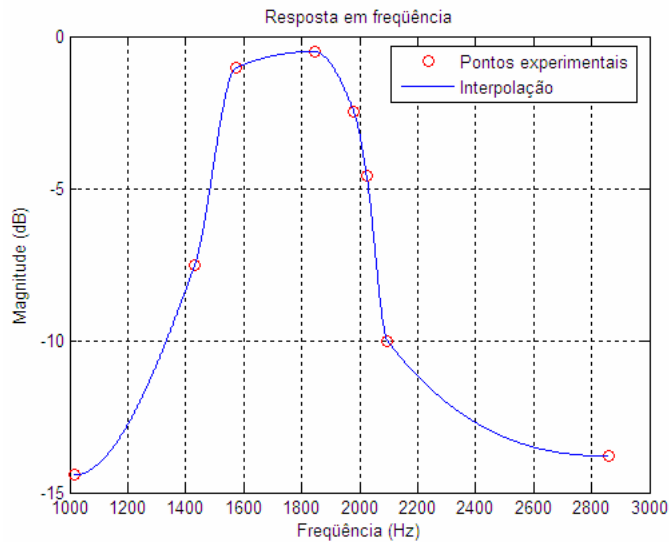


Figura 3.3.2.6 – Resposta em frequência da saída do filtro FIR passa-faixa centrado em 1750Hz, obtido por meio da interpolação.

### 3.3.3 Exemplo 3: Implementação de dois filtros FIR rejeita-faixa para a recuperação de uma entrada de voz corrompida.

Esse exemplo analisa o funcionamento simultâneo de dois filtros FIR rejeita-faixa que recuperam uma entrada de voz que inicialmente está corrompida, através da implementação do código *Notch2.c* e da utilização de alguns recursos do CCS.

```
//Notch2.c
//Implementação de dois filtros FIR do tipo rejeita-faixa para a remoção dos ruídos.

#include "BS900.cof"           // arquivo de coeficiente BS com frequência de
                               // 900 Hz
#include "BS2700.cof"          // arquivo de coeficiente BS com frequência de
                               // 2700 Hz

short dly1[N]={0};             // dly1 determina o atraso das amostras do primeiro filtro
short dly2[N]={0};             // dly2 determina o atraso das amostras do primeiro filtro
int y1out = 0, y2out = 0;       // inicia a saída de cada filtro
short out_type = 1;            // indicador para o tipo de saída

interrupt void c_int11()       // ISR
{
    short i;
    dly1[0] = input_sample();   // mais nova entrada no topo do buffer
    y1out = 0;                  // início da saída do primeiro filtro
    y2out = 0;                  // início da saída do segundo filtro
    for (i = 0; i < N; i++)
        y1out += h900[i]*dly1[i]; // y1(n)+=h900(i)*x(n-i)
```

```

dly2[0]=(y1out >>15);          // saída do primeiro filtro, entrada para o segundo filtro
for (i = 0; i < N; i++)
    y2out += h2700[i]*dly2[i]; // y2(n)+=h2700(i)*x(n-i)

    for (i = N-1; i > 0; i--)    // final do buffer
    {
        dly1[i] = dly1[i-1];    // atualiza as amostras do primeiro buffer
        dly2[i] = dly2[i-1];    // atualiza as amostras do segundo buffer
    }
    if (out_type==1)             // o indicador está na posição 1
        output_sample(dly1[0]); // entrada corrompida (voz+sinais)
if (out_type==2)
    output_sample(y2out>>15);   // saída do segundo filtro (voz)
    return;                     // retorna do ISR
}
void main()
{
    comm_intr();                // inicializa o DSK, codec, McBSP
while(1);                      // laço infinito
}

```

//BS900.cof coeficientes do filtro FIR rejeita- faixa centrado em 900Hz

```

#define N 89                // tamanho do filtro

short h900[N]={-495,-1367,1234,-681,-849,-840,-373,139,467,502,
298,11,-201,-249,-158,-35,20,-27,-106,-107,42,294,491,452,101,-445,
-903,-966,-493,363,1190,1518,1081,5,-1212,-1931,-1718,-601,908,2049,
2212,1269,-337,-1807,30371,-1807,-337,1269,2212,2049,908,-601,-1718,
-1931,-1212,5,1081,1518,1190,363,-493,-966,-903,-445,101,452,491,294,42,
-107,-106,-27,20,-35,-158,-249,-201,11,298,502,467,139,-373,-840,-849,
-681,1234,-1367,-495};

```

//BS2700.cof coeficientes do filtro FIR rejeita- faixa centrado em 2700Hz

```

#define N 89                // tamanho do filtro

short h2700[N]= {5255,-4173,-3204,-4194,27,-845,-2073,1043,
-403,-1301,1233,-404,-917,1214,-495,-672,1139,-584,-483,1065,
-660,-330,983,-721,-197,913,-765,-76,835,-810,30,776,-845,124,
703,-874,222,630,-891,311,559,-904,399,483,31861,483,399,-904,
559,311,-891,630,222,-874,703,124,-845,776,30,-810,835,-76,-765,
913,-197,-721,983,-330,-660,1065,-483,-584,1139,-672,-495,1214,
-917,-404,1233,-1301,-403,1043,-2073,-845,27,-4194,-3204,-4173,5255};

```

Verifica-se no código *Notch2.c* que se utiliza um *buffer* para cada amostra atrasada de cada filtro. A saída do primeiro filtro rejeita-faixa centrado em 900Hz, torna-se a entrada do segundo filtro rejeita-faixa centrado em 2700Hz.

## Criação do projeto

O projeto *Notch2.pjt* é construído como verificado na figura (3.3.3.1), e depois utilizando as mesmas configurações dos parâmetros do compilador e do *linker* como fora feito nos exemplos anteriores presente neste tutorial, o executável é gerado e o programa é carregado.

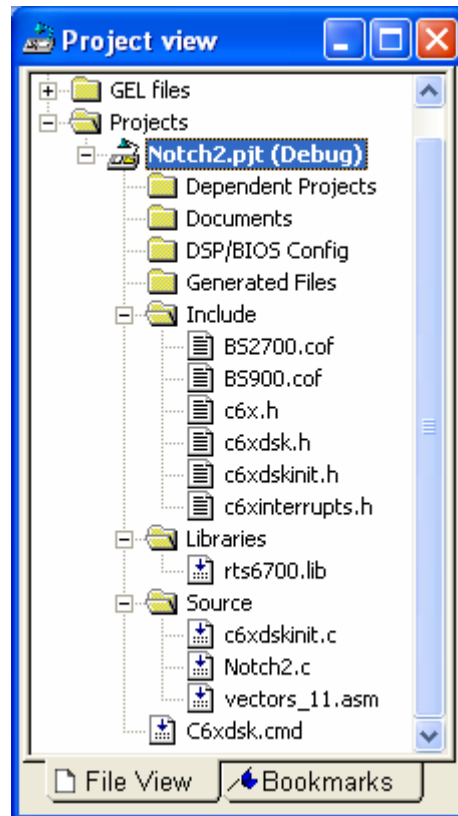


Figura 3.3.3.1 – Visualização do projeto.

## Resultados Experimentais

Após a construção do projeto aplicou-se à entrada *J7* do *DSK* o sinal de voz corrompido, *corruptvoice.wav*, que se encontra-se no diretório *notch2* do CD que acompanha esse trabalho.

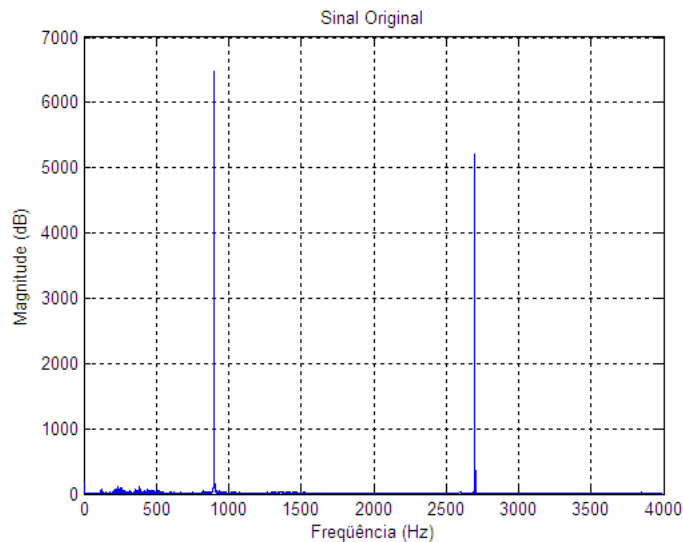


Figura 3.3.3.2 – Espectro do sinal de voz corrompido por dois sinais senoidais centrados em 900Hz e 2700Hz.

O sinal de voz corrompido, Figura 3.3.3.2, cujo código fonte fora implementado na ferramenta MATLAB 7.0, encontra-se no anexo I, apresenta dois sinais senoidais centrados em 900 e 2700Hz, adicionados ao sinal de voz original. Ao passar por dois filtros rejeita-faixa, cujas frequências são as mesmas dos ruídos senoidais presentes no sinal corrompido, obtém-se uma saída recuperada que pode ser capturada por meio da saída *J6* do *DSK*.

Ao se aplicar como entrada do filtro o mesmo ruído pseudo-randômico, *ruído.m*, utilizado nos exemplos anteriores, obteve-se com o auxílio do Osciloscópio – Agilent 5AG21A, uma resposta em frequência que explicita as faixas de rejeição impostas pelos dois filtros conectados em série (Figura 3.3.3.3).

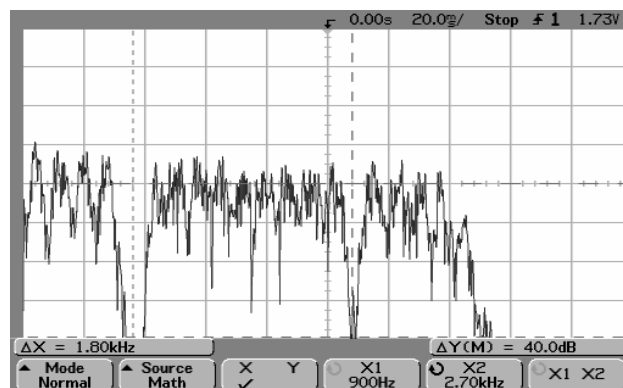


Figura 3.3.3.3 – Resposta em frequência da saída dos filtros FIR rejeita-faixas em série centrados em 900Hz e 27000Hz, obtida no osciloscópio.

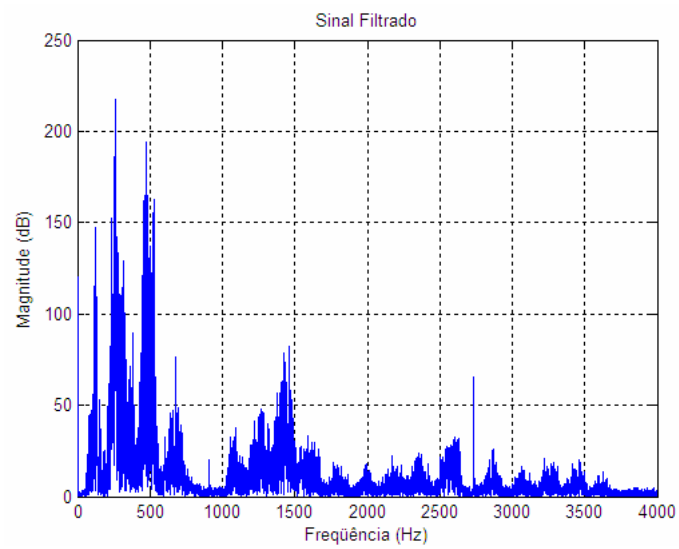


Figura 3.3.3.4 – Resposta em frequência do sinal recuperado.

## 4 - TUTORIAL 3: FILTROS COM RESPOSTA IMPULSIONAL INFINITA (IIR)

### 4.1 OBJETIVO

- Introdução a estruturas de filtros IIR: forma direta do tipo I, forma direta do tipo II, forma cascata e forma paralela.
- Introdução aos conceitos de transformação bilinear.
- Projeto e implementação de filtros com resposta impulsional infinita (IIR)
- Programação de exemplos usando código C e o TMS320C6711 DSK

Esse tutorial tem por finalidade apresentar os conceitos básicos sobre filtros com resposta impulsional infinita, filtros recursivos. Projetar dois exemplos de filtro IIR a partir da estrutura de forma direta do tipo II em cascata no TMS320C6711 DSK.

### 4.2 INTRODUÇÃO

#### 4.1 Filtros com Resposta Impulsional Infinita (IIR)

Um filtro IIR é aquele cujas respostas aos impulsos são de duração infinita. Este tipo de filtro também é conhecido como recursivo, uma vez que a sua saída  $y(n)$  depende dos sinais de entrada corrente, e dos sinais de saída passados [Marven & Ewers, 1996].

$$\begin{aligned} y(n) &= b_0 x(n) + b_1 x(n-1) + \dots + b_{L-1} x(n-L+1) - a_1 y(n-1) - \dots - a_M y(n-M) \\ &= \sum_{k=0}^{N-1} b_k x(n-k) - \sum_{m=1}^M a_m y(n-m) \end{aligned} \quad (4.1)$$

A transformada Z da equação de diferenças do filtro do tipo recursivo, equação (4.1), considerando todas as condições iniciais nulas é representada por:

$$\begin{aligned} Y(z) &= b_0 X(z) + b_1 z^{-1} X(z) + b_2 z^{-2} X(z) + \dots + b_{N-1} z^{-(N-1)} X(z) \\ &\quad - a_1 z^{-1} Y(z) - a_2 z^{-2} Y(z) - \dots - a_M z^{-M} Y(z) \end{aligned} \quad (4.2)$$

Assim:

$$Y(z) = \left( \sum_{k=0}^{N-1} b_k z^{-k} \right) X(z) + \left( \sum_{m=1}^M a_m z^{-m} \right) Y(z) \quad (4.3)$$

A função de transferência do filtro recursivo é:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{N-1} b_k z^{-k}}{1 + \sum_{m=1}^M a_m z^{-m}} = \frac{B(z)}{1 + A(z)} \quad (4.4)$$

A partir da equação (4.4) verifica-se que o filtro do tipo IIR é composto por dois filtros do tipo FIR  $A(z)$  e  $B(z)$  [Chen & Sorensen, 1997].

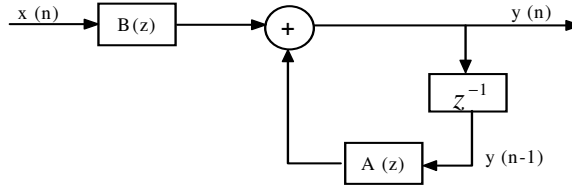


Figura 4.1.1 – Filtro IIR composto por dois filtros FIR.

Considerando  $M = N - 1$  na equação (4.2) obtém-se:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots + b_M z^{-M}}{1 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_M z^{-M}} = \frac{N(z)}{D(z)} \quad (4.5)$$

Multiplicando o numerador e o denominador por  $z^M$ ,  $H(z)$  torna-se:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 z^M + b_1 z^{M-1} + b_2 z^{M-2} + \dots + b_M}{z^M + a_1 z^{M-1} + a_2 z^{M-2} + \dots + a_M} = C \prod_{i=1}^M \frac{z - z_i}{z - p_i} \quad (4.6)$$

A função de transferência, equação (4.6), possui  $M$  zeros e  $M$  pólos. Caso todos os coeficientes  $a_i$  sejam nulos a função de transferência será composta apenas por  $M$  pólos na origem do plano  $z$  representando um filtro FIR que é não-recursivo. O filtro IIR é considerado estável quando os seus pólos encontram-se dentro do círculo unitário.

Existem várias estruturas que representam um filtro do tipo IIR, dentre elas estão a forma direta do tipo I, forma direta do tipo II, forma cascata, e forma paralela.

A Figura 4.1.2 representa um filtro IIR na forma direta do tipo I, obtida a partir da equação (4.2).

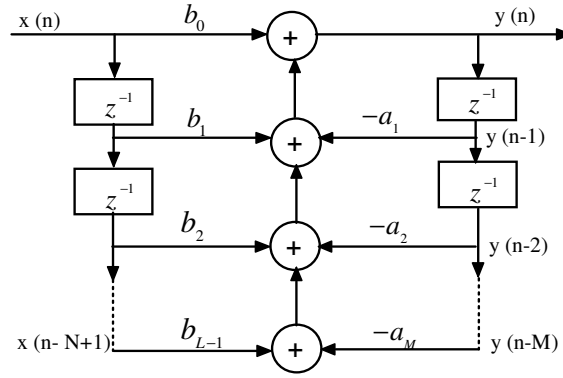


Figura 4.1.2 – Forma direta do Tipo I do filtro IIR.

Este tipo de filtro pode ser interpretado como a cascata de duas funções de transferência onde  $H_1(z) = N(z)$  e  $H_2(z) = \frac{1}{D(z)}$ , assim:

$$H(z) = H_1(z)H_2(z) \quad (4.7)$$

O que representa um filtro de ordem  $N$  com  $2N$  elementos atrasados, representados por  $z^{-1}$ . Caso o filtro utilizado seja de segunda ordem com  $N = 2$ , o qual possui 4 elementos atrasados, onde,  $H_1(z) = b_0 + b_1(z) + b_2(z)$  e  $H_1(z) = b_0 + b_1z^{-1} + b_2z^{-2}$  e  $H_2(z) = 1/(1 + a_1z^{-1} + a_2z^{-2})$ , obtém-se:

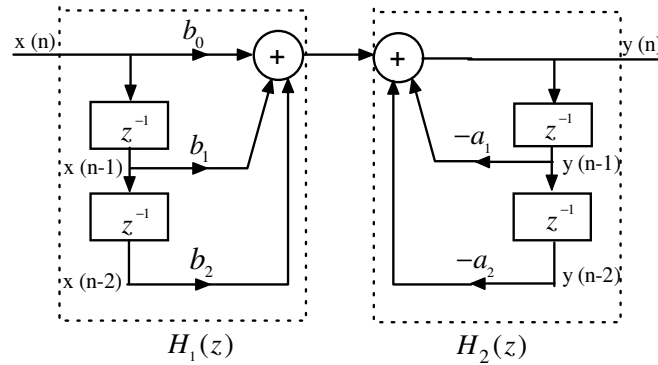


Figura 4.1.3 – Forma direta do Tipo I de segunda ordem.

Como a equação (4.7) é comutativa então a Figura 4.1.3 pode ser representada como:



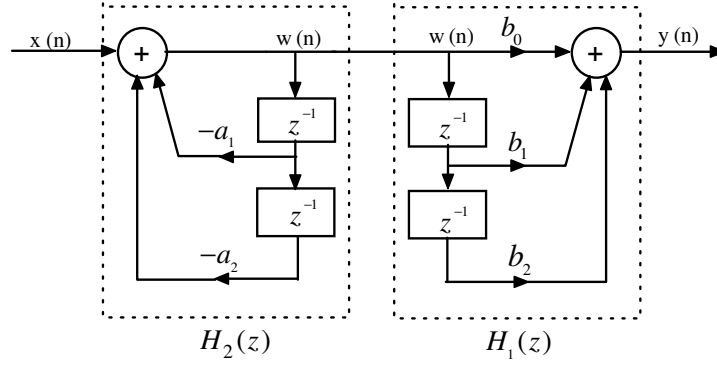


Figura 4.1.4 – Forma direta do Tipo I de segunda ordem  $H(z) = H_2(z)H_1(z)$ .

A existência do sinal intermediário  $w(n)$ , comum entre ambos os sinais de *buffers*  $H_1(z)$  e  $H_2(z)$ , elimina a necessidade da utilização de dois *buffers* separados. Logo, eles podem ser combinados em um, e compartilhado por ambos os filtros, como pode ser verificado abaixo:

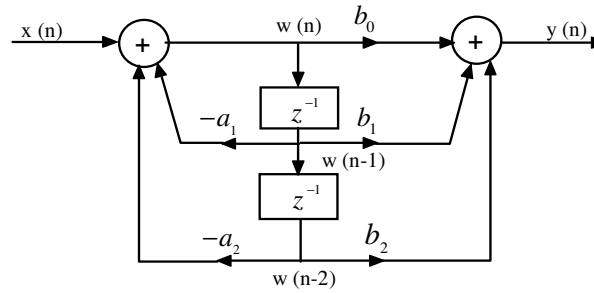


Figura 4.1.5 – Forma direta do Tipo II de segunda ordem.

Verifica-se que para esta realização necessita-se apenas de três locações de memória o que a diferencia da estrutura observada na Figura 4.1.3, afinal para esta seriam requisitadas seis alocações de memória. Este tipo de estrutura é classificado como filtro IIR na forma direta do tipo II. Sua saída  $y(n)$  é representada por [Tretter, 1995]:

$$y(n) = b_0 w(n) + b_1 w(n-1) + b_2 w(n-2) \quad (4.8)$$

Onde:

$$w(n) = x(n) - a_1 w(n-1) - a_2 w(n-2) \quad (4.9)$$

Aplicando a transformada Z em ambas as equações.

$$Y(z) = W(z)(b_0 + b_1 z^{-1} + b_2 z^{-2})$$

$$X(z) = W(z)(1 + a_1 z^{-1} + a_2 z^{-2})$$
(4.10)

A função de transferência é representada por:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$
(4.11)

De forma geral têm-se:

$$w(n) = x(n) - \sum_{m=1}^M a_m w(n-m)$$
(4.12)

$$y(n) = \sum_{k=0}^{N-1} b_k w(n-k)$$
(4.13)

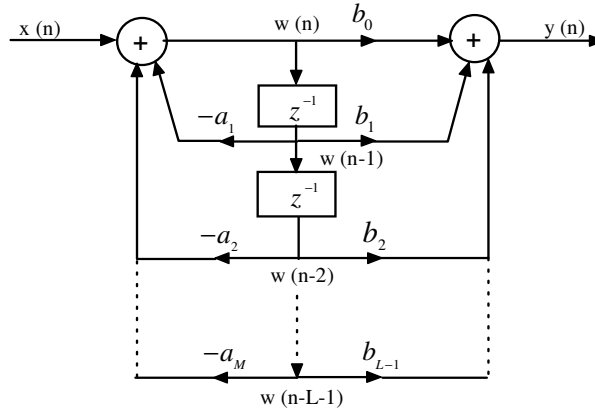


Figura 4.1.6 – Forma direta do tipo II.

A realização em cascata de um filtro IIR é composta pelo produto de funções de transferência simples de primeira ou de segunda ordem. Essas funções são obtidas a partir da fatoração polinomial do numerador e do denominador de uma função de transferência  $H(z)$  qualquer. Considerando a função de transferência  $H(z)$  dada em (4.6), esta pode ser expressa como [Bateman & Yates,1991]:

$$H(z) = CH_1(z)H_2(z)...H_K(z) = C \prod_{k=1}^K H_k(z)$$
(4.14)

Onde  $k$  corresponde ao número total de seções. Cada seção pode ser representada por uma estrutura de forma direta do tipo II.

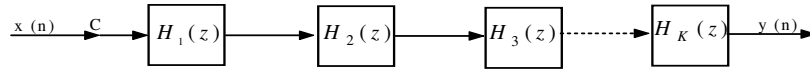


Figura 4.1.7 – Forma cascata de um filtro IIR.

Considerando um filtro IIR de quarta ordem a função de transferência do tipo cascata vai ser da forma [Parks & Burrus, 1987]:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{(b_{01} + b_{11}z^{-1} + b_{21}z^{-2})(b_{02} + b_{12}z^{-1} + b_{22}z^{-2})}{(1 + a_{11}z^{-1} + a_{21}z^{-2})(1 + a_{12}z^{-1} + a_{22}z^{-2})} \quad (4.15)$$

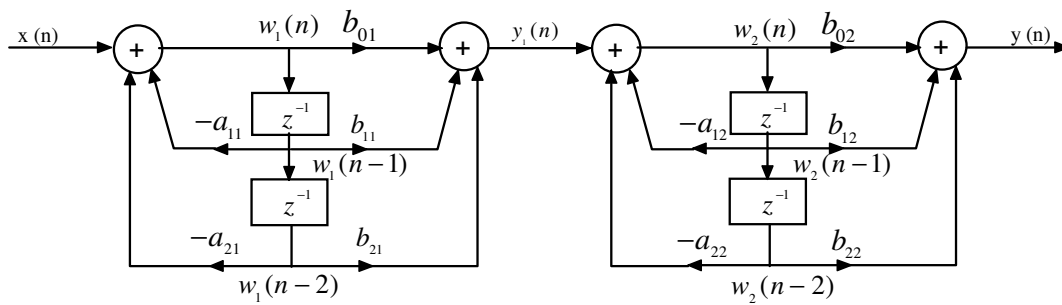


Figura 4.1.8 – Filtro IIR de quarta ordem com duas seções na forma direta do tipo II em cascata.

Diferentes ordens e casamentos, agrupamentos entre zeros e pólos, matematicamente não afeta o resultado da saída  $y(n)$ . Entretanto do ponto de vista prático a disposição de cada seção de segunda ordem pode minimizar o ruído de quantização, afinal a saída da primeira seção torna-se a entrada da segunda. Com o resultado da saída intermediária  $y_1(n)$  armazenado em um dos registradores, o truncamento da saída intermediária torna-se insignificante. Conclui-se que a implementação da estrutura cascata é muito menos sensível a erros de quantização de coeficientes do que a estrutura de forma direta do tipo II.

Outra estrutura bastante utilizada em filtros digitais do tipo IIR é a forma paralela cuja função de transferência é obtida a partir da decomposição em frações parciais de (4.6), sendo representada por:

$$H(z) = C + H_1(z) + H_2(z) + \dots + H_K(z) \quad (4.16)$$

Cada função de transferência presente na forma paralela pode ser de primeira ou de segunda ordem.

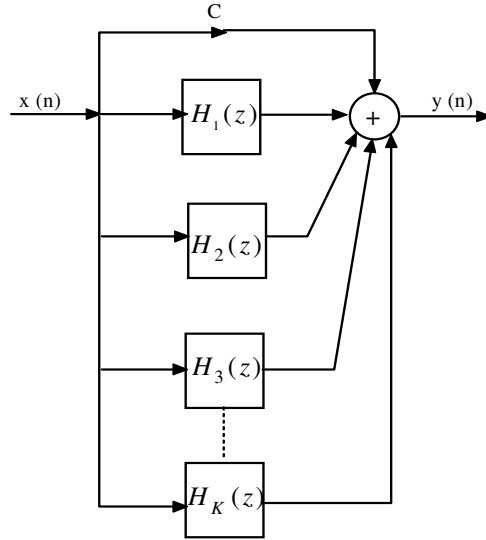


Figura 4.1.9 – Forma paralela de um filtro IIR.

## 4.2 Transformação Bilinear

A transformação bilinear é uma técnica de mapeamento, ou transformação, ponto a ponto do plano  $s$  analógico para o plano  $z$ , convertendo, assim, os pólos e zeros analógicos em digitais, conservando a magnitude da resposta em frequência do filtro analógico, uma vez que não ocorre o *aliasing*, e introduzindo uma distorção na fase. Esta transformação é definida por:

$$s = K \frac{z-1}{z+1} \quad (4.17)$$

Para manter a mesma ordem dos sistemas tanto no sistema analógico quanto no digital é necessário que os polinômios sejam de primeiro grau, como pode ser observado em (4.17). A constante  $K$  é normalmente determinada como  $K = 2/T$ , onde  $T$  representa uma variável amostrada. Ao escolher, por conveniência,  $T = 2$ , a transformação bilinear fica reduzida a [Williams, 1986]:

$$s = \frac{z-1}{z+1} \quad (4.18)$$

Logo a transformação permite que:

- A região a esquerda no plano  $s$ , o que corresponde a  $\sigma < 0$ , esteja dentro do círculo unitário no plano  $z$ .
- A região a direita no plano  $s$ , o que corresponde a  $\sigma > 0$ , esteja fora do círculo unitário no plano  $z$ .

- O eixo imaginário  $j\omega$  no plano  $s$  encontra-se sobre o círculo unitário no plano  $z$ .

Um filtro analógico estável deve ser transformado em um filtro digital estável. Logo verifica-se que o método da transformada bilinear consiste apenas em mapear a metade esquerda do plano  $s$ .

Se  $\omega_A$  e  $\omega_D$  representam respectivamente as frequências analógica e digital, então  $s = j\omega_A$  e  $z = e^{j\omega_D T}$ . Substituindo em (4.18):

$$j\omega_A = \frac{e^{j\omega_D T} - 1}{e^{j\omega_D T} + 1} = \frac{e^{j\omega_D T/2} (e^{j\omega_D T/2} - e^{-j\omega_D T/2})}{e^{j\omega_D T/2} (e^{j\omega_D T/2} + e^{-j\omega_D T/2})} \quad (4.19)$$

Acarretando em:

$$\omega_A = \tan \frac{\omega_D T}{2} \quad (4.20)$$

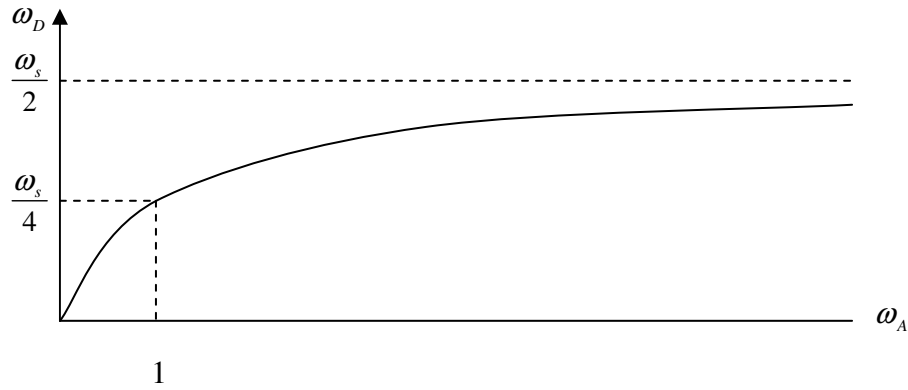


Figura 4.2.1. – Relação entre frequências analógicas e digitais.

Observa-se na figura acima que a partir de  $\omega_A > 1$ , a qual corresponde a região entre  $\omega_s/4$  e  $\omega_s/2$  para  $\omega_D$ , onde  $\omega_s$  é a frequência amostrada em radianos, a região é considerada completamente não-linear. Esta não-linearidade corresponde a uma grande distorção no módulo ou na magnitude na frequência do filtro digital, também conhecida como efeito *warping*.

## 4.3 EXEMPLOS DE FILTROS COM RESPOSTA IMPULSIONAL INFINITA (IIR)

Dois exemplos utilizando a forma direta do tipo II em cascata foram desenvolvidos para ilustrar a implementação de filtros IIR.

### 4.3.1 Exemplo 1: Filtro Passa-Baixa

Esse exemplo analisa as características do filtro IIR do tipo passa-baixa através da implementação do código *IIR.c* e da utilização de alguns recursos do *CCS*.

```
//IIR.c
//Filtro IIR usando forma direta do tipo II em cascata

#include "lp2000.cof"           // arquivo de coeficiente LP com frequência de 2000Hz
short dly[stages][2] = {0};    // amostras atrasadas por estágio
interrupt void c_int11()        // ISR
{
    int i, input;
    int wn, yn;

    input = input_sample();      // entrada do primeiro estágio
    for (i = 0; i < stages; i++) // repete para cada estágio
    {
        wn=input-((a[i][0]*dly[i][0])>>15) - ((a[i][1]*dly[i][1])>>15);

        yn=((b[i][0]*un)>>15)+((b[i][1]*dly[i][0])>>15)+((b[i][2]*dly[i][1])>>15);

        dly[i][1] = dly[i][0]; // atualização dos atrasos
        dly[i][0] = wn;        // atualização dos atrasos
        input = yn;            // saída intermediária->entrada para o próximo estágio
    }
    output_sample(yn);          // resultado da saída final para um tempo n
    return;                     // retorna para ISR
}
void main()
{
    comm_intr();                // inicializa o DSK, codec, McBSP
    while(1);                   // loop infinito
}
```

//lp2000.cof Coeficientes do filtro IIR passa-baixa com frequência de corte de 2 kHz

```
#define stages 4                // número de estágios de segunda ordem

int a[stages][3] =              { //coeficientes do numerador
{ 304, 608, 304},              //b10, b11, b12 para o primeiro estágio
{ 32768, 66530, 33778},        //b20, b21, b22 para o segundo estágio
```

```

{32768, 65518, 32765},          //b30, b31, b32 para o terceiro estágio
{32768, 64542, 31788} };       //b40, b41, b42 para o quarto estágio

int b[stages][2] =               { //coeficientes do denominador
{0, 318},                       //a11, a12 para o primeiro estágio
{0, 3015},                      //a21, a22 para o segundo estágio
{0, 9362},                      //a31, a32 para o terceiro estágio
{0, 22070} };                   //a41, a42 para o quarto estágio

```

## Considerações sobre o programa

O arquivo *lp2000.cof* incluso no código *IIR.c* contém os coeficientes de um filtro IIR de oitava ordem.

O programa *IIR.c* implementa um filtro genérico usando, em cascata, seções de segunda ordem. Logo, se observa a associação de duas equações em cada seção [Lynn & Fuerst, 1994]:

$$w(n) = x(n) - a_1 w(n-1) - a_2 w(n-2)$$

$$y(n) = b_0 w(n) + b_1 w(n-1) + b_2 w(n-2)$$

O *loop* que há dentro do código do programa é processado quatro vezes (número de seções) para cada valor de  $n$ , ou período de amostragem. Para o primeiro estágio,  $x(n)$  é a entrada amostrada recém adquirida. Assim, a saída do estágio  $y(n)$  será a entrada  $x(n)$  da próxima seção.

Os coeficientes  $a[i][0]$  e  $a[i][1]$  corresponde a  $a_1$  e  $a_2$  respectivamente, onde  $i$  represente cada estágio. Os atrasos  $dly[i][0]$  e  $dly[i][1]$  correspondem a  $w(n-1)$  e  $w(n-2)$  respectivamente.

## Criação do projeto

Para criar este projeto no *Code Composer Studio™ IDE*, é preciso adicionar os arquivos necessários à construção do projeto IIR, que se encontra no material fornecido, como fora feito nos exemplos do tutorial 2.

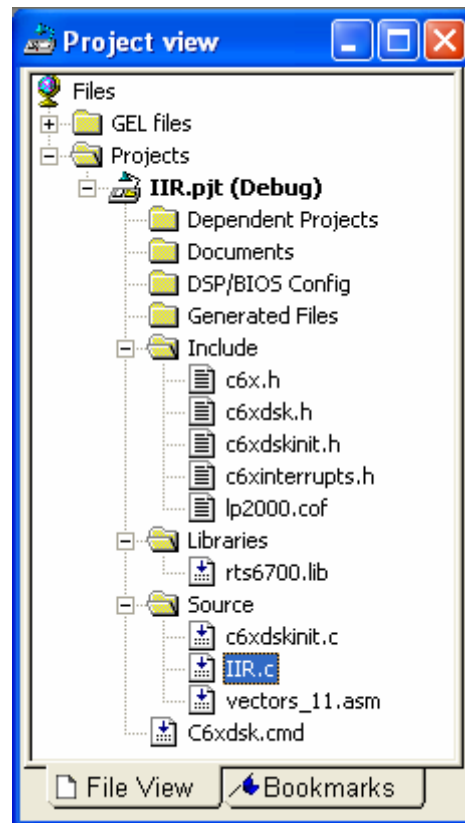


Figura 4.3.1.1 – Visualização do projeto.

Para gerar o executável e carregar o programa utilizam-se as mesmas configurações dos parâmetros do compilador e do *linker* como fora feito no tutorial 2.

## Resultados Experimentais

Ao se implementar no conector IN (J7) da placa do DSK o ruído pseudo-randômico, *ruído.m*, anexo I, como entrada deste filtro IIR passa-baixa com frequência de corte igual a 2000Hz obteve-se com o auxílio do Osciloscópio – Agilent 5AG21A, que se encontra conectado na saída da placa do DSK, conector OUT (J6), a resposta apresentada na Figura 4.3.1.2..

A Figura 4.3.1.3 representa a resposta do filtro construída por meio de interpolação cujo código fonte, que fora implementado na ferramenta MATLAB 7.0, encontra-se no anexo I.



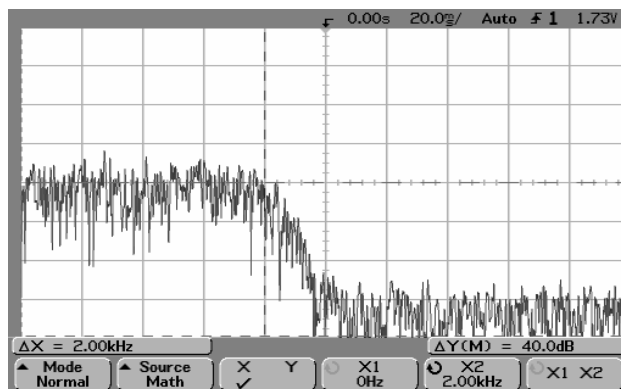


Figura 4.3.1.2 – Resposta em frequência da saída do filtro IIR passa-baixa com frequência de corte de 2000Hz, obtida no osciloscópio.

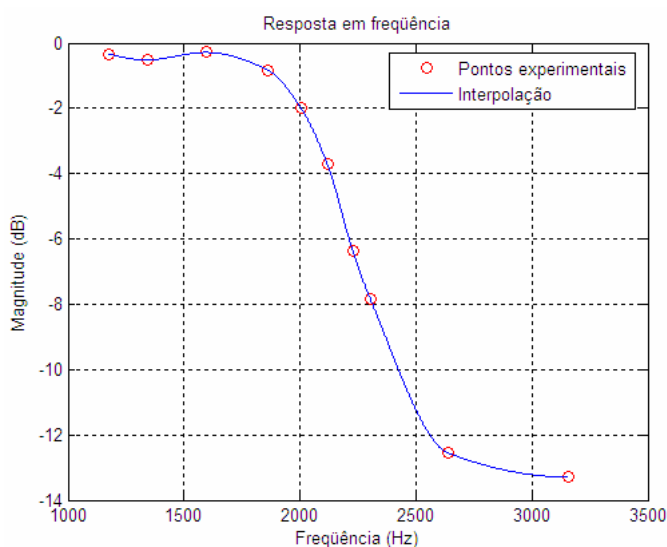


Figura 4.3.1.3 – Resposta em frequência da saída do filtro IIR passa-baixa com frequência de corte de 2000Hz, obtida por meio da interpolação.

## 4.3.2 Exemplo 2: Filtro Passa-Faixa

Esse exemplo analisa as características do filtro IIR do tipo passa-faixa com frequência centrada em 2000Hz a partir da modificação do arquivo dos coeficientes incluso no código *IIR.c*, ou seja, troca-se *lp2000.cof* por *bp2000.cof* como fora feito no exemplo 2 do tutorial 2.

// lp2000.cof Coeficientes do filtro IIR passa-faixa com frequência centrada em 2000Hz

// Os coeficientes a's e b's do MATLAB são os b's e a's utilizados aqui

```
#define stages 18      // número de estágios de segunda ordem
int a[stages][3]=      {      // coeficientes do numerador

    { 1, -3, 1},          // b10, b11, b12 para o primeiro estágio
    {32768, 63298, 32768}, // b20, b21, b22 para o segundo estágio
    {32768, -51261, 32768}, // b30, b31, b32 para o terceiro estágio
    {32768, 51261, 32768}, //.....
    {32768, -39945, 32768},
    {32768, 39945, 32768},
    {32768, -32302, 32768},
    {32768, 32302, 32768},
    {32768, 27364, 32768},
    {32768, -27364, 32768},
    {32768, 24164, 32768},
    {32768, -24164, 32768},
    {32768, -22117, 32768},
    {32768, 22117, 32768},
    {32768, 20895, 32768},
    {32768, -20895, 32768},
    {32768, -20322, 32768}, // b170,b171,b172 para o 17th estágio
    {32768, 20322, 32768} }; // b180,b181,b182 para o 18th estágio

int b[stages][2]=      { // coeficientes do denominador
    {-3196, 13135},      // a11, a12 para o primeiro estágio
    {3196, 13135},       // a21, a22 para o segundo estágio
    {-8611, 15257},      //...
    {8611, 15257},
    {-12148, 18330},
    {12148, 18330},
    {-14126, 21429},
    {14126, 21429},
    {15125, 24179},
    {-15125, 24179},
    {15602, 26520},
    {-15602, 26520},
    {-15855, 28520},
    {15855, 28520},
    {16068, 30287},
    {-16068, 30287},
    {-16350, 31943},
    {16350, 31943}      }; // a181,a182 para 18th estágio
```

O arquivo *bp2000.cof* representa dezoito seções, ou seja, 36th- ordem.

## Criação do projeto

Após a modificação proposta no código *IIR.c* cria-se o seguinte projeto;

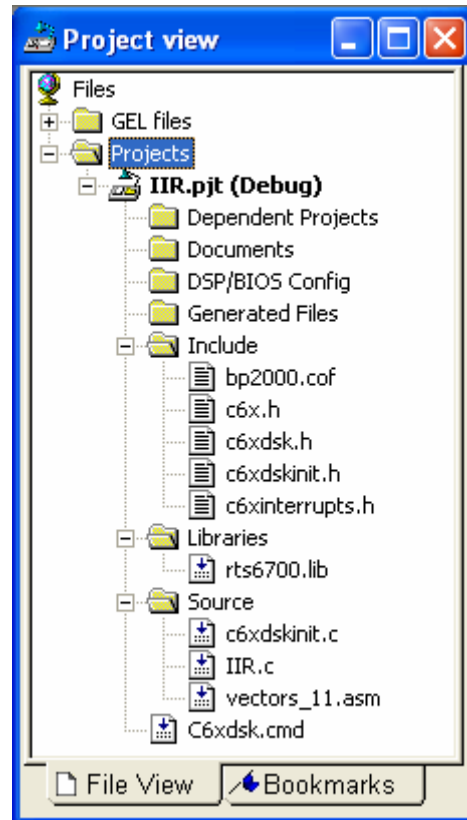


Figura 4.3.2.1 – Visualização do projeto.

Para gerar o executável e carregar o programa utilizam-se as mesmas configurações dos parâmetros do compilador e do *linker* como fora feito no exemplo anterior.

## Resultados Experimentais

A Figura 4.3.2.2 mostra a resposta em frequência do filtro obtida com o auxílio do Osciloscópio – Agilent 5AG21A.

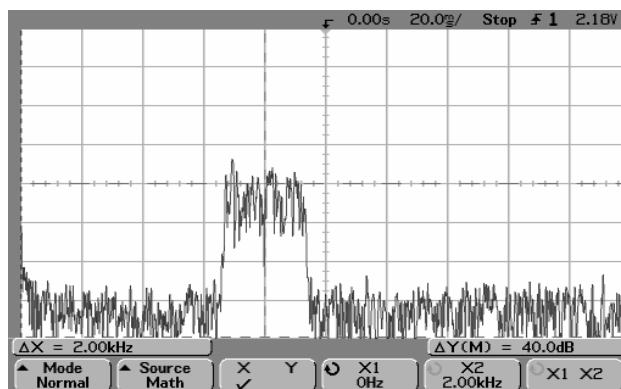


Figura 4.3.2.2 – Resposta em frequência da saída do filtro IIR passa-faixa centrado em 2000Hz, obtida no osciloscópio.

### Construção gráfica no Matlab

Utilize a ferramenta *sptool*, e determine as seguintes características do filtro IIR passa-faixa na janela *Filter Designer*:

- *Sampling frequency* → 8000.
- *Algorithm* → *Chebyshev Type II IIR*.
- *Order* → 18.
- *Fc1* → 1600 e *Fc2* → 2400, determinando a faixa onde existe uma probabilidade do filtro estar centrado.
- *Rs* → 101.5

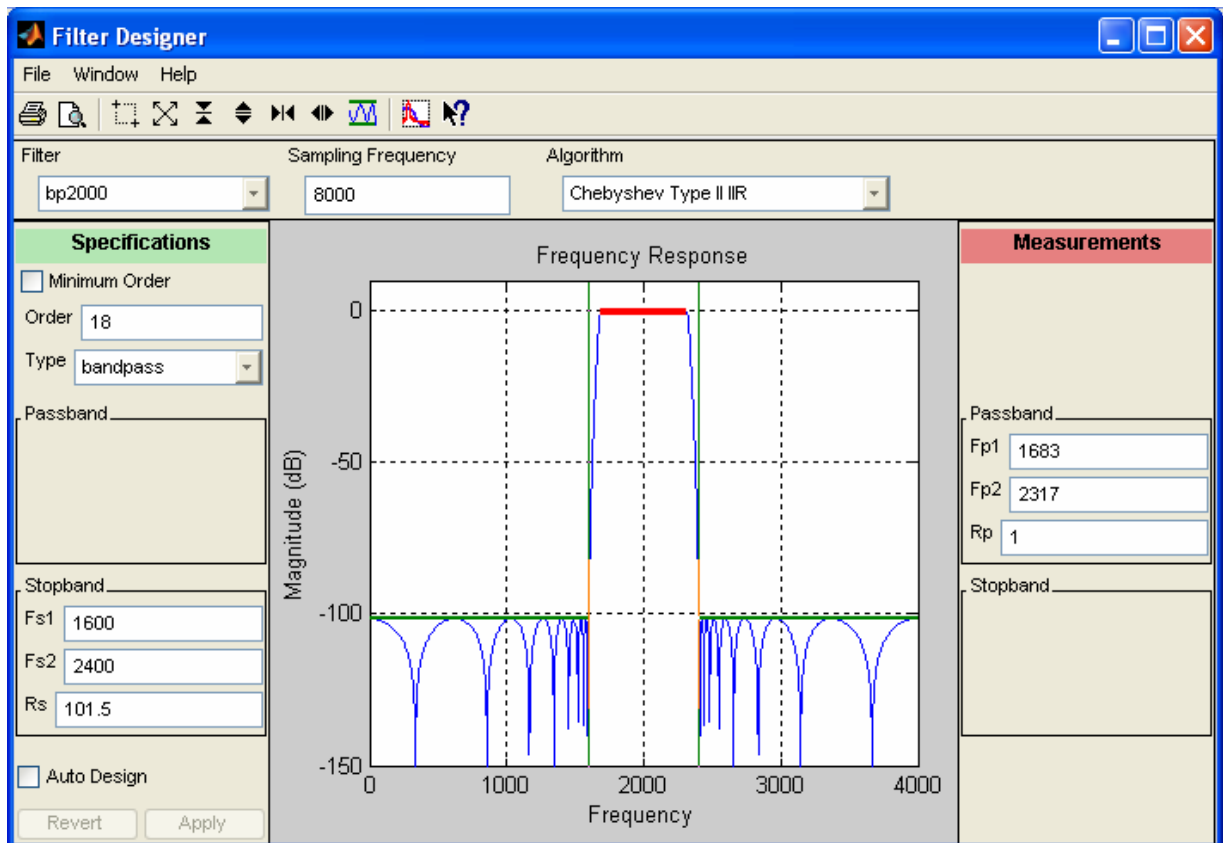


Figura 4.3.2.3 – Características do filtro IIR passa-faixa centrado em 2000Hz.

Exporte o arquivo *Filter:bp2000 [design]* para a área de trabalho do Matlab e verifique no mesmo, através dos comandos abaixo, a obtenção dos coeficientes contidos no arquivo *bp2000.cof*.

```
>> [z,p,k]=tf2zp (bp2000.tf.num, bp2000.tf.den);
>> sec_ord_sec=zp2sos(z,p,k);
>> sec_ord_sec=round(sec_ord_sec*2^15)
```

O primeiro comando acha as raízes do numerador e do denominador, ou seja, os pólos e os zeros, e converte-os em um formato de seções de segunda ordem para a implementação.

Logo se observa que os coeficientes, escalados por  $2^{15}$ , do filtro IIR passa-faixa listados dentro da área de trabalho do Matlab são os mesmos apresentados no arquivo de coeficientes *bp2000.cof*.

## 5 - TUTORIAL 4: TRANSFORMADA RÁPIDA DE FOURIER (FFT)

### 5.1 OBJETIVO

- Introdução aos conceitos da Transformada de Fourier Discreta
- Introdução à transformada rápida de Fourier usando raiz 2 e raiz 4.
- Decimação na frequência e no tempo.
- Programação de exemplos usando código C e o TMS320C6711 DSK

Esse tutorial tem por finalidade apresentar os conceitos sobre a transformada rápida de Fourier (FFT), um algoritmo muito eficiente que é utilizado para converter um sinal no domínio do tempo em um sinal equivalente no domínio da frequência, baseado na transformada discreta de Fourier (DFT). Projetar dois exemplos no TMS320C6711 DSK.

### 5.2 INTRODUÇÃO

#### 5.2.1 Transformada Discreta de Fourier (DFT)

A transformada discreta de Fourier (DFT) converte uma sequência discreta no domínio do tempo numa sequência equivalente no domínio da frequência.

Para uma sequência complexa  $x(n)$  de tamanho  $N$  a transformada de Fourier Discreta é representada por:

$$X(w_k) \triangleq \sum_{n=0}^{N-1} x(n)e^{-jw_k t_n} = \sum_{n=0}^{N-1} x(n)e^{-j2\pi kn/N}, \quad k=0,1,2,\dots,N-1 \quad (5.1)$$

Onde:

$$t_n \triangleq nT = \text{n-ésimo instante de amostragem (s)}$$

$$w_k \triangleq k\Omega = \text{k-ésima amostra de frequência (rad/s)}$$

$$T \triangleq 1/f_s = \text{intervalo de amostragem do tempo (s)}$$

$$\Omega \triangleq 2\pi f_s/N = \text{intervalo de amostragem da frequência (rad/s)}$$

Logo  $X(w_k)$  é a  $k$ -ésima amostra do espectro na frequência  $w_k$ . Então a  $k$ -ésima amostra de  $X(w_k)$  do espectro de  $x$  é definido como produto interno de  $x$  com a  $k$ -ésima senóide DFT.

Ao considerar o fator  $W_N = e^{-j2\pi/N}$ , o qual representa a fase, a equação (5.1) pode ser escrita da seguinte forma:

$$X(w_k) \triangleq \sum_{n=0}^{N-1} x(n)W_N^{kn}, \quad k = 0, 1, \dots, N-1 \quad (5.2)$$

Logo:

$$X(w_k) = x(0) + x(1)W_N^k + x(2)W_N^{2k} + \dots + x(N-1)W_N^{(N-1)k} \quad (5.3)$$

A Equação 5.3 mostra que é possível calcular os  $k$  elementos  $X(w_k)$  por meio do produto entre uma matriz  $N \times N$  e um vetor  $N \times 1$ , conforme mostra a Equação 5.4.

$$\begin{bmatrix} X(w_0) \\ X(w_1) \\ X(w_2) \\ \vdots \\ X(w_{N-1}) \end{bmatrix} = \begin{bmatrix} W_N^0 & W_N^0 & \dots & W_N^0 \\ W_N^0 & W_N^1 & \dots & W_N^{(N-1)} \\ W_N^0 & W_N^2 & \dots & W_N^{2(N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ W_N^0 & W_N^{(N-1)} & \dots & W_N^{(N-1)^2} \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ \vdots \\ x(N-1) \end{bmatrix} \quad (5.4)$$

Cada valor complexo de  $x(n)$  em (5.3), é composto por  $N$  multiplicações complexas e  $(N-1)$  adições complexas, para calcular todos os  $N$  valores da DFT são necessárias  $N^2$  multiplicações complexas e  $N(N-1)$  adições complexas.

Caso a propriedade distributiva seja aplicada, a Equação 5.2 poder ser expressa em termos de operações reais, obtendo-se:

$$X(w_k) = \sum_{n=0}^{N-1} [(Re\{x(n)\}Re\{W_N^{kn}\} - Im\{x(n)\}Im\{W_N^{kn}\}) + j(Re\{x(n)\}Im\{W_N^{kn}\} + Im\{x(n)\}Re\{W_N^{kn}\})], \quad k=0, 1, \dots, N-1 \quad (5.5)$$

Onde uma multiplicação complexa é composta por quatro multiplicações reais e duas adições reais, e cada adição complexa possui duas adições reais. Tem-se para cada valor de  $k$ ,  $4N$  multiplicações reais e  $(4N-2)$  adições reais. Conseqüentemente, para  $N$  valores a transformada discreta de Fourier de  $x(n)$  requer  $4N^2$  multiplicações reais e  $N(4N-2)$  adições reais.

Para reduzir esta quantidade de operações em 1965, Cooley e Tukey, propuseram uma eficiente técnica algorítmica denominada transformada rápida de Fourier (FFT) baseada na transformada discreta de Fourier mas que requer bem menos cálculos, já que a

FFT reduz a complexidade dos cálculos de  $N^2$  para  $N \log_2 N$ . A Figura 5.2.1.1 mostra o comportamento das funções  $f(N) = N^2$  e  $f(N) = N \log_2 N$  a medida que  $N$  cresce [Mitra & Kaiser, 1993].

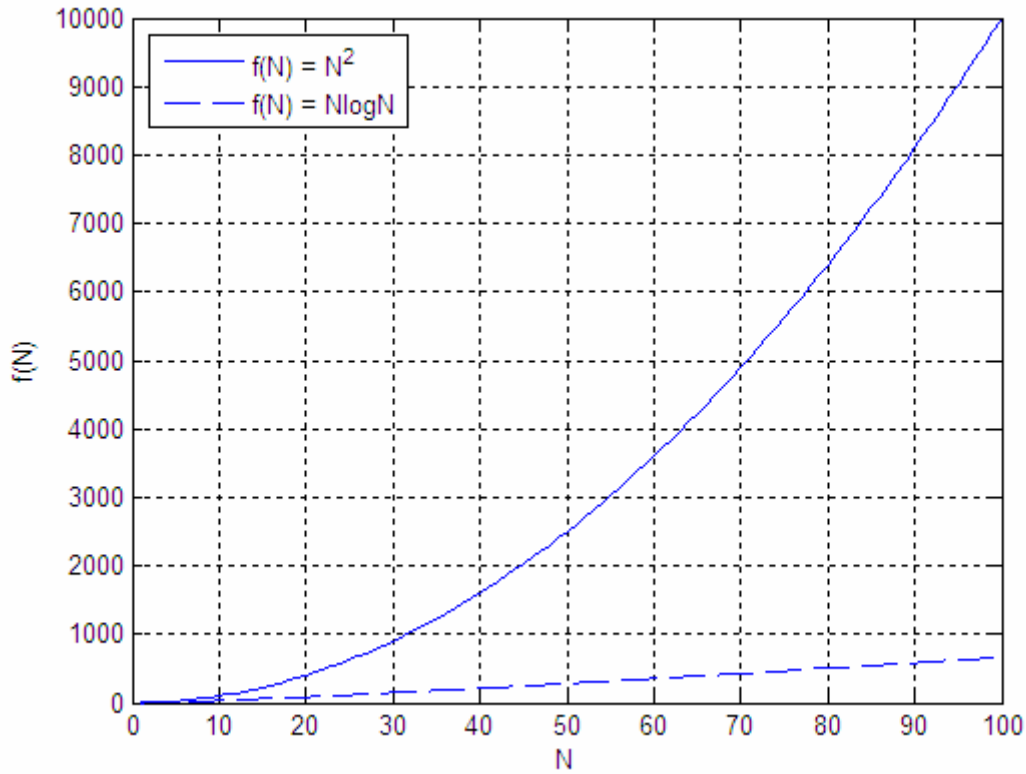


Figura 5.2.1.1 – Comparação entre as funções  $f(N) = N^2$  e  $f(N) = N \log_2 N$ .

A redução da complexidade da FFT requisita a utilização da simetria e da periodicidade, propriedades de  $W_N^{kn}$ .

A periodicidade de  $W$  é determinada como:

$$W_N^{k+N} = W_N^k \quad (5.6)$$

A simetria de  $W$  expressa-se por meio de:

$$W_N^{k+N/2} = -W_N^k \quad (5.7)$$

Considerando  $N = 8$  e  $k = 2$  aplicando a propriedade da periodicidade  $W^{10} = W^2$  e da simetria  $W^6 = -W^2$ , obtém-se o diagrama da Figura 5.2.1.2.



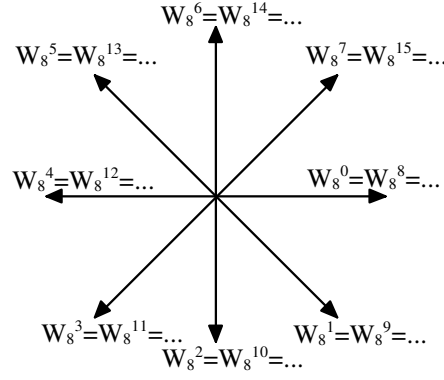


Figura 5.2.1.2 – Visualização das propriedades de simetria e de periodicidade.

Na base 2, a FFT decompõem a DFT de  $N$  pontos em duas DFT's menores compostas por  $N/2$  pontos cada.

Logo para otimizar o processamento de uma DFT existem vários tipos de algoritmos rápidos cujo conjunto é conhecido como algoritmos de FFT.

## 5.2.2 Desenvolvimento do algoritmo raiz-2 com decimação no tempo

Decimação no tempo é o processo no qual a sequência de entrada  $x(n)$  é decomposta em seqüências menores.

Supondo que o comprimento  $N$  da sequência  $x(n)$  seja uma potência de dois,  $N = 2^l$ , onde  $l$  representa o número de estágios, pode-se decompor  $x(n)$  em duas outras seqüências menores, uma com os elementos de  $x(n)$  para  $n$  par e outra com elementos  $x(n)$  para  $n$  ímpar, conforme mostra a Equação 5.8 [Mitra,1998].

$$X(w_k) = \sum_{n=0}^{(N/2)-1} x(2n)W_N^{2nk} + \sum_{n=0}^{(N/2)-1} x(2n+1)W_N^{(2n+1)k} \quad (5.8)$$

Considerando  $W_N^2 = W_{N/2}$ ,

$$W_N^2 = e^{-j2\pi/N} = e^{-j2\pi/(N/2)} = W_{N/2} \quad (5.9)$$

Substituindo (5.9) em (5.8),

$$X(w_k) = \sum_{n=0}^{(N/2)-1} x(2n)W_{N/2}^{nk} + W_N^k \sum_{n=0}^{(N/2)-1} x(2n+1)W_{N/2}^{nk} \quad (5.10)$$

Onde:

$$\begin{aligned} C(w_k) &= \sum_{n=0}^{(N/2)-1} x(2n)W_{N/2}^{nk} \\ D(w_k) &= \sum_{n=0}^{(N/2)-1} x(2n+1)W_{N/2}^{nk} \end{aligned} \quad (5.11)$$

Desta forma:

$$X(w_k) = C(w_k) + W_N^k D(w_k) \quad (5.12)$$

A Equação 5.12 representa duas DFTs com tamanho  $N/2$ , acarretando em apenas  $(N/2)^2$  multiplicações complexas para cada somatório. Entretanto a seqüência composta por elementos de índice ímpar requer também  $N$  multiplicações por  $W_N^k$ . Assim a Equação 5.12 implica em  $2(N/2)^2 + N = (N^2/2) + N$  multiplicações complexas. Após estas multiplicações são realizadas  $2[(N/2)^2 - N/2] + N = N^2/2$  adições. Estas características são aplicadas de uma forma recursiva em cada uma das DFTs até que obtenham-se DFTs de comprimento 1, o que reduz de forma bastante significativa a complexidade dos cálculos [Johnson, 1989].

Ao considerar  $w_k > (N/2) - 1$  e utilizando a propriedade de simetria (Equação 5.7), a Equação 5.12 resulta em,

$$X(w_{k+N/2}) = C(w_k) - W_N^k D(w_k) \quad (5.13)$$

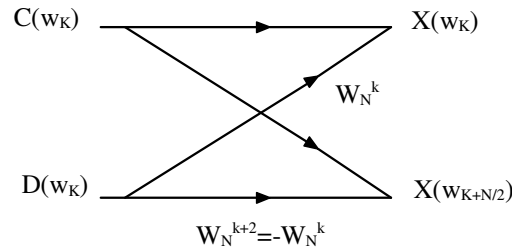


Figura 5.2.2.1 – Célula Básica do algoritmo de FFT com decimação no tempo.

Este tipo de célula básica, também conhecida como borboleta, neste caso, utiliza uma única multiplicação complexa, resultando num algoritmo com  $\frac{N}{2} \log_2 N$  multiplicações complexas.

Logo para  $N = 8$  tem-se:

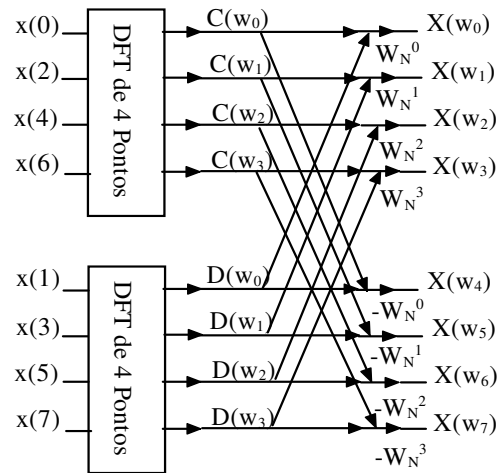


Figura 5.2.2.2 – Decomposição de uma DFT com 8 pontos em duas DFTs de 4 pontos com decimação no tempo.

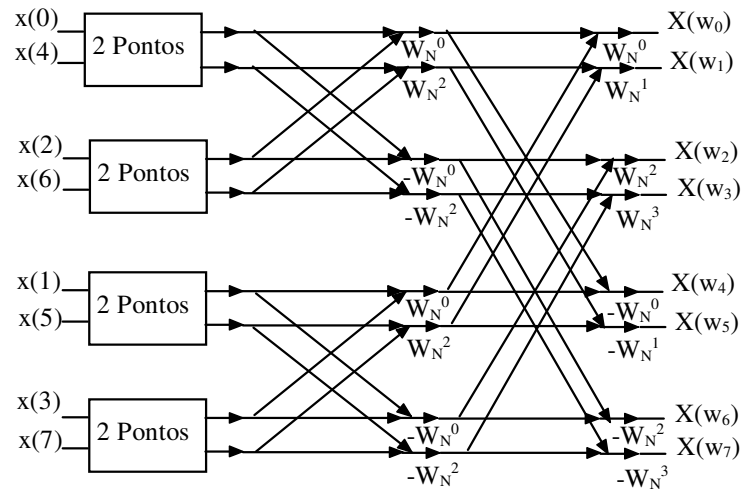


Figura 5.2.2.3 – Decomposição de duas DFTs de 4 pontos em quatro DFTs de 2 pontos com decimação no tempo.

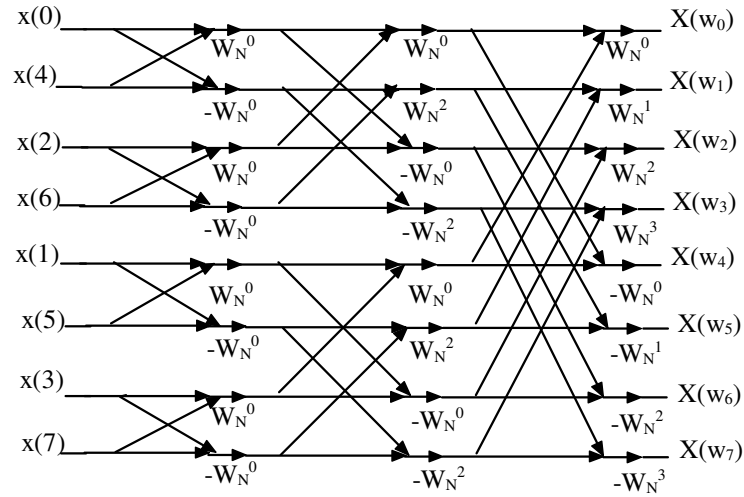


Figura 5.2.2.4 – FFT de oito pontos usando decimação no tempo.

### 5.2.3 Desenvolvimento do algoritmo raiz-2 com decimação na frequência

O algoritmo raiz de 2 com decimação na frequência corresponde a divisão da sequência de saída  $X(w_k)$  em sequências cada vez menores.

Ao dividir a sequência  $x(n)$  da Equação (5.2) em duas partes iguais obtém-se:

$$X(w_k) = \sum_{n=0}^{(N/2)-1} x(n)W_N^{nk} + \sum_{n=N/2}^{N-1} x(n)W_N^{nk} \quad (5.14)$$

Considerando  $n = n + N/2$  no segundo somatório,

$$\begin{aligned} X(w_k) &= \sum_{n=0}^{(N/2)-1} x(n)W_N^{nk} + \sum_{n=0}^{(N/2)-1} x(n + N/2)W_N^{nk}W_N^{kN/2} \\ &= \sum_{n=0}^{(N/2)-1} x(n)W_N^{nk} + (W_N^{kN/2}) \sum_{n=0}^{(N/2)-1} x(n + N/2)W_N^{nk} \end{aligned} \quad (5.15)$$

Como  $W_N^{kN/2} = (-1)^k$  a equação (5.15) reduz-se a;

$$X(w_k) = \sum_{n=0}^{(N/2)-1} \left[ x(n) + (-1)^k x(n + \frac{N}{2}) \right] W_N^{nk} \quad (5.16)$$

Separando os termos pares e ímpares de  $X(w_k)$  obtém-se:

$$X(w_{2k}) = \sum_{n=0}^{(N/2)-1} \left[ x(n) + x(n + \frac{N}{2}) \right] W_N^{2nk} \quad (5.17)$$

$$X(w_{2k+1}) = \sum_{n=0}^{(N/2)-1} \left[ x(n) - x(n + \frac{N}{2}) \right] W_N^n W_N^{2nk} \quad (5.18)$$

Para  $k = 0, 1, 2, \dots, (N/2) - 1$ .

Ao admitir que  $W_N^2 = W_{N/2}$  e considerando,

$$\begin{aligned} g(n) &= x(n) + x(n + N/2) \\ h(n) &= x(n) - x(n - N/2) \end{aligned} \quad (5.19)$$

As Equações (5.17) e (5.18) podem ser escritas como DFTs de  $(N/2)$  pontos,

$$X(w_{2k}) = \sum_{n=0}^{(N/2)-1} g(n) W_{N/2}^{nk} \quad (5.20)$$

$$X(w_{2k+1}) = \sum_{n=0}^{(N/2)-1} h(n) W_N^n W_{N/2}^{nk} \quad (5.21)$$

Primeiramente são calculadas as seqüências  $g(n)$  e  $h(n)$ , então é calculado  $h(n)W_N^n$ , e finalmente calculam-se os  $(N/2)$  pontos, acarretando na obtenção das saídas dos termos pares e ímpares respectivamente.

Sendo considerado  $a(n) = g(n)$  e  $b(n) = h(n)W_N^n$  a célula básica da decimação na frequência é representada por:

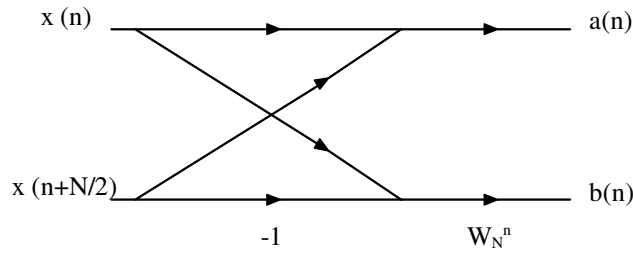


Figura 5.2.3.1 – Célula Básica do algoritmo de FFT com decimação na frequência.

Este tipo de procedimento pode ser repetido para cada uma das DFTs de  $(N/2)$  pontos, gerando DFTs de comprimento  $(N/4)$ , e este gerando DFTs de comprimento  $(N/8)$ , e assim por diante.

Logo para  $N = 8$ , tem-se:

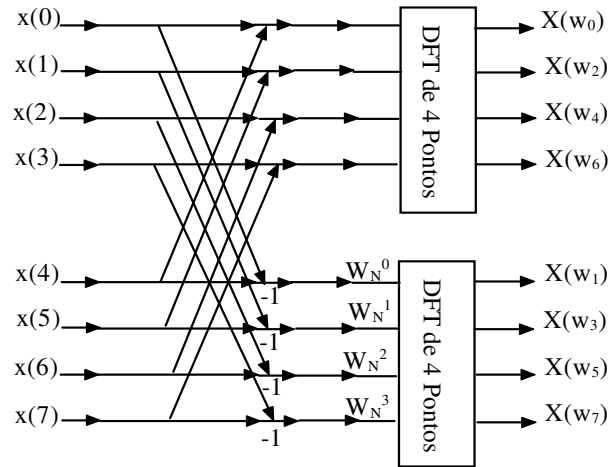


Figura 5.2.3.2 – Decomposição de uma DFT com 8 pontos em duas DFTs de 4 pontos com decimação na frequência.

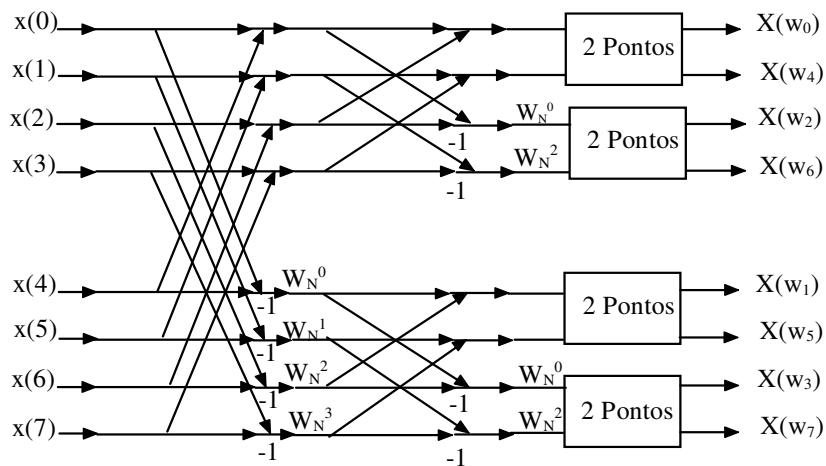


Figura 5.2.3.3 – Decomposição de duas DFTs de 4 pontos em quatro DFTs de 2 pontos com decimação na frequência.

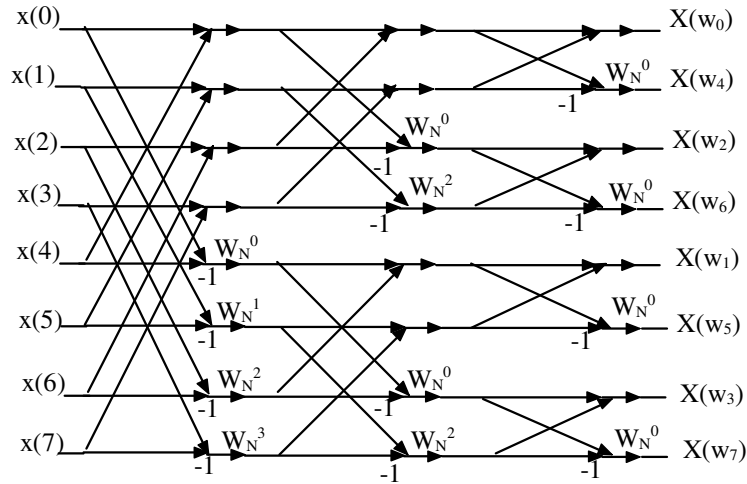


Figura 5.2.3.4 – FFT de oito pontos usando decimação na frequência.

A Figura 5.2.3.4 requer  $(N/2)\log_2 N$  multiplicações complexas e  $N\log_2 N$  adições complexas. Logo o número de cálculos realizados na decimação na frequência corresponde ao da decimação no tempo.

#### 5.2.4 Algoritmo de raiz-4

Quando o algoritmo utilizado for de raiz de 4 cujo comprimento é  $N = 2^{2l}$ , a DFT (5.2) é decomposta em quatro seqüências.

$$X(w_k) = \sum_{n=0}^{(N/4)-1} x(n)W_N^{nk} + \sum_{n=N/4}^{(N/2)-1} x(n)W_N^{nk} + \sum_{n=N/2}^{(3N/4)-1} x(n)W_N^{nk} + \sum_{n=3N/4}^{N-1} x(n)W_N^{nk} \quad (5.22)$$

Considerando  $n = n + N/4$ ,  $n = n + N/2$ ,  $n = n + 3N/4$  no segundo, terceiro e quarto somatório respectivamente obtém-se;

$$\begin{aligned} X(w_k) = & \sum_{n=0}^{(N/4)-1} x(n)W_N^{nk} + W_N^{kN/4} \sum_{n=0}^{(N/4)-1} x(n + N/4)W_N^{nk} \\ & + W_N^{kN/2} \sum_{n=0}^{(N/4)-1} x(n + N/2)W_N^{nk} + W_N^{3kN/4} \sum_{n=0}^{(N/4)-1} x(n + 3N/4)W_N^{nk} \end{aligned} \quad (5.23)$$

Onde,

$$\begin{aligned} W^{kN/4} &= (-j)^k \\ W^{kN/2} &= (-1)^k \\ W^{3kN/4} &= (j)^k \end{aligned} \quad (5.24)$$

Assim (5.23) fica reduzida a,

$$X(w_k) = \sum_{n=0}^{(N/4)-1} [x(n) + (-j)^k x(n + N/4) + (-1)^k x(n + N/2) + (j)^k x(n + 3N/4)] W_N^{nk} \quad (5.25)$$

Uma vez que  $W_N^4 = W_{N/4}$ , (5.25) pode ser escrita como;

$$\begin{aligned} X(w_{4k}) &= \sum_{n=0}^{(N/4)-1} [x(n) + x(n + N/4) + x(n + N/2) + x(n + 3N/4)] W_{N/4}^{nk} \\ X(w_{4k+1}) &= \sum_{n=0}^{(N/4)-1} [x(n) - jx(n + N/4) - x(n + N/2) + jx(n + 3N/4)] W_N^n W_{N/4}^{nk} \\ X(w_{4k+2}) &= \sum_{n=0}^{(N/4)-1} [x(n) - x(n + N/4) + x(n + N/2) - x(n + 3N/4)] W_N^{2n} W_{N/4}^{nk} \\ X(w_{4k+3}) &= \sum_{n=0}^{(N/4)-1} [x(n) + jx(n + N/4) - x(n + N/2) - jx(n + 3N/4)] W_N^{3n} W_{N/4}^{nk} \end{aligned} \quad (5.26)$$

Verifica-se que quanto menor o comprimento da DFT da célula básica de um algoritmo de FFT, mais otimizado ele torna-se, exceto para algoritmos de raízes múltiplos de dois, onde se pode obter um número de multiplicações progressivamente inferior ao dos algoritmos de raiz de 2.

## 5.2.5 Transformada Inversa Discreta de Fourier (IDFT)

A transformada inversa discreta de Fourier converte a sequência no domínio da frequência numa frequência equivalente no domínio do tempo [Bellanger, 1989].

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[w_k] e^{j2\pi kn/N} \quad (5.27)$$

$$\begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ \vdots \\ x(N-1) \end{bmatrix} = \frac{1}{N} \begin{bmatrix} W_N^0 & W_N^0 & \dots & W_N^0 \\ W_N^0 & W_N^{-1} & \dots & W_N^{-(N-1)} \\ W_N^0 & W_N^{-2} & \dots & W_N^{-2(N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ W_N^0 & W_N^{-(N-1)} & \dots & W_N^{-(N-1)^2} \end{bmatrix} \begin{bmatrix} X(w_0) \\ X(w_1) \\ X(w_2) \\ \vdots \\ X(w_{N-1}) \end{bmatrix} \quad (5.28)$$



### 5.3 EXEMPLOS DE TRANSFORMADA DISCRETA DE FOURIER (DFT) E TRANSFORMADA RÁPIDA DE FOURIER (FFT)

Dois exemplos são realizados nesta seção, o primeiro ilustra a DFT de duas entradas distintas, e o segundo exemplo implementa um algoritmo com duas entradas diferentes para ilustrar a FFT formulada por Danielson e Lanczos.

#### 5.3.1 Exemplo 1: DFT de uma Sequência de números reais com saída para a janela do CCS

Esse exemplo tem como objetivo ilustrar a transformada discreta de Fourier (DFT) de uma sequência de  $N$ - pontos através da implementação do código *dft.c* e da utilização de alguns recursos do CCS.

```
//dft.c
//DFT de N-pontos. Saída é verificada pela janela Watch Window

#include <stdio.h>
#include <math.h>
void dft(short *x, short k, int *outRe, int *outIm); // Protótipo de função
#define N 8 // Comprimento da DFT
float pi = 3.1416;

short x[N] = { 1000,707,0,-707,-1000,-707,0,707 }; // função cosseno de um ciclo

//short x[N]={0,602,974,974,602,0,-602,-974,-974,-602,
//           0,602,974,974,602,0,-602,-974,-974,-602}; // função seno de dois ciclos

int outRe[N] = {0,0,0,0,0,0,0,0}; // valor inicial da saída outRe
int outIm[N] = {0,0,0,0,0,0,0,0}; // valor inicial da saída outIm

void dft(short *x, short k, int *outRe, int *outIm) // função DFT
{
    int sumRe = 0; // valor inicial da componente real
    int sumIm = 0; // valor inicial da componente imaginária
    int i = 0;
    float cs = 0; // valor inicial da componente cosseno
    float sn = 0; // valor inicial da componente seno

    for (i = 0; i < N; i++) // para DFT de N-pontos
    {
        cs = cos(2*pi*(k)*i/N); // componente real
        sn = sin(2*pi*(k)*i/N); // componente imaginária
        sumRe = sumRe + x[i]*cs; // cálculo dos componentes reais
        sumIm = sumIm - x[i]*sn; // cálculo dos componentes imaginários
    }
}
```

```

    }

    outRe[k] = sumRe;           // cálculo dos componenetes reais
    outIm[k] = sumIm;          // cálculo dos componentes imaginários

}

void main()
{
    int j;

    for (j = 0; j < N; j++)
    {
        dft(x, j, outRe, outIm);           // função denominada DFT
    }
}

```

### Considerações sobre o programa

A DFT da seqüência de uma entrada  $x(n)$  é calculada por,

$$X(w_k) = \sum_{n=0}^{N-1} x(n)W_N^{kn}, \quad k = 0, 1, \dots, N-1$$

Onde  $W_N = e^{-j2\pi/N}$  são constantes de giro. Logo, esta equação pode ser decomposta em dois outros somatórios um com componentes reais e outro contendo componentes imaginários,

$$\begin{aligned} \text{Re}\{X(w_k)\} &= \sum_{n=0}^{N-1} x(n) \cos(2\pi nk/N) \\ \text{Im}\{X(w_k)\} &= \sum_{n=0}^{N-1} x(n) \text{sen}(2\pi nk/N) \end{aligned}$$

Assim ao utilizar uma seqüência de números reais com um número inteiro de ciclos  $m$ , tem-se  $X(w_k) = 0$  para todo o  $k$  exceto para  $k = m$  e  $k = N - m$ .

O programa. *dft.c* possui duas entradas  $x(n)$  distintas, em ambos os casos a frequência de amostragem é considerada como  $F_s = 8000$  Hz.

A primeira entrada é um sinal cosseno de comprimento  $N = 8$ , ou seja, composta por um único ciclo, multiplicado por 1000 e cuja frequência  $f = 1000$  kHz, uma vez que  $f = F_s (\text{numero de ciclos})/N = 1\text{kHz}$ , assim,

$$x(n) = 1000 \times \cos(2\pi n/8) = x(n) = 1000 \times \cos(\pi n/4)$$

Logo através da implementação na ferramenta MATLAB 7.0, do código que se encontra no anexo I, obtém-se  $x[N] = \{1000, 707, 0, -707, -1000, -707, 0, 707\}$ , como pode ser verificado graficamente na Figura 5.3.1.1.

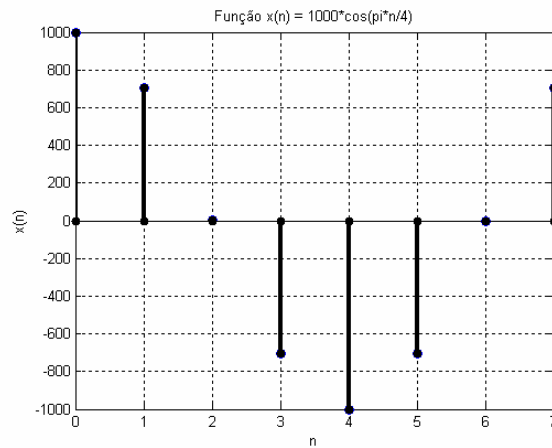


Figura 5.3.1.1 – Gráfico da função  $x(n) = 1000 * \cos\left(\frac{\pi}{4}n\right)$ ; Frequência do sinal  $f = 1000$  Hz e Frequência de amostragem  $F_s = 8000$  Hz.

As Figuras 5.3.1.2 e 5.3.1.3 representam graficamente os valores da componente real da DFT, (outRe), obtidos a partir do cálculo de  $\text{Re}\{X(w_k)\}$ , e dos valores da componente imaginária da DFT, (outIm), calculados por  $\text{Im}\{X(w_k)\}$ .

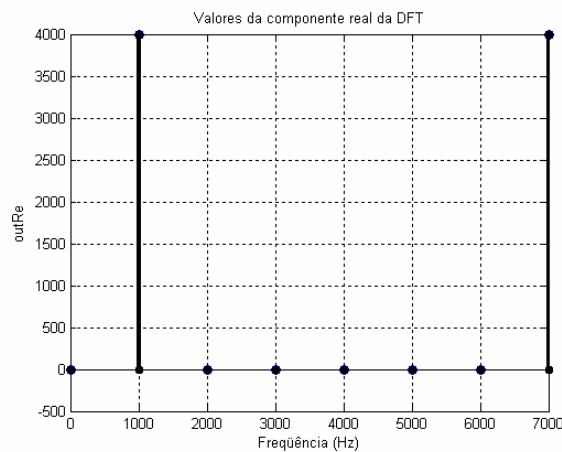


Figura 5.3.1.2 – Valores da componente real (outRe) da DFT de  $N = 8$  gerada no MATLAB.

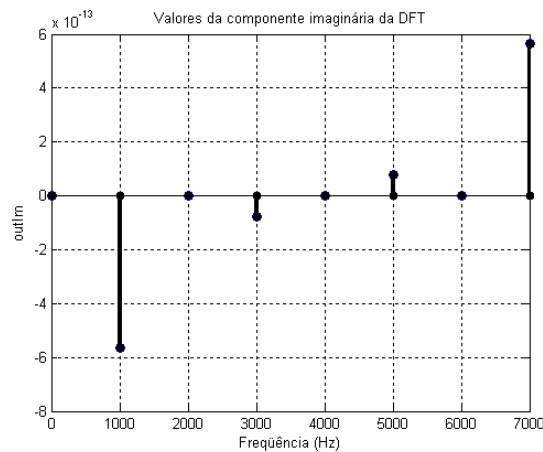


Figura 5.3.1.3 – Valores da componente imaginária (outIm) da DFT de  $N = 8$  gerada no MATLAB.

Verifica-se que por ser um sinal cosseno, a parte imaginária possui valores muito pequenos quando comparada aos valores da componente real, sendo assim considerada quase zero.

Os picos da parte real, como definidos anteriormente, ocorrem em  $k = m = 1$ ,  $\text{outRe}[1]$ , e  $k = N - m = 8 - 1 = 7$ ,  $\text{outRe}[7]$ , como pode ser verificado na Figura 5.3.1.2.

A segunda entrada composta no programa *DFT.c* é um sinal seno de comprimento  $N = 20$ , composto por dois ciclos, multiplicado por 1000, cuja frequência  $f = 800\text{Hz}$ , assim,

$$x(n) = 1000 \times \sin(2\pi n 2/20) = x(n) = 1000 \times \cos(\pi n/5)$$

Logo, através da implementação na ferramenta MATLAB 7.0, do código que se encontra no anexo I, como pode ser visto na Figura 5.3.1.4 os valores obtidos pelo seno são:

$$x[N] = \{0, 602, 974, 974, 602, 0, -602, -974, -974, -602, 0, 602, 974, 974, 602, 0, -602, -974, -974, -602\}$$

Ao contrário do sinal cosseno, o sinal seno possui a parte real próxima de zero quando comparada à parte imaginária como pode ser observado nas Figuras 5.3.1.5 e 5.3.1.6.

Os picos da parte imaginária ocorrem em  $k = m = 2$ ,  $\text{outIm}[2]$ , e  $k = N - m = 18$ ,  $\text{outIm}[18]$ , como pode ser verificado na Figura 5.3.1.6.

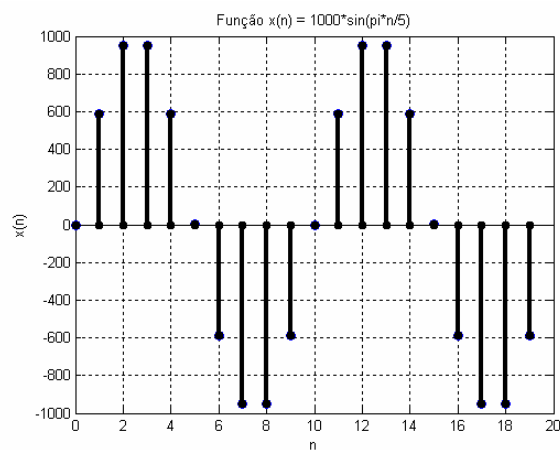


Figura 5.3.1.4 – Gráfico da função  $x(n) = 1000 \times \sin\left(\frac{\pi}{5} n\right)$ ; Freqüência do sinal  $f = 800\text{Hz}$  e Freqüência de amostragem  $F_s = 8000\text{ Hz}$ .

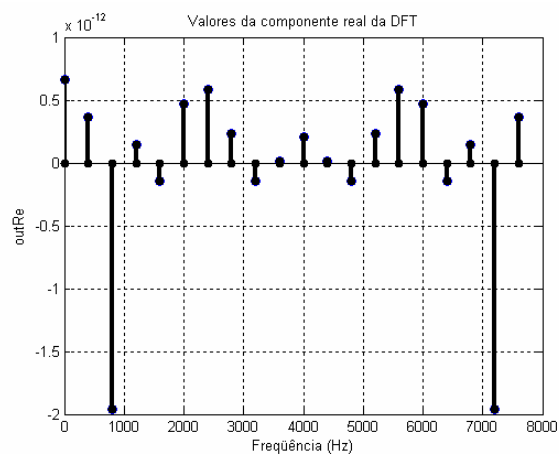


Figura 5.3.1.5 – Valores da componente real (outRe) da DFT de  $N = 20$  gerada no MATLAB.

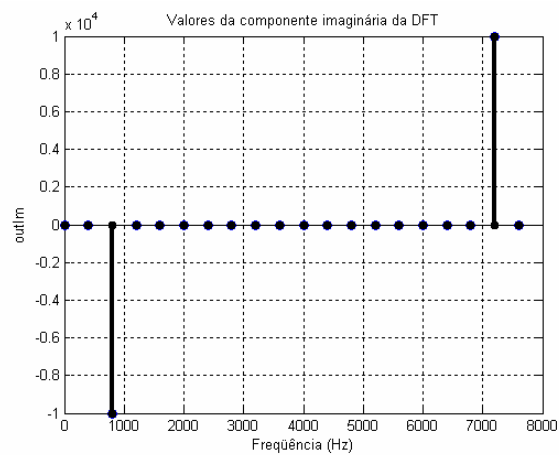


Figura 5.3.1.6 – Valores da componente imaginária (outIm) da DFT de  $N = 20$  gerada no MATLAB.

## Criação do projeto

Para criar este projeto no *Code Composer Studio™ IDE*, é preciso adicionar os arquivos necessários à construção do projeto DFT, como fora feito nos exemplos anteriores.

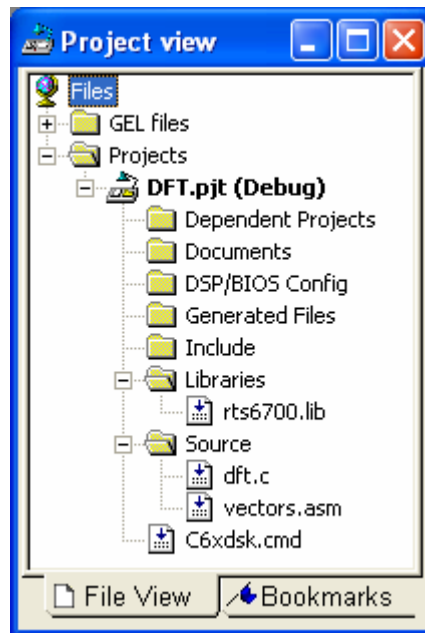


Figura 5.3.1.7 – Visualização do projeto.

Para gerar o executável e carregar o programa utilizam-se as mesmas configurações dos parâmetros do compilador e do *linker* como fora feito no terceiro tutorial.

## Obtenção da DFT no CCS

Para obter a DFT no CCS é necessário que você:

1. Selecione View ➔ *Watch Window*. Ao abrir a janela clique na aba *Watch 1* e escreva outRe e outIm.



Figura 5.3.1.8 – Visualização da janela Watch Window para  $N = 8$ .

Como a função cosseno corresponde à primeira entrada, então se observa na Figura 5.3.1.8 que a coluna *Type* refere-se ao comprimento da DFT.

Logo ao expandir na coluna *Name* outRe e outIm obtém-se os valores respectivos das componentes real e imaginária da DFT.

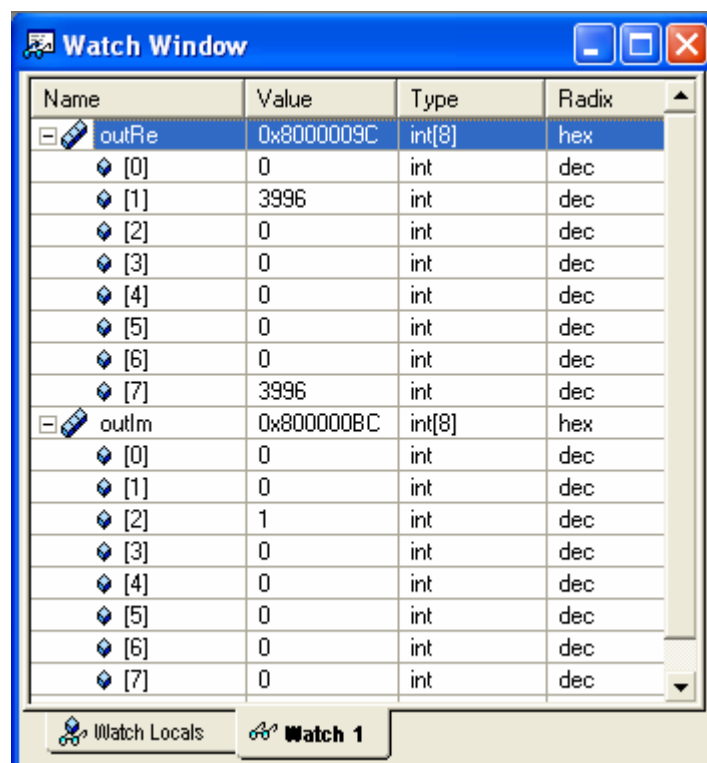


Figura 5.3.1.9 – Componentes real (outRe) e imaginária (outIm) da DFT de  $N = 8$ .

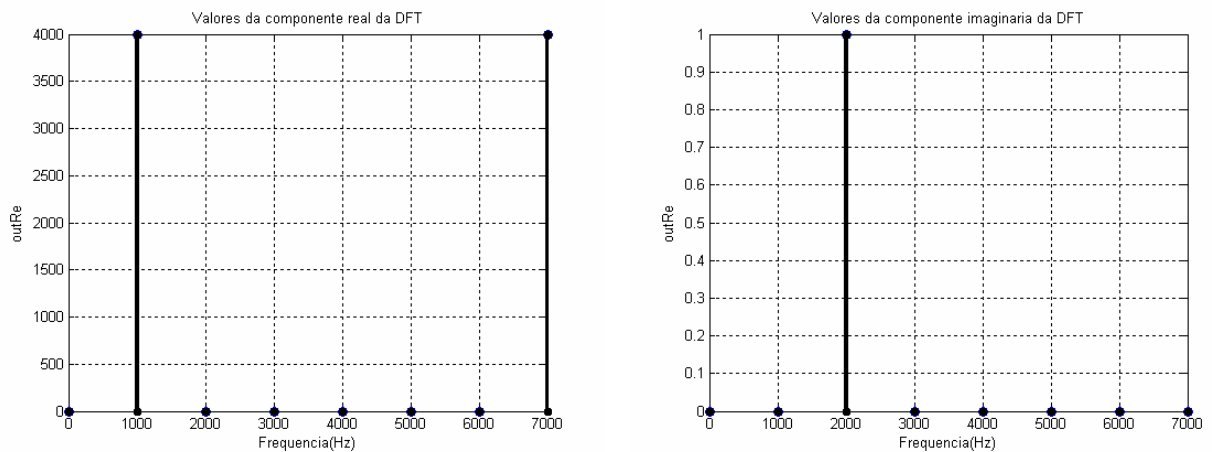


Figura 5.3.1.10 – Componentes real (outRe) e imaginária (outIm) da DFT de  $N = 8$  geradas no MATLAB.

A Figura 5.3.1.11 mostra a análise da complexidade da função “dft” por meio do *Profile Viewer* apresentado no Capítulo 2, Tutorial 1, Exemplo 5.

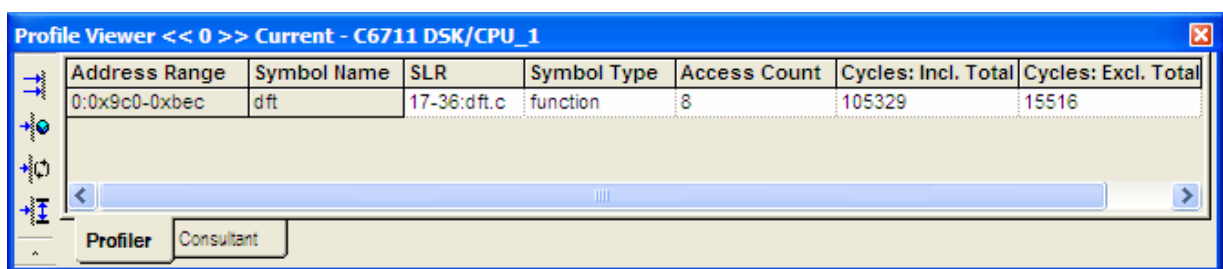


Figura 5.3.1.11 – Complexidade da função dft.c de uma DFT de comprimento  $N = 8$ .

A função *dft.c* neste caso foi acessada 8 vezes, foram necessários 15516 ciclos para execução da função, entretanto incluindo as sub-rotinas foram necessários 105329 ciclos.

Considerando o sinal seno como entrada, e fazendo as modificações necessárias no código *dft.c* é possível obter:



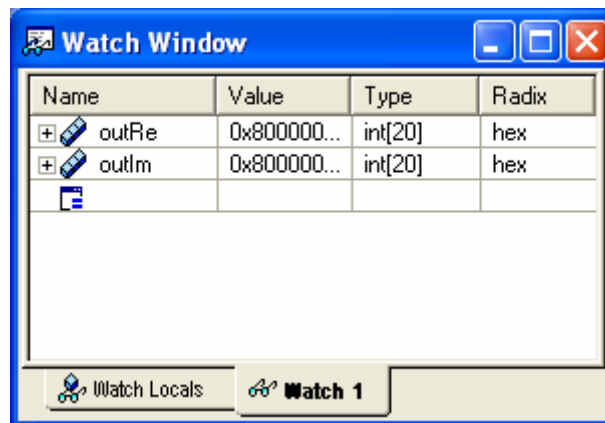


Figura 5.3.1.12 – Visualização da janela Watch Window para  $N = 20$ .

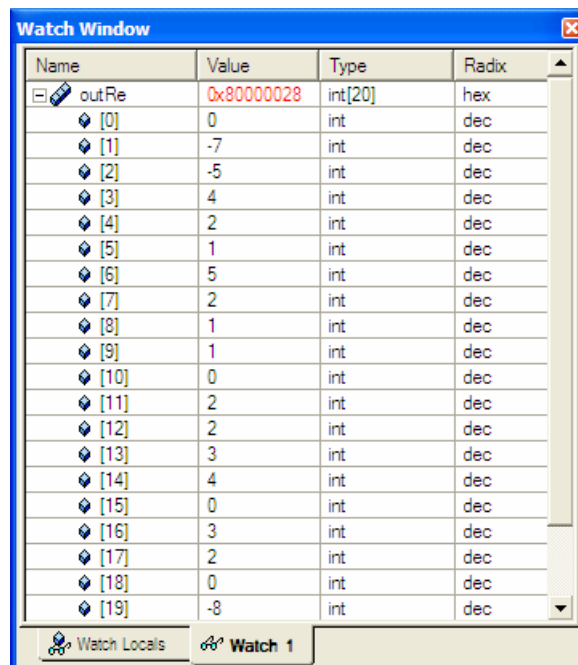


Figura 5.3.1.13 – Valores da componente real (outRe) da DFT de  $N = 20$ .

Watch Window			
Name	Value	Type	Radix
outRe	0x80000028	int[20]	hex
outIm	0x80000078	int[20]	hex
[0]	0	int	dec
[1]	0	int	dec
[2]	-10232	int	dec
[3]	-1	int	dec
[4]	0	int	dec
[5]	0	int	dec
[6]	0	int	dec
[7]	1	int	dec
[8]	0	int	dec
[9]	-1	int	dec
[10]	0	int	dec
[11]	-1	int	dec
[12]	0	int	dec
[13]	0	int	dec
[14]	0	int	dec
[15]	0	int	dec
[16]	0	int	dec
[17]	1	int	dec
[18]	10232	int	dec
[19]	1	int	dec

Figura 5.3.1.14 – Valores da componente imaginária (outIm) da DFT de  $N = 20$ .

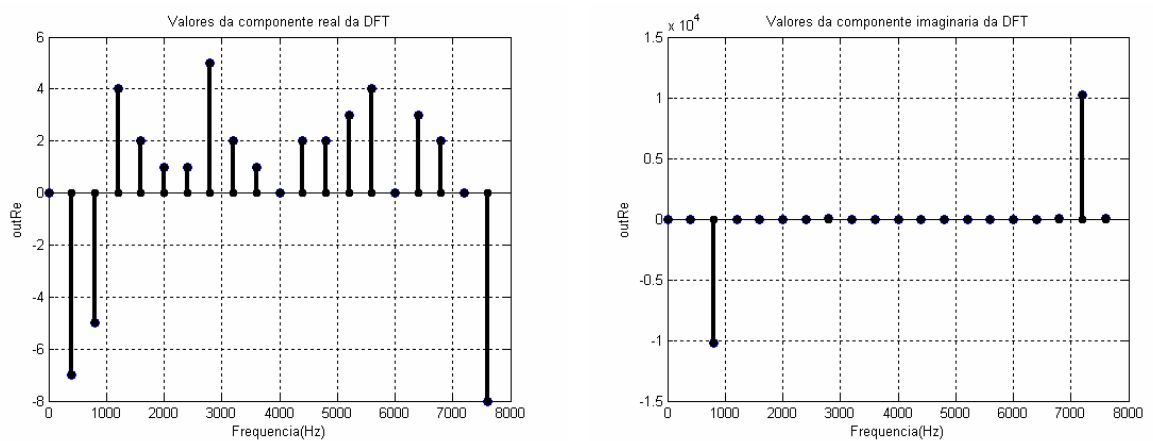


Figura 5.3.1.15 – Componentes real (outRe) e imaginária (outIm) da DFT de  $N = 20$  geradas no MATLAB.

A Figura 5.3.1.16 mostra a análise da complexidade da função “dft”.

Profile Viewer << 0 >> Current - C6711 DSK/CPU_1						
Address Range	Symbol Name	SLR	Symbol Type	Access Count	Cycles: Incl. Total	Cycles: Excl. Total
0:0x9c0-0xbf0	dft	17-36:dft.c	function	20	806480	95161

Figura 5.3.1.16 – Complexidade da função dft.c de uma DFT de comprimento  $N = 20$ .

A função *dft.c* para o sinal seno foi acessada 20 vezes, foram necessários 95161 ciclos para execução da função, entretanto ao incluir as sub-rotinas foram necessários 806480 ciclos.

Devido à limitação na precisão do *DSP* o gráfico da componente imaginária da função cosseno obtida, Figura 5.3.1.10, é diferente do gráfico apresentado na parte teórica, Figura 5.3.1.3, mas ambos quando comparados a componente real tendem a zero. Este tipo de limitação também pode ser observado quando comparados os gráficos da componente real do sinal seno, Figura 5.3.1.15 e Figura 5.3.1.5.

### 5.3.2 Exemplo 2: FFT formulada por Danielson -Lanczos.

Esse exemplo tem como objetivo ilustrar a transformada rápida de Fourier (FFT) através da implementação do código *fft.c*, algoritmo este formulado por Danielson-Lanczos, de uma sequência de *N*- pontos , utilizando alguns recursos do CCS.

```
//fft.c
//FFT de N-pontos. Saída é verificada pela janela Watch Window

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

//Dados de entrada //
//SINAL 1 //
#define N_x 17
#define N_fft 8
float x[N_x] = {0,1000,0,707,0,0,0,-707,0,-1000,0,-707,0,0,0,707,0};

//SINAL 2 //
//#define N_x 65
//#define N_fft 32
//float x[N_x]={0, 0,0, 383,0, 707,0, 924,0, 1000,0, 924,0, 707,0, 383,0, 0,0,
//              -383,0, -707,0, -924,0,-1000,0,-924,0,-707,0, -383,0, 0,0,
//              383,0, 707,0, 924,0, 1000,0, 924,0, 707,0, 383,0, 0,0,
//              -383,0, -707,0, -924,0,-1000,0,-924,0,-707,0, -383,0};

// Saida
float outRe[N_fft];
float outIm[N_fft];

void fft(float data[], unsigned long nn, int isign) {

    unsigned long n,mmax,m,j,istep,i;
```

```

double wtemp,wr,wpr,wpi,wi,theta;
float tempr,tempi;

n=nn << 1;
j=1;
for (i=1;i<n;i+=2)
{
    if (j > i)
    {
        SWAP(data[j],data[i]);
        SWAP(data[j+1],data[i+1]);
    }

    m=n >> 1;
    while (m >= 2 && j > m) {
        j -= m;
        m >>= 1;
    }
    j += m;
}
mmax=2;
while (n > mmax)
{
    istep=mmax << 1;
    theta=isign*(6.28318530717959/mmax);
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0;
    wi=0.0;
    for (m=1;m<mmax;m+=2)
    {
        for (i=m;i<=n;i+=istep)
        {
            j=i+mmax;
            tempr=wr*data[j]-wi*data[j+1];
            tempi=wr*data[j+1]+wi*data[j];
            data[j]=data[i]-tempr;
            data[j+1]=data[i+1]-tempi;
            data[i] += tempr;
            data[i+1] += tempi;
        }
        wr=(wtemp=wr)*wpr-wi*wpi+wr;
        wi=wi*wpr+wtemp*wpi+wi;
    }
    mmax=istep;
}
}
main()
{
    int i;
    fft(x,N_fft,1);
    for (i=1;i<=N_fft;i++)
    {
        outRe[i-1] = x[2*i-1];

```

```

        outIm[i-1] = x[2*i];
    }
}

```

### Considerações sobre o programa

Sabe-se que Danielson e Lanczos mostraram que a DFT poderia ser reescrita como a soma de duas outras, uma formada por pontos pares (e), e outra por pontos ímpares (o), cada uma de tamanho  $N/2$ , como fora feito em (5.8). Logo a Equação (5.12) é representada por.

$$F_k = F_k^e + W_N^k F_k^o$$

A aplicação deste Teorema só é possível quando  $N$  é potência de 2, caso a FFT em questão não tenha potência de dois, zeros serão acrescentados até que FFT atinja a próxima potência de dois.

Uma vez que  $N$  da FFT é uma potência de dois o Teorema de Danielson e Lanczos é utilizado até que o comprimento desta seja igual a um. Neste caso uma forma de solucionar o principal problema, ou seja, o de combinar as transformadas de ordem 2, depois as de ordem 4 e assim sucessivamente, seria combinar os pares adjacentes de elementos, depois combinar elementos resultantes em pares adjacentes e assim por diante. Este tipo de combinação, entre pares adjacentes, é possível através da reordenação dos elementos.

Logo ao aplicar-se sucessivas divisões entre pares e ímpares até obter uma transformada de 1 termo:

$$F_k^{eooooe...oe} = f_n$$

Neste caso  $f_n$  não depende de  $k$ , então é preciso descobrir quem é  $n$  no conjunto original. Substituindo os e's e os o's por 0's e 1's, respectivamente, um número binário é obtido. O valor de  $n$  é exatamente este número com a ordem dos bits invertida. Isto acontece porque as subdivisões sucessivas em pares e ímpares são testes sucessivos de bits de baixa ordem  $n$ .

A primeira divisão entre pares e ímpares de um sinal de 8 pontos, como o *SINAL 1*, do código *fft.c*,  $\{f(0), f(1), f(2), f(3), f(4), f(5), f(6), f(7)\}$  resulta em dois conjuntos  $\{f(0), f(2), f(4), f(6),\}$  e  $\{f(1), f(3), f(5), f(7)\}$ , já a divisão destes resultará em quatro

conjuntos  $\{f(0), f(4)\}, \{f(2), f(6)\}, \{f(1), f(5)\}, \{f(3), f(7)\}$ . Este tipo de reordenação é verificado na primeira parte do código *fft.c*.

Tabela 5.3.1 – Reordenação utilizando a inversão de Bits

Conjunto Original	Índice	Conjunto Reordenado	Índice
f(0)	000	f(0)	000
f(1)	001	f(4)	100
f(2)	010	f(2)	010
f(3)	011	f(6)	110
f(4)	100	f(1)	001
f(5)	101	f(5)	101
f(6)	110	f(3)	011
f(7)	111	f(7)	111

A segunda parte do código *fft.c* possui um laço externo que é executado  $\log_2 N$  vezes e calcula, em ordem, a transformada do comprimento  $2, 4, 8, \dots, N$ . Para cada estágio deste processo, dois laços internos aninhados escalam sobre as transformadas já computadas e os elementos de cada um transformam, implementando desta forma o Teorema de Danielson-Lanczos. A operação é feita mais eficiente por chamadas externas de restrição para senos e cossenos trigonométricos ao laço exterior, aonde ele é feito somente  $\log_2 N$  vezes. A computação de senos e cossenos de ângulos múltiplos são através da simples relação de recorrência no laço interno.

O código *fft.c* tem como entrada o número de pontos de dados complexos  $nn$ , o *array* de dados (`data[1...2*nn]`), e *isign*, o qual deve ser ajustado para  $\pm 1$  o que representa o sinal de  $i$  na exponencial em (5.1). Quando *isign* for igual a -1 a rotina calculará a transformada inversa sem multiplicação da mesma pelo termo  $1/N$ , equação (5.27).

O *array* de entrada contém um comprimento de  $2 \times nn$ , onde cada valor complexo ocupa duas localizações consecutivas, ou seja, `data[1]` é a parte real de  $f_0$ , e `data[2]` a parte imaginária do mesmo, e assim consecutivamente até `data[2*nn-1]` que representa a parte real de  $f_{N-1}$  e `data[2*nn]` que é a parte imaginária deste.

O *array* de saída contém o espectro complexo de Fourier com  $N$  valores de frequência. Ou seja, a parte real e imaginária da frequência zero (componentes de  $F_0$ ) estão localizados em  $data[1]$  e  $data[2]$ , logo a menor frequência positiva tem suas componentes real e imaginária localizadas em  $data[3]$  e  $data[4]$ , com o aumento da magnitude as partes real e imaginária são armazenadas em  $data[5]$  e  $data[6]$ , assim sucessivamente até  $data[nn-1]$ ,  $data[nn]$ . Consequentemente a menor frequência negativa diferente de zero tem parte real em  $data[2*nn-1]$  e a parte imaginária em  $data[2*nn]$ , a próxima frequência é armazenada em  $data[2*nn-3]$ ,  $data[2*nn-2]$ , e assim sucessivamente até  $data[nn+3]$  e  $data[nn+4]$ . Logo  $data[nn+1]$  e  $data[nn+2]$  contém as partes reais e imaginárias das frequências mais positivas e mais negativas, acarretando desta forma num ponto de *alias*.

O *SINAL 1* observado no código *fft.c* é o mesmo sinal cosseno utilizado no exemplo anterior no código *dft.c*, como apresentado na Figuras (5.3.1.1), (5.3.1.2) e (5.3.1.3).

O *SINAL 2* a ser processado é um sinal seno de comprimento  $N = 32$ , multiplicado por 1000, cuja frequência é  $f = 500\text{Hz}$ , assim,

$$x(n) = 1000 \times \sin(2\pi n/32) = x(n) = 1000 \times \cos(\pi n/8)$$

Através da implementação na ferramenta MATLAB 7.0, do código que se encontra no anexo I, obtém-se:

$$x[N] = \{0, 0, 0, 383, 0, 707, 0, 924, 0, 1000, 0, 924, 0, 707, 0, 383, 0, 0, 0, -383, 0, -707, 0, -924, 0, -1000, 0, -924, 0, -707, 0, -383, 0, 0, 0, 383, 0, 707, 0, 924, 0, 1000, 0, 924, 0, 707, 0, 383, 0, 0, 0, -383, 0, -707, 0, -924, 0, -1000, 0, -924, 0, -707, 0, -383, 0\}$$

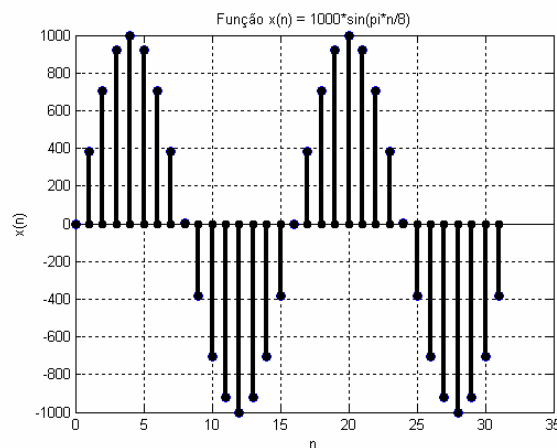


Figura 5.3.2.1 –Gráfico da função  $x(n) = 1000 * \sin\left(\frac{\pi}{8}n\right)$ ; Frequência do sinal  $f = 500\text{ Hz}$  e Frequência de amostragem  $F_s = 8000\text{ Hz}$ .

A Figura 5.3.2.2 representa graficamente os valores das componentes real da FFT, (outRe), e dos valores da componente imaginária desta FFT, (outIm).

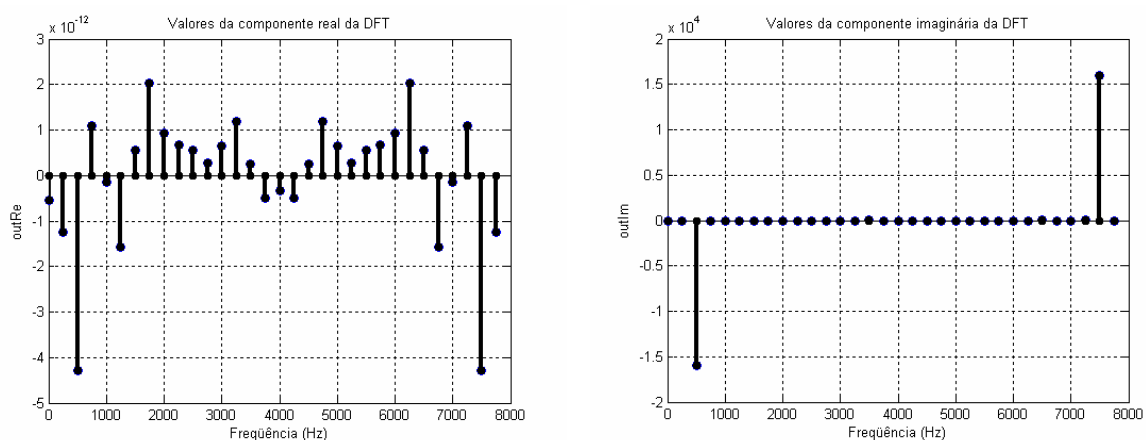


Figura 5.3.2.2 – Componentes real (outRe) e imaginária (outIm) da FFT de  $N = 32$  geradas no MATLAB.

### Criação do projeto

Para criar este projeto no *Code Composer Studio™ IDE*, é preciso adicionar os arquivos necessários à construção do projeto FFT, ou seja, *fft.c*, *vector.asm*, *C6xdsk.cmd* e *rts6700.lib*, que estão presente na pasta *FFT* do *cd* fornecido.

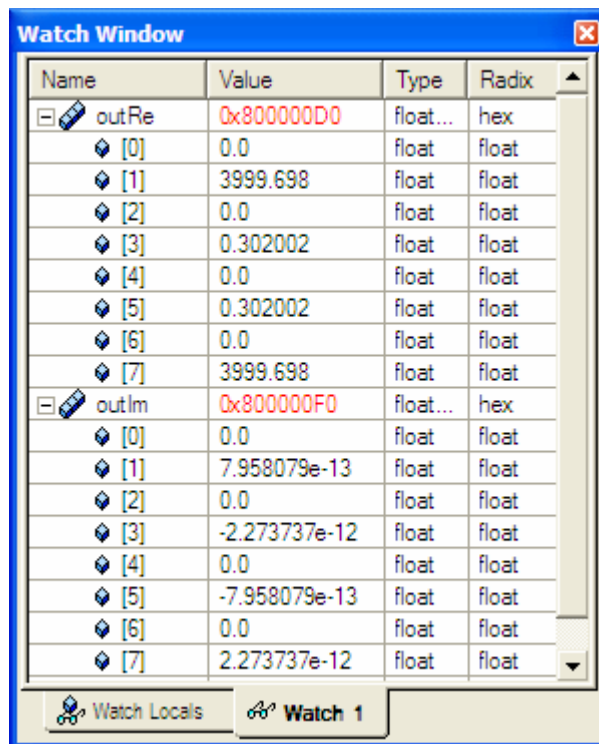
Após a construção do projeto gere o executável e carregue o programa utilizando as mesmas configurações dos parâmetros do compilador e do *linker* como fora feito no projeto da DFT.

### Obtenção da FFT no CCS

Para obter a FFT no *CCs* é necessário que você realize os mesmos comandos realizados na DFT e assim selecione a janela *Watch Window*, indicando na barra *Watch 1* outRe e outIm. Logo ao expandir a coluna *Name*, outRe e outIm obtém-se os valores respectivos das componentes real e imaginária da FFT.

Logo ao considerar o *SINAL 1*, obtém-se a seguinte resposta.

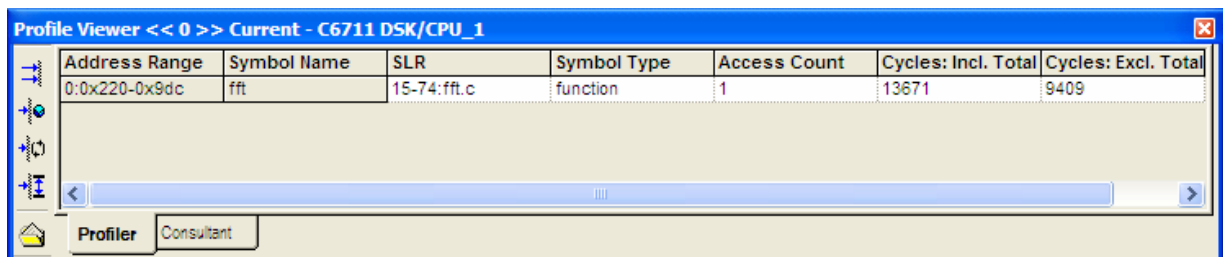




Name	Value	Type	Radix
[-] outRe	0x800000D0	float...	hex
[0]	0.0	float	float
[1]	3999.698	float	float
[2]	0.0	float	float
[3]	0.302002	float	float
[4]	0.0	float	float
[5]	0.302002	float	float
[6]	0.0	float	float
[7]	3999.698	float	float
[-] outIm	0x800000F0	float...	hex
[0]	0.0	float	float
[1]	7.958079e-13	float	float
[2]	0.0	float	float
[3]	-2.273737e-12	float	float
[4]	0.0	float	float
[5]	-7.958079e-13	float	float
[6]	0.0	float	float
[7]	2.273737e-12	float	float

Figura 5.3.2.3 – Componentes real (outRe) e imaginária (outIm) da FFT de  $N = 8$ .

A Figura 5.3.2.4 mostra a análise da complexidade da função “fft”.



Address Range	Symbol Name	SLR	Symbol Type	Access Count	Cycles: Incl. Total	Cycles: Excl. Total
0:0x220-0x9dc	fft	15-74:fft.c	function	1	13671	9409

Figura 5.3.2.4 – Complexidade da função `fft.c` de uma FFT de comprimento  $N = 8$ .

Logo pode ser verificada na Figura 5.3.2.4 a diminuição tanto do *Cycles: Incl.Total* quanto do *Cycles: Excl.Total*, ou seja, dos custos dos ciclos de *clock*, quando comparada com a Figura 5.3.1.11 que representa a complexidade da função `dft.c` de uma DFT de comprimento de  $N = 8$ .

Fazendo as modificações necessárias no código `fft.c` e implementando o sinal seno, *SINAL 2*, obtém-se as seguintes componentes:

Watch Window			
Name	Value	Type	Radix
outRe	0x80000104	float...	hex
[0]	0.0	float	float
[1]	0.0	float	float
[2]	0.0002475658	float	float
[3]	0.0	float	float
[4]	0.0	float	float
[5]	0.0	float	float
[6]	9.289657e-05	float	float
[7]	0.0	float	float
[8]	0.0	float	float
[9]	0.0	float	float
[10]	-9.289657e-05	float	float
[11]	0.0	float	float
[12]	0.0	float	float
[13]	0.0	float	float
[14]	-0.0002475658	float	float
[15]	0.0	float	float
[16]	0.0	float	float
[17]	0.0	float	float
[18]	-0.0002475658	float	float
[19]	0.0	float	float
[20]	0.0	float	float
[21]	0.0	float	float
[22]	-9.289657e-05	float	float
[23]	0.0	float	float
[24]	0.0	float	float
[25]	0.0	float	float
[26]	9.289657e-05	float	float
[27]	0.0	float	float
[28]	0.0	float	float
[29]	0.0	float	float
[30]	0.0002475658	float	float
[31]	0.0	float	float

Figura 5.3.2.5 – Valores da componente real (outRe) e da FFT de  $N = 32$ .

Name	Value	Type	Radix
outIm	0x80000184	float...	hex
[0]	0.0	float	float
[1]	0.0	float	float
[2]	16001.26	float	float
[3]	0.0	float	float
[4]	0.0	float	float
[5]	0.0	float	float
[6]	1.36691	float	float
[7]	0.0	float	float
[8]	0.0	float	float
[9]	0.0	float	float
[10]	2.574918	float	float
[11]	0.0	float	float
[12]	0.0	float	float
[13]	0.0	float	float
[14]	2.463379	float	float
[15]	0.0	float	float
[16]	0.0	float	float
[17]	0.0	float	float
[18]	-2.463379	float	float
[19]	0.0	float	float
[20]	0.0	float	float
[21]	0.0	float	float
[22]	-2.574918	float	float
[23]	0.0	float	float
[24]	0.0	float	float
[25]	0.0	float	float
[26]	-1.36691	float	float
[27]	0.0	float	float
[28]	0.0	float	float
[29]	0.0	float	float
[30]	-16001.26	float	float
[31]	0.0	float	float

Figura 5.3.2.6 – Valores da componente imaginária (outIm) da FFT de  $N = 32$ .

A Figura 5.3.1.16 mostra a análise da complexidade da função “fft”.

Address Range	Symbol Name	SLR	Symbol Type	Access Count	Cycles: Incl. Total	Cycles: Excl. Total
0:0x220-0x9dc	fft	26-85:fft.c	function	1	60294	53456

Figura 5.3.2.7 – Complexidade da função fft.c de uma FFT de comprimento  $N = 32$ .

## **6 - TUTORIAL 5: FILTROS ADAPTATIVOS**

### **6.1 OBJETIVO**

- Introdução do conceito sobre estruturas adaptáveis
- Estudo do Algoritmo do gradiente estocástico - Least Mean Square
- Programação de identificação de sistema utilizando C e o TMS320C6711 DSK

Esse tutorial tem por finalidade apresentar o conceito sobre filtros adaptativos, de suas estruturas e características básicas, de algoritmos adaptativos, e aplicações importantes. Projetar um exemplo de filtro adaptativo utilizando processador DSP de ponto flutuante, TMS320C6711 DSK.

### **6.2 INTRODUÇÃO**

#### **6.2.1 Filtros adaptativos**

Nos filtros digitais convencionais FIR e IIR, como visto anteriormente, os parâmetros do processo que determinam as características dos filtros já são conhecidos e podem variar com o tempo, uma vez que a natureza da variação é também conhecida. Entretanto muitos problemas práticos não podem ser solucionados com a implementação dos filtros digitais fixos ou por não possuir informação suficiente para projetar o filtro com coeficientes fixos, havendo assim uma grande incerteza com relação a alguns parâmetros, ou porque os critérios do projeto mudam durante a operação. Para tanto um tipo especial de filtro que modifica a sua resposta automaticamente, adaptando-se a uma dada situação, para que seu desempenho melhore durante a operação, denominado de filtro adaptativo, é utilizado.

O filtro adaptativo IIR possui a mesma performance que o filtro adaptativo FIR com baixa complexidade, entretanto caso os pólos existentes se locomovam para fora do círculo unitário durante o processo de adaptação, o filtro adaptativo IIR torna-se potencialmente instável. Logo, o filtro adaptativo não-recursivo, ou combinador linear adaptativo (CLA), é utilizado na maioria dos filtros adaptativos e é o elemento mais importante em sistemas de aprendizado e processos adaptativos em geral [Widrow, 1985].

A saída do CLA, para um conjunto fixo de pesos, é a combinação linear dos elementos de entrada. Caso os pesos estejam em processo de adaptação, sendo também função dos componentes de entrada, a saída do combinador deixa de ser uma função linear da entrada.

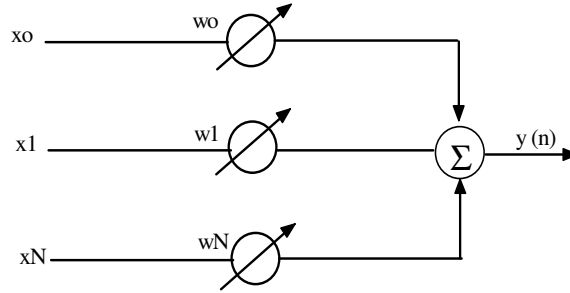


Figura 6.2.1.1 – Forma geral de um combinador linear adaptativo.

Logo a Equação 6.1 é a relação de entrada e saída de um filtro adaptativo FIR.

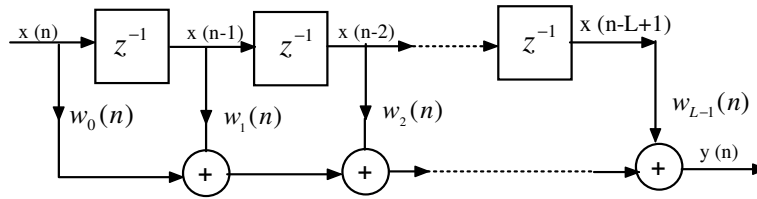


Figura 6.2.1.2 – Diagrama de um filtro adaptativo FIR.

$$y(n) = \sum_{i=0}^{L-1} w_i(n)x(n-i) = \mathbf{w}^T(n)\mathbf{x}(n) \quad (6.1)$$

Onde  $w_i(n)$  representa os  $i$ 's coeficientes ajustáveis ou pesos para um tempo específico  $n$ . O vetor de coeficientes  $\mathbf{w}(n)$  constitui de  $L$  coeficientes.

$$\mathbf{w}(n) = [w_0(n) \ w_1(n) \dots w_{L-1}(n)]^T \quad (6.2)$$

$$\mathbf{x}(n) = [x(n) \ x(n-1) \dots x(n-L+1)]^T \quad (6.3)$$

No processo de adaptação o vetor de pesos do combinador linear é ajustado, de acordo com alguns critérios estatísticos, de tal forma que à minimizar o sinal de erro,  $e(n)$ , ou seja, que produza uma saída  $y(n)$  o mais próximo possível do sinal desejado  $d(n)$ .

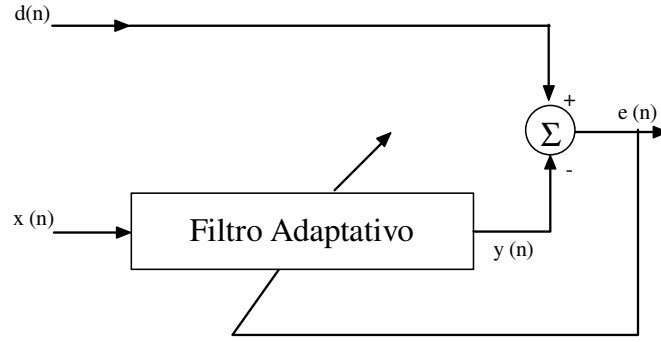


Figura 6.2.1.3 – Estrutura Básica de um filtro Adaptativo.

Onde,

$$e(n) = d(n) - y(n) \quad (6.4)$$

A atualização dos coeficientes é feita através do critério de minimização do erro quadrático médio acarretando, desta forma, no processo de minimização do critério predeterminado. Para a obtenção do erro quadrático médio calcula-se o valor esperado da seguinte forma;

$$\xi = E[e^2(n)] \quad (6.5)$$

Onde  $E[.]$  é o operador que retorna o valor esperado.

Substituindo as Equações 6.1 e 6.4 na Equação 6.5, obtém-se:

$$\xi = E[d^2(n)] + \mathbf{w}^T(n)\mathbf{R}\mathbf{w}(n) - 2\mathbf{w}^T(n)\mathbf{p} \quad (6.6)$$

A matriz  $L \times L$  de autocorrelação do vetor de amostras do sinal de entrada é representada por,

$$\mathbf{R} = E[\mathbf{x}(n)\mathbf{x}^T(n)] = \begin{bmatrix} r_{xx}(0) & r_{xx}(1) & \dots & r_{xx}(L-1) \\ r_{xx}(1) & r_{xx}(0) & & \\ \vdots & & \ddots & r_{xx}(1) \\ r_{xx}(L-1) & r_{xx}(1) & r_{xx}(0) & \end{bmatrix} \quad (6.7)$$

Os termos da diagonal principal da matriz  $\mathbf{R}$  são os quadrados dos componentes da entrada, e os termos fora desta diagonal indicam a correlação entre os elementos de entrada.

O vetor  $\mathbf{p}$  da Equação 6.6 representa o vetor  $L \times 1$  de correlação cruzada.

$$\xi = E[d(n)\mathbf{x}(n)] = [r_{dx}(0) \quad r_{dx}(1) \quad \dots \quad r_{dx}(L-1)]^T \quad (6.8)$$

Este vetor quantifica a dependência entre o sinal desejado e cada elemento do valor de sinais de entrada. Onde a função de autocorrelação é definida como;

$$r_{dx}(k) = E[d(n)x(n-k)] \quad (6.9)$$

Onde  $k=0,1,...,L-1$ . Logo se verifica que a equação (6.6) é uma função convexa garantindo, desta forma, a existência de um mínimo local. Uma maneira de se determinar o mínimo da superfície de erro quadrático é através do método do gradiente [Widrow, 1985].

$$\nabla(\xi) = \frac{\partial \xi}{\partial \mathbf{w}(n)} = 0 \quad (6.10)$$

Assim para a função de erro definida em (6.6) obtém-se o vetor de coeficiente ótimo  $\mathbf{w}^0$ .

$$\mathbf{w}^0 = \mathbf{R}^{-1}\mathbf{p} \quad (6.11)$$

Esta equação é conhecida como Wiener-Hopf, e o filtro com coeficientes ótimos dados nesta Equação 6.9 é chamado de filtro de Wiener.

Ao substituir a Equação 6.11 em 6.6 obtém-se o erro médio quadrático mínimo, resultando, desta forma, num  $y(n)$  com uma estimativa bastante próxima de  $d(n)$ .

Uma maneira bastante intuitiva de se construir este algoritmo é inicializar os coeficientes com um valor qualquer, e a cada iteração dar pequenos passos contrários ao gradiente de desempenho. Deste modo a cada iteração se terá uma melhor aproximação de  $\mathbf{w}^0$  [Born, 2000].

## 6.2.2 Algoritmo LMS - Least Mean Square

O algoritmo LMS é importante pela simplicidade e facilidade de computação. Se o sistema adaptativo é um combinador linear adaptativo, e o vetor de entrada  $\mathbf{x}(n)$  e a resposta desejada  $d(n)$  estão disponíveis a cada iteração, o algoritmo LMS é geralmente a melhor escolha para muitas aplicações de sinais [Widrow, 1985].

Este algoritmo utiliza o valor instantâneo do erro quadrado,  $e^2(n)$ , como estimativa da função de custo por  $E[e^2(n)]$ . Logo,

$$\nabla(n) = \frac{\partial^2 e(n)}{\partial \mathbf{w}(n)} = -2e(n)\mathbf{x}(n) \quad (6.12)$$

Como ao gradiente é o vetor que aponta no sentido máximo da função de custo, desloca-se o vetor de pesos na direção oposta com o objetivo de procurar o mínimo. Assim o algoritmo LMS, ou algoritmo de gradiente estocástico, é expresso como:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + 2\mu e(n)\mathbf{x}(n) \quad (6.13)$$

Nas implementações práticas a constante  $2\mu$  é usualmente substituídas por  $\mu$ . A constante  $\mu$  é denominada de ganho (ou passo) de adaptação, e tem como características o controle de estabilidade e da velocidade de convergência do algoritmo.

Para que haja estabilidade no algoritmo é preciso que o valor do ganho de adaptação obedeça a seguinte desigualdade [Opeheim, 1993],

$$0 < \mu < \frac{1}{LP_x} \quad (6.14)$$

Onde  $L$  é comprimento e a potência do sinal de entrada  $x(n)$  é  $P_x$ . O valor de  $P_x$  é calculado através de um simples método recursivo que estima a potência de um sinal de amostras.

Uma importante técnica para otimizar a velocidade de convergência enquanto é mantida a independência da potência do sinal é conhecida como algoritmo normalizado de LMS, ou seja, NLMS [Nascimento, 2005].

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu e(n)\mathbf{x}(n) \quad (6.15)$$

Neste caso  $\mu$  é normalizado por  $L$  e pela potência do sinal  $x(n)$ , ou seja,

$$\mu = \frac{\alpha}{L\hat{P}_x} \quad (6.16)$$

Onde  $\alpha$  é uma constante positiva de valor inferior a uma unidade, e  $\hat{P}_x$  é a estimativa da potência do sinal de referência, ou seja, de  $x(n)$  no tempo  $n$ .

Por definição, em [Kuo & Gan, 2005], tem-se;

$$\hat{P}_x = (1 - \beta)\hat{P}_x(n-1) + \beta x^2(n) \quad (6.17)$$

Onde  $\beta < 1$ , afinal  $\beta = 1/N$ , sendo  $N$  o comprimento da janela.

A Equação 6.15 apresenta melhor comportamento que o LMS simples na presença de sinais de voz por adaptar o ganho em função da potência do sinal. Este tipo de sinal apresenta grande variação de potência ao longo do tempo, tornando bastante difícil à escolha de um valor fixo para o ganho de adaptação: caso o ganho seja muito pequeno o algoritmo aproxima-se da solução parando a adaptação antes de atingir uma posição razoável devido à precisão finita. Por outro lado se o valor da potência estimada for zero ou muito pequeno, o ganho de adaptação é suficientemente elevado e o algoritmo torna-se instável. Para tanto, nas implementações básicas são utilizados softwares limitados.

A Equação 6.13 pode ser reescrita na forma escalar [Haykin, 1991].

$$w_i(n+1) = w_i(n) + \mu e(n)x(n-i) \quad (6.18)$$



Onde  $i=0,1,\dots,L-1$ . Este tipo de algoritmo requer somente  $2L$  multiplicações e adições, sendo, desta forma, considerado o algoritmo mais eficiente em termos de cálculos e armazenamento. A complexidade do algoritmo é considerada mais baixa quando comparada com outros algoritmos adaptativos tais como Kalman e o algoritmo recursivo de mínimo quadrado, RLS. Logo o algoritmo LMS é o mais amplamente utilizado para aplicações práticas.

### 6.2.3 Estruturas Adaptativas

Nesta secção serão introduzidas quatro aplicações de estruturas básicas que são utilizadas para diferentes aplicações na filtragem adaptativa.

- **Identificação de Sistemas**

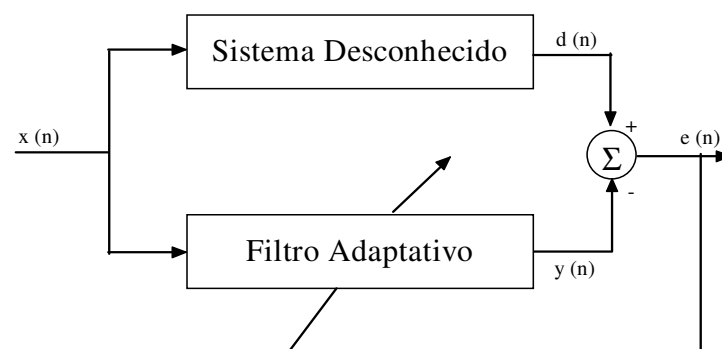


Figura 6.2.3.1 – Identificação de Sistema utilizando filtro adaptativo.

Como verificado na Figura 6.2.3.1, a mesma entrada  $x(n)$  excita tanto o sistema desconhecido quanto o filtro adaptativo, que se encontra em paralelo. O sinal de erro definido na Equação 6.4 retorna para o filtro adaptativo e é utilizado para atualizar os coeficientes do filtro adaptativo até que a saída seja  $y(n) = d(n)$ . Quando isto acontece o processo adaptativo finaliza e  $e(n)$  aproxima-se de zero.

- **Modelamento de Inversão**

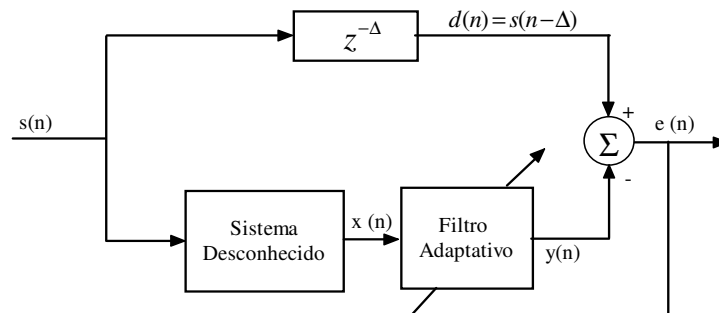


Figura 6.2.3.2 – Sistema de modelamento de inversão utilizando um filtro adaptativo.

Neste tipo de estrutura, Figura 6.2.3.2, o sistema desconhecido encontra-se em cascata com o filtro adaptativo. O sinal desejado  $d(n)$  é obtido através do sinal atrasado  $s(n)$  usando a unidade de atraso  $z^{-\Delta}$ , onde  $\Delta \approx L/2$ . O objetivo do atraso é para compensar o atraso de propagação ocorrido através do sistema desconhecido e do filtro adaptativo. O atraso permite que o filtro adaptativo convirja para um filtro causal, o qual é o inverso do sistema desconhecido. O filtro adaptativo equaliza o sistema desconhecido, ou canal, recuperando, conseqüentemente, a versão atrasada do sinal,  $s(n)$ , na saída do filtro,  $y(n)$ .

Caso  $s(n)$  tenha um espectro plano e um ruído pequeno, o filtro pode se adaptar para um modelo inverso preciso de um sistema desconhecido.

- **Cancelamento de ruído**

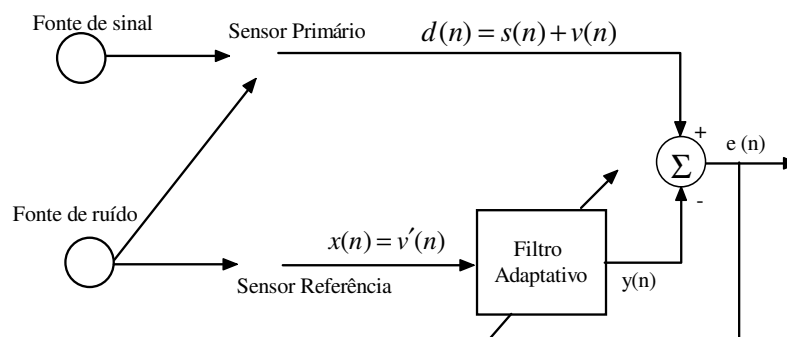


Figura 6.2.3.3 – Estrutura de um Cancelamento de Ruído.

O sinal primário,  $d(n)$ , adquirido pelo sensor primário, Figura 6.2.3.3, contém o sinal desejado,  $s(n)$ , e um ruído indesejável,  $v(n)$ . Para remover este ruído, o sensor de referência é utilizado para obter um ruído  $v'(n)$ , o qual é utilizado na entrada do filtro adaptativo  $x(n)$ . Como  $v'(n)$  e  $s(n)$  não são correlatados o filtro adaptativo pode ser somente ajustar o ruído referência  $v'(n)$  de tal forma que produza uma saída  $y(n)$  que se aproxime do ruído  $v(n)$ . Assim o componente  $v(n)$  é cancelado por  $y(n)$ , acarretando na convergência gradual do sinal  $e(n)$  aproximando-o para o sinal limpo  $s(n)$ .

- **Predição Adaptativa**

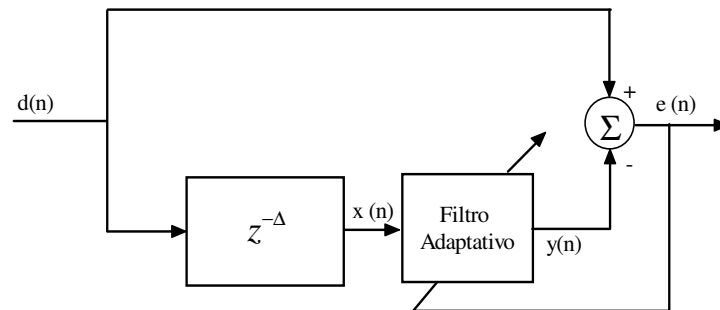


Figura 6.2.3.4 – Estrutura de Predição Adaptativa.

O sinal desejado,  $d(n)$ , é atrasado por  $\Delta$  amostras, logo  $x(n) = d(n - \Delta)$  é o sinal de entrada do filtro adaptativo, o qual adapta os coeficientes para minimizar o sinal de erro,  $e(n)$ . Este filtro adaptativo consiste em disponibilizar a melhor predição do valor presente dum sinal aleatório. O valor presente do sinal serve de resposta desejada para um filtro adaptativo. É verificado na Figura 6.2.3.4 que o sinal de referência do filtro consiste apenas nos valores passados do processo.

Em alguns casos o sinal de saída do filtro adaptativo serve como saída do sistema, nestes casos o filtro funciona como preditor. Em outros casos, o erro de predição é utilizado como saída, e assim o filtro funciona como um filtro de erro de predição.

## 6.3 EXEMPLOS DE FILTROS ADAPTATIVOS

### 6.3.1 Exemplo 1: FIR adaptativo para identificação de sistemas.

Esse exemplo implementa um filtro adaptativo utilizando o algoritmo do gradiente estocástico, ou algoritmo de erro médio quadrático, no código *ADAPTC.C*.

```
//ADAPTC.C
//Filtro Adaptativo utilizando LMS

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "sinais.h"

#define beta 0.01           // ganho de adaptação ( velocidade de convergência)
#define N 21               // ordem do filtro
#define Fs 8000            // frequência de amostragem
#define pi 3.1415926

float error[NS], Y_out[NS];

main()
{
    long I, T;
    float Y, E, D;
    float W[N+1] = {0.0};
    float X[N+1] = {0.0};

    for (T = 0; T < NS; T++)           // início do algoritmo adaptativo
    {
        X[0] = NOISE[T];               // nova amostra de ruído
        D = DESIRED[T];                // sinal desejado
        Y = 0;                         // saída do filtro igual a zero
        for (I = 0; I <= N; I++)
            Y += (W[I] * X[I]);         // cálculo da saída do filtro
        E = D - Y;                     // cálculo do sinal de erro

        error[T] = E;
        Y_out[T] = Y;

        for (I = N; I >= 0; I--)
        {
            W[I] = W[I] + (beta*E*X[I]); // atualização dos coeficientes do filtro
        }
    }
}
```

```

if (I != 0)
X[I] = X[I-1];          // atualização da amostra de dados
}

}

}

```

### Criação do projeto

A criação deste projeto no *Code Composer Studio™ IDE* requer a adição de arquivos composto na pasta de *FiltroAdapt* do cd fornecido; *ADAPTC.C*, *vector.asm*, *C6xdsk.cmd*, *rts6700.lib*, e *sinais.h*.

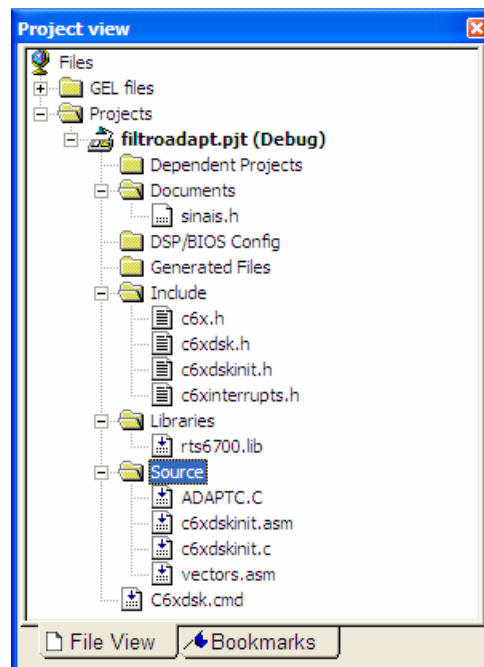


Figura 6.3.1.1 – Visualização do projeto.

Após a construção do projeto *filtroadapt.pjt*, gere o executável e carregue o programa utilizando as mesmas configurações dos parâmetros do compilador e do *linker* como fora feito nos exemplos do Tutorial 4.

## Construção gráfica no CCS

Para construir os gráficos no domínio do tempo e da frequência é necessário que você:

1. Selecione View → Graph → Time/Frequency. Ao abrir *Graph Property Dialog* faça as seguintes modificações:
  - *Start Address – upper display* → *DESIRED*, como pode ser verificado no código *fir.c*.
  - *Start Address – lower display* → *Y\_out*.
  - *Acquisition Buffer Size* → 128, representando o tamanho do *buffer*.
  - *DisplayData Size* → 40.
  - *DSP Data Size* → 32-bit float point.
  - *Sampling Rate (Hz)* → 8000.

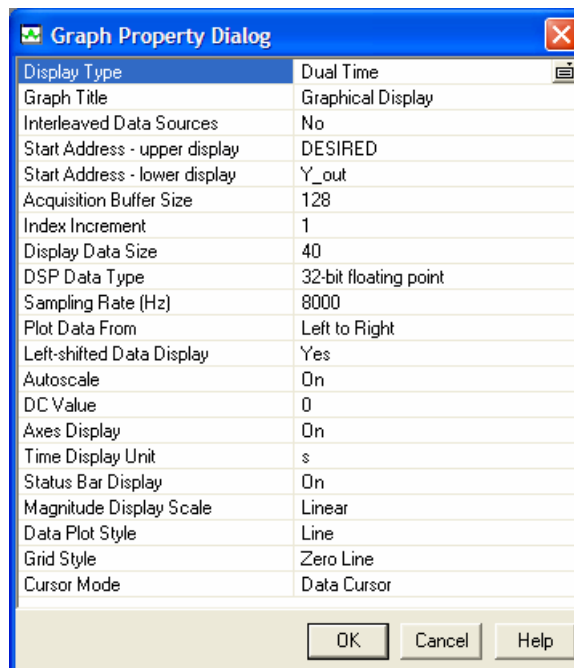


Figura 6.3.1.2 – Janela *Graph Property Dialog* com modificações.

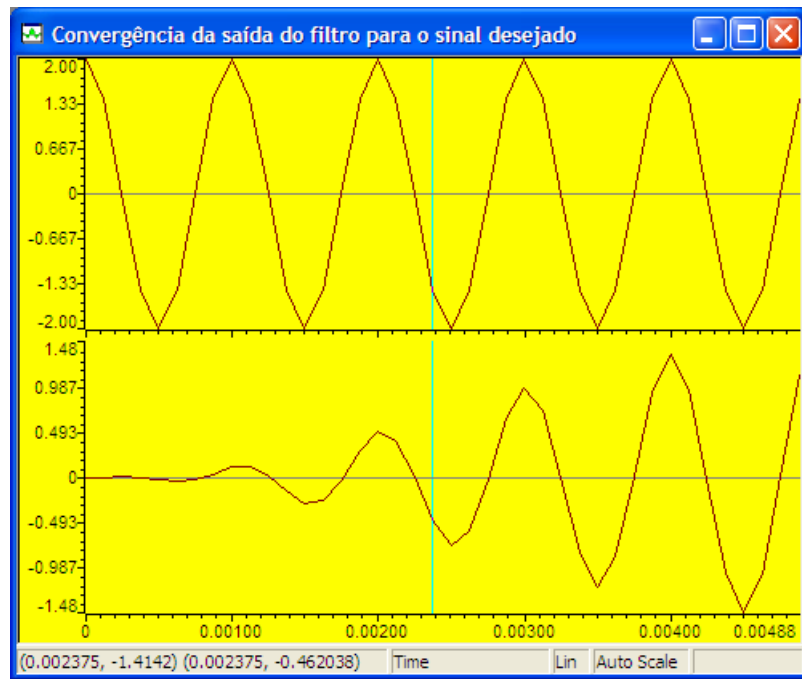


Figura 6.3.1.3 – Sinal de cima: Sinal desejado (DESIRE); Sinal de baixo saída do filtro (Y\_out).

## 7 - CONCLUSÃO

A motivação inicial desse trabalho era de abordar o tema DSP's sob a óptica da família TMS320C6x, desenvolvida pela Texas Instruments para dar suporte a disciplina de processamento digital de sinais (PDS).

Primeiramente partiu-se de conceitos gerais a respeito das características inerentes a qualquer DSP, em seguida fez-se uma abordagem sobre o pacote DSK, (DSP *starter kit*), que oferece as ferramentas de suporte de software e hardware da família de processadores que é alvo deste trabalho, em conjunto com práticas de laboratório que demonstraram os comandos necessários para a construção e implementação de projetos no ambiente oferecido pelo *Code Composer Studio* (CCS), ferramenta que acompanha o TMS320C6x.

Posteriormente, a partir da conceituação de alguns tópicos em processamento digital de sinais, tais como filtro FIR, filtros IIR, transformada discreta de Fourier (DFT), transformada rápida de Fourier, e filtros adaptativos, e em posse dos conhecimentos adquiridos no primeiro tutorial fora possível criar, construir e implementar práticas de laboratório.

Observou-se que a conclusão obtida ao final do desenvolvimento de cada experimento fora bastante satisfatória uma vez que apresentou ao leitor uma série de experimentos que consolidaram a teoria abordada em cada capítulo e ao mesmo tempo formaram a base para o desenvolvimento de outras aplicações em DSP's. Sendo assim, pode-se afirmar que o presente trabalho atende a uma necessidade de ordem acadêmica e ao mesmo tempo de mercado, uma vez que tais dispositivos constituem uma solução tecnológica presente em uma diversidade muito grande de dispositivos comerciais.

Por fim, trabalhos futuros poderiam desenvolver-se no sentido de adaptar os experimentos aqui apresentados para DPS's mais modernos, bem como incorporar novas práticas que poderiam surgir de necessidades específicas detectadas pelo professor da disciplina de Processamento Digital de Sinais.



## REFERÊNCIA BIBLIOGRÁFICA

- (Ahmed & Natarajan, 1983) – Ahmed, N., Natarajan, T. *Discrete-Time Signal and Systems*, Reston Publishing, Reston, VA, 1983.
- (Aziz, Sorensen & Van Der Spiegel, 1996) – Aziz, P. M., Sorensen, H. V., Van Der Spiegel, J. *An Overview of Sigma Delta Converters*, IEEE Signal Processing, Piscataway, Jan.1996.
- (Bateman & Yates, 1991) – Bateman, A., Yates, W. *Digital Signal Processing Design*, Computer Science Press, New York, 1991.
- (Bellanger, 1989) – Bellanger, M. *Digital Processing of Signals*, Wiley, New York, 1989.
- (Born, 2000) – Born, R.S., *Filtros Adaptativos Aplicados a Sinais Biomédicos*. Monografia de Curso de Bacharelado em Informática. Instituto de Física e Matemática da Universidade Federal de Pelotas, 2000.
- (Candy & Temes, 1992) – Candy, J. C., Temes, G. *Oversampling Delta-Sigma Data Converters: Theory, Design and Simulation*, IEEE Press, Piscataway, New Jersey, 1992.
- (Chassaing, 1999) – Chassaing, R. *Digital Signal Processing Laboratory Experiments Using C and the TMS320C31*, Wiley, New York, 1999.
- (Chassaing, 2000) – Chassaing, R. *DSP Applications Using C and the TMS320C6x DSK*. New York, NY: John Wiley. 2000.
- (Chassaing & Horning, 1990) – Chassaing, R., Horning, D.W. *Digital Signal Processing Laboratory Experiments Using C and the TMS320C5*, Wiley, New York, 1990.
- (Chen & Sorensen, 1997) – Chen, J., Sorensen, H. V. *A Digital Signal Processing Laboratory Using the TMS320C30*, Prentice Hall, Upper Saddle River, New Jersey, 1997.
- (Dahnoum, 2000) – Dahnoum, N. *DSP Implementation Using the TMS320C6x Processors*, Prentice Hall, Upper Saddle River, New Jersey, 2000.
- (Eyre, 2001) – Eyre, J. *The Newest Breed Trade off Speed, Energy Consumption, and cost to vie for an ever bigger piece of the action*, IEEE Spectrum, June 2001.
- (Gold & Radere, 1969) – Gold, B., Rader, C. M. *Digital Signal Processing*, Prentice Hall, Upper Saddle River, New Jersey, 1969.
- (Gray & Markel, 1973) – Gray, A. H., Markel, J.D. *Digital Lattice and Ladder Filter Synthesis*, IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-21, 1973, pp 491-500.
- (Gray & Markel, 1975) – Gray, A. H., Markel, J.D. *A Normalized Digital Filter Structure*, IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-23, 1975, pp 268-277.

(Jackson, 1969) – Jackson, L.B. *An Analysis of Limit Cycles due to Multiplicative Rounding in Recursive Digital Filters*, Proceeding of the 7<sup>th</sup> Allerton Conference on Circuit and System Theory, 1969, pp 69-78.

(Jackson, 1996) – Jackson, L.B. *Digital Filters and Signal Processing*, Kluwer Academic, Norwell, MA, 1996.

(Johnson, 1989) – Johnson, J. R. *Introduction to Digital Signal Processing*, Prentice Hall, Upper Saddle River, New Jersey, 1989.

(Hamming, 1983) – Hamming, R. W. *Digital Filters*, Prentice Hall, Upper Saddle River, New Jersey, 1983.

(Haykin, 1991) – Haykin, S. *Adaptive Filter Theory*, 2<sup>nd</sup>. Edition, Prentice-Hall. 1991.

(Kaiser, 1974) – Kaiser, J. F. *Some practical considerations in the realization of linear digital filters*, Proceedings of the 3<sup>rd</sup> Allerton Conference on Circuit System Theory, Oct. 1965, pp. 621-633.

(Kehtarnavaz & Keramat, 2001) – Kehtarnavaz, N., Keramat, M. *DSP System Design Using the TMS320C6000*, Prentice Hall, Upper Saddle River, New Jersey, 2001.

(Kehtarnavaz & Simsek, 2000) – Kehtarnavaz, N., Simsek, B. *C6x-Based Digital Signal Processing*, Prentice Hall, Upper Saddle River, New Jersey, 2000.

(Kernigan & Ritchie, 1998) – Kernigan, B.W., Ritchie, D. M. *The C Programming Language*, Prentice Hall, Upper Saddle River, New Jersey, 2001.

(Kuo & Gan, 2005) – Kuo, S.M., Gan, W. *Digital Signal Processors*. Prentice-Hall. 2005.

(Lynn & Fuerst, 1994) – Lynn, P. A., Fuerst, W. *Introduction Digital Signal Processing with Computer Application*, Wiley, New York, 1994.

(Lyons, 1997) – Lyons, R. G. *Understanding Digital Signal Processing*, Addison-Wesley, Reading, MA, 1996.

(Marven & Ewers, 1996) – Marven, C., Ewers, G. *A Simple Approach to Digital Signal Processing*, Wiley, New York, 1999.

(Matlab, 2004) - Mathworks®, *Using Matlab*. Version 7. The Mathworks, Inc., 2004.

(Mitra, 1998) – Mitra, S. K. *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, New York, 1998.

(Mitra & Kaiser, 1993) – Mitra, S. K., Kaiser, J. F. *Handbook for Digital Signal Processing*, Wiley, New York, 1993.

(Nascimento, 2005) – Nascimento, V.H. *Implementação de Filtros Adaptativos em Aritmética de Ponto Fixo II*. 2005.

(Norsworthy, Schreier & Temes, 1999) – Norsworthy, S., Schreier R., G. Temes, *Delta-Sigma Data Converters: Theory, Design and Simulation*, IEEE Press, Piscataway, New Jersey, 1997.

(Oppenheim & Schaffer, 1989) – Oppenheim, A. V., Schaffer, R. *Discrete-Time Signal Processing*, Prentice Hall, Upper Saddle River, New Jersey, 1989.

(Oppenheim, 1993) – Oppenheim, R.W. *Digital Signal Processing*, Prentice-Hall. 1993.

(Orfanidis, 1996) – Orfanidis, S. J. *Introduction to Signal Processing*, Prentice Hall, Upper Saddle River, New Jersey, 1993.

(Parks & Burrus, 1987) – Parks, T. W., Burrus, C. S. *Digital Filter Design*, Wiley, New York, 1997.

(Piedra & Fritsh, 1996) – Piedra, R. M., Fritsh, A. *Digital Signal Processing Comes of Age*, IEEE Spectrum, June 2001.

(Porat, 1997) – Porat, B. *A Course in Digital Signal Processing*, Wiley, New York, 1997.

(Proakis & Manolakis, 1996) – Proakis, J. G., Manolakis, D. G. *Digital Signal Processing: Principles, Algorithms and Application*, Prentice Hall, Upper Saddle River, New Jersey, 1996.

(Rabiner & Gold, 1975) – Rabiner, L.R., Gold, B. *Theory and Application of Digital Signal Processing*, Prentice Hall, Upper Saddle River, New Jersey, 1975.

(Solomon, 1988) – Solomon, C. W. *Switched-capacitor Filters*, IEEE Spectrum, June 1988.

(Stearns & David, 1993) – Stearns, S. D., David, R. A. *Signal Processing in Fortrans and C*, Prentice Hall, Upper Saddle River, New Jersey, 1993.

(Tretter, 1995) – Tretter, S. A. *Communication System Using DSP Algorithms*, Plenum Press, New York, 1995.

(TMS320C6000, 1999) – *TMS320C6000 Assembly Language Tools User's Guide*, SPRU198D, Texas Instruments, Dallas, Texas, 1999.

(TMS320C6000C, 1998) – *TMS320C6000 CPU and Instruction Set Reference Guide*, SPRU198D, Texas Instruments, Dallas, Texas, 1998.

(TMS320C6000, 2000) – *TMS320C6000 Programmer's Guide*, SPRU198D, Texas Instruments, Dallas, Texas, 2000.

(TMS320C6000O, 1999) – *TMS320C6000 Optimizing C Compiler User's Guide*, SPRU198D, Texas Instruments, Dallas, Texas, 1999.

(TMS320C6211, 2000) – *TMS320C6211 Fixed-point Digital Signal Processor-TMS320C6711 Floating-point Digital Signal Processor*, Texas Instruments, Dallas, Texas, 2000.

(Williams, 1986) – Williams, C. S. *Designing Digital Filters*, Prentice Hall, Upper Saddle River, New Jersey, 1986.

(Widrow & Stearns, 1986) – Widrow, B., Stearns, S. *Adaptive Signal Processing*, Prentice-Hall. 1985.

(Young, 1985) –Young, T. *Linear Systems and Digital Signal Processing*, Prentice Hall, Upper Saddle River, New Jersey, 1985.

## ANEXO I

- **Código Fonte para a geração de um ruído pseudo-randômico de 10 segundos.**

```
%ruído.m
%Programa para gerar ruído pseudo-randômico de 10 segundos

% Tempo de duração
T = 10;

% Frequência de amostragem
FS = 8000;

% Gera o ruído
y = rand(1,T*FS);

% Escreve o arquivo ruído.wav
wavwrite(y,8000,16,'ruído.wav')
```

- **Código Fonte para a implementação da resposta em frequência da saída do filtro FIR rejeita-faixa centrado em 2700Hz.**

```
%fir.2700
%Programa Gerador de Resposta em Frequência dos Filtros FIR

% Prepara o ambiente
clear all;
close all;
clc;

% Pontos para o filtro FIR rejeita-faixa centrado em 2700 Hz
entradas = [1.137 1.130 1.138 1.133 1.136 1.134 1.128 1.131 1.129
1.128 1.132 1.133 1.131 1.131];
saidas = [1.065 1.020 1.039 1.020 1.026 0.614 0.060 0.366 1.024
1.011 0.692 0.434 0.186 0.058];
```

```

frequencias = [1065 1302 1764 2024 2358 2488 2720 2870 3050 3300
3450 3550 3650 3910];

% Resposta em frequência do filtro
Ganho = 10*log10(saidas./entradas);
xx = min(frequencias):max(frequencias);
yy = pchip(frequencias,Ganho,xx);

plot(frequencias,Ganho,'ro',xx,yy)
ylabel('Magnitude (dB)')
xlabel('Frequência (Hz)')
title('Resposta em frequência')
grid on

legend('Pontos experimentais', 'Interpolação')

```

- **Código Fonte para a implementação da resposta em frequência da saída do filtro FIR passa-faixa centrado em 1750Hz.**

```

%fir.1750
%Programa Gerador de Reposta em Frequência dos Filtros FIR

% Pontos para o filtro FIR passa-faixa centrado em 1750 Hz
entradas = [1.136 1.137 1.137 1.135 1.131 1.136 1.134 1.128];
saidas = [0.041 0.201 0.893 1.012 0.639 0.397 0.113 0.047];
frequencias = [1016 1431 1572 1845 1980 2024 2096 2860];

% Resposta em frequência do filtro
figure(2)
Ganho = 10*log10(saidas./entradas);
xx = min(frequencias):max(frequencias);
yy = pchip(frequencias,Ganho,xx);

plot(frequencias,Ganho,'ro',xx,yy)
ylabel('Magnitude (dB)')
xlabel('Frequência (Hz)')

```

```

title('Resposta em frequência')
grid on

legend('Pontos experimentais', 'Interpolação')

```

- **Código Fonte para a implementação da resposta em frequência da saída do filtro IIR passa-baixa centrado em 1750Hz.**

```

%iir.2000
%Programa Gerador de Reposta em Frequência dos Filtros FIR

% Pontos para o filtro IIR passa-baixas frequencia de corte 2000
Hz
entradas = [1.132 1.137 1.135 1.134 1.134 1.134 1.135 1.134 1.133
1.131];
saidas = [1.049 1.010 1.063 0.939 0.716 0.481 0.263 0.186 0.063
0.053];
frequencias = [1175 1342 1595 1862 2004 2119 2227 2304 2639 3160];

% Resposta em frequência do filtro
figure(3)
Ganho = 10*log10(saidas./entradas);
xx = min(frequencias):max(frequencias);
yy = pchip(frequencias,Ganho,xx);

plot(frequencias,Ganho,'ro',xx,yy)
ylabel('Magnitude (dB)')
xlabel('Frequência (Hz)')
title('Resposta em frequência')
grid on

legend('Pontos experimentais', 'Interpolação')

```

- **Código Fonte para a geração do espectro do sinal corrompido por dois sinais senoidais centrados em 900Hz e 2700Hz, e a FFT do sinal recuperado.**

```
%Programa Gerador do Espectro de voz.

% Prepara o ambiente
clear all;
close all;
clc;

% Abre o arquivo com o som ruidoso
[CV,FS1,NBITS1] = wavread('corruptvoice.wav');
N1 = length(CV);

% Abre o arquivo com o som filtrado
[FV,FS2,NBITS2] = wavread('filteredvoice.wav');
N2 = length(FV);

% Faz a FFT do sinal original
CVFFT = fft(CV);
freq = linspace(0, (N1-1)*FS1/N1, N1);
plot(freq(1:N1/2), abs(CVFFT(1:N1/2)));
ylabel('Magnitude (dB)')
xlabel('Frequência (Hz)')
title('Sinal Original')
grid on

% Faz a FFT do sinal filtrado
FVFFT = fft(FV);
freq = linspace(0, (N2-1)*FS2/N2, N2);
figure
plot(freq(1:N2/2), abs(FVFFT(1:N2/2)));
ylabel('Magnitude (dB)')
xlabel('Frequência (Hz)')
title('Sinal Filtrado')
grid on
```



- **Código Fonte do comportamento teórico do sinal do tipo**

$x(n) = 1000 \cdot \cos\left(\frac{\pi}{4}n\right)$  de comprimento  $N = 8$ .

```
%Prepara o ambiente
clear all
close all

% Número de amostras
N = 8;

% Frequencia de amostragem
FS = 8000; % Hz

% Ângulos em radianos
theta = 0:2*pi/N:2*pi;

% Função x(n)
x = 1000*cos(theta(1:N));
stem([0:N-1],x)
grid on
xlabel('n')
ylabel('x(n)')
title('Função x(n) = 1000*cos(pi*n/4)')

% FFT do sinal x
figure
xFFT = fft(x);

% Ajuste do eixo das frequencias
freq = linspace(0, (N-1)*FS/N, N);

% Parte real
stem(freq, real(xFFT));
grid on
xlabel('Frequência (Hz)')
ylabel('outRe')
title('Valores da componente real da DFT')
figure

% Parte imaginária
stem(freq, imag(xFFT));
grid on
xlabel('Frequência (Hz)')
ylabel('outIm')
title('Valores da componente imaginária da DFT')
```

- **Código Fonte do comportamento teórico do sinal do tipo  $x(n) = 1000 * \sin\left(\frac{\pi}{5}n\right)$  de comprimento  $N = 20$ .**

```
%Prepara o ambiente
clear all
close all

% Número de amostras
N = 20;

% Frequencia de amostragem
FS = 8000; % Hz

% Ângulos em radianos
theta = 0:(4*pi/N):4*pi;

% Função x(n)
x = 1000*sin(theta(1:N));
stem([0:N-1],x)
grid on
xlabel('n')
ylabel('x(n)')
title('Função x(n) = 1000*sin(pi*n/5)')

% FFT do sinal x
xFFT = fft(x);

% Ajuste do eixo das frequencias
freq = linspace(0, (N-1)*FS/N, N);

% Parte real
figure
stem(freq, real(xFFT));
grid on
xlabel('Frequência (Hz)')
ylabel('outRe')
title('Valores da componente real da DFT')

% Parte imaginária
figure
stem(freq, imag(xFFT));
grid on
xlabel('Frequência (Hz)')
ylabel('outIm')
title('Valores da componente imaginária da DFT')
```

- **Código Fonte do comportamento teórico do sinal do tipo  $x(n) = 1000 * \sin\left(\frac{\pi}{8}n\right)$  de comprimento  $N = 32$ .**

```
%Prepara o ambiente
clear all
close all

% Número de amostras
N = 32;

% Frequencia de amostragem
FS = 8000; % Hz

% Ângulos em radianos
theta = 0:(4*pi/N):4*pi;

% Função x(n)
x = 1000*sin(theta(1:N));
stem([0:N-1],x)
grid on
xlabel('n')
ylabel('x(n)')
title('Função x(n) = 1000*sin(pi*n/8)')

% FFT do sinal x
xFFT = fft(x);

% Ajuste do eixo das frequencias
freq = linspace(0, (N-1)*FS/N, N);

% Parte real
figure
stem(freq, real(xFFT));
grid on
xlabel('Frequência (Hz)')
ylabel('outRe')
title('Valores da componente real da DFT')

% Parte imaginária
figure
stem(freq, imag(xFFT));
grid on
xlabel('Frequência (Hz)')
ylabel('outIm')
title('Valores da componente imaginária da DFT')
```