COPPE
Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia
UFRJ

ESTIMAÇÃO DA AGRESSIVIDADE DA CAVITAÇÃO EM TURBINAS HIDROELÉTRICAS BASEADA EM MODELAGEM CICLOESTACIONÁRIA

Rafael Linhares Marinho

Tese de Doutorado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Doutor em Engenharia Elétrica.

Orientador: Fernando Antônio Pinto Barúqui

Rio de Janeiro
Junho de 2015

ESTIMAÇÃO DA AGRESSIVIDADE DA CAVITAÇÃO EM TURBINAS HIDROELÉTRICAS BASEADA EM MODELAGEM CICLOESTACIONÁRIA

Rafael Linhares Marinho

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

Examinada por:

_____

Prof. Fernando Antônio Pinto Barúqui, D.Sc.

_____

Prof. Antonio Petraglia, Ph.D.

_____

Prof. Carlos Augusto Duque, D.Sc.

_____

Prof. João Baptista de Oliveira e Souza Filho, D.Sc.

_____

Prof. José Gabriel Rodriguez Carneiro Gomes, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
JUNHO DE 2015

Marinho, Rafael Linhares

Estimação da Agressividade da Cavitação em Turbinas Hidroelétricas Baseada em Modelagem Cicloestacionária/ Rafael Linhares Marinho. – Rio de Janeiro: UFRJ/COPPE, 2015.

IX, 235p.: il.; 29,7 cm.

Orientador: Fernando Antônio Pinto Barúqui

Tese (doutorado) – UFRJ/ COPPE/ Programa de Engenharia Elétrica, 2015.

Referências Bibliográficas: p. 122-128.

1. Cavitação Erosiva. 2. Turbinas Hidráulicas. 3. Cicloestacionariedade. I. Barúqui, Fernando Antônio Pinto. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Titulo.

AGRADECIMENTOS

Ninguém faz nada sozinho. Ninguém. Nem mesmo coisas simples como andar, falar, ler ou escrever. Em algum momento alguém nos ensinou a ler e a escrever. Em algum momento alguém serviu de exemplo e inspiração para que começássemos a andar e a falar. Assim também é quando o homem faz a ciência progredir.

O progresso da ciência é como a construção de uma grande catedral: envolve o trabalho de muitas mãos e o suor de muitos rostos. Algumas mãos materializam o alicerce, e outras os pavimentos de uma obra que parece ser sempre inacabada.

Há também aquelas mãos que servem o alimento e a água para as mãos que trabalham na construção, e para os rostos que suam. Assim também é o progresso da ciência: muitas vezes são necessárias horas e horas de elucubrações e estudos com muito afinco. Muitas vezes porém, tudo o que é preciso é uma palavra de incentivo, ou uma injeção de bom ânimo, que são tão essenciais quanto a água e o alimento para as mãos construtoras, e que dão resultados em minutos ou até segundos. É o alimento para o pensamento, que nutre as ideias e as faz se desenvolver. É a água de arrefecimento que impede o excesso de calor de nos fazer parar de construir.

É impossível construir uma parede sequer sem um alicerce, e sem água e alimento. Da mesma forma, seria impossível para mim assentar mais um tijolo na construção dessa grande catedral, que é o progresso da ciência, sem meus inúmeros colaboradores. Para eles, que são todas essas mãos que participaram diretamente ou indiretamente, deixo minhas palavras de gratidão:

Ao meu professor orientador, **Fernando Antônio Pinto Barúqui**, que além de ajudar no aperfeiçoamento desta obra, foi grande colaborador na hora da aquisição de dados.

Aos engenheiros **Milton Marcelo Klavin** e **Mansur**, que acompanharam e apoiaram o processo de aquisição de dados.

A todos os professores, servidores administrativos e meus alunos do **IFRJ**, que apoiaram o meu engajamento em uma pesquisa de doutorado, em especial aos

professores **Elton Flach** e **Marcelo Bittencourt de Lacerda** e aos servidores **Paulo, Batista e Milton.**

À minha mãe, **Alice Linhares Marinho** e avó **Maria de Lourdes Linhares** (in memorian).

Ao **prof. Luiz Fernando Aramis de Mattos** (in memorian), e toda equipe de professores do Macedão (extinto).

Aos amigos **Alberto Paletta**, **Fabrizio Paletta**, **Igor de Souza François**, **Armando Tavares**, **Júlio Cézar Gomes Ardente, Cláudia de Souza e Silva, Marilucia Rocha** e **Rosane Sarmento Figueiredo**.

Aos colaboradores **Suzana Carolina Lima**, **Marcelino Lago**, **Leandro Manhanini Simião**, **Ana Lúcia**, **Paulo Barbosa Sousa**, **Carla Ferreira da Silveira**, **Bruno Sercundo, Andrea Dias Alves** e **Pablo Enrique Pua Techera.**

Aos alunos e ex-alunos **Gabriel Morgado, Bruna Reis, Caio Muzzi e Marialda.**

É claro que eu não poderia esquecer de agradecer ao **Criador**, que com Seu **Espírito Santo** nos traz iluminação e entendimento, fundamentais na elaboração de soluções para problemas complexos. Não é incomum ver ou ouvir pessoas pedindo ajuda a**O Criador** ou a**O Salvador**, ao se depararem com problemas aparentemente sem solução. Comigo não foi diferente. Por fim, deixo uma frase de reflexão de um dos meus autores preferidos.

"A fé desempenha em nossa vida um papel mais importante do que supomos, e é o que nos permite fazer mais do que pretendemos. Creio que aí está o elemento precursor de nossas ideias. Sem a fé não se teriam elaborado jamais hipóteses e teorias, nem se teriam inventado as ciências ou as matemáticas. Estou convencido de que a fé é um prolongamento do espírito: negar a fé é condenar-se e condenar o espírito que engendra todas as forças criadoras de que dispomos."

Sir Charles Spencer Chaplin

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

ESTIMAÇÃO DA AGRESSIVIDADE DA CAVITAÇÃO EM TURBINAS HIDROELÉTRICAS BASEADA EM MODELAGEM CICLOESTACIONÁRIA

Rafael Linhares Marinho

Junho/2015

Orientador: Fernando Antônio Pinto Barúqui

Programa: Engenharia Elétrica

O presente trabalho propõe uma nova metodologia para detecção e identificação da cavitação erosiva em turbinas de geração hidroelétrica. A metodologia baseia-se na modelagem dos sinais vibracionais induzidos pela cavitação como processos cicloestacionários. Diferentes tipos de cavitação afetam diferentes partes da turbina, e induzem assinaturas vibracionais que podem ser usadas para a identificação e localização da cavitação. Adicionalmente, a agressividade da cavitação, que é seu poder erosivo, é estimada com base na potência medida do sinal vibracional. Acelerômetros de alta frequência foram usados para captar os sinais em duas turbinas reais durante operação normal. A metodologia foi implementada em software e um hardware dedicado foi desenvolvido para a execução no local. Um conjunto de sinais foi sintetizado conforme a modelagem cicloestacionária, e foi usado para validação da metodologia proposta. Resultados obtidos com os sinais reais foram semelhantes aos obtidos com sinais sintetizados, e corroboraram a viabilidade do emprego da metodologia em sistemas de monitoramento da cavitação.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

ESTIMATION OF CAVITATION AGGRESSIVENESS IN HYDRO TURBINES BASED ON CYCLOSTATIONARY MODELING

Rafael Linhares Marinho

June/2015

Advisor: Fernando Antônio Pinto Barúqui

Department: Electrical Engineering

This work proposes a new methodology of detection and identification of erosive cavitation in hydro turbines. This methodology is based on cyclostationary modeling of the cavitation induced vibrational signals. Different cavitation types cause damage to different turbine parts and induce different vibrational signatures, which can be employed to identify and locate the cavitation. Additionally, the cavitation aggressiveness can be estimated based on the measured power of vibrational signal. High frequency accelerometers picked up the signals from two real turbines under normal operation. The methodology was implemented in software and a specific hardware was developed to run the software locally. Signals were synthesized in accord with the cyclostationary modeling and employed to validate the proposed methodology. Results obtained from real signals were similar to the ones obtained from synthetic signals, and corroborate the feasibility of this methodology in cavitation monitoring systems.

SUMÁRIO

1 INTRODUÇÃO

O presente trabalho tem como objetivo apresentar os resultados de uma pesquisa de tese de doutorado, que tem como escopo o desenvolvimento de uma ferramenta destinada à detecção e monitoramento da cavitação erosiva em turbinas de geração hidroelétrica.

A cavitação é um fenômeno bastante complexo, e causa prejuízos de enorme magnitude em diversas áreas da indústria. As máquinas hidráulicas da indústria hidroelétrica constituem um dos exemplos onde a cavitação é notoriamente prejudicial, e portanto, onde um melhor entendimento do fenômeno seria de grande valor na elaboração de uma estratégia de redução de custos devidos à cavitação erosiva.

Na data da escrita deste documento, o Brasil está passando por séria crise hídrica e portanto, uma ferramenta que possibilite redução do custo da energia elétrica gerada bem como economia de água, certamente constitui vantagem no enfrentamento da crise.

Uma parada de manutenção corretiva devido à erosão pode custar milhares de reais e durar várias semanas, constituindo prejuízo para as companhias geradoras não somente por ter uma máquina geradora parada, mas também por ter outra(s) máquina(s) do Sistema Interligado Nacional (SIN) possivelmente em sobrecarga para suprir a ausência daquela em manutenção.

Atualmente as paradas de manutenção são feitas em intervalos programados de acordo com o que se estima ser o melhor momento para efetuar uma manutenção, baseando-se nos históricos de perda de massa de cada unidade. Como estas paradas são feitas por uma estimativa, não é incomum uma unidade parar no intervalo programado sem apresentar um nível de erosão que justifique a parada, ou ao contrário, parar quando a perda de massa já está além do que seria economicamente viável reparar. Nota-se que para minimizar os custos devido à cavitação erosiva, é necessário determinar o melhor momento para realizar a manutenção, mas infelizmente até hoje não há meios de determinar a real perda de massa de uma turbina.

A cavitação além de causar perda de massa provoca outros efeitos indesejáveis, como por exemplo, anomalias operacionais, ruídos acústicos e vibrações.

1.1 CONTRIBUIÇÕES DESTA OBRA

Como contribuições inovadoras desta obra, o autor pode citar que, inicialmente, foi reunida uma quantidade considerável de informações sobre o fenômeno da cavitação hidrodinâmica, principalmente no âmbito das turbinas hidroelétricas. A partir dessas informações reunidas, foi possível construir um corpo de conhecimento, o qual é indispensável para o bom entendimento dos mecanismos de produção da cavitação. Este conhecimento foi construído a partir de descrições físicas e fenomenológicas da cavitação, e habilitou o autor a propor um modelo que descreve as vibrações induzidas pela cavitação nas turbinas hidroelétricas, ainda que este modelo seja simplificado.

Estas vibrações produzidas pela cavitação, de acordo com o modelo proposto, constituem os sinais vibracionais de interesse e transportam as informações sobre a cavitação. Tais sinais vibracionais foram caracterizados pelo autor, em um segundo momento de sua pesquisa, como sinais cicloestacionários de segunda ordem. Esta caracterização permitiu que o autor estabelecesse correspondências entre propriedades e características (do ponto de vista da cicloestacionariedade) do sinal vibracional e as características físicas da cavitação que provoca tal sinal. Além disso, o fato de o sinal produzido pela cavitação ser caracterizado como cicloestacionário permitiu o desenvolvimento de uma metodologia específica para a o diagnóstico da cavitação erosiva em turbinas. Esta metodologia tem por base o emprego de ferramentas de análise cicloestacionária (que surgiram há décadas no ramo das telecomunicações, mas cujo emprego no processamento de sinais mecânicos é ainda considerado pobre). Adicionalmente, o emprego de ferramentas de análise cicloestacionária possibilita a discriminação de várias formas de ruído, os quais ocorrem muito frequentemente no ambiente industrial. Finalmente, um diagnóstico qualitativo pode ser feito com base nas ferramentas empregadas, e um diagnóstico quantitativo da “intensidade da cavitação”, foi proposto pelo autor, com base na identificação das componentes relevantes do sinal vibracional, através da similaridade espectral com outras componentes chave.

A metodologia desenvolvida não serve para determinar diretamente a real perda de massa de uma turbina, mas serve como uma ferramenta para diagnosticar a cavitação em turbinas a partir da análise das vibrações induzidas. Entenda-se diagnosticar aqui como identificar o(s) *tipo*(*s*) de cavitação e a(s) sua(s) *localização*(*ões*), bem como

estimar o(s) seu(s) *poder*(*es*) *erosivo*(*s*). A análise das vibrações induzidas pode ser empregada de forma totalmente não invasiva e sem interferir com o funcionamento normal da unidade geradora.

Apesar de o diagnóstico não determinar diretamente a real perda de massa, as estimativas do poder erosivo fornecidas por ele podem ser usadas como informações auxiliares, ou como ponto de partida em pesquisa complementar, cujo objetivo seja a determinação da real perda de massa. Adicionalmente, este diagnóstico é útil na investigação do fenômeno da cavitação em si, ou na comparação entre o desempenho de uma turbina antes e após uma manutenção corretiva, avaliando a eficácia da manutenção. Finalmente, por ser uma metodologia que não interfere de forma alguma com a operação das unidades geradoras, e também por ser uma metodologia capaz de detectar e medir o poder erosivo da cavitação, ainda que este seja muito pequeno, a metodologia desenvolvida é apropriada para promover a otimização na operação das turbinas, contribuindo para a economia de água e redução no custo da energia gerada.

Cabe ressaltar que a metodologia foi implementada em *software,* e um *hardware* específico foi desenvolvido. O custo total do equipamento é consideravelmente baixo, e os sensores empregados são pouco numerosos e também de custo mediano. Estas características contribuem para a criação de equipamentos de diagnóstico da cavitação erosiva, de tal forma que seja viável a implantação desta instrumentação em larga escala. A metodologia proposta foi testada inicialmente com sinais sintéticos (um simulador de sinais foi desenvolvido com base na modelagem proposta). Finalmente, a metodologia foi testada também em duas turbinas reais (com níveis de cavitação notavelmente baixos), e se mostrou apropriada para a detecção e monitoramento da cavitação erosiva em turbinas hidroelétricas.

1.1.1 Organização deste trabalho

Ainda neste capítulo introdutório serão abordados de forma sucinta alguns conceitos básicos que são de fundamental importância para o desenvolvimento da metodologia. O leitor que desejar uma abordagem mais profunda sobre o fenômeno da cavitação pode encontrar no Apêndice A – Cavitação. Uma descrição mais detalhada

sobre tipos e funcionamento de turbinas pode ser vista no Apêndice B – Turbinas hidráulicas. Os principais mecanismos de erosão causados pela cavitação estão descritos no Apêndice C – Erosão. Serão abordadas também demandas em todo mundo no sentido de detectar e monitorar a cavitação, com um levantamento das metodologias desenvolvidas até hoje.

No capítulo 2 será apresentada a modelagem do sinal vibracional de uma turbina com cavitação erosiva como um *processo cicloestacionário* (ANTONI, 2009). A modelagem desenvolvida é totalmente baseada no entendimento da dinâmica dos escoamentos e das turbinas, e cada um dos principais tipos de cavitação erosiva será caracterizado pela influência que causa no padrão vibracional da turbina.

O capítulo 3 apresenta o desenvolvimento teórico da metodologia para a identificação e localização da cavitação erosiva, e estimação do seu poder erosivo.

No capítulo 4 serão descritos os experimentos de aquisição de sinais com sensores apropriados, a implementação da metodologia desenvolvida em *software*, e o desenvolvimento de um sintetizador de sinais a fim de produzir dados para a validação da metodologia.

O capítulo 5 apresenta os resultados obtidos com os sinais reais e uma discussão dos resultados, e no capítulo 6 serão mostradas as conclusões desta pesquisa e sugestões para futuros aperfeiçoamentos.

1.2 O FENÔMENO DA CAVITAÇÃO HIDRODINÂMICA

A cavitação hidrodinâmica foi descrita pelo professor Knapp da Universidade da Califórnia (EUA), em seu livro (KNAPP *et al.*, 1970) como "*an unpleasant hydrodynamic phenomenon, the harmful effects of which often creates serious difficulties in solving many scientific and engineering problems*". Basicamente, a cavitação é um fenômeno físico, que consiste em uma maneira alternativa à ebulição de converter uma substância de seu estado líquido para seu estado de vapor.

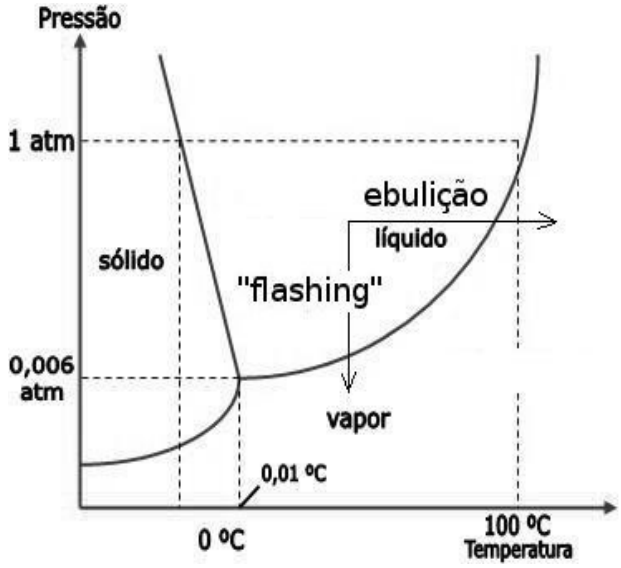


Figura 1 - Diagrama de fases da água e suas transformações entre líquido e vapor.

Fonte: Elaboração própria.

A figura 1 mostra o diagrama de fases da água e suas linhas de equilíbrio entre as fases. A ebulição, representada no diagrama como a seta horizontal nos mostra que, ao aquecer uma certa quantidade de água, lentamente esta se transforma em vapor na medida que a energia térmica é fornecida ao sistema, e cavidades de vapor se formam no interior da massa líquida. A transformação inversa também é possível, pois ao resfriar o sistema bifásico água/vapor, o vapor é convertido em líquido e o sistema se torna de uma só fase. As velocidades destas transformações estão limitadas pela potência térmica fornecida ou retirada da massa de água.

Há uma outra possibilidade de converter água no estado líquido em vapor, representada pela seta vertical no diagrama de fases, mas que envolve o abaixamento da pressão a valores inferiores à pressão de vapor da água. Neste caso a transformação é tão rápida que é chamada de evaporação *flash*, ou *flashing*. A transformação inversa também é possível com a elevação de pressão no sistema bifásico, o que faz as cavidades preenchidas com vapor se tornarem instáveis e implodirem violentamente. Tipicamente, o processo de implosão dura alguns microssegundos (ESCALER *et al.*, 2006a), e por ser tão rápido, é considerado adiabático, o que resulta em temperaturas nos pontos centrais das implosões da ordem de milhares de graus Celsius e pressões de milhares de atmosferas. As mudanças de estado físico provocadas pela cavitação

ocorrem em temperatura constante, e de modo geral, temperaturas mais baixas dificultam a formação de cavidades de vapor.

Atualmente existem várias teorias a respeito do mecanismo de destruição dos materiais expostos à violência dessas implosões. As elevadas temperaturas nos pontos centrais das implosões deslocam o equilíbrio termodinâmico de muitos materiais usados na indústria, como as ligas de ferro, e facilitam a oxidação (GENTIL, 1982). É um mecanismo de corrosão facilitado pela cavitação.

As altas pressões e temperaturas também são capazes de dissociar a água em hidrogênio e oxigênio, caso em que o hidrogênio pode se difundir entre os grãos do aço e formar bolsas de gás no interior do metal. O resultado é conhecido como *hydrogen embrittlement* (figura 2).

Figura 2 - Trinca no aço induzida por hidrogênio.

Fonte:<http://en.wikipedia.org/wiki/File:Steel-with-Hydrogen-Induced-Cracks-01.jpg>. Acesso em 15/04/2015.

Figura 3 - Corrosão alveolar em aço inoxidável.

Fonte:<http://www.ndt.net/article/v07n07/ginzel_r/ginzel_r.htm>. Acesso em 15/04/2015.

Outro processo é o denominado corrosão alveolar, ou *pitting*. Neste caso, as altas pressões produzidas pelas implosões são suficientes para remover a camada de

passivação do aço inoxidável. Isto expõe material quimicamente ativo logo abaixo da superfície ao meio corrosivo, e nova camada de passivação é formada no local. Com a repetição deste ciclo, há a formação de um padrão de destruição em pequenos alvéolos (figura 3).

O mecanismo de erosão mais aceito atualmente é um mecanismo puramente físico: a força dos impactos é grande o suficiente para causar deformações plásticas no metal, o que produz o endurecimento e o *encruamento* do aço. Também há um aumento na densidade de *deslocações* do material, e uma diminuição das suas mobilidades, o que torna o material quebradiço. Com a repetição das deformações, o material acaba por sofrer falha por um mecanismo semelhante à *fadiga de contato*, e uma trinca é nucleada. Com a progressão da trinca, há perda de partes macroscópicas do material (figura 4). Uma característica peculiar da erosão devido à cavitação é que a superfície afetada tem aspecto áspero e poroso, lembrando neve.

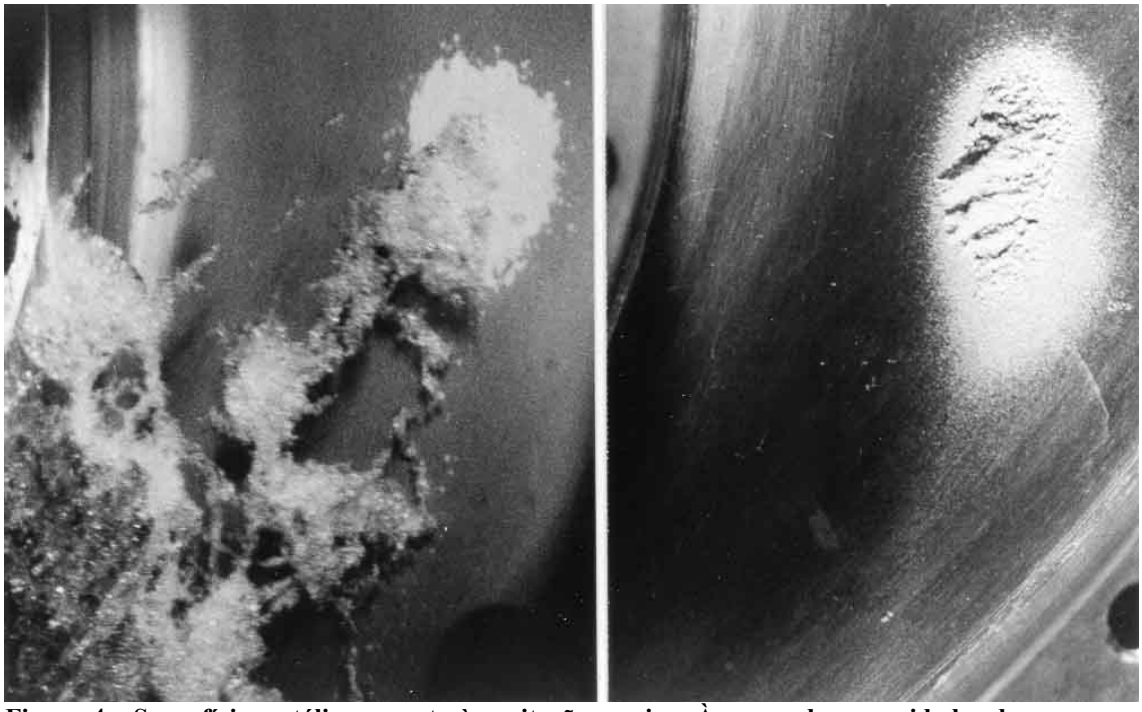


Figura 4 - Superfície metálica exposta à cavitação erosiva. À esquerda as cavidades de vapor em contato com a superfície. À direita a mesma superfície seca mostrando padrão peculiar da erosão.

Fonte:(KHURANA *et al.*, 2012)

Não obstante, pode haver ainda o efeito sinergístico entre a perda de massa por fadiga e a corrosão propriamente dita, o que além de tornar mais grave os danos, torna

também muito mais complexa a determinação da perda de massa real. Este e outros fatores expostos no Apêndice C – Erosão – levaram o autor a decidir não buscar a determinação da perda de massa em si, mas ter como objetivo estimar a *agressividade da cavitação* (CHOI *et al.*, 2012; PETKOVŠEK, DULAR, 2013).

1.3 A AGRESSIVIDADE DA CAVITAÇÃO E A DEMANDA DE SUA ESTIMAÇÃO A NÍVEL NACIONAL E INTERNACIONAL

A agressividade da cavitação (BERCHICHE, 2002) é um termo cunhado na literatura especializada, e que tem o significado de poder erosivo da cavitação. A quantidade real de massa perdida por uma turbina está *indiretamente* relacionada à agressividade da cavitação.

A erosão é somente um dos efeitos nocivos da cavitação hidrodinâmica que podem surgir nas turbinas hidráulicas. Não é incomum uma turbina que opere com cavitação apresentar emissão sonora em níveis anormalmente altos, queda no rendimento energético, anomalias operacionais ou vibrações em níveis perigosos. Ademais, uma turbina que sofre com cavitação pode apresentar erosão ou não, pois nem toda cavitação é erosiva. A cavitação pode surgir em uma turbina de diversas formas, caracterizando diferentes *tipos* de cavitação (ESCALER *et al.*, 2006a). Cada tipo de cavitação tem um mecanismo de formação diferente, e apresenta também efeitos nocivos diferenciados, como por exemplo, alguns tipos são mais erosivos e outros são mais ruidosos.

A cavitação em si por sua vez é um fenômeno ainda pouco entendido pela ciência. Não é incomum encontrar duas turbinas operando em condições supostamente iguais, mas com tipos e agressividades de cavitação completamente diferentes (ESCALER *et al.*, 2004). Além disso, apesar de os fabricantes de turbinas limitarem as faixas de potência gerada, vazão e pressão para forçar a operação com níveis *aceitáveis* de cavitação (CLINE *et al.*, 2009), ela causa prejuízos em 65% dos casos, segundo um levantamento feito aqui no Brasil (CALAINHO, HORTA, 1999).

O desafio encontrado atualmente pelas empresas geradoras nacionais, e também pelas empresas geradoras de países que têm a geração de energia baseada em

hidroelétricas (por exemplo Noruega e Canadá), é não somente detectar a cavitação, mas também identificar quando a cavitação é erosiva, e quanto de agressividade está presente. Atualmente a cavitação já pode ser detectada e seu tipo identificado com sucesso (ESCALER *et al.*, 2006a), mas ainda não há meios confiáveis e não invasivos de determinar se a cavitação é erosiva, e qual é o seu poder erosivo.

1.4 METODOLOGIAS DESENVOLVIDAS ATÉ O PRESENTE

Vários pesquisadores já realizaram esforços no sentido de detectar e monitorar a cavitação erosiva, empregando diferentes técnicas e sensores, como acelerômetros de alta frequência, sensores de emissão acústica e transdutores de pressão *flush-mounted*. Algumas técnicas empregadas são não invasivas e não envolvem alterar a operação normal da unidade geradora, e portanto são elegíveis para um monitoramento *online* da agressividade da cavitação. As técnicas mais avançadas envolvem o uso de muitos sensores (aproximadamente 20 sensores), e mais caros (normalmente são sensores de emissão acústica com frequência de corte acima de 1 MHz), e também exigem variar o ponto de operação da turbina em análise (BAJIC, 2003).

1.4.1 Metodologias não invasivas

A mais simples metodologia empregada para detectar a cavitação é a retificação em onda completa dos sinais de vibração induzidos pela cavitação erosiva, provenientes de acelerômetros de alta frequência e filtrados por filtros passa faixa (FARHAT, BOURDON, 1999). Apesar de nenhuma correlação direta entre a potência dos sinais retificados e a agressividade da cavitação ter sido estabelecida, esta metodologia revelou os locais mais apropriados para a instalação de acelerômetros. Também foram identificados sinais em frequências chave que modulam em amplitude o sinal aleatório da cavitação. Devido a não similaridades entre os protótipos de turbinas a agressividade da cavitação não pode ser estimada, mas foi feita uma comparação entre a potência dos sinais vibracionais antes e após uma parada de manutenção corretiva.

Um aprimoramento da metodologia anterior foi realizar a extração do envelope

dos sinais vibracionais previamente filtrados usando a transformada de Hilbert discreta (ESCALER *et al.*, 2006a, 2006b). Os autores foram bem-sucedidos em identificar vários tipos de cavitação, bem como suas localizações, através das frequências detectadas no conjunto de sinais modulantes. Também foram empregados sensores de emissão acústica e sensores de pressão no tubo de descarga (de forma invasiva), e um melhor conhecimento sobre os mecanismos de formação das cavidades foi adquirido.

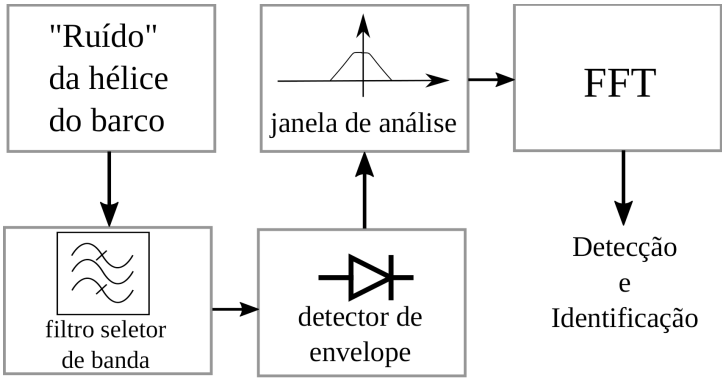


Figura 5 - Diagrama de blocos do processamento DEMON, aplicado à detecção e identificação de barcos.

Fonte: Elaboração própria com *freeware* Inkscape v_0.91 for Linux.

As metodologias citadas até aqui empregam variações do processamento denominado *Detection of Envelope Modulation On Noise* (DEMON), que é uma metodologia empírica, pois depende ou de um conhecimento prévio sobre a faixa onde o sinal característico da cavitação se sobressai, ou da intervenção humana para sintonizar o filtro passa faixa corretamente. A própria filtragem passa faixa representa perda de informação útil, pois o sinal da cavitação ocupa uma banda larga (de dezenas de quilohertz a alguns megahertz), e filtrar em banda significa descartar componentes importantes do sinal (figura 5).

De M. e Hammit F. conseguiram estabelecer uma relação entre a agressividade da cavitação e a potência medida do sinal vibracional induzido. Esta relação depende das propriedades mecânicas do material exposto à cavitação erosiva e os autores a denominaram *eficiência erosiva da cavitação* (DE, HAMMITT, 1982).

1.4.2 Metodologias invasivas ou interferentes na operação

As metodologias mais eficazes para detectar a cavitação, identificar o seu tipo e estimar a sua agressividade contam com alguma técnica invasiva, como por exemplo: sensores ópticos (BALDASSARRE *et al.*, 1998), câmeras, sensores de pressão e outros instalados no interior da turbina; ou com alguma interferência na operação normal (parar a turbina, variar a vazão ou a potência, etc).

A análise multidimensional (BAJIC, 2002) emprega sensores de emissão acústica e em mesmo número que as palhetas diretrizes da turbina. Os sinais são amostrados de forma sincronizada com o giro do eixo da máquina, e as suas potências são medidas em função da posição angular *absoluta*, (envolve parar a operação da turbina e marcar uma lâmina do rotor como a lâmina de referência). A turbina também é operada deliberadamente em várias potências entre 0% e 100% de sua capacidade nominal, o que faz alguns tipos de cavitação surgir e outros desaparecer em algumas faixas de potência. Os sinais são analisados em função da posição angular do rotor, da posição do sensor, da potência elétrica gerada, e da frequência espectral, por isso é uma análise multidimensional. O autor Branko Bajic emprega este método comercialmente para realizar diagnósticos que identificam tipos de cavitação presentes em uma turbina, e suas respectivas agressividades.

A análise vibroacústica inicialmente estabeleceu a correlação entre a potência do sinal vibroacústico de interesse induzido pela cavitação e a agressividade da cavitação. Posteriormente a potência total $I(P)$ do sinal vibroacústico foi modelada como o somatório das potências I_m dos sinais causados por cada um dos M mecanismos individuais de cavitação (BAJIC, 2003):

$$I(P) = \sum_{m=1}^{M} I_m \tag{1}$$

Quando a potência P gerada pela turbina varia, cada termo I_m também varia e a assinatura vibroacústica pode variar consideravelmente.

1.4.3 Metodologia desenvolvida

O projeto surgiu então, conforme mencionado na secção 1.3, da necessidade de monitorar a perda de massa através da estimação da agressividade da cavitação, com a finalidade de se determinar o melhor momento para realizar uma parada de manutenção. Inicialmente a ideia era desenvolver um instrumento de medição dedicado, e que fosse baseado em um *Application Specific Integrated Circuit* (ASIC) analógico. Com o processamento realizado de forma analógica em um *chip* dedicado, o instrumento de medição se tornaria de fabricação mais simples, com menor consumo de energia, de menor tamanho e também possivelmente com maior repetibilidade de fabricação.

O processamento DEMON, por ser mais simples, é elegível para a implementação em um *chip*, entretanto é um processamento empírico e que não é independente da intervenção humana, pelo menos no caso de cavitação em turbinas. Ademais, o processamento DEMON até hoje se mostrou adequado para detectar e identificar o(s) tipo(s) de cavitação presente(s) em uma turbina, mas para a grande necessidade que é estimar a perda de massa, ou a agressividade da cavitação, ele se mostrou ineficiente.

A metodologia desenvolvida neste trabalho consiste em um aperfeiçoamento do processamento DEMON em vários aspectos, e surgiu a partir de uma inspiração do autor em modelar os sinais vibracionais de interesse (SVIs) produzidos por uma turbina sofrendo de cavitação como um processo cicloestacionário.

A modelagem cicloestacionária permite analisar o processo com base em rigorosos critérios estatísticos (não é um processamento empírico), permite traçar paralelos entre parâmetros estatísticos calculados e *grandezas físicas* reais, e habilita o emprego de ferramentas de análise como o *Cyclic Modulation Spectrum* (CMS) (ANTONI, 2009) e o *Cyclic Modulation Coherence* (CMC) (ANTONI, HANSON, 2010, 2012). A necessidade de um operador humano (como no processamento DEMON) com conhecimento prévio das faixas de frequências onde o SVI se sobressai é eliminada, constituindo uma vantagem na possível automação do processamento. Além disso, os padrões vibracionais de uma turbina mudam significativamente ao variar a potência gerada, caso em que a modelagem cicloestacionária não necessita ter seu

processamento ressintonizado. Entretanto, devido à complexidade da metodologia desenvolvida ser muito maior que a do processamento DEMON, a implementação em um ASIC analógico é tecnicamente inviável, até o presente momento. A implementação foi realizada entretanto por meio de um *software* aplicativo específico, e um *hardware* dedicado para a aquisição de sinais.

2 MODELAGEM DA CAVITAÇÃO EM TURBINAS HIDRÁULICAS

A definição simplificada de cavitação, que é a mais utilizada pela engenharia durante o projeto de máquinas hidráulicas, é baseada na pressão de vapor da água, de modo que se a pressão cai abaixo da mesma, há formação de cavidades cheias de vapor e inevitavelmente implosões posteriores. Sabe-se que baixas pressões e altas velocidades de escoamento (que resultam em abaixamento da pressão devido à equação de Bernoulli) favorecem o aparecimento da cavitação. Esta tendência que um escoamento tem de produzir cavidades de vapor é representada pelo *número de cavitação* σ (RAABE, 1985) ou *número de Thoma* (VIVIER, 1966), e depende fundamentalmente da velocidade do escoamento V, da densidade do fluido ρ, da pressão p no local a ser analisado e da pressão de vapor p_v:

$$\sigma = \frac{p - p_v}{\frac{1}{2}\rho V^2} \tag{2}$$

Quanto menor o valor de σ, que é um número adimensional, maior a tendência *global* de um escoamento apresentar cavitação. Um valor importante de σ é o *número de cavitação incipiente* σ_i, que representa a condição limítrofe entre os escoamentos com e sem cavitação. Nota-se também que todas as variáveis envolvidas em (2) são determinísticas, e que teoricamente o comportamento de um escoamento seria como o representado pela figura 6, podendo apresentar cavitação em vários graus (incipiente, limitada, desenvolvida e supercavitação). Maiores detalhes sobre os diferentes graus de cavitação podem ser encontrados no Apêndice A – Cavitação.

Na prática entretanto, observa-se grandes dificuldades em reproduzir experimentalmente fenômenos envolvendo escoamentos com cavitação (pouca repetibilidade). Por exemplo, pode-se demonstrar (STINEBRING *et al.*, 2001) que a água pode suportar pressões inferiores à pressão de vapor sem produzir cavitação de imediato. Logo conclui-se que meramente submeter a água a tensões não garante a mudança de estado físico, e a água é dita em *estado de equilíbrio metastável*. Adicionalmente, esta modelagem não explica porque dois escoamentos apresentam um mesmo valor de σ, mas um apresenta cavitação e o outro não.

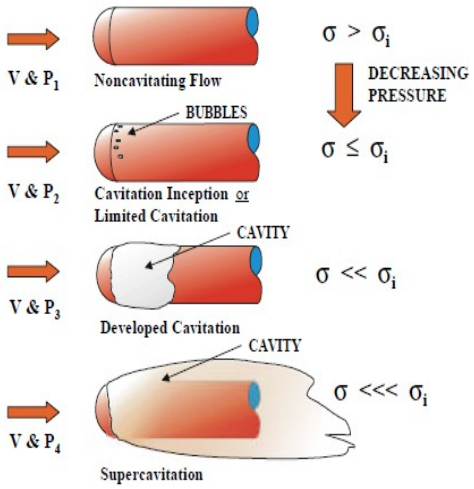


Figura 6 - Representação dos vários graus de cavitação em função da relação entre σ e σ_i.

Fonte: (FRANC, MICHEL, 2004)

Conclui-se que apesar de as pressões e velocidades *locais* (variáveis determinísticas relacionadas a fenômenos macroscópicos) influenciarem diretamente no surgimento e no grau de desenvolvimento da cavitação, esta modelagem apresenta falhas, e algum fenômeno não determinístico e provavelmente microscópico deve estar envolvido. De fato, no século XX cientistas descobriram que os valores da pressão mínima que uma determinada amostra de água pode suportar sem cavitar variam bastante de um experimento para outro e têm correlação com o tratamento que foi dado à água antes do experimento, bem como com o procedimento experimental e a presença de impurezas nos recipientes dos experimentos. Surgiu assim a teoria, posteriormente comprovada, da existência dos *nuclei* (BRENNEN, 1995; FRANC, MICHEL, 2004; MØRCH, 2009).

2.1 OS *NUCLEI* DE CAVITAÇÃO

Um *nucleus* pode ser entendido de forma simplificada como inclusões microscópicas na massa líquida, e que a tornam não homogênea. As inclusões pode ser compostas por gás não dissolvido (caso em que caracterizam microbolhas), partículas sólidas, micro-organismos, ou ainda serem produzidas pela ação dos raios cósmicos. Essas inclusões funcionam como pontos fracos no *continuum* da água, e são os pontos

que romperão (vaporizando) primeiro, caso a massa líquida seja submetida a um abaixamento gradual de pressão.

De fato, todo escoamento líquido real possui uma *população* de *nuclei* finita e não nula. Ademais, a população de *nuclei* presente em um líquido é capaz de influenciar significativamente o surgimento e o desenvolvimento da cavitação (CHAHINE, 2004). Adicionalmente, a população de *nuclei* é, exceto nos experimentos em que se controla a mesma, sempre *aleatória* (tanto no número de *nuclei* quanto na distribuição de seus diâmetros).

Não é o objetivo deste capítulo descrever detalhadamente o comportamento de um *nucleus* quando submetido a um abaixamento de pressão, caso em que ele se torna uma cavidade eventualmente macroscópica e instável (passa por um crescimento explosivo). Também o *nucleus* quando está em equilíbrio instável pode responder de diferentes formas a um aumento repentino de pressão (pode produzir oscilações não lineares, ou pode implodir produzindo pulsos de pressão). As equações que descrevem o comportamento de um *nucleus* estão detalhadamente deduzidas no Apêndice A – Cavitação. Para o contexto deste capítulo entretanto, é importante ter em mente algumas propriedades dos *nuclei*, no que se refere à produção de pulsos de pressão acústica.

2.1.1 Características principais dos *nuclei*

O raio de um *nucleus* influencia no grau de estabilidade do mesmo. De certa forma, um *nucleus* inicialmente menor aguenta abaixamentos de pressão maiores sem sofrer o crescimento explosivo e se instabilizar. Entretanto, as implosões produzidas pelos *nuclei* inicialmente menores são mais violentas que as dos *nuclei* inicialmente maiores. Um *nucleus* menor significa uma menor quantidade de gás não condensável, e por isso menos estabilidade, pois é justamente a quantidade de gás que lhe confere estabilidade (FRANC, MICHEL, 2004).

Os *nuclei* também estão distribuídos aleatoriamente ao longo do tempo e do espaço. *Nuclei* mais afastados entre si têm menor grau de interação, e são mais acuradamente descritos pelas equações diferenciais. Já uma aproximação dos *nuclei* entre si ou de superfícies sólidas produz interações mais fortes, e as equações típicas

não descrevem acuradamente seus comportamentos. Como resultado, os *nuclei* deixam de ser esféricos, e suas implosões deixam de ser simétricas, caracterizando outros tipos de implosão. Também a forte interação entre os *nuclei* causa colapsos coerentes (HALLANDER, 2002) e uma elevação das frequências emitidas pelas oscilações não lineares, devido ao fenômeno *harmonic cascading* (KUMAR, BRENNEN, 1992, 2006; REISMAN *et al.*, 1998). Os mecanismos de concentração de energia e causadores de erosão mais importantes (nomeadamente as ondas de choque e os microjatos) estão descritos com mais detalhes no Apêndice C – Erosão.

Outro detalhe importante é que os *nuclei* podem estar livres dentro da massa líquida, caso em que quando instabilizados produzem a denominada *nucleação homogênea.* De outra forma, os *nuclei* podem estar presos a superfícies sólidas, em suas ranhuras ou defeitos microscópicos, caso em que a nucleação é chamada *heterogênea*. Esta última também acontece quando a tensão a que o líquido é submetido somada à força de coesão do líquido superam a força de adesão entre o líquido e o sólido, e uma bolsa de vapor se forma entre o sólido e o líquido. Ambas nucleações homogêneas e heterogêneas são precursoras da cavitação erosiva.

Resumidamente, a ação conjunta de inúmeras implosões microscópicas com localizações, ocorrências temporais, durações e intensidades aleatórias produz um sinal de pressão nas superfícies sólidas próximas que é composto por um trem de pulsos exponenciais aleatórios, e que pode ser descrito pelo modelo de Morozov (MOROZOV, 1969; ZHANG *et al.*, 1989).

2.2 FENÔMENOS QUE CAUSAM MODULAÇÕES DE PRESSÃO EM TURBINAS

A palavra turbina foi introduzida pela primeira vez pelo engenheiro francês Claude Burdin em 1822, derivada do termo em latim *turbo* ou vórtice. Turbina é, por definição, uma máquina rotativa construída com a finalidade de extrair energia mecânica do escoamento de um fluido. As turbinas hidráulicas atuais têm em geral rendimentos superiores a 90%, e extraem a energia potencial (pressão) e/ou cinética (velocidade) da água. Basicamente podemos dividir as turbinas em turbinas de impulso

e turbinas de reação (LUCKY, 2012), mas no Brasil a grande maioria das turbinas hidroelétricas é do tipo de reação. Dentre as turbinas de reação, o tipo mais usado no Brasil é a turbina tipo *Francis*, e a minoria é do tipo *Kaplan*. Ambos os tipos *Francis* e *Kaplan* são máquinas hidráulicas, e portanto têm como principais elementos um rotor com um número determinado de lâminas *b*, e um estator (ou distribuidor) com um número de palhetas diretrizes *v*. As turbinas de reação também têm uma caixa caracol (também chamada caixa espiral ou caixa voluta), e um tubo de descarga (também chamado de tubo de sucção), como mostrado nas figuras 7 e 8.

Figura 7 - Turbina tipo Francis conectada a um gerador. Notar a caixa voluta em azul e o regulador de potência em amarelo.

Fonte:<http://en.wikipedia.org/wiki/Francis_turbine> Acesso em 17/04/2015.

As experiências realizadas neste trabalho somente envolveram a aquisição de dados de turbinas do tipo Francis, de modo que uma maior ênfase na modelagem dos sinais será dada para este tipo de turbina. A mesma metodologia entretanto, pode ser aplicada às turbinas *Kaplan* com pouca ou nenhuma modificação, pois as diferenças operacionais do ponto de vista da cavitação são mínimas. Um maior detalhamento sobre o assunto pode ser encontrado no Apêndice B – Turbinas hidráulicas.

O principal fenômeno causador de flutuações *localizadas* de pressão no escoamento dentro de uma turbina é sem dúvida a *interação entre rotor e estator* (RSI) (RUCHONNET *et al.*, 2006a). A RSI de uma máquina hidráulica é formada a partir da combinação dos efeitos que as palhetas do distribuidor e as lâminas do rotor provocam no escoamento, por exemplo, suas *esteiras de turbulência*. As esteiras de turbulência

produzidas pelas palhetas do distribuidor são estáticas para um certo grau de abertura das palhetas guia, pois o distribuidor é estático. Por outro lado, as esteiras de turbulência produzidas pelas lâminas giram na mesma velocidade angular que o rotor.

Figura 8 - Corte de uma turbina Francis mostrando as palhetas do distribuidor em amarelo e o rotor em vermelho.

Fonte:<http://en.wikipedia.org/wiki/Francis_turbine> Acesso em 17/04/2015.

A interação entre os dois campos de velocidade e pressão locais se dá de forma não linear (figura 9), e há um efeito multiplicativo (semelhante à modulação em amplitude com portadora suprimida), de modo que a pressão em um ponto X arbitrário localizado no estreito vão entre o rotor e o distribuidor depende fundamentalmente do instante de tempo e da posição angular θ do ponto X (figura 10).

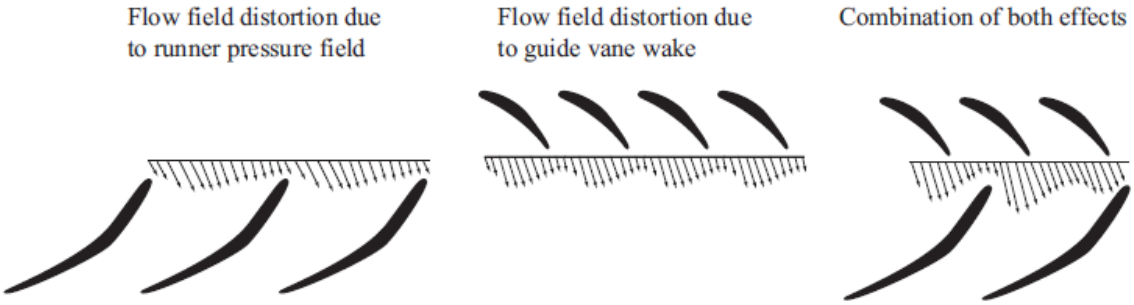


Figura 9 - Composição da RSI a partir das perturbações causadas pelo rotor e pelo distribuidor.

Fonte:(RUCHONNET *et al.*, 2006a)

A RSI em uma turbina de reação produz flutuações de pressão localizadas, e em frequências múltiplas da frequência α_f de rotação fundamental do eixo. As principais frequências observadas são denominadas na literatura especializada como a *frequência de passagem das lâminas* ($\alpha_b = b\alpha_f$) e a *frequência de passagem das palhetas* ($\alpha_v = v\alpha_f$).

Eventualmente são observadas harmônicas de α_b e α_v, ou ainda *produtos de intermodulação* (NICOLET *et al.*, 2010).

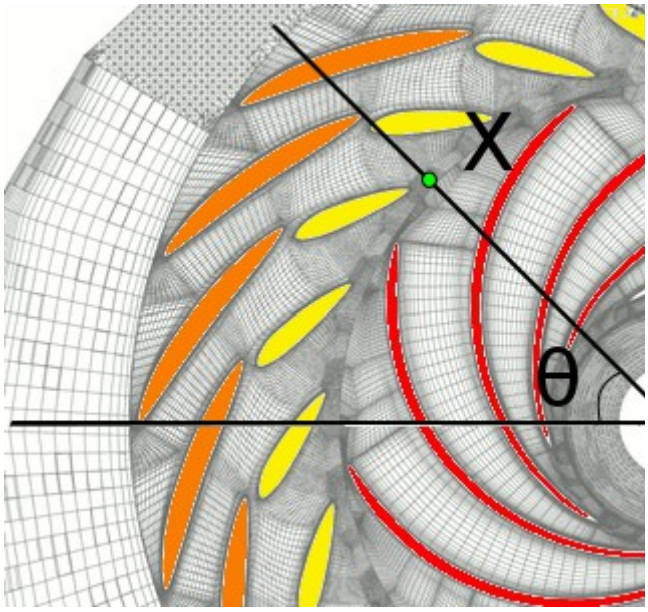


Figura 10 - Diagrama representando o vão entre o distribuidor (em amarelo) e o rotor (em vermelho) de uma turbina Francis.

Fonte: adaptado de (ZOBEIRI *et al.*, 2006)

Estas flutuações de pressão induzidas pela RSI da máquina hidráulica podem eventualmente excitar modos de ressonância da caixa voluta, e produzir ondas estacionárias de pressão (RUCHONNET *et al.*, 2006b). Este fenômeno é mais comum em turbinas projetadas aproximadamente antes do ano 1978, que foi quando a análise computacional da dinâmica de fluidos (CFD) surgiu, e permitiu a simulação da RSI da máquina e eventuais ondas estacionárias na caixa voluta (ČARIJA *et al.*, 2008). As turbinas projetadas nos dias de hoje têm menos tendência a desenvolver ondas estacionárias, pois estas são indesejáveis, e portanto o projeto é modificado a fim de mitigar seu surgimento (KECK, SICK, 2008).

Um outro fenômeno, pouco documentado, é o derramamento de vórtices de Von Kármán (figura 11). Basicamente, a esteira turbulenta produzida por uma lâmina do rotor ou por uma palheta do distribuidor é composta de *vórtices* (formação de *vortex street*). Estes vórtices são derramados em uma frequência determinada pelo *número de Strouhal* (AITBOUZIAD, 2005; AUSONI *et al.*, 2007), que nada tem a haver com a frequência de rotação α_f do rotor, mas somente com a velocidade da água. Vórtices

possuem regiões de baixa pressão em seu interior, e portanto, as flutuações de pressão provocadas pelo derramamento de vórtices de Von Kármán sempre estão na mesma frequência do derramamento dos vórtices. Eventualmente esta frequência pode coincidir com a frequência de ressonância de algum componente da turbina, e um ruído acústico conhecido como "*singing noise*" é emitido.

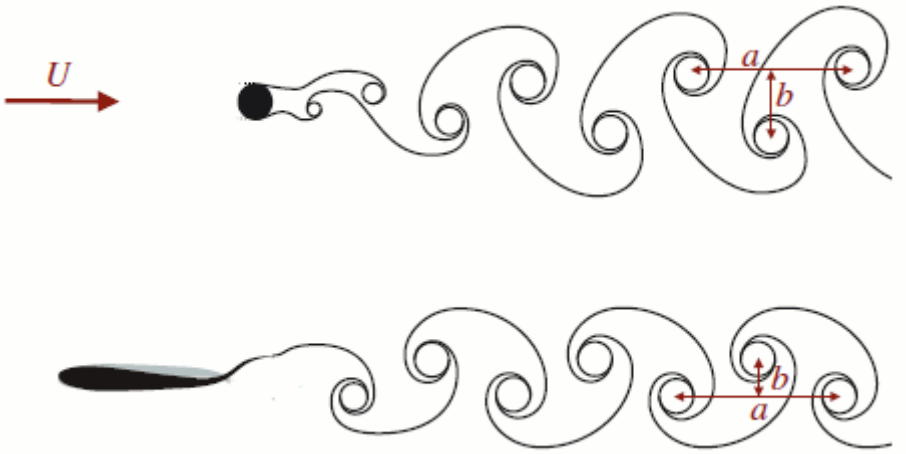


Figura 11 - Produção de vórtices de Von Kármán em um escoamento por obstáculo. Acima: obstáculo com seção circular. Abaixo: hidrofólio.

Fonte: (ELOY, 2012)

O último fenômeno causador de flutuações de pressão em turbinas hidráulicas, e que deve ser citado neste capítulo, é o vórtice de precessão no tubo de descarga (ZOBEIRI, 2009). Este vórtice normalmente é formado no tubo de descarga de turbinas do tipo Francis, que têm o rotor com geometria fixa, e portanto, têm um menor grau de adaptação às variadas combinações de pressão e vazão, em relação às turbinas do tipo Kaplan.

Na operação nas condições de projeto (ponto ótimo), o fluxo da água no tubo de descarga de uma turbina é teoricamente irrotacional. Quando uma turbina opera em condições de pressão e vazão muito diferentes das condições de projeto, o escoamento no tubo de descarga pode ter uma componente residual rotacional, o que dá origem a um vórtice "perpétuo" no tubo de descarga. Este vórtice normalmente abriga uma cavidade de vapor em seu interior, em formato de corda (*vortex rope* ou *draft tube swirl*), que começa no cubo do rotor e se estende pelo tubo de descarga. Eventualmente, oscilações de pressão tão intensas são produzidas a nível *global* dentro da turbina, que podem ser detectadas no conduto forçado (tubo que alimenta a turbina com água do reservatório).

Esta condição é normalmente perigosa, pois as oscilações de pressão podem excitar os modos de ressonância de alguma estrutura próxima (por exemplo, o gerador ou a casa de força), e provocar uma falha catastrófica. A frequência das oscilações é denominada *frequência de Rheingans*, e é sempre uma fração (de 25% a 40%) da frequência α_f de rotação do eixo da turbina.

2.3 CICLOESTACIONARIEDADE

O sinal de pressão acústica produzido pela cavitação hidrodinâmica, conforme exposto na secção 2.1.1, é um processo aleatório composto por inúmeros eventos individuais, que são as implosões. Estas implosões produzem o sinal descrito pelo modelo de Morozov, um trem de pulsos exponenciais aleatórios, e cada pulso é decorrente de um evento individual.

Outros processos também aleatórios e compostos por eventos individuais podem ser citados, como por exemplo: o decaimento radioativo de uma amostra, onde cada evento individual corresponde à desintegração de um único átomo instável; o recebimento de chamadas telefônicas em um *call-center*; a ocorrência de terremotos; o *ruído Barkhausen* e o ruído *shot*. Todos estes processos aleatórios citados são caracterizados como *processos aleatórios tipo Poisson* (VOKURKA, 1980).

Processos aleatórios do tipo Poisson são caracterizados por ter o intervalo entre dois eventos consecutivos quaisquer independente de quaisquer outros intervalos entre eventos. Adicionalmente, o intervalo entre eventos consecutivos é uma variável aleatória com distribuição exponencial de parâmetro λ (o parâmetro taxa de eventos, que tem significado de frequência *média* de eventos). No caso de terremotos, em média ocorre um terremoto a cada 40 minutos, se considerado todo o planeta.

Alguns processos são considerados *processos Poisson homogêneos*, quando o parâmetro λ é constante. O exemplo da ocorrência de terremotos é aproximadamente um caso de processo Poisson homogêneo, pois a frequência dos terremotos diminui muito lentamente ao longo do tempo. O ruído *shot*, desde que mantidas constantes as condições elétricas do circuito, também constitui um exemplo (figura 12).

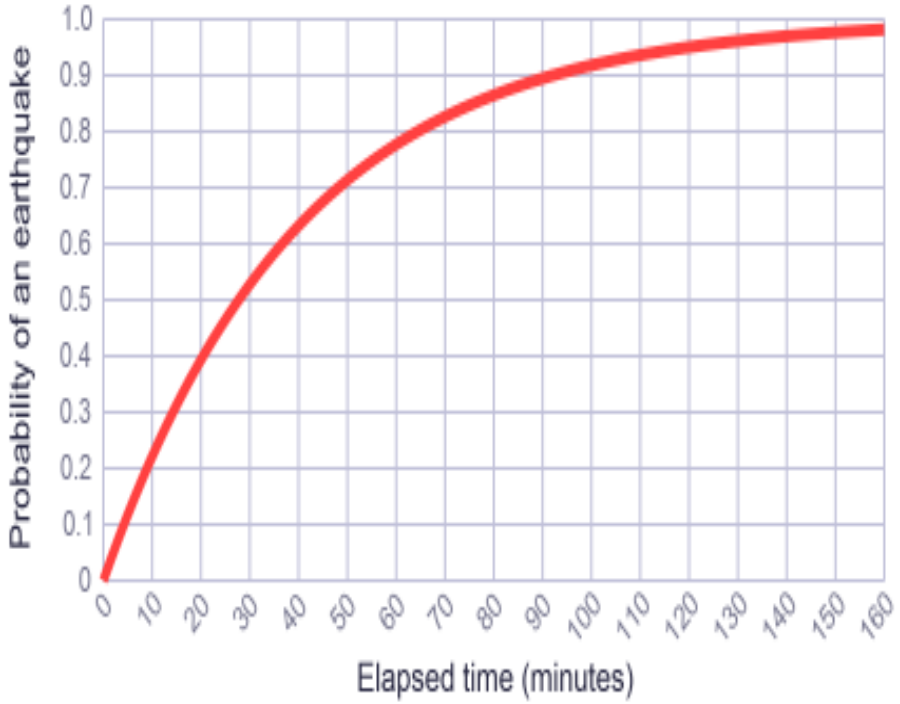


Figura 12 - Função de probabilidade acumulada para um processo Poisson homogêneo simples (tempo entre ocorrência de terremotos).

Fonte:<http://preshing.com/20111007/how-to-generate-random-timings-for-a-poisson-process> Acesso em 20/05/2013.

Alternativamente, outros processos têm o parâmetro λ variável, como por exemplo, em função do tempo: $\lambda(t)$. O decaimento radioativo é um exemplo onde o parâmetro $\lambda(t)$ varia de forma previsível, caindo pela metade a cada meia-vida da amostra. Este tipo de processo é chamado *processo Poisson não homogêneo.*

Um tipo particular de processo Poisson não homogêneo é quando o parâmetro λ varia *periodicamente* em função de outra grandeza. Um exemplo de fácil entendimento é a recepção de chamadas em um *call-center* de abrangência regional, pois λ é maior durante o turno da tarde, e atinge valores mínimos pela madrugada. Neste caso, λ varia de acordo com o ritmo circadiano humano, e o processo é denominado *processo Poisson periódico*.

O professor. Karel. Vokurka em seus artigos (VOKURKA, BUOGO, 2008; VOKURKA, 1980, 2002) afirma que a cavitação hidrodinâmica emite sinais (nos artigos do autor consta "*noise*") acústicos modelados como um processo Poisson homogêneo. Naturalmente, esta afirmação é válida desde que as condições de escoamento sejam estacionárias (como um hidrofólio em um túnel de testes). Em uma

máquina hidráulica rotativa entretanto, a RSI e outras flutuações *periódicas* tornam o escoamento não estacionário, não justificando o uso do modelo Poisson homogêneo, mas sim do modelo Poisson periódico. Outros autores (ANTONI, HANSON, 2012; CARLTON, 2007) caracterizaram a emissão acústica devido à cavitação em escoamentos estacionários como um "ruído branco".

Se, por hipótese, os pulsos exponenciais que compõem o sinal acústico produzido pela cavitação hidrodinâmica fossem todos iguais em amplitude e em constante de tempo, o simples fato de variar a frequência de ocorrência de pulsos λ já caracteriza um tipo de modulação, pois a quantidade média de pulsos acústicos varia, e portanto a *potência média* que o sinal acústico transporta varia também. Obviamente os pulsos exponenciais não são todos idênticos, pois as flutuações de pressão periódicas em uma turbina afetam não só a taxa de implosões, mas suas energias e durações individuais. Apesar de o processo ser aleatório, nota-se claramente que não é *estacionário* propriamente dito, e sua análise não é apropriada se abordada desta forma. Entretanto, o processo de enquadra como *cicloestacionário*, pois de forma simplificada, um sinal é cicloestacionário se de alguma forma "oculta" há periodicidade na energia que o mesmo transporta. Analogamente, as informações transportadas por um sinal de áudio (sinais de fala ou música) são devidas às sucessões de transientes, e não às parcelas estacionárias.

2.3.1 Cicloestacionariedade em diversos graus

A definição de cicloestacionariedade é uma extensão da definição de processo estacionário. Na verdade, processos estacionários são *casos particulares* de processos cicloestacionários (formam um subconjunto). Pode-se definir genericamente como um "sinal que *exibe* cicloestacionariedade" (maiores detalhes sobre a diferença entre sinais que *exibem* e sinais que *são* cicloestacionários serão dados mais adiante na taxonomia dos sinais cicloestacionários) todo sinal $x(t)$ que se processado por uma cascata de dois blocos consecutivos, sendo um linear $h(t,f,\Delta f)$ e outro não linear (figura 13), resulta em um sinal com uma componente senoidal na frequência α, a qual é denominada *frequência de ciclo*. O grau da transformação não linear necessário para evidenciar a

periodicidade oculta em $x(t)$ é o grau da cicloestacionariedade que $x(t)$ exibe.

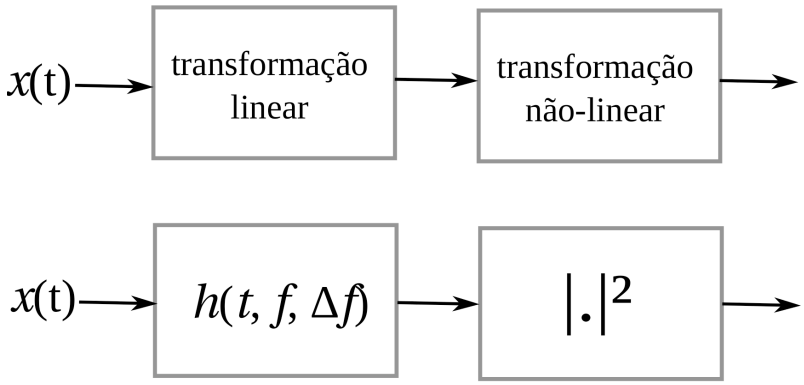


Figura 13 - Definição genérica de sinal $x(t)$ cicloestacionário. As saídas devem conter sinais senoidais, revelando as periodicidades escondidas em $x(t)$.

Fonte: Elaboração própria com *freeware* Inkscape v_0.91 for Linux.

2.3.1.1 Sinais aleatórios estacionários

Sinais classificados como *estacionários em amplo senso*, por sua natureza aleatória, não podem ser determinados com exatidão como um sinal determinístico, mas podem ter seus parâmetros estatísticos determinados. No caso de sinais estacionários em amplo senso, seus parâmetros estatísticos de todos os graus (primeiro momento ou valor médio μ, segundo momento ou variância σ^2, etc) são valores constantes, e portanto, não seguem tendências (em outras palavras: α=0). A análise destes sinais pode ser feita utilizando ferramentas estatísticas comuns. Um exemplo é o ruído térmico gerado em um resistor.

2.3.1.2 Sinais cicloestacionários de primeira ordem

Sinais considerados *cicloestacionários de primeira ordem* são sinais aleatórios, mas com uma característica peculiar: o seu valor médio (primeiro cumulante) não é uma constante como no caso estacionário, mas sim uma função periódica de uma outra variável, como por exemplo, o tempo. Fica definida a função $\mu(t)$, que é a *média instantânea*. A título de exemplo, podemos citar o sinal de saída $x(t)$ de um oscilador senoidal (figura 14) com ruído branco gaussiano aditivo. Este sinal pode ser decomposto em dois termos, um previsível $p(t)=A\cos(\omega t+\varphi)$ e outro aleatório $a(t)$, que é

o ruído branco gaussiano aditivo, tal que $x(t)=p(t)+a(t)$.

Como o ruído branco gaussiano tem valor esperado zero, se fizermos uma média de N sinais (realizações) sincronizados, teremos:

$$\bar{x}(t)=\frac{1}{N}\sum_{i=1}^{N} x_i(t)=\frac{1}{N}\sum_{i=1}^{N} A\cos(\omega t+\varphi)+\frac{1}{N}\sum_{i=1}^{N} a(t) \tag{3}$$

O primeiro somatório tem seu resultado independente do número de repetições enquanto o segundo tenderá a zero quando N tender ao infinito. Portanto:

$$\mu(t)=\lim_{N\to\infty}\bar{x}(t)=A\cos(\omega t+\varphi) \tag{4}$$

Nos casos em que a função $\mu(t)$ puder ser deduzida de um número finito de realizações, denomina-se o processo como *ergódico no primeiro momento* ou *ergódico médio*. Para estes processos, a análise pode ser feita com ferramentas comumente encontradas na teoria de processamento de sinais, como a transformada de Fourier. Também, nota-se que na figura 13 o bloco não linear pode na verdade ser reduzido a um bloco linear, ou mesmo ser eliminado, pois a extração da(s) componente(s) periódica(s) é realizada por um filtro *linear* comum.

Figura 14 - Decomposição de um sinal cicloestacionário de primeira ordem (à esquerda) em suas componentes determinística (no centro) e aleatória (à direita).

Fonte:(ANTONI, 2009)

Finalmente, esta decomposição em termo determinístico e termo aleatório pode também ser feita no domínio da frequência, sendo que o termo determinístico tem um espectro caracterizado por componentes de frequências *discretas*, e o termo aleatório tem seu espectro *contínuo*. O significado físico desta decomposição é que o termo determinístico é decorrente de macrofenômenos dentro do processo (em máquinas rotativas, pode ser por exemplo, um desbalanceamento de uma roda, um eixo torto, ou

resultado do próprio processo, como no caso de um oscilador), e o termo aleatório é decorrente de microfenômenos (ruído, atrito, etc) aleatórios.

2.3.1.3 Sinais cicloestacionários de segunda ordem

Há processos onde o sinal produzido $x(t)$ não apresenta seu valor médio como uma função periódica, mas sim momentos de ordem mais alta, como por exemplo, a variância (figura 15). Neste caso, se $x(t)$ for a representação temporal de uma pressão acústica, então $x^2(t)$ terá significado físico de *potência instantânea*, a potência que a onda acústica carrega ($x^2(t)$ normalizado pela impedância acústica do meio). Neste caso faz sentido definir a variância (sincronizada) instantânea $\sigma^2(t)$ de N realizações como:

$$\sigma^2(t) = \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} \left(x_i^2(t) - \mu^2(t) \right) \tag{5}$$

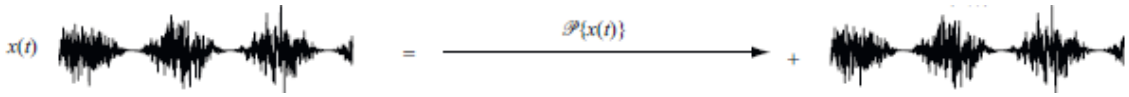


Figura 15 - Decomposição de um sinal cicloestacionário de segunda ordem (à esqueda) em suas partes determinística (no centro, nula) e aleatória (à direita).

Fonte:(ANTONI, 2009)

Processos em que *potência média* transferida varia periodicamente são chamados de *processos cicloestacionários de segunda ordem*, e a análise destes é bem mais complexa que processos com cicloestacionariedade de primeira ordem. Ferramentas estatísticas comuns são uma má escolha, pois são apropriadas para processos estacionários e por isso ignoram as variações cíclicas da potência. Ferramentas apropriadas para análise de processos cicloestacionários de primeira ordem também não têm a capacidade de analisar corretamente estes processos, pois também ignoram as variações de potência, retornando sempre valores de *potência média* ao longo dos ciclos de sinal (e não a *potência instantânea média* entre ciclos do sinal).

Foi realizada uma análise com a transformada rápida de Fourier (FFT) em um sinal real (ruído Barkhausen obtido com ferro para transformadores e um agitador

magnético de laboratório) modulado em amplitude, e os resultados podem ser vistos na figura 16. A FFT sozinha é incapaz de revelar a periodicidade escondida no sinal.

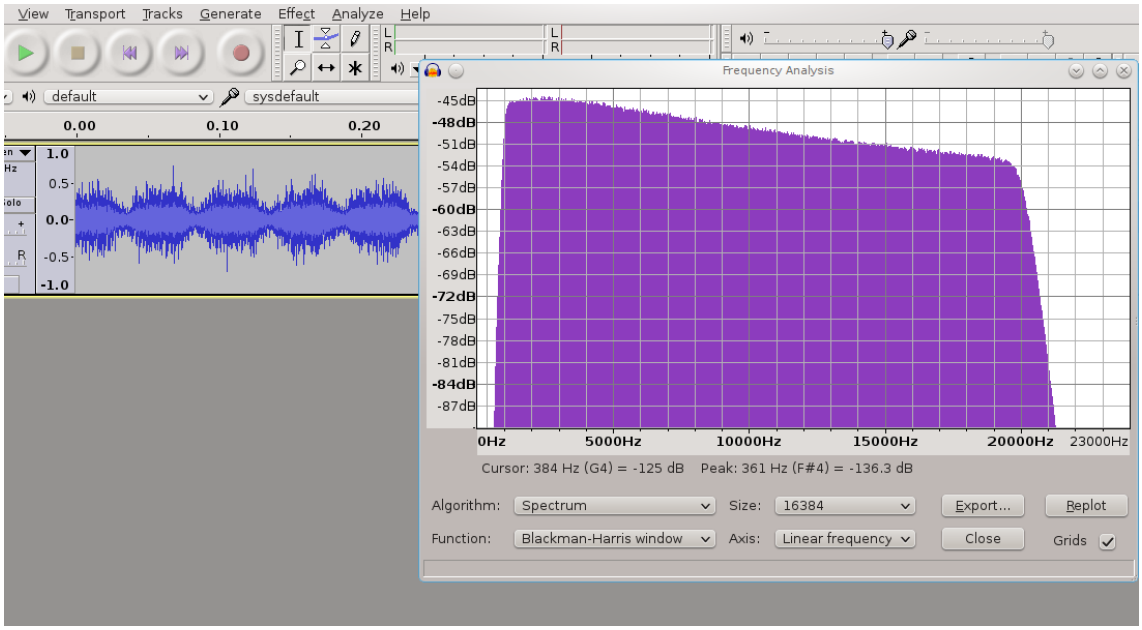


Figura 16 - Sinal real cicloestacionário de segunda ordem. Em azul no canto superior esquerdo: a representação temporal onde é evidente a modulação em amplitude. Em roxo à direita: o espectro de frequências não mostra qualquer sinal de periodicidade.

Fonte: Elaboração própria com o *freeware* Audacity 2.0.5 Linux.

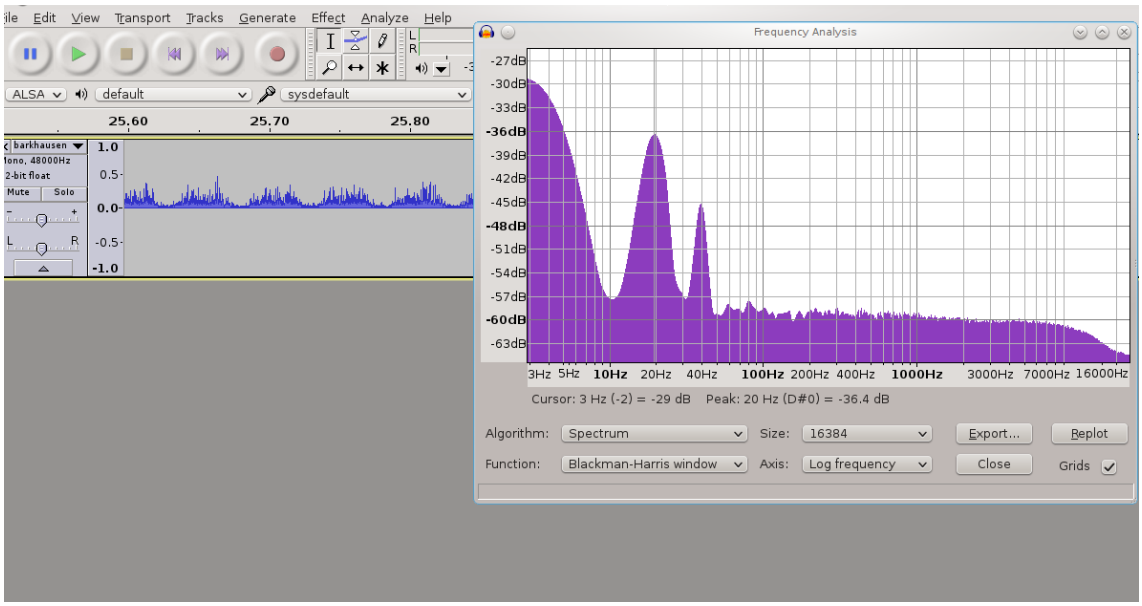


Figura 17 - Periodicidade de um sinal cicloestacionário de segunda ordem evidenciada por meio de uma operação não linear no sinal original. À esquerda: o sinal original da figura 16 elevado ao quadrado. À direita: o espectro revela picos nas frequências de ciclo.

Fonte: Elaboração própria com o *freeware* Audacity 2.0.5 Linux.

Maiores detalhes sobre o ruído Barkhausen estão disponíveis no Apêndice D – Ruído Barkhausen e magnetostrição. Uma extensão do experimento da figura 16 foi feita, mas com o sinal elevado ao quadrado, e neste caso, a operação não linear de ordem dois revelou a periodicidade escondida (o agitador magnético estava girando a aproximadamente 600 rpm, ou 10 voltas por segundo). Neste caso fica provado que o sinal possui pelo menos uma componente cicloestacionária de segunda ordem (figura 17).

No exemplo ilustrado pelas figuras 16 e 17, a periodicidade do sinal original não pode ser evidenciada pela representação no domínio da frequência, apesar de estar evidente na representação no domínio do tempo. Não é difícil, entretanto, produzir um sinal aleatório com uma periodicidade escondida, de tal forma que seja impossível identificá-la visualmente nas representações no domínio da frequência, e no domínio do tempo separadamente. É necessária então uma representação mista em um *domínio de duas grandezas simultaneamente*, e esta representação será a base para o desenvolvimento da metodologia de detecção do sinal da cavitação e medida da sua potência.

2.3.1.4 Taxonomia da cicloestacionariedade

Um sinal é considerado cicloestacionário quando há um conjunto $\mathbb{A}$ de frequências de ciclo não nulas, que são as frequências em que a potência transmitida pelo sinal oscila. Estas frequências são também chamadas *frequências cíclicas*. Na figura 17 é possível identificar duas frequências cíclicas: 20 e 40 Hz.

Se o conjunto $\mathbb{A}$ é composto por uma componente fundamental de frequência cíclica α_f, suas harmônicas e ainda outros elementos, então o sinal é dito *exibir* cicloestacionariedade. Caso o conjunto $\mathbb{A}$ seja formado por α_f, e suas harmônicas *somente*, então diz-se que o sinal *é* cicloestacionário (ANTONI, 2009).

2.3.1.5 Cicloestacionariedade pura versus impura

A definição de cicloestacionariedade segundo o critério descrito na página 24 é propícia a uma má interpretação por um detalhe sutil: o sinal mostrado na figura 14, que é cicloestacionário de primeira ordem, se processado pela cascata de blocos mostrados na figura 13 (metade inferior com o bloco não linear de segunda ordem), vai certamente resultar em um sinal com uma componente senoidal, e portanto seria considerado cicloestacionário de segunda ordem também. Na verdade a cicloestacionariedade de segunda ordem que o sinal exibe é somente em razão do seu termo determinístico, ou seja, devido à cicloestacionariedade de primeira ordem. Neste caso a cicloestacionariedade exibida de segunda ordem é dita *impura* (ANTONI, 2009), pois há uma componente cicloestacionária de primeira ordem (ordem inferior a dois).

Já o sinal mostrado na figura 15 é considerado um sinal que é cicloestacionário *puro* de segunda ordem (ANTONI, 2009), pois a componente cicloestacionária de primeira ordem é nula.

Uma conclusão importante para este estudo é que, para sinais cicloestacionários de primeira ordem puros a parte aleatória (também chamada de *residual*) não exibe cicloestacionariedade de ordem maior ou igual a dois. Já para sinais cicloestacionários de segunda ordem puros a parte determinística é nula, e a parte residual é cicloestacionária de segunda ordem (ANTONI, 2009).

2.4 TIPOS DE CAVITAÇÃO EM TURBINAS HIDRÁULICAS

A combinação de um ou mais mecanismos de formação de cavidades descritos na secção 2.1, com um ou mais fatores que causam modulação devido às flutuações de pressão descritos na secção 2.2, resulta em diferentes tipos de cavitação, bem documentados na literatura (ESCALER *et al.*, 2006a). Estes tipos se diferenciam em tamanho e forma das cavidades, bem como nos locais da erosão provocada, e outros efeitos nocivos. Por serem combinações de microfenômenos aleatórios (formação e implosão dos *nuclei*) com macrofenômenos determinísticos (as flutuações periódicas de pressão), cada um dos tipos de cavitação produz sinais acústicos com uma combinação

única de termos determinístico e aleatório.

Outra característica importante dos sinais induzidos pela cavitação é que a maioria dos sinais modulantes não são sinais periódicos no domínio do tempo, mas sinais periódicos no domínio da *posição angular* do eixo da turbina. Alguns sinais modulantes são de fato sinais temporais, mas estes não serão analisados neste estudo, pois os sinais característicos da cavitação erosiva são modulados por sinais no domínio angular. Portanto, é de se esperar que os sinais de interesse exibam *cicloestacionariedade angular ou em relação ao ângulo*, e portanto o conjunto A referido na secção 2.3.1.4 não é um conjunto de frequências cíclicas, mas sim um conjunto de *ordens de máquina*. Ordem de máquina é o domínio da transformada de Fourier calculada de um sinal no domínio angular, assim como a frequência é o domínio da transformada de Fourier calculada de um sinal no domínio do tempo (ANTONI, 2009). Portanto, se um sinal está na ordem de máquina de número 2, significa que ele se repete 2 vezes a cada revolução do eixo de referência.

Os tipos de cavitação mais importantes dentro de uma turbina são: a cavitação em bolhas itinerantes (*traveling bubble cavitation*), a cavitação em nuvem (*cloud cavitation*), a cavitação em vórtice de Von Kármán (*Von Kármán vortex cavitation*, um tipo bem raro de cavitação) e a cavitação no tubo de descarga (*draft tube swirl* ou *vortex rope*). Destes, os mais erosivos são a cavitação em bolhas itinerantes e a cavitação em nuvem. A cavitação em vórtices de Von Kármán pode causar quebra de partes da turbina por fadiga, em raras ocasiões. A cavitação no tubo de descarga limita severamente a faixa de potências que uma turbina pode operar, induzindo vibrações em níveis inaceitáveis, e que podem provocar uma falha catastrófica.

2.4.1 Cavitação em bolhas itinerantes

A cavitação em bolhas itinerantes (figura 18) é o resultado da interação entre a nucleação homogênea e a RSI da máquina, podendo eventualmente a nucleação heterogênea também estar envolvida. A rápida passagem de uma lâmina do rotor produz um abaixamento brusco de pressão no lado da sucção da lâmina (o lado visível pelo tubo de descarga, também visível em vermelho na figura 8), que estimula o crescimento

explosivo dos *nuclei* presentes na água. Os *nuclei* se mantém aproximadamente esféricos, pois há pouca interação entre eles devido ao afastamento relativamente alto, tanto temporalmente quanto espacialmente, e isso permite o sinal acústico ser descrito aproximadamente pelo modelo de Morozov.

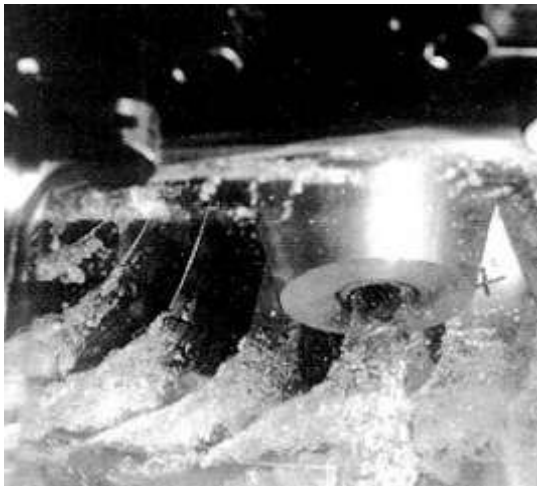


Figura 18 - Cavitação em bolhas itinerantes no bordo de fuga das lâminas do rotor.
Fonte:(ESCALER *et al.*, 2006a).

Este tipo de cavitação é muito sensível ao número de cavitação σ e às características da população de *nuclei.* Portanto, sempre que uma turbina opera com níveis anormalmente baixos de reservatório de descarga, o número de cavitação cai e a tendência *global* de apresentar a cavitação em bolhas itinerantes aumenta. Para este tipo de cavitação erosiva, a agressividade aumenta com a vazão e com a potência gerada pela turbina (ESCALER *et al.*, 2006a).

Este tipo de cavitação produz erosão somente nas lâminas do rotor, no lado da sucção e próximo à borda de fuga, conforme indicado pela região "B", na figura 19. Outros efeitos nocivos que a cavitação em bolhas produz são: queda no rendimento da geração de energia, forte emissão de ruído sonoro e intensas vibrações induzidas. A existência simultânea de derramamento de vórtices de Von Kármán ou do vórtice em precessão no tubo de descarga pode interferir com este tipo de cavitação, introduzindo outros sinais de modulação.

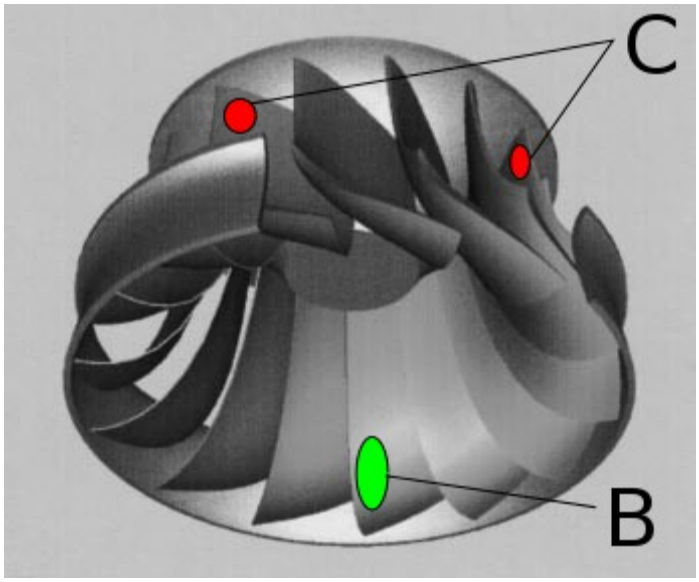


Figura 19 - Rotor de uma turbina Francis com corte na cinta. Em "B" (verde) o local danificado pela cavitação em bolhas. Em "C" (vermelho) os locais danificados pela cavitação em nuvem.

Fonte: Adaptado de (DRTINA, SALLABERGER, 1999).

Tanto a RSI da máquina quanto as ondas estacionárias na caixa voluta podem ser modeladas como uma série de Fourier, e para o caso particular da cavitação em bolhas itinerantes, a *ordem de máquina fundamental* desta série de Fourier é a ordem de máquina *b*, onde *b* é o número de lâminas do rotor (*blades*). O sinal modulante $m_B(\theta)$ é então:

$$m_B(\theta) = \sum_{i=-\infty}^{\infty} B_i e^{jbi\,\theta} \tag{6}$$

Em (6), *j* é a unidade imaginária, B_i é o coeficiente complexo da componente de ordem de máquina de número *bi* e θ é a posição angular do rotor da turbina. O sinal modulante $m_B(\theta)$ deve ser sempre não negativo, pois a modulação é em amplitude com portadora, e portanto o coeficiente B_0 deve ser positivo.

O sinal acústico produzido pelas implosões (resultando em ondas de choque ou microjatos) serve como estímulo mecânico nas peças da turbina, e induz vibrações que podem ser captadas por acelerômetros instalados externamente. A modelagem exata da produção de vibrações mecânicas externas por excitações acústicas internas é um tanto complexa, pois envolve fatores como múltiplas ressonâncias, múltiplos percursos (incluindo em meio sólido e em meio líquido), transmissões mecânicas não lineares e

variantes no tempo.

Outro fator de complexidade é que o sinal excitador acústico descrito pelo modelo de Morozov é essencialmente um sinal no domínio do tempo, perdendo seu significado no domínio angular. O sinal de pressão acústica $p(t)$ segundo o modelo de Morozov é:

$$p(t)=\sum_{i=1}^{N} P_i e^{-\frac{|t-t_i|}{\Psi_i}} \tag{7}$$

As variáveis P_i e Ψ_i representam respectivamente o pico de pressão produzido pela *i*-ésima implosão e a constante de tempo que influencia na duração da *i*-ésima implosão, sendo ambas aleatórias por dependerem do raio crítico do *i*-ésimo *nucleus*. A *i*-ésima implosão acontece no instante t_i, e no total são somadas N implosões. Quando N é um número muito grande ($N>>1$), o sinal $p(t)$ se torna indistinguível de um ruído gaussiano.

Uma aproximação razoável para transmissão mecânica ligando o(s) ponto(s) onde o(s) sinal(is) acústico(s) são produzidos ao ponto onde está instalado o sensor externo, é considerar que a excitação acústica é um pequeno sinal, e por isso modelar como uma função de transferência linear. A complexidade da função de transferência, bem como se é ou não invariável no tempo é irrelevante para a análise em questão.

A função de transferência (ou a sua transformada de Fourier inversa, a resposta ao impulso) também só tem sentido físico se expressa no domínio da frequência (ou do tempo), não havendo equivalente para o domínio da ordem de máquina (ou domínio angular). Uma maneira de contornar este problema é assumir, por questões de simplicidade, que a velocidade angular do eixo é constante, portanto $\theta(t)$ é uma função linear. Simplificadamente $\theta(t)=\omega t$.

Seja $h_B(\theta)$ a resposta ao impulso correspondente ao caminho mecânico ligando o local da erosão ao sensor, já incluindo também a resposta impulsiva do sensor; e $n_B(\theta)$ a função excitação acústica aleatória como na equação (7), só que no domínio angular. O sinal vibracional $c_B(\theta)$ produzido (estacionário, sem modulação) é:

$$c_B(\theta)=(h_B * n_B)(\theta) \tag{8}$$

O sinal * denota a convolução, e o subscrito "*B*" denota "*bubble cavitation*". Cabe ressaltar que as implosões têm durações de microssegundos ou nanosegundos, e portanto o sinal excitatório acústico se estende até alguns megahertz. Já as ressonâncias mecânicas de uma turbina se estendem a no máximo algumas centenas de quilohertz. Uma boa aproximação portanto, é considerar o sinal $n_B(\theta)$ como um sinal aleatório gaussiano e branco (para a faixa de frequências em questão) (ANTONI, HANSON, 2012).

Finalmente, o processo de modulação da taxa das implosões e de suas durações e intensidades é um processo não linear. O sinal aleatório $c_B(\theta)$ é estacionário, portanto tem a variância (potência) constante; e o sinal modulante $m_B(\theta)$ deve modular a potência (variância) da portadora aleatória (*carrier*) $c_B(\theta)$ diretamente. Portanto o sinal vibracional de interesse (SVI) produzido pela cavitação em bolhas itinerantes $x(\theta)_{\text{SVI-B}}$ é uma função não linear f_B de $m_B(\theta)$ e $c_B(\theta)$:

$$x(\theta)_{SVI-B}=f_B(m_B(\theta), c_B(\theta)) \tag{9}$$

2.4.2 Cavitação em nuvem

A cavitação em nuvem resulta da combinação das flutuações de pressão devido à RSI da máquina ou de ondas estacionárias de pressão na caixa voluta (ou ambas), com a nucleação heterogênea que provoca o "descolamento" da massa líquida das paredes sólidas de uma lâmina ou palheta diretriz. Este descolamento forma uma bolsa preenchida com vapor, e solidária ao bordo de ataque da lâmina ou palheta, a denominada *cavidade fixa* ou *attached cavity* (figura 20).

A cavidade fixa pode assumir diferentes regimes dependendo das condições do escoamento, podendo ser estável (*sheet cavitation*), oscilatória ou instável. As flutuações de pressão da RSI ou ondas estacionárias modulam o tamanho e a forma dessa cavidade fixa, e dela se desprendem partes em forma de *nuvens* de vapor.

Figura 20 - Cavidade fixa no bordo de ataque de um hidrofólio em túnel de testes e formação da cavitação em nuvem. O escoamento é da esquerda para a direita.

Fonte:<http://www.heliciel.com/images/cavitation_bulles.jpg>. Acesso em 23/04/2015.

Estas nuvens de vapor são formadas por inúmeros *nuclei* muito próximos entre si, e portanto com forte interação. A cavidade fixa tem a capacidade de aumentar em muito a população de *nuclei* à sua jusante, o que não só torna a cavitação em nuvem pouco dependente da população original de *nuclei* e do número de cavitação σ, mas também provoca a forte interação entre os *nuclei* instáveis, que deixam de ser esféricos. A forte interação provoca fenômenos como o *harmonic cascading*, que desloca a emissão de sinais acústicos para faixas de frequências superiores, e os *colapsos coerentes*, que amplificam os impactos produzidos pelas ondas de choque e microjatos de forma similar à amplificação da luz em um LASER. A cavitação em nuvem é portanto, um tipo de cavitação com muito poder erosivo e que pode induzir comportamento anormal na operação da máquina, haja vista que esta não foi projetada para operar com bolsas de fluido compressível (vapor) solidárias às lâminas do rotor.

A cavitação em nuvem, por ter como precursora a cavidade fixa, sempre danifica o bordo de ataque de lâminas ou palhetas diretrizes (ESCALER *et al.*, 2006a). Quando uma turbina opera com níveis de reservatório de entrada muito acima do nominal, a cavidade fixa se desenvolve na face de sucção das lâminas do rotor. Por outro lado, quando a turbina opera com níveis de reservatório muito abaixo do nominal, a cavidade fixa aparece na face de pressão das lâminas do rotor (figura 19). Este tipo pode

eventualmente danificar as palhetas do distribuidor, e também pode sofrer interferência do vórtice em precessão no tubo de descarga e do derramamento de vórtices de Von Kármán.

A modelagem do SVI produzido pela cavitação em nuvem é semelhante à feita para a cavitação em bolhas itinerantes, porém a ordem de máquina fundamental da série de Fourier é a ordem de máquina *v*, que é o número de palhetas no distribuidor (*vanes*). Isto é devido à cavidade fixa sofrer forte influência da esteira turbulenta produzida pelo distribuidor. O sinal modulante $m_C(\theta)$ é então:

$$m_C(\theta) = \sum_{i=-\infty}^{\infty} C_i e^{jvi\theta} \tag{10}$$

De forma análoga à cavitação em bolhas, C_i é o coeficiente complexo da componente de ordem de máquina *vi*, e $m_C(\theta)$ deve sempre ser não negativo, portanto C_0 deve ser não nulo. O subscrito "*C*" denota "*cloud cavitation*". O sinal vibracional $c_C(\theta)$ (a portadora aleatória estacionária) é modelado de forma similar:

$$c_C(\theta) = (h_C * n_C)(\theta) \tag{11}$$

Ambos $h_C(\theta)$ e $n_C(\theta)$ são diferentes dos seus equivalentes produzidos pela cavitação em bolhas. O primeiro é devido ao local da erosão ser diferente, portanto o caminho de propagação das vibrações é diferente, resultando em outra resposta ao impulso. O segundo é devido à forte interação entre os *nuclei* na cavitação em nuvem, e consequente *harmonic cascading* e produção de colapsos coerentes. Ademais, a função não linear f_C também é diferente do caso da cavitação em bolhas, pois os mecanismos de modulação são ligeiramente diferentes. O sinal vibracional produzido pela cavitação em nuvem é então modelado como:

$$x(\theta)_{SVI-C} = f_C(m_C(\theta), c_C(\theta)) \tag{12}$$

2.4.3 Cavitação em vórtice no tubo de descarga

O *draft tube swirl* é um tipo peculiar, resultante da cavitação em vórtice com um mecanismo de nucleação qualquer. O vórtice em precessão no tubo de descarga possui uma região de baixa pressão em seu interior, capaz de abrigar cavidades de vapor formadas por outros tipos de cavitação ou nucleação. A cavidade formada assume a forma de um cordão (figura 21) que começa no cubo do rotor e se estende pelo tubo de descarga.

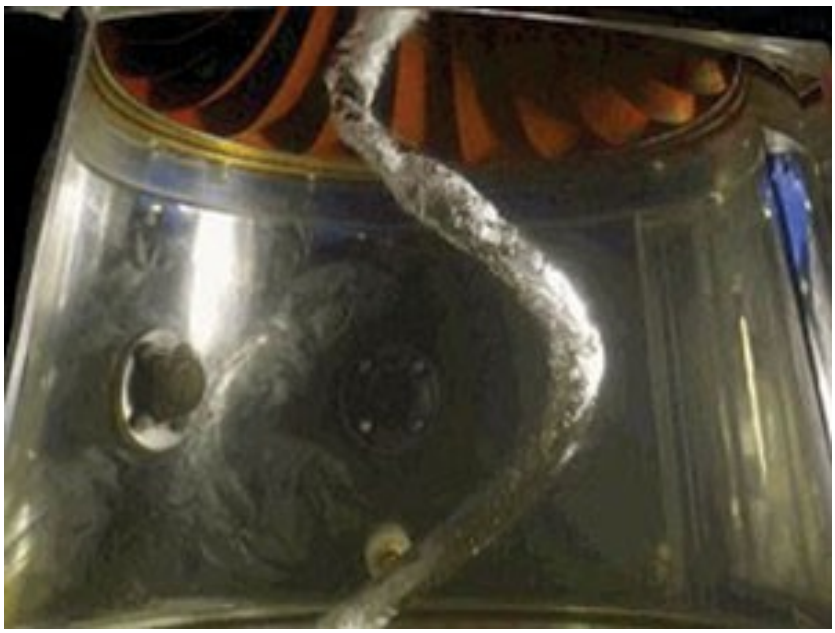


Figura 21 - Formação de vórtice em cordão no tubo de descarga de acrílico de uma turbina Francis.

Fonte:<http://www.powermag.com/a-breakthrough-in-hydroturbine-design-2/?pagenum=3> Acesso em 23/04/2015.

Este tipo de cavitação é pouco erosivo em si, e a pouca erosão é produzida nos pontos onde a cavidade toca as peças da turbina, como o cubo (*hub*) do rotor e eventualmente o interior do tubo de descarga. A frequência de rotação deste cordão de vapor é denominada *frequência de Rheingans*, e é uma fração da frequência de rotação do eixo da turbina. O efeito mais nocivo e que é notável neste tipo de cavitação é a produção de oscilações de pressão na mesma frequência de Rheingans, e tão intensas que podem ser sentidas até no conduto forçado (*penstock*). Esta condição anormal obriga a turbina a operar com faixa limitada de potência, vazão e nível.

Como este tipo de cavitação é pouco erosivo, pois só produz impactos no cubo do rotor, e as frequências modulantes não são múltiplas inteiras da de rotação do eixo, a sua modelagem é um pouco mais complexa, e a *detecção direta* é complicada. Não obstante, a *detecção indireta* é possível nos casos onde há simultaneamente cavitação em bolhas itinerantes ou cavitação em nuvem, pois as intensas flutuações de pressão na frequência de Rheingans interferem globalmente na máquina, afetando outros tipos de cavitação preexistentes.

Definindo R como a ordem de máquina equivalente à frequência de Rheingans (sendo R, segundo o descrito na página 22, algum valor dentro da faixa entre 0,25 e 0,40), esta interferência pode também ser modelada simplificadamente como uma série de Fourier, onde D_i é o coeficiente complexo da componente de ordem de máquina Ri:

$$m_D(\theta) = \sum_{i=-\infty}^{\infty} D_i e^{jRi\theta} \tag{13}$$

O sinal $m_D(\theta)$ acaba por multiplicar (ou exercer outra operação não linear sobre) os sinais modulantes $m_B(\theta)$ e $m_C(\theta)$, e também modular diretamente os sinais aleatórios de portadora $c_B(\theta)$ e $c_C(\theta)$. Surgem então sinais modulantes das ordens de máquina R e múltiplos inteiros, e também seus produtos de intermodulação com as ordens de máquina b e v (por exemplo: $b\pm R$, $2b\pm R$, $b\pm 2R$, $v\pm R$, $2v\pm R$, $v\pm 2R$, etc).

Tecnicamente falando, e seguindo os critérios apresentados na taxonomia da cicloestacionariedade (secção 2.3.1.4), estas novas componentes moduladas produzidas pelo vórtice no tubo de descarga introduzem frequências cíclicas que não são harmônicas de α_f, (na verdade introduzem ordens de máquina que não são inteiras) no conjunto $\mathbb{A}$. Portanto, o sinal vibracional de interesse resultante somente exibe cicloestacionariedade de segunda ordem, mas não é cicloestacionário. Alguns autores observaram a formação de bandas laterais acima e abaixo de α_b e α_v, em turbinas com o *draft tube swirl* (ESCALER *et al.*, 2004, 2006a, 2014, 2006b).

A estimação da agressividade da cavitação neste caso é impossível por este meio indireto de detecção, mas felizmente, a erosão causada pela cavitação em vórtice no tubo de descarga raramente é motivo de preocupação.

2.4.4 Cavitação em vórtices de Von Kármán

Os vórtices derramados à jusante de uma palheta diretriz ou lâmina podem também abrigar cavidades instáveis de vapor em seu interior (figura 22), e a flutuação de pressão induzida nas imediações pode interferir com outros tipos de cavitação preexistentes, da mesma forma que a cavitação em vórtice no tubo de descarga o faz. A ação é local entretanto (no bordo de fuga ou próximo), e o derramamento de vórtices pode interferir com a cavitação em nuvem ou em bolhas somente, mas não com ambas.

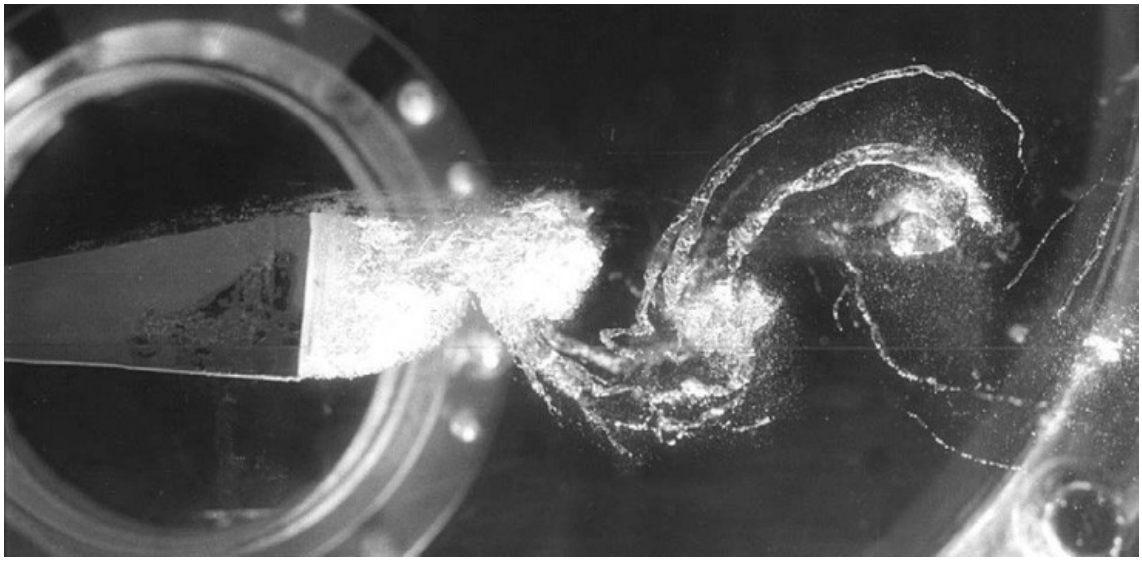


Figura 22 - Cavitação em vórtice de Von Kármán na esteira turbulenta de um hidrofólio em túnel de testes.

Fonte: (FRANC, MICHEL, 2004)

As flutuações de pressão são produzidas na frequência de derramamento dos vórtices, que por sua vez depende do número de Strouhal (que depende da velocidade do fluido e outros fatores). De fato, alguns medidores de vazão velocimétricos medem a frequência de derramamento dos vórtices, que é determinada basicamente pela velocidade do fluido.

Os vórtices de Von Kármán não são perpétuos como o vórtice em precessão no tubo de descarga, e raramente são observados em turbinas hidráulicas. Frequentemente são observados em hidrofólios instalados em túneis de teste, e mesmo assim quando o escoamento atinge condições extremas, caso em que se observa a frequência *natural* de derramamento dos vórtices.

Para a cavitação em vórtices de Von Kármán surgir em uma turbina, uma condição peculiar é necessária: a frequência natural de derramamento de vórtices deve coincidir com alguma outra frequência natural do sistema, ou alguma frequência produzida pela RSI da máquina. Esta condição de *oscilação forçada* também é perigosa, pois causa o fenômeno conhecido como travamento de frequências (*lock-in*) e produz vibrações conhecidas como "*singing noise"*, bem como quebra do bordo de fuga por fadiga do material (BALDASSARRE *et al.*, 1998; ESCALER *et al.*, 2006a).

Da mesma forma que ocorre na cavitação em vórtice no tubo de descarga, há a formação de componentes moduladas de novas ordens de máquina, mas devido à condição necessária de oscilação forçada, estas novas componentes estão em ordens de máquina de número inteiro, não necessariamente múltiplos de *b* ou *v*. Um exemplo encontrado na literatura caracteriza as vibrações induzidas pela cavitação em vórtice de Von Kármán interferindo com a cavitação em bolhas (componente de ordem de máquina *b*) pela presença peculiar de uma componente de ordem de máquina *b*+1 (ESCALER *et al.*, 2006b). Os autores entretanto descreveram a cavitação de Von Kármán como a responsável pela produção de uma componente na vigésima harmônica da frequência de rotação do eixo de uma turbina com 19 lâminas. Conclui-se que o SVI não deixa de *ser* cicloestacionário com a introdução da cavitação von Kármán, pois no conjunto Ⱥ continuam a existir somente ordens de máquina de números inteiros. A estimação da agressividade da cavitação, entretanto, não é trivial.

2.4.5 O sinal vibracional de interesse completo

A composição do SVI completo, incluindo as contribuições de todos os tipos de cavitação citados, pode ser considerada como a soma de ambos os sinais da cavitação em bolhas e em nuvem somados, haja vista que não deve haver interação entre os *nuclei* de tipos diferentes de cavitação, e a composição é linear:

$$x(\theta)_{SVI} = x(\theta)_{SVI-B} + x(\theta)_{SVI-C} \tag{14}$$

As características de todos os tipos de cavitação citados neste capítulo estão sumarizadas na tabela 1, para facilitar o entendimento do leitor e para servir como

orientação no capítulo seguinte, que trata do desenvolvimento da metodologia de estimação da agressividade da cavitação.

Tabela 1 - Características dos principais tipos de cavitação.

Tipo de cavitação	Predominância de emissão acústica	Modulação do volume das cavidades	Características notáveis
Bolhas itinerantes	Freq. < 100 kHz	Na frequência de passagem das lâminas (ordem de máquina *b*)	Muito ruidosa, menos erosiva.
Nuvem	Freq. > 100 kHz	Na frequência de passagem das palhetas (ordem de máquina *v*)	Muito erosiva, menos ruidosa.
Vórtice no tubo de descarga	Não avaliada	Na frequência de Rheingans (ordem de máquina fracionária)	Limita a faixa de operação da turbina. Induz fortes vibrações.
Vórtice de Von Kármán	Não avaliada	Na frequência de derramamento de vórtices (ordem de máquina arbitrária e inteira)	Pode causar "*singing noise*". Danifica o bordo de fuga de lâminas e palhetas por fadiga.

3 METODOLOGIA DESENVOLVIDA

Conforme exposto no capítulo 2, o SVI produzido pela cavitação erosiva em turbinas de geração de energia hidroelétrica é um sinal que exibe cicloestacionariedade de segunda ordem, e em relação à posição angular do eixo da turbina. Para a detecção e identificação dos tipos de cavitação mostrados, é necessário então observar as ordens de máquina (não nulas) presentes no conjunto $\mathbb{A}$. Dependendo dos elementos encontrados em $\mathbb{A}$, a detecção e identificação das cavitações em bolhas, nuvem, no tubo de descarga e em vórtice de Von Kármán pode ser realizada.

Para a estimação da agressividade da cavitação, é necessário avaliar a potência de cada componente de ordem de máquina que seja produzida por cada tipo de cavitação. As diferenças entre as portadoras aleatórias moduladas em cada componente de ordem de máquina podem auxiliar neste processo de *extração de informações* do SVI.

Adicionalmente, os sinais vibracionais *reais* provenientes de turbinas instaladas em usinas geradoras estão contaminados, como é comum em ambientes industriais, por sinais interferentes intensos e de diversas origens, como por exemplo: mecânica, eletromagnética, etc. Considerando que os impactos produzidos pela cavitação são microfenômenos, os sinais interferentes (que serão tratados doravante como *ruídos*) eventualmente podem atingir intensidades até algumas ordens de grandeza acima das intensidades dos SVIs.

Todo sinal vibracional real pode ser decomposto em suas partes determinística (periódica) e aleatória (resíduo) (ANTONI, 2009). Obviamente, o SVI da cavitação ajuda a compor a parte residual do sinal, devido à sua natureza aleatória. Outros fenômenos aleatórios (por exemplo: fricção e ruído mecânico produzido pelo escoamento) também ajudam a compor a parte residual dos sinais vibracionais reais:

$$x(\theta) = x_{CS1}(\theta) + x_{CS2+}(\theta) + \eta(\theta) \tag{15}$$

O sinal $x(\theta)$ representa o sinal vibracional de um sensor arbitrário; $x_{CS1}(\theta)$ é o termo cicloestacionário de primeira ordem que representa fenômenos determinísticos (periódicos); $x_{CS2+}(\theta)$ é o termo que exibe cicloestacionariedade de ordem maior ou igual

a dois e representa os microfenômenos aleatórios; e $\eta(\theta)$ é um termo também aleatório, mas representa os sinais interferentes que não são cicloestacionários ou que são poli-cicloestacionários. O SVI é somente a parcela que exibe cicloestacionariedade pura de segunda ordem, e faz parte de $x_{CS2+}(\theta)$. Algumas fontes de ruído contribuem com $x_{CS1}(\theta)$, como é o caso do ruído eletromagnético conhecido como *hum*, e outras contribuem com $\eta(\theta)$, como por exemplo, o escoamento turbulento e entrada de areia junto com a água na turbina.

Apesar de os sinais vibracionais reais só terem significado no domínio do tempo, e da aquisição de sinais ser controlada por uma *base de tempo* fixa, os SVIs exibem cicloestacionariedade angular, de modo que na equação (15), todos os sinais são do domínio angular. A reconstrução de sinais *discretos* no domínio angular a partir de sinais *discretos* no domínio do tempo e de forma conveniente a esta aplicação de análise vibracional é o assunto da próxima secção.

3.1 DOMÍNIO DO TEMPO VERSUS DOMÍNIO ANGULAR

Os SVIs exibem cicloestacionariedade em relação ao domínio angular, e, a menos que haja um equipamento dedicado à amostragem e gravação sincronizada com o giro do eixo da máquina, os sinais obtidos não são sinais no domínio angular. O autor não dispôs de nenhum equipamento com estas características durante esta pesquisa, e, segundo levantamento feito pelo mesmo, normalmente um equipamento assim é construído sob demanda, significando que ele serve para fazer testes em uma turbina específica, mas pode não servir para outra(s).

Uma alternativa tecnicamente viável é a aquisição de sinais no domínio do tempo, com a amostragem controlada por um oscilador a cristal. Infelizmente, a conversão de um sinal vibracional do domínio do tempo para o domínio angular não é direta, no caso de sinais provenientes de turbinas hidráulicas, devido ao efeito chamado *droop speed control*, que será abordado na próxima secção. Não obstante, a conversão pode ser feita através da *reamostragem angular*, se juntamente com os sinais vibracionais no domínio do tempo, for gravado pelo menos um sinal $\theta(t)$ que represente a posição angular instantânea.

3.1.1 O *droop speed control*

Uma turbina hidráulica produz energia mecânica necessária para fazer girar o eixo de um gerador elétrico (alternador), que é uma máquina rotativa *síncrona*. Normalmente, vários geradores estão conectados ao SIN, e por serem máquinas síncronas, devem necessariamente gerar tensões alternadas sincronizadas em fase, e portanto, na mesma frequência, que no Brasil é *nominalmente* igual a 60 Hz. Isto significa que dois conjuntos turbina-gerador (TG) podem até girar em velocidades angulares diferentes (pois isto depende das características do gerador), mas se a velocidade angular de um conjunto TG se altera, a velocidade do outro conjunto TG deve necessariamente se alterar na mesma proporção.

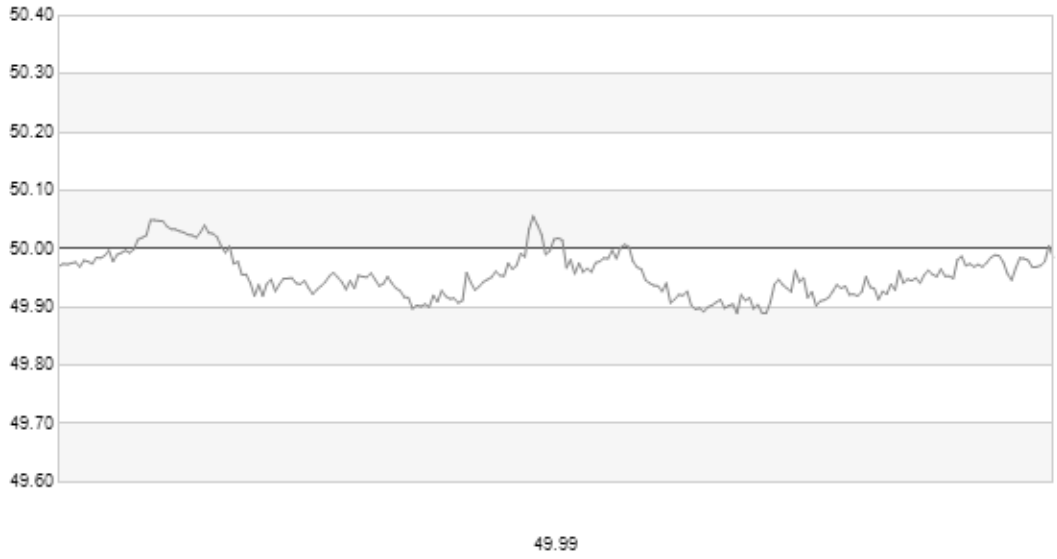


Figura 23 - Variação da frequência instantânea da rede elétrica do Reino Unido, devido ao *droop speed control*. O valor nominal é 50 Hz. No eixo das abcissas está o tempo (sem escalas), e no eixo das ordenadas a frequência instantânea em Hz. O valor 49,99 Hz é a frequência da última amostra.

Fonte:<http://www2.nationalgrid.com/uk/Industry-information/electricity-transmission-operational-data/> Acesso em 25/04/2015 às 19:21.

Se a potência mecânica produzida por todas as turbinas do SIN excede a demanda de potência instantânea a ser entregue pelo SIN aos consumidores, há uma tendência de as turbinas acelerarem, e a frequência da rede elétrica aumenta ligeiramente. Por outro lado, se a potência mecânica é insuficiente, a frequência da rede elétrica cai abaixo do valor nominal (figura 23)

No Brasil é permitido um desvio de até 1 Hz (segundo a norma IEC6100-2-2) acima ou abaixo do valor nominal de 60 Hz (±1,667%), e este desvio é um dos indicadores da qualidade da geração de energia elétrica. Esta margem (denominada *droop speed control*) de desvio na frequência, e portanto, na velocidade angular das turbinas geradoras, é necessária para adequar a potência instantânea gerada à demanda instantânea, que é uma variável aleatória.

3.1.2 O pré-processamento por reamostragem angular

Antes de iniciar o processamento propriamente dito, que realiza a extração de informações dos sinais de vibração, é necessário realizar a conversão dos sinais amostrados no domínio do tempo, para sinais (re)amostrados no domínio angular.

A comparação entre amostragem direta angular e reamostragem angular de sinais vibracionais é assunto de debate entre vários autores do campo de processamento de sinais (ANDRE *et al.*, 2010; ANTONI, 2009; BOURDON *et al.*, 2014; EL BADAOUI, BONNARDOT, 2014). À primeira vista, a amostragem direta parece ser o meio mais conveniente de aquisição de sinais vibracionais que exibem cicloestacionariedade angular. Entretanto, ambos os métodos apresentam suas desvantagens. No caso de sinais de cavitação, a amostragem deve ser feita em taxa elevada o suficiente para a aquisição de vibrações a dezenas de quilohertz. Este critério leva a dezenas de milhares de amostras a cada volta completa do eixo da máquina, o que é inviável, pois a construção de um *encoder* ou a colagem de uma fita zebra com tantas divisões no eixo é muito complicada. A amostragem direta também sofre de não uniformidades, pois nenhum *encoder* é ideal.

Já a amostragem no domínio do tempo pode ser feita com equipamentos mais simples e pode atingir taxas de amostragem elevadas o suficiente para a aplicação. Ainda é necessário um *encoder* para fornecer sinais indicando a posição angular instantânea do eixo, mas este *encoder* pode ser bem mais simples que o citado para o caso da amostragem angular direta. As desvantagens da reamostragem angular são: a necessidade de adquirir e gravar um sinal a mais, e o pré-processamento que é computacionalmente caro.

A própria metodologia empregada para realizar a reamostragem angular também é outro objeto de várias discussões entre autores do campo de processamento de sinais mecânicos (ANTONI, ELTABACH, 2013; BORGHESANI *et al.*, 2012; CAPDESSUS, ANTONI, 2014; GUBRAN, SINHA, 2014; URBANEK *et al.*, 2013), de modo que o autor desta obra decidiu por uma metodologia mais direta e intuitiva a fim de não fugir do escopo da pesquisa: a interpolação com a função *sinc* e reamostragem em intervalos angulares uniformes. Este método de reamostragem angular é denominado na literatura especializada como *computer order tracking* (COT) ou simplesmente *order tracking* (OT).

Basicamente, o COT consiste em usar um sinal que seja conhecidamente sincronizado com a posição angular do eixo de referência, e a partir deste sinal de sincronismo, determinar os instantes em que o(s) novo(s) sinal(is) reamostrado(s) será(ão) computado(s).

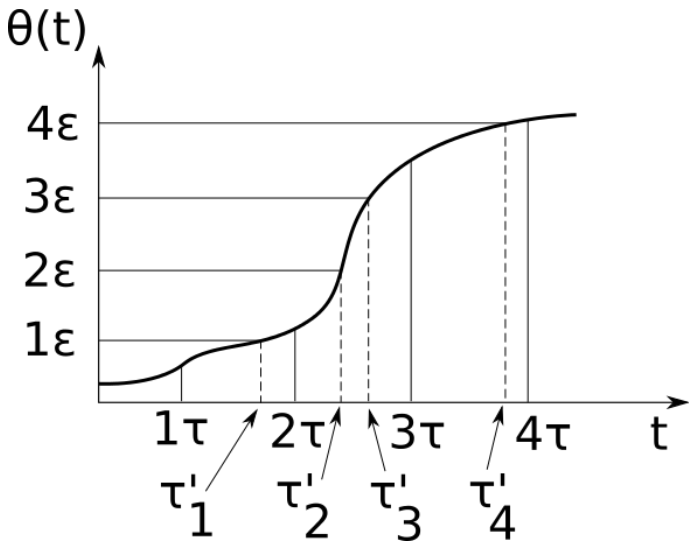


Figura 24 - Determinação dos instantes de reamostragem.

Fonte: Elaboração própria com *freeware* Inkscape v_0.91 Linux.

No gráfico da figura 24, a máquina tem a posição angular instantânea do eixo dada por $\theta(t)$, uma função não linear e bijetora. Os sinais vibracionais são todos amostrados em instantes de tempo múltiplos de τ, mas deseja-se reamostrar os sinais vibracionais em intervalos regulares de ângulo ε. Usa-se então a função inversa de $\theta(t)$ para obter os instantes de tempo correspondentes às posições angulares múltiplas de ε (τ_1', τ_2', τ_3', τ_4', etc).

O nome *order tracking* é sugestivo, pois um sinal de uma determinada ordem de máquina é "seguido" ou "rastreado" em frequência. No caso de uma turbina hidráulica, é possível colar uma fita em seu eixo e usar um tacômetro ótico para gerar um pulso por cada passagem da fita pelo sensor, ou seja, um pulso por volta. Neste caso, a ordem de máquina de número 1 está sendo seguida, e sabe-se que a cada intervalo entre dois pulsos consecutivos o eixo completou uma volta. Portanto, pode-se dividir o intervalo entre dois pulsos consecutivos em quantos instantes de amostragem forem necessários e realizar a reamostragem. Obviamente, qualquer variação da velocidade angular que existir dentro do intervalo entre os dois pulsos introduzirá erros.

Uma alternativa melhorada para o caso de um conjunto TG, é que, se o gerador é síncrono, a tensão alternada gerada está necessariamente em uma frequência múltipla da frequência de rotação do eixo do conjunto TG. Em outras palavras, a tensão alternada gerada constitui uma componente de ordem de máquina. Mais especificamente, se o gerador possui P polos magnéticos (P sempre é par), então a cada giro do eixo passarão em um determinado enrolamento da armadura, $P/2$ polos norte e $P/2$ polos sul, produzindo respectivamente $P/2$ semiciclos positivos e $P/2$ semiciclos negativos na tensão gerada. Conclui-se que, por definição, a tensão alternada senoidal de saída do gerador é (exceto por eventuais variações no ângulo de carga do gerador) um sinal na ordem de máquina $P/2$.

Pode-se usar então, por exemplo, as transições positivas do sinal senoidal como os instantes de referência, e entre 2 transições positivas determinar quantos instantes de reamostragem forem necessários. Esta modificação por si só já permite um melhor "rastreio" da velocidade angular instantânea, pois usando a ordem de máquina $P/2$ como referência, é possível corrigir a nova reamostragem $P/2$ vezes por volta do eixo, ao invés de somente uma vez por volta, como no caso do tacômetro óptico.

Após determinados os instantes desejados para reamostragem, é necessário reconstruir os sinais vibracionais em tempo contínuo, para que sejam reamostrados. Pode-se empregar qualquer método de reconstrução de sinal, como a interpolação linear, porém a interpolação *sinc* é o método que teoricamente resulta na reconstrução perfeita de um sinal de banda limitada, a partir das suas amostras discretas.

O diagrama de blocos da reamostragem pode ser visto na figura 25.

Inicialmente, o sinal (de um sensor arbitrário) temporal, discreto e limitado no tempo (somente um trecho finito para permitir a reconstrução com um número finito de amostras) $x(m\tau)$, onde τ é o inverso da taxa de amostragem e m é um número natural (o número da amostra), é processado por um filtro de reconstrução de sinal com *kernel* tipo função *sinc*. O resultado, que é o sinal contínuo no tempo $x(t)$ e teoricamente é a reconstrução perfeita do sinal $x(m\tau)$, é reamostrado nos instantes que correspondem a posições angulares regularmente espaçadas. Desta forma, o novo sinal discreto no domínio angular $x[n]$, onde n é o número da amostra, pode ter um número de amostras que é independente do número de amostras do sinal original.

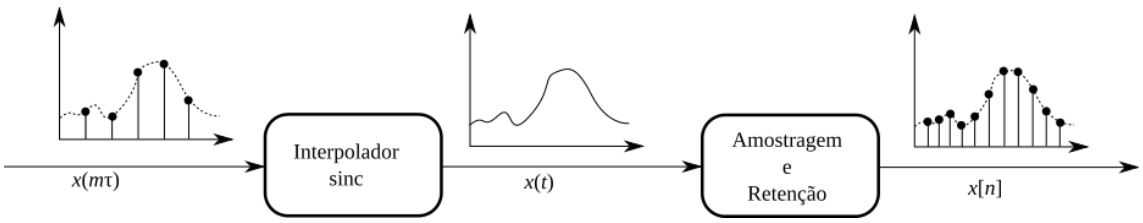


Figura 25 - Reconstrução do sinal contínuo a partir da interpolação *sinc*, e posterior reamostragem do sinal na nova taxa de amostragem.

Fonte: Elaboração própria com *freeware* Inkscape v_0.91 Linux.

O novo sinal vibracional, agora reamostrado no domínio angular, está pronto para o processamento propriamente dito. O autor decidiu por utilizar as transições positivas da tensão senoidal do gerador como os instantes de referência para a reamostragem angular, e também que a reamostragem deve sempre produzir um número M (sendo M fixo e potência de dois) de amostras entre duas transições da tensão senoidal. Adicionalmente, se a cada volta do eixo são produzidos $P/2$ ciclos senoidais, então são produzidas L amostras a cada volta do eixo, com $L= M \cdot P/2$. O número M escolhido é a menor potência de 2 que seja superior ao número original de amostras, caracterizando uma sobreamostragem. Todos estes critérios serão importantes na etapa de processamento do sinal, onde serão empregadas janelas de análise com N amostras, onde N é uma potência de 2 (por questões de performance computacional), e portanto, L é um múltiplo inteiro de N (a cada volta do rotor haverá sempre um número inteiro de janelas de análise).

3.2 DECOMPOSIÇÃO DETERMINÍSTICA/ALEATÓRIA DOS SINAIS

Sabendo que o SVI é somente uma parcela do termo cicloestacionário de segunda ordem na equação (15), é natural iniciar o processamento dos sinais reais e já convenientemente reamostrados, pela separação dos seus termos determinístico e aleatório (ANTONI, 2009; D'ELIA, DELVECCHIO, 2011). Esta separação pode ser feita pela média sincronizada das amostras, que na literatura é denominada *time synchronous average* (TSA), ou mais propriamente *synchronous averaging* (SA), já que os sinais processados não são sinais temporais.

O SA é um estimador da parte determinística do sinal cicloestacionário, e é apropriado para sinais cicloestacionários que têm um número inteiro de amostras a cada ciclo, condição que já foi garantida pela reamostragem angular descrita na secção 3.1.2. Outra estimativa da parte determinística (periódica) do sinal é obtida pela transformação do sinal no domínio angular para o domínio da ordem de máquina através da transformada de Fourier, e converter novamente para o domínio angular com uma *série* de Fourier inversa. Neste caso entretanto, é necessário selecionar os picos em ordens de máquina inteiras antes de realizar a conversão para o domínio angular (ANTONI, 2009).

A estimativa obtida com o SA é mais intuitiva e direta, e consiste em tomar a média de amostras que têm as mesmas propriedades estatísticas. Esta intuição vem do entendimento de um processo cicloestacionário, com L amostras por ciclo, como sendo equivalente a *L processos estacionários intercalados*. Por exemplo: a temperatura máxima diária na cidade do Rio de Janeiro é um processo cicloestacionário, pois o ciclo é de um ano. Há então L=365 amostras de temperatura ao longo de um ciclo. Calcular a média entre a temperatura do primeiro dia de março e o primeiro dia de junho, por exemplo, não faz qualquer sentido, pois estas temperaturas não possuem as mesmas características estatísticas. Entretanto, se ao longo de muitos anos (ciclos) for feita a média das temperaturas do primeiro dia de março (que por uma aproximação razoável são amostras que possuem as mesmas características estatísticas), esta média é a *estimativa da parte determinística* da temperatura do primeiro dia de março. Pode-se calcular esta média para cada um dos 365 dias do ano, e os resultados intercalados formarão o termo determinístico da temperatura ao longo do ano (ciclo), aquele devido aos fenômenos determinísticos (inclinação e proximidade da Terra em relação ao

Sol, ...).

Este processamento pode ser feito também para sinais amostrados no domínio angular. Por exemplo, a figura 26 mostra os sinais angulares de pressão acústica obtidos de vários ciclos diferentes de um motor a diesel. Nota-se que, para a faixa angular compreendida entre 386º e 387,5º aproximadamente, os sinais têm boa repetibilidade (a parte determinística é dominante). Já para a faixa angular entre 388º a 390º, é notável a diferença entre os ciclos (a parte aleatória é dominante).

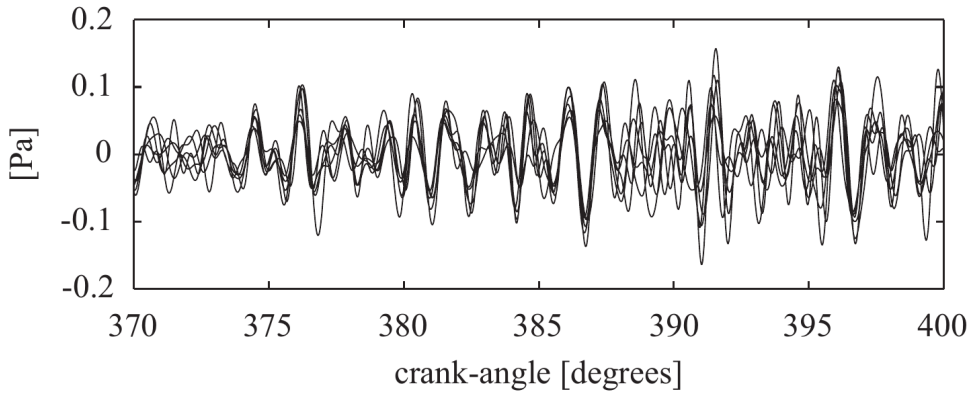


Figura 26 - Sobreposição de vários ciclos de um sinal cicloestacionário (pressão acústica produzida por um motor a diesel) dentro de uma faixa angular estreita.

Fonte: (ANTONI, 2009).

Supondo que o sinal vibracional de um sensor arbitrário $x(\theta)$, definido na equação (15), seja representado em forma discreta $x[n]$ após a reamostragem angular, e que K revoluções do eixo foram gravados, com L amostras por revolução, então o termo determinístico $x_{CSI}[n]$ (cicloestacionário de primeira ordem) pode ser estimado por:

$$x_{CSI}[n]=\frac{1}{K}\sum_{k=0}^{K-1} x[kL+m] \tag{16}$$

O sinal $x_{CSI}[n]$ se repete a cada L amostras, pois m é o resto da divisão inteira entre n e L (na figura 27 há um exemplo com K=7 ciclos, onde sinais vibracionais de uma caixa de engrenagens são processados).

O processamento com o estimador SA é o equivalente a filtrar o sinal $x[n]$ com um *comb filter*, que seleciona somente as componentes harmônicas da frequência de

revolução do eixo da turbina. O termo residual $x_r[n]$ é portanto, a diferença entre o sinal vibracional real e a sua parte determinística:

$$x_r[n] = x[n] - x_{CS1}[n] = x_{CS2+}[n] + \eta[n] \tag{17}$$

Tanto o termo residual $x_r[n]$ quando o termo determinístico $x_{CS1}[n]$ têm o mesmo número de amostras do sinal vibracional real $x[n]$: $K \cdot L$ amostras. No resíduo restaram o termo cicloestacionário de segunda ordem (ou ordem maior) $x_{CS2+}[n]$ e o termo não cicloestacionário e/ou poli-cicloestacionário $\eta[n]$.

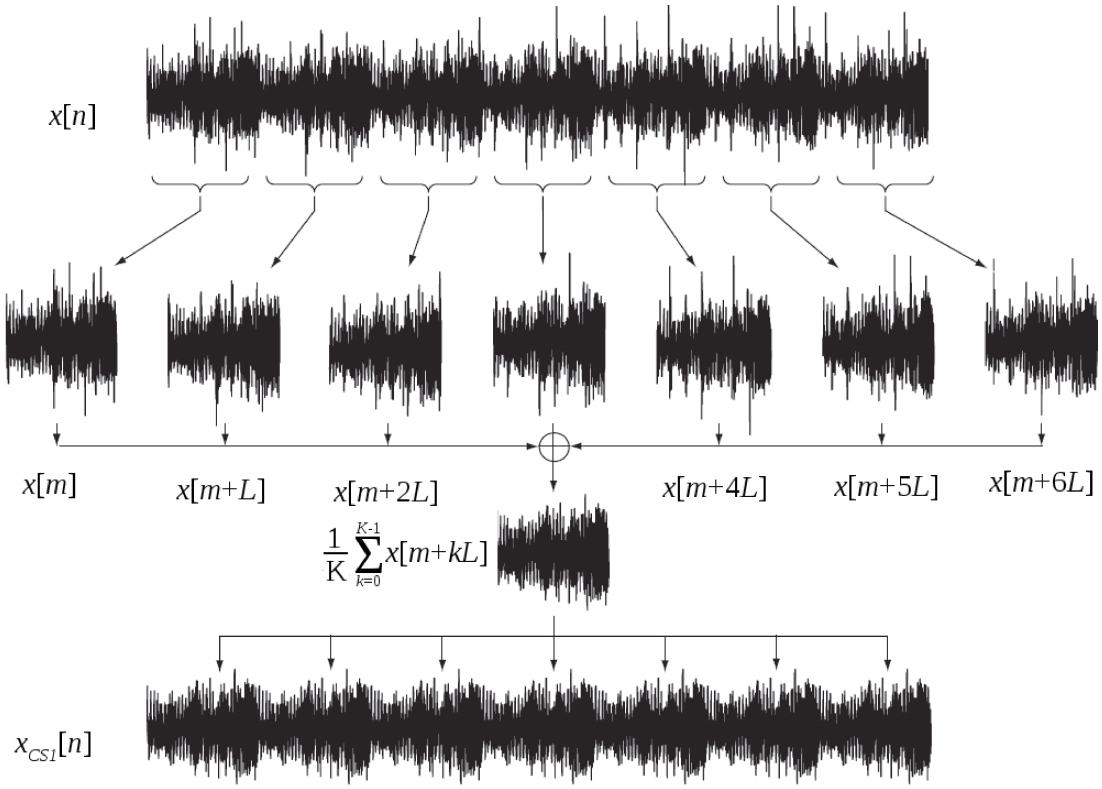


Figura 27 - Ilustração do princípio da estimação da parte determinística do sinal *x*[*n*].

Fonte: Adaptado de (ANTONI, 2009) com o *freeware* GIMP v_2.8 Linux.

O termo residual é chamado de *sinal resíduo,* e é composto pelo SVI da cavitação e pelas outras fontes de ruídos aleatórios. Interferências determinísticas como o *ruído hum* (eletromagnético ou mecânico) são eliminadas nesta parte do processamento. O sinal resíduo deve ser processado para revelar as periodicidades escondidas (e com isso discriminar entre diferentes fenômenos que provocam modulações), e para realçar a parte cicloestacionária de segunda ordem pura.

3.3 REALCE DO TERMO CICLOESTACIONÁRIO DE SEGUNDA ORDEM PURO

O sinal resíduo $x_r[n]$, como resultado da remoção do termo determinístico do sinal $x[n]$, é composto por parcelas que exibem cicloestacionariedade de ordem maior ou igual a dois, além de possivelmente conter termos não cicloestacionários ou poli-cicloestacionários. Entretanto, o SVI da cavitação é um sinal cicloestacionário de segunda ordem puro, no caso da cavitação em bolhas itinerantes, cavitação em nuvem e no caso particular da cavitação em vórtices de Von Kármán acompanhada do fenômeno *lock-in*. Já no caso da cavitação em vórtice no tubo de descarga, espera-se que o sinal indiretamente produzido exiba cicloestacionariedade de segunda ordem pura.

A separação dos termos de interesse neste caso somente pode ser feita com ferramentas de análise cicloestacionárias específicas para cicloestacionariedade de segundo grau, como por exemplo, a representação do sinal em um domínio misto entre duas grandezas, como o *domínio tempo-frequência* (GARDNER, 1986). A transformada de Fourier de tempo curto (*short time Fourier transform* ou STFT) é um exemplo de representação no domínio tempo-frequência, e é o mais intuitivo e simples, se comparado com outras representações tempo-frequência como a transformada *wavelet*. A STFT é computada a partir da transformada discreta de Fourier (DFT) ou da equivalente transformada rápida de Fourier (FFT, que é computacionalmente mais eficiente e produz exatamente o mesmo resultado da DFT), e janelas de análise.

A transformada de Fourier de um sinal no domínio do tempo resulta na representação do mesmo sinal no domínio da frequência, ou seja, a sua decomposição em componentes (de frequência) espectrais. A STFT então, como uma extensão da DFT ou FFT para processos não estacionários, tem o significado de *evolução temporal da composição em frequências espectrais* do sinal. É importante enfatizar aqui que o termo "frequências espectrais" está sendo empregado, apesar de parecer redundante, para diferenciação das frequências cíclicas citadas na secção 2.3.1.4 (ANTONI, 2009).

No caso dos sinais vibracionais de domínio angular, o significado é ligeiramente diferente, haja vista que a transformada de Fourier resulta na representação do sinal em componentes de ordem de máquina. Entretanto, no caso dos sinais de vibração das

turbinas, não é conveniente a representação em ordens de máquina, pois não há sentido físico em tal representação.

Novamente, uma forma de contornar este problema e produzir um resultado com significado físico, é assumir que a turbina está a uma *velocidade constante e igual à velocidade nominal de rotação*. A reamostragem angular descrita na secção 3.1.2 já produz um sinal que permite assumir com boa aproximação que esta condição é verdadeira, exceto por poucas imperfeições na reamostragem. Além disso, a STFT dos sinais vibracionais reamostrados nesta condição tem duplo significado físico: a *evolução angular da composição em frequências espectrais do sinal*, ou a evolução *temporal* da composição em frequências espectrais do sinal, exatamente como no caso de sinais vibracionais temporais (pois tempo e posição angular são equivalentes quando a turbina está a velocidade constante).

Para tornar evidentes as periodicidades escondidas em sinais que exibem cicloestacionariedade de segunda ordem, é necessário pois que haja um bloco de processamento não linear de ordem dois. Conectar em cascata um bloco que realiza uma transformação linear (a transformada STFT, que resulta em uma matriz de números complexos) com um bloco que eleva ao quadrado o módulo dos números complexos é exatamente o mostrado na metade inferior da figura 13. A função linear $h(t,f,\Delta f)$ representa a STFT de um sinal temporal, pois depende de t (o instante de tempo), f (a frequência espectral) e Δf (a resolução em frequência, que depende da janela de análise empregada).

Tecnicamente, a ligação em cascata dos dois blocos produz como resultado uma matriz conhecida como o *espectro de potência instantâneo*, ou *instantaneous power spectrum* (IPS) (CHEN *et al.*, 2005). O significado físico desta matriz é a distribuição da energia do sinal entre valores *discretos* de frequência espectral ao longo do tempo (ou da posição angular). Uma diferença conceitual (e sutil) entre o espectrograma e o IPS, é que o espectrograma é a evolução temporal da *densidade espectral de potência*, e portanto é uma função *contínua* no domínio da frequência espectral (HEINZEL, 2002).

O IPS do sinal resíduo é calculado da seguinte forma: $x_r[n]$ é dividido em K segmentos correspondentes às K revoluções do eixo da turbina, com L amostras em cada segmento. Para cada um dos K segmentos, uma janela de análise $w[n]$, com

comprimento N amostras é introduzida e deslocada por passos de R amostras, sendo $R<N$. A janela deslocada $w_N[n]=w[n-iR]$ realiza um corte suave do segmento entre as amostras $n=iR,...,iR+N-1$, e tipicamente é uma janela do tipo Hanning. A FFT do produto $w_N[n]\cdot x_r[n]$ é calculada, sendo que há $I=1+(L-N)/R$ janelas de análise a cada revolução do eixo, e o mesmo número de FFTs de comprimento N. A fração de *overlap* entre duas janelas consecutivas, neste caso, é $1-R/N$.

Para janelas do tipo Hanning e frações de *overlap* igual a 50% ou 75%, a sobreposição de janelas consecutivas produz ganho em amplitude com planura perfeita, ou *perfect amplitude flatness* (HEINZEL, 2002). Na figura 28 é mostrado um gráfico com janelas tipo Hanning e 50% de *overlap*. A sobreposição das janelas produz um ganho constante e unitário, conforme indicado pela linha tracejada.

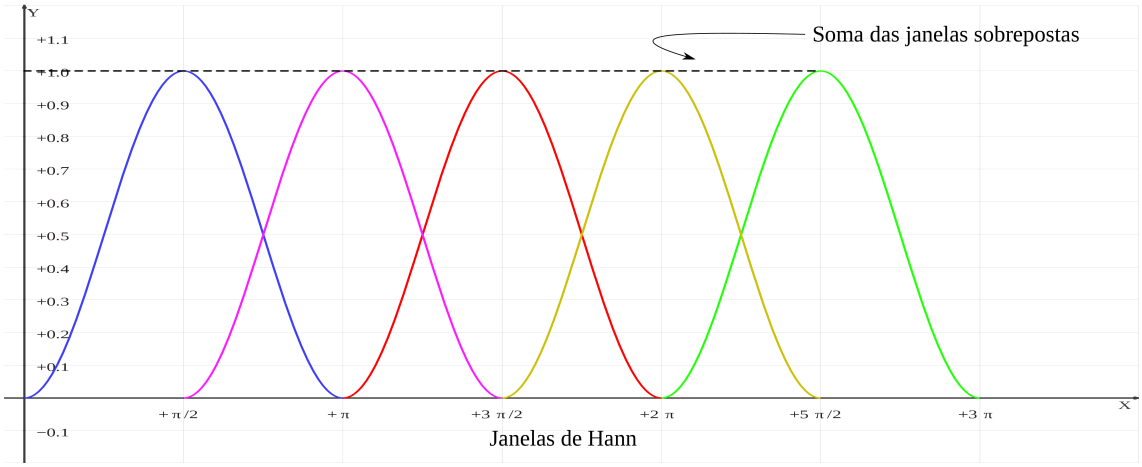


Figura 28 - Sobreposição de janelas Hanning com 50% de *overlap*, e planura perfeita no ganho em amplitude, que permanece unitário.

Fonte: Elaboração própria com *freeware KmPlot 1.2.1 forLinux*.

Uma fração de *overlap* igual ou maior a 75% é necessária para evitar posterior *vazamento cíclico* (fenômeno análogo ao *spectral leakage*), e também o vazamento cíclico resultante da amostragem no domínio angular, mas agora na nova taxa que é R vezes menor que a taxa original (em outras palavras: *aliasing*, mas no domínio da frequência cíclica) (ANTONI, HANSON, 2012).

A matriz complexa resultante tem I colunas e $N/2+1$ linhas, e é a STFT do sinal resíduo. O IPS do resíduo relativo à k-ésima volta é o módulo ao quadrado de cada elemento da matriz complexa:

$$IPS_k[i,h] = \left| \sum_{n=iR}^{iR+N-1} x_r[n] w_N[n] e^{j2\pi h \frac{n-iR}{N-1}} \right|^2 \quad (18)$$

Na equação (18), *i* é a variável discreta do domínio angular e *h* é a variável discreta representando a frequência espectral. O comprimento *N* da janela afeta o IPS, pois a resolução no domínio angular cai *N* vezes (piora), mas permite a discriminação de *N*/2 bandas de frequências espectrais.

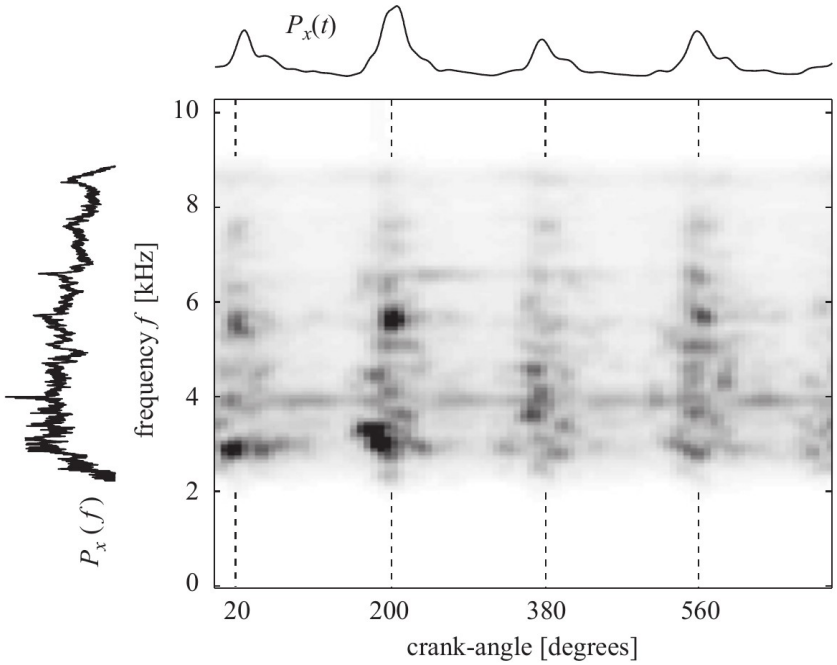


Figura 29 - Exemplo de IPS computado em sinal acústico de motor a diesel. A janela de análise empregada permite resolução angular de 21º. Os pulsos de energia acústica se repetem a cada 180º.

Fonte: (ANTONI, 2009).

O comprimento *N* deve ser tal que permita a discriminação de eventos produzidos pela passagem de uma única lâmina ou palheta, e para isso, a janela de análise deve ter duração temporal duas ordens de grandeza abaixo do período de revolução do eixo da turbina (para o caso de turbinas com dezenas de lâminas e palhetas). Um exemplo de IPS pode ser visto na figura 29, e adicionalmente, a evolução da potência ao longo do eixo angular (temporal) $P_x(t)$ e a distribuição ao longo do eixo de frequências espectrais $P_x(f)$, que é a *power spectral density* (PSD) do sinal.

O IPS médio, ou *averaged instantaneous power spectrum* (AIPS), é computado levando todas as *K* revoluções do eixo em conta, em uma média aritmética simples:

$$AIPS[i,h]=\frac{1}{K}\sum_{k=0}^{K-1} IPS_k[i,h] \tag{19}$$

Cada elemento das matrizes $IPS_k[i,h]$ tem o significado de potência do sinal aleatório presente na frequência discreta h e durante a i-ésima janela de análise angular. O cálculo da média mostrado na equação (19), usando raciocínio análogo ao aplicado à equação (16), é o *estimador do termo determinístico da potência instantânea* (*angular*)*, mas discriminada em frequências espectrais*.

Sinais que *são* cicloestacionários de segunda ordem em relação ao ângulo têm a potência variando de forma sincronizada com as revoluções do eixo, e a operação de média aritmética realça esses sinais. Por outro lado, sinais que *exibem* cicloestacionariedade não são sincronizados, e por isso são atenuados nesta etapa do processamento, o mesmo acontecendo com o termo não cicloestacionário $\eta[n]$ da equação (17). A primeira conclusão prática que pode ser obtida é que os sinais produzidos pela cavitação em bolhas itinerantes, cavitação em nuvem e em vórtice de Von Kármán são realçados em relação aos sinais que não *são* cicloestacionários. Infelizmente, o sinal produzido pela cavitação em vórtice no tubo de descarga será atenuado também, não permitindo doravante a sua detecção indireta.

A segunda conclusão é que, da mesma forma que o AIPS calcula o termo determinístico da potência, faz sentido falar da existência de um termo residual. O AIPS é uma matriz com I colunas e $N/2+1$ linhas, calculada em função de K revoluções do eixo, e portanto, a matriz resíduo do cálculo do AIPS é uma matriz com $N/2+1$ linhas, mas com $K \cdot I$ colunas. Apesar de as informações sobre a cavitação em vórtice no tubo de descarga constarem na matriz resíduo, eventualmente o processamento dessa matriz se torna computacionalmente caro demais.

A reamostragem angular realizada anteriormente ao cálculo do AIPS e descrita na secção 3.1.2, por configurar uma sobreamostragem, produz linhas na matriz AIPS que correspondem a frequências espectrais que estão acima da *frequência de Nyquist do sinal original*, no domínio do tempo (se o sinal foi sobreamostrado, a sua frequência de Nyquist aumenta na mesma relação de reamostragem). Estas linhas não contém informações úteis, e portanto, são descartadas.

3.4 DISCRIMINAÇÃO DAS COMPONENTES DE ORDEM DE MÁQUINA

O AIPS do sinal residual é uma representação da energia distribuída em ambos os domínios angular e de frequência espectral, e que atenua os termos que *exibem* cicloestacionariedade, sem atenuar os termos que *são* cicloestacionários. Portanto, o sinal representado pelo AIPS é uma combinação de componentes de ordens de máquina de número inteiro, e pode ser analisado com ferramentas específicas, como por exemplo a *Cyclic Modulation Spectrum* (CMS) e a *Cyclic Modulation Coherence* (CMC).

A CMC, como proposta originalmente por J. Antoni, é calculada a partir do espectrograma do sinal (ANTONI, HANSON, 2010, 2012; ANTONI, 2009). No presente trabalho, a CMS e a CMC são computadas de uma maneira ligeiramente diferente, pois o cálculo é realizado a partir do AIPS do sinal, e será exposto com detalhes na secção 3.4.2. O autor prefere neste ponto enfatizar o significado da CMS aplicado ao escopo desta pesquisa.

3.4.1 O significado da CMS

A CMS é capaz de tornar evidente qualquer modulação em amplitude oculta, pois o sinal passa ser representado como a potência distribuída em dois domínios: o domínio da frequência espectral (ou frequência da portadora) e o domínio da frequência cíclica (ou frequência da modulante). Seu cálculo é feito aplicando a transformada de Fourier no espectrograma do sinal, o que converte o domínio do tempo no domínio da frequência cíclica α.

A título de exemplo, na figura 30 estão representados dois sinais modulados em amplitude (AM) pelo mesmo sinal modulante senoidal $m(t)=1+sen(2\pi\alpha_0 t)$, exibido em linhas tracejadas. O *offset* unitário adicionado à função seno evita que $m(t)$ seja negativo, e produz modulação em amplitude com portadora.

Na metade superior da figura 30, a portadora é determinística: um sinal senoidal puro na frequência f_0. O sinal modulado em AM é composto somente por componentes periódicas, nomeadamente sinais senoidais nas frequências f_0 (a portadora), $f_0+\alpha_0$ (a banda lateral superior) e $f_0-\alpha_0$ (a banda lateral inferior). Conclui-se que o sinal modulado

em AM com portadora senoidal exibe cicloestacionariedade de primeiro grau.

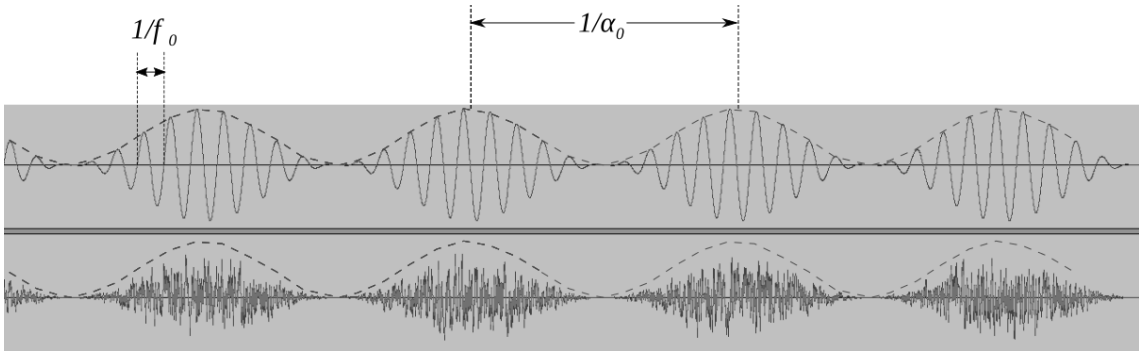


Figura 30 - Dois sinais cicloestacionários e suas componentes modulantes senoidais em linhas tracejadas. Acima: Portadora determinística. Abaixo: Portadora aleatória.

Fonte: Elaboração própria com Audacity 2.0.5, InkScape 0.91 e SoX 14.4.0 Linux.

Na metade inferior da figura 30, a portadora é aleatória, e supondo que seja um sinal com densidade espectral de potência constante entre as frequências f_1 e f_2, e nula fora deste intervalo, o sinal resultante da modulação em AM é cicloestacionário de segunda ordem *puro*.

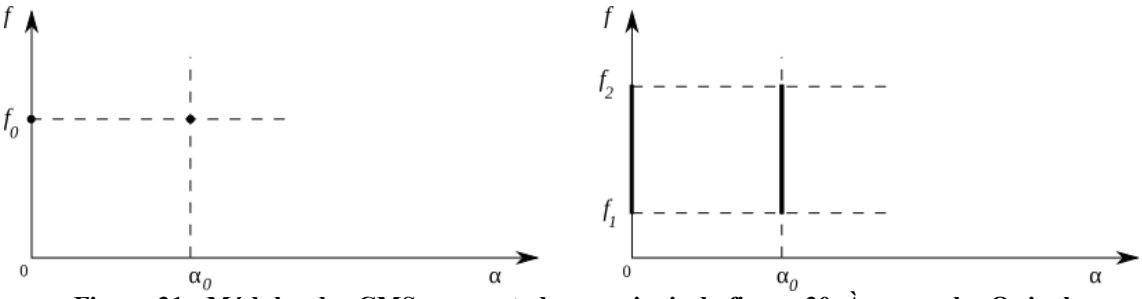


Figura 31 - Módulos das CMSs computadas nos sinais da figura 30. À esquerda: O sinal cicloestacionário de primeiro grau (portadora senoidal). À direita: o sinal cicloestacionário de segundo grau puro (portadora aleatória).

Fonte: Elaboração própria com o *freeware* InkScape v_0.91 Linux.

Ao aplicar a transformada de Fourier nos espectrogramas de ambos os sinais da figura 30 e plotar os módulos dos elementos individuais da matriz complexa resultante (CMS), as imagens formadas podem ser vistas na figura 31. Com a portadora determinística, a imagem formada pela matriz CMS mostra somente dois pontos: um em (α_0,f_0), que representa o termo cuja potência varia na frequência α_0, e outro nas coordenadas $(0,f_0)$, indicando o termo estacionário (a portadora que não foi modulada devido à adição do 1 no sinal modulante). Já no caso da portadora aleatória (à direita), a

imagem formada mostra duas *linhas* paralelas ao eixo das frequências espectrais, nas abcissas 0 e α_0. Isto significa que, apesar de a portadora ser aleatória, a potência (variância) transportada pelo sinal varia na frequência α_0, exatamente como no caso da portadora determinística. A diferença é que a portadora tem a sua potência distribuída na faixa de frequências espectrais de vai de f_1 a f_2. A linha na abcissa zero é o termo estacionário do sinal modulado.

Duas conclusões podem ser tiradas deste exemplo: a primeira é que a CMS é uma maneira de evidenciar sinais modulados em amplitude, não importando se a portadora é determinística ou aleatória. Esta característica da CMS é possível, pois o processo de modular em amplitude sinais no domínio do tempo é o equivalente a produzir sinais no domínio da frequência que possuem componentes correlacionadas. No caso da portadora determinística, as duas bandas laterais sempre têm a mesma amplitude, caracterizando uma correlação espectral. No caso da portadora aleatória, as bandas laterais foram espalhadas, mas não perderam a correlação espectral entre si.

A segunda conclusão, e que tem efeitos práticos para esta pesquisa, é que os sinais da cavitação por serem cicloestacionários de segundo grau puro aparecem como linhas paralelas ao eixo da frequência espectral, deixando uma "sombra" na abcissa zero. Sinais cicloestacionários de primeiro grau aparecem como pontos isolados, e sinais estacionários em amplo senso (frequência cíclica nula) devem necessariamente aparecer como pontos ou linhas sobre o eixo da frequência espectral.

3.4.2 Cálculo da CMS a partir do AIPS

No presente trabalho, a discriminação das potências das componentes de ordem de máquina (cálculo da CMS) é conseguida inicialmente aplicando a transformada de Fourier ao AIPS, mas desta vez no sentido de transformar a representação no domínio angular em representação no domínio da ordem de máquina. A variável discreta a representa o domínio da ordem de máquina, análoga à variável α do domínio da frequência cíclica:

$$CMS[a,h]=\frac{1}{I}\sum_{i=0}^{I-1} AIPS[i,h]e^{j2\pi a\frac{i}{I-1}} \quad (20)$$

A variável *a* assume valores adimensionais e inteiros somente, pois o cálculo da CMS foi feito com base no AIPS, que contém as informações equivalentes a uma revolução completa do eixo da turbina. Portanto, atenção especial deve ser dada aos valores calculados da CMS para *a* igual a múltiplos inteiros de *b* ou *v*, pois são as componentes de ordem da máquina produzidas pela RSI da turbina.

A matriz CMS é uma matriz de números complexos, e cada elemento *CMS*[*a*,*h*] representa a potência da componente (de fase aleatória) de frequência espectral *h* que é modulada na ordem de máquina *a*. A representação complexa fornece o módulo e a fase da potência em relação à ordem de máquina de referência (um). O módulo ao ser plotado como uma imagem em tons de cinza, produz gráficos como o da figura 31. Para o caso particular de *a*=0 (sinais sem modulação), *CMS*[*0*,*h*] é um estimador do *espectro de potência* do sinal resíduo $x_r[n]$, ou seja *PS*[*h*], o equivalente discreto da PSD. Obviamente, para todos os valores de *h*, *CMS*[*0*,*h*] é um número real, pois representa uma componente estacionária.

3.4.3 Cálculo da CMC e o seu significado

Eventualmente, algumas faixas espectrais do sinal $x_r[n]$ podem conter sinais modulados em amplitude, mas com potência relativamente muito baixa em relação a outras faixas espectrais (que podem ou não apresentar modulação também). Pode ocorrer então uma espécie de "mascaramento" das componentes nas faixas com menor potência, o que torna a visualização na imagem formada por |*CMS*[*a*,*h*]| difícil. Este fenômeno de "mascaramento" de certas faixas espectrais pode ocorrer, por exemplo, se a função de transferência mecânica ligando a fonte de excitação ao sensor tiver zeros na transmissão, dentro ou próximos dessas faixas.

Uma solução para contornar essa condição desfavorável é computar a relação entre a quantidade de potência que varia devido à modulação e a potência da parcela estacionária; algo semelhante ao índice de modulação em amplitude, porém aplicado à modulação em potência.

A CMC então é calculada a partir da CMS, dividindo-se cada linha da matriz complexa $CMS[a,h]$ pelo valor de $CMS[0,h]$, ou seja, o sinal sofre branqueamento:

$$CMC[a,h]=\frac{CMS[a,h]}{CMS[0,h]} \tag{21}$$

A coluna $CMC[0,h]$ tem todos os seus elementos iguais a 1, e portanto é descartada, e com ela, a contribuição das parcelas aleatórias estacionárias do sinal. Adicionalmente, se no sinal resíduo $x_r[n]$ existirem componentes periódicas e não moduladas (obviamente não sincronizadas com o eixo da turbina), estas também serão eliminadas, pois a energia que essas componentes transportam flui de maneira constante (estacionária). Conclui-se que a imagem formada por $|CMC[a,h]|$ é uma representação das componentes cicloestacionárias do sinal somente, excluindo as componentes estacionárias em amplo senso. Finalmente, $|CMC[a,h]|$ significa o *grau de cicloestacionariedade* da componente de frequência espectral h modulada na ordem de máquina a, e a imagem formada por $|CMC[a,h]|$ constitui um *indicador visual* da presença de cicloestacionariedade do sinal. Fatalmente, a cavitação erosiva (ou até mesmo outra falha), se estiver presente, produzirá componentes que aparecerão na imagem formada, mesmo sob relações sinal/ruído desfavoráveis (ANTONI, HANSON, 2010).

3.5 ESTIMAÇÃO DA AGRESSIVIDADE DE CADA TIPO DE CAVITAÇÃO

Conforme exposto na secção 3.4.2, as componentes de sinal produzidas pela cavitação em nuvem e em bolhas devem aparecer como linhas verticais na imagem formada por $|CMS[a,h]|$, para valores de a iguais a múltiplos inteiros de v e b, respectivamente. No exemplo ilustrado pela figura 31 (metade direita), o sinal cicloestacionário de segunda ordem puro foi produzido a partir de uma portadora aleatória com densidade espectral de potência constante entre f_1 e f_2, o que resultou em uma linha contínua na imagem formada por $|CMS[a,h]|$.

No caso de sinais reais, é improvável que a portadora aleatória possua densidade espectral de potência constante, pois ela é o resultado da combinação entre a fonte de

excitação e a função de transferência mecânica. Como a cavitação em nuvem e a cavitação em bolhas itinerantes têm mecanismos de formação diferentes, então as excitações acústicas produzidas por elas são diferentes. Ademais, ambos tipos de cavitação causam erosões em lugares diferentes da turbina, obrigando os sinais a se propagarem por caminhos diferentes até o sensor, o que resulta em funções de transferência também diferentes.

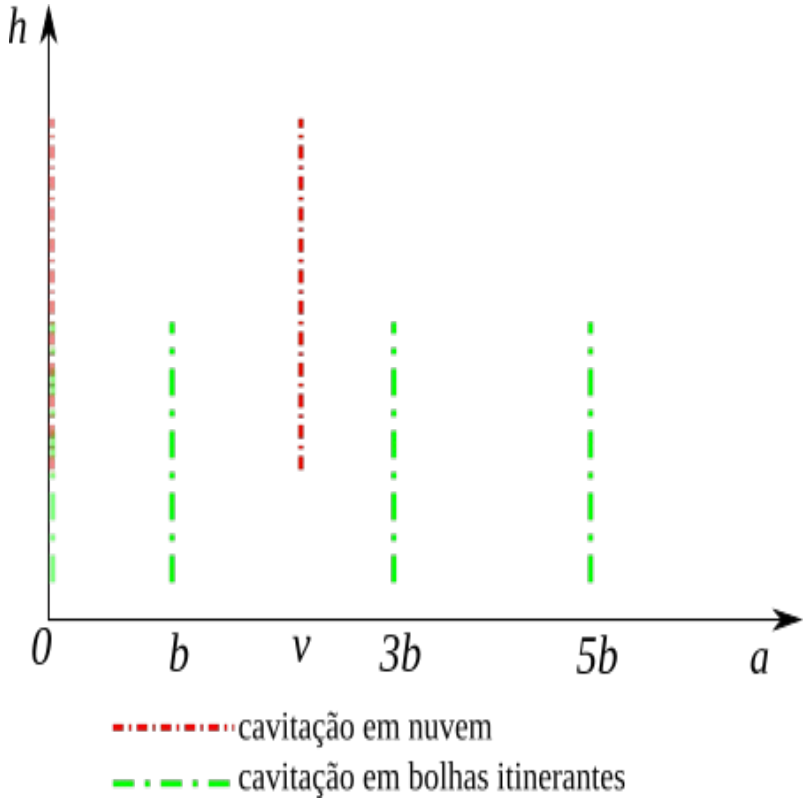


Figura 32 - CMS de sinais produzidos por diferentes tipos de cavitação.

Fonte: Elaboração própria com o *freeware* InkScape v_0.91 Linux.

Na prática, ambos os tipos de cavitação possuem uma combinação única, e que se reflete na portadora aleatória produzida. Na imagem formada por $|CMS[a,h]|$, estas portadoras aparecem como linhas *descontínuas*, porém com padrões de descontinuidade diferentes entre si, o que permite identificar em quais ordens de máquina estão as componentes produzidas pelo mesmo tipo de cavitação (figura 32).

Além disso, a cavitação em vórtices de Von Kármán influencia indiretamente a cavitação em bolhas ou a cavitação em nuvem, e a sua influência pode ser distinguida pelas características da portadora aleatória. Na figura 32, a cavitação em vórtices de Von

Kármán apareceria como componentes nas ordens de máquina b+1 ou v+1 (ESCALER *et al.*, 2006b).

Obviamente, o diagnóstico da cavitação pode ser feito visualmente através das imagens formadas por $|CMS[a,h]|$ e $|CMC[a,h]|$. Entretanto, este diagnóstico é apenas qualitativo, pois ainda não fornece informações sobre a agressividade da cavitação. Ademais, o diagnóstico visual exige a intervenção de um operador humano, o que inviabiliza a elaboração de laudos técnicos de forma automatizada e contínua.

Um diagnóstico quantitativo sobre a agressividade da cavitação, ou mais propriamente, a estimativa da agressividade de cada tipo de cavitação pode ser calculada a partir da matriz CMS, lembrando que a potência da componente estacionária é a agressividade da cavitação (BAJIC, 2002; HAMMITT, DE, 1979), mas esta componente é impossível de ser separada do ruído estacionário e dos outros sinais de cavitação (a potência da componente estacionária é a própria potência média do SVI). Assim, as potências de todos os pontos em verde que estão sobre o eixo das frequências espectrais na figura 32 são somadas para compor a estimativa da agressividade da cavitação em bolhas itinerantes (P_B). Da mesma forma, as potências de todos os pontos vermelhos sobre o eixo das frequências são somadas para fornecer a estimativa da agressividade da cavitação em nuvem (P_C).

Os pontos verdes e vermelhos na figura 32, que não estão sobre o eixo das frequências cíclicas, representam as potências das componentes cicloestacionárias dos SVIs da cavitação em bolhas itinerantes e em nuvem, respectivamente. As potências das componentes cicloestacionárias correspondem às variações das potências instantâneas do SVIs, acima e abaixo dos valores das potências médias, algo *análogo* ao conceito de *potência reativa* em um circuito elétrico (em um circuito elétrico, a potência real flui somente da fonte para a carga, e a potência reativa flui alternadamente em ambos os sentidos, mas sem alterar a potência média total transferida da fonte para a carga). Na realidade as componentes cicloestacionárias indicam se há ou não cavitação e identificam o seu tipo, mas a soma das potências destas componentes *não* é um estimador da agressividade da cavitação. Não obstante, as agressividades da cavitação P_B e P_C podem ser calculadas a partir das potências de cada uma das componentes cicloestacionárias de cada tipo de cavitação, conforme será exposto na secção 3.5.3,

para os casos em que a cavitação é incipiente ou limitada, e portanto intermitente. É preciso pois, identificar primeiro quais componentes de ordem de máquina correspondem aos sinais pertencentes a cada tipo de cavitação, para depois calcular as potências de cada uma dessas componentes, e a partir delas estimar P_B e P_C.

3.5.1 Identificação das componentes pertencentes a cada tipo de cavitação

O principal problema de uma análise automatizada é distinguir na imagem formada por $|CMS[a,h]|$ quais são as componentes de ordem que máquina que pertencem a qual tipo de modulação. A distinção pode ser feita com base na *similaridade espectral* entre diferentes componentes de ordem de máquina. Esta similaridade pode ser avaliada através do coeficiente de correlação de Pearson (PPMC).

No caso da cavitação em bolhas itinerantes, a distribuição espectral da componente de ordem de máquina de número *b* é tomada como referência para comparação de similaridade espectral. As componentes de ordem de máquina de número *mb*, onde *m* é um número inteiro e maior que um, somente são consideradas como pertencentes ao SVI da cavitação em bolhas itinerantes se o coeficiente de correlação entre a distribuição espectral da componente de ordem de máquina *mb* e a componente de referência for maior que um valor pré-definido. Da mesma forma, a componente de ordem de máquina de número *v* é tomada como referência para a identificação das componentes cicloestacionárias pertencentes ao SVI da cavitação em nuvem.

3.5.2 Obtenção da potência de cada componente cicloestacionária

A componente de ordem de máquina $i \neq 0$ (cicloestacionária) tem, de forma similar à componente estacionária (i=0), um espectro de potências "reativas" dado por:

$$PS_i[h] = CMS[i,h] \tag{22}$$

Diferentemente da componente estacionária, $CMS[i,h]$ é um número complexo,

representando o módulo e a fase da potência da componente cicloestacionária de ordem de máquina i e frequência espectral h. A componente estacionária tem espectro de potências $PS[h] = CMS[0,h]$, que é sempre um número real.

Devido ao processo de modulação, e decorrente similaridade espectral, $PS_i[h]$ é igual a uma constante complexa A_i multiplicada por $PS[h]$. Esta constante complexa determina a amplitude e a fase das oscilações da componente cicloestacionária:

$$PS_i[h] = A_i PS[h] \tag{23}$$

Lembrando que os sinais modulantes são periódicos e não-negativos para todos os valores de θ, o espectro de potência instantâneo do SVI da cavitação pode ser expresso por uma série de Fourier (supondo que a modulação afeta *linearmente* a potência instantânea do sinal):

$$IPS[\theta, h] = PS[h] + \sum_{i=1}^{\infty} \left(\Re(A_i) PS[h] \cos(i\theta) + \Im(A_i) PS[h] \sin(i\theta) \right) \tag{24}$$

As funções $\Re(.)$ e $\Im(.)$ representam respectivamente a parte real e a parte imaginária dos coeficientes complexos. O primeiro termo é decorrente da parcela constante do sinal modulante (componente estacionária), e o somatório produz as componentes cicloestacionárias. Se na equação (24) for calculado o somatório no domínio das frequências espectrais h, resulta em:

$$P(\theta) = \sum_{h} PS[h] = \overline{P} + \sum_{i=1}^{\infty} \left(\Re(A_i) \overline{P} \cos(i\theta) + \Im(A_i) \overline{P} \sin(i\theta) \right) \tag{25}$$

onde $P(\theta)$ é a potência instantânea (angular) do SVI da cavitação, $\overline{P}$ é a potência média do SVI da cavitação (ou seja, a própria agressividade da cavitação), e os coeficientes dos senos e cossenos $\Re(A_i)\overline{P}$ e $\Im(A_i)\overline{P}$ compõem a potência instantânea da componente cicloestacionária de ordem de máquina i. Desta forma, a potência "reativa" de cada componente de ordem de máquina i é $\overline{P} \cdot A_i$.

A cavitação em uma turbina em regime de operação normal está sempre em nível incipiente ou limitado, e portanto intermitente, não existindo em todos os instantes

de tempo considerados. Neste caso, o índice de modulação é 100%.

O motivo de o índice de modulação ser 100% quando a cavitação é intermitente não é evidente, mas se for considerada a forma como o cálculo da matriz AIPS "reconstrói" a potência média do sinal (levando em conta inúmeras revoluções), é possível demonstrar que o resultado é de fato um sinal representado como se a potência média variasse de forma periódica, de 0 a 100% do valor máximo.

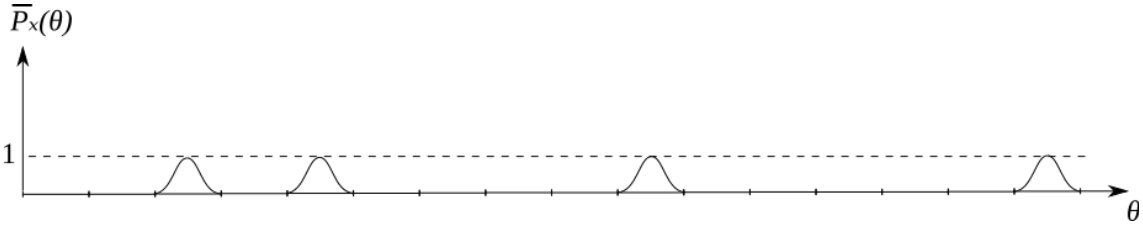


Figura 33 - Distribuição irregular da potência do sinal ao longo da posição angular, devido à intermitência.

Fonte: Elaboração própria com *freeware* InkScape v_0.91 Linux.

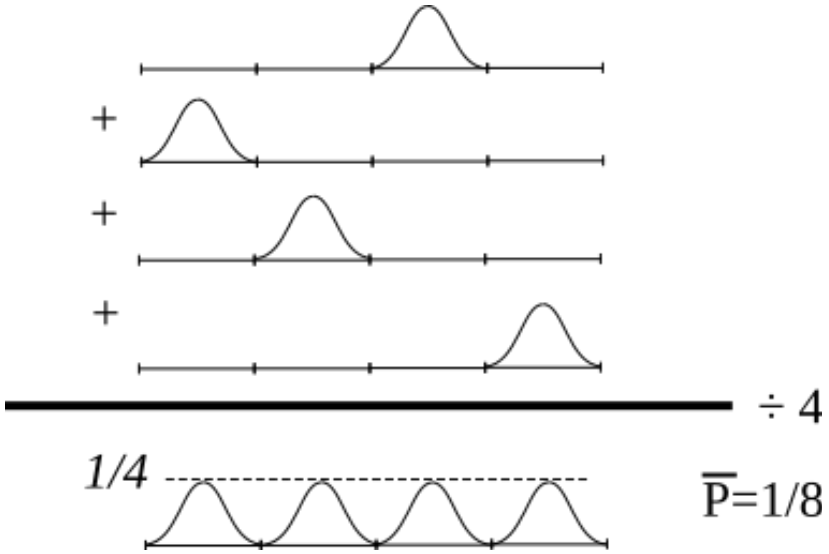


Figura 34 - Processo de reconstrução da informação (potência média instantânea), no sinal exemplo da figura 33.

Fonte: Elaboração própria com *freeware* InkScape v_0.91 Linux.

Como exemplo ilustrativo deste processo, considere a figura 33. A potência média $\bar{P}_x(\theta)$ do SVI da cavitação, já filtrado por um dos canais do *comb-filter* (transformada STFT) com largura de banda Δf, está representada como pulsos na forma 0,5-0,5$cos(4\theta)$, mas de ocorrência intermitente. Cada pulso tem potência máxima unitária e potência média 0,5. Na realidade, os pulsos da cavitação não têm envelope

cossenoidal, mas por ora e por questões de simplicidade, pode-se considerar como cossenoidal, sem perda de generalidade para frequências múltiplas. A única exigência é que a potência média instantânea tenda a zero nos extremos dos pulsos, o que pode ser conseguido com uma série de cossenos. Ainda neste exemplo, considera-se que uma revolução completa da máquina consiste em 4 divisões do eixo θ (como se por exemplo, o número de lâminas do rotor fosse igual a 4). Estão representadas no gráfico quatro revoluções do eixo.

Ao computar a média das potências instantâneas das 4 revoluções, a informação, que está de fato na potência média instantânea que varia periodicamente de 0% a 100%, é reconstruída (figura 34). Nota-se, apesar de as implosões serem de ocorrência aleatória, que há posições angulares $\theta=\{0, \pi/2, \pi, 3\pi/2,...\}$ em que a probabilidade de ocorrência é anulada, e o fenômeno da cavitação erosiva não acontece, anulando a potência do sinal. O formato dos pulsos que compõem o sinal reconstruído é resultado direto da influência que a RSI da máquina e outras flutuações periódicas de pressão exercem sobre a probabilidade de produção de *nuclei* instáveis.

Obviamente, os pulsos individuais induzidos pelas implosões de uma única nuvem ou conjunto de bolhas de vapor não têm o formato cossenoidal mostrado nas figuras anteriores, tendo também formato aleatório e diferente a cada revolução. Mas novamente, ao computar a média de muitas revoluções, a potência média instantânea será maior nas posições angulares de maior probabilidade, tendendo a zero em determinados pontos (NENNEMANN, 2007).

3.5.3 Cálculo das estimativas das agressividades da cavitação

Tendo todas as potências das componentes cicloestacionárias para um determinado SVI da cavitação, e levando em conta que $\Re(A_i)\bar{P}=\Re(A_i\bar{P})$ e $\Im(A_i)\bar{P}=\Im(A_i\bar{P})$, o somatório contido na equação (25) pode ser calculado, e a parte oscilatória de $P(\theta)$ determinada.

Lembrando que $P(\theta)$ deve ser um sinal modulado em amplitude com índice de modulação igual a 100%, o que é verdadeiro para a cavitação incipiente, $P(\theta)$ deve ser

sempre não-negativo para todos os valores de θ, e ser igual a zero em pelo menos dois valores de θ, dentro do intervalo $0 \leq \theta \leq 2\pi$. Esta condição é obtida fazendo $\bar{P}$ igual ao mínimo global da parte oscilatória da equação (25) multiplicado por -1, conforme a equação (26). O valor de $\bar{P}$ é a própria agressividade da cavitação. A função *min*(.) retorna o mínimo global da função no intervalo de $0 \leq \theta \leq 2\pi$ (que é o valor mais negativo que a potência "reativa" assume).

$$\overline{P} = -min\left(\sum_{i=1}^{\infty}\left(\Re\left(\sum_{h} CMS[i,h]\right)\cos(i\theta) + \Im\left(\sum_{h} CMS[i,h]\right)\sin(i\theta)\right)\right) \tag{26}$$

Aplicando este resultado à cavitação em bolhas itinerantes, a agressividade P_B é calculada segundo a equação (27). É importante relembrar que a componente de ordem de máquina de número *b* é tomada como referência para comparação da similaridade espectral, e as componentes de ordem de máquina *bi* só são computadas se forem previamente identificadas como espectralmente similares à componente de referência, conforme descrito na secção 3.5.1.

$$P_B = -min\left(\sum_{i=1}^{\infty}\left(\Re\left(\sum_{h} CMS[bi,h]\right)\cos(bi\theta) + \Im\left(\sum_{h} CMS[bi,h]\right)\sin(bi\theta)\right)\right) \tag{27}$$

Similarmente a P_B, a agressividade da cavitação em nuvem (P_C) é calculada pela equação (28), mas tomando a componente de ordem de máquina de número *v* como referência:

$$P_C = -min\left(\sum_{i=1}^{\infty}\left(\Re\left(\sum_{h} CMS[vi,h]\right)\cos(vi\theta) + \Im\left(\sum_{h} CMS[vi,h]\right)\sin(vi\theta)\right)\right) \tag{28}$$

Caso a cavitação esteja em nível desenvolvido, a potência média instantânea dos SVIs produzidos não se anula em nenhuma posição angular, e o fenômeno não é mais intermitente (como a cavitação que ocorre em hélices de barcos). Nestes casos, a melhor estimativa possível da agressividade da cavitação é ainda considerar que a potência da componente estacionária é igual à soma das potências das componentes cicloestacionárias, pois não há como estimar a verdadeira potência total. A agressividade estimada será sempre um valor inferior ao valor verdadeiro, entretanto.

A cavitação, quando ocorre em uma turbina em nível desenvolvido, produz

queda no rendimento, sendo de fácil detecção. A cavitação intermitente, entretanto, produz erosão de forma insidiosa, e é eventualmente assintomática, dando sentido a esta pesquisa.

Estes estimadores das potências *absolutas* dos sinais podem sofrer influência de vários fatores, como por exemplo, mau casamento de impedância mecânica na instalação dos acelerômetros, má escolha do local da instalação dos sensores, diferenças entre turbinas e/ou instalações, etc. Neste caso, é necessário normalizar P_B e P_C pela potência da parcela estacionária do sinal, que é por definição a potência média que o sinal transporta:

$$P_{B\%} = \frac{P_B}{\sum_h |CMS[0,h]|} \tag{29}$$

$$P_{C\%} = \frac{P_C}{\sum_h |CMS[0,h]|} \tag{30}$$

Os estimadores $P_{B\%}$ e $P_{C\%}$ portanto, fornecem as potências *relativas* dos sinais modulados produzidos por cada tipo de cavitação, significando qual a porcentagem da potência total é produzida por somente aquele tipo de cavitação. Tanto os estimadores das potências absolutas, quanto os estimadores relativos, juntamente com a imagem formada por $|CMC[a,h]|$ servem para fornecer informações sobre o tipo, localização e agressividade das cavitações presentes na turbina. Além disso, os estimadores e as imagens formadas por $|CMC[a,h]|$ e $|CMS[a,h]|$ podem ser arquivados, e ao longo do tempo constituir um histórico de forma totalmente automatizada. Tal histórico poderá ser usado em futuras pesquisas, que não fazem parte do escopo deste trabalho, mas que serão citadas no capítulo 6 (conclusões).

4 IMPLEMENTAÇÃO DA METODOLOGIA

O escopo original deste projeto de pesquisa era o desenvolvimento de um *circuito integrado de aplicação específica* (ASIC), para detecção e identificação (em tempo real) dos tipos de cavitação existentes em uma turbina, bem como extrair informações sobre a agressividade da cavitação.

Inicialmente, o levantamento bibliográfico feito pelo autor não foi profundo o suficiente para possibilitar a modelagem de uma turbina hidráulica, que sofre efeitos da cavitação erosiva, como um processo cicloestacionário, haja vista que o conceito de processo cicloestacionário era algo desconhecido pelo autor até então. Ademais, todas as descrições sobre a cavitação erosiva em turbinas disponíveis na literatura especializada são descrições fenomenológicas, focadas no combate dos seus efeitos nocivos, e na identificação de condições *globais* que *favorecem* o seu surgimento.

Normalmente, o fenômeno cavitação erosiva é descrito como algo que produz inicialmente bolhas, depois um "ruído" como se houvesse entrado cascalho na turbina, seguido pela erosão e finalmente queda no rendimento da máquina. O processamento mais comum atualmente, que é o processamento DEMON, também é empírico e baseado na extração de informações contidas no envelope do "ruído". Este processamento é elegível para uma implementação em ASIC. Atualmente, a detecção do sinal envelope que modula uma portadora aleatória pode ser feita em circuito integrado, por meio de retificadores ou transformadores de Hilbert.

Entretanto, além de ser uma técnica empírica, a técnica DEMON necessita da intervenção humana e não fornece resultados conclusivos sobre a agressividade da cavitação, servindo somente para detecção e eventualmente identificação do tipo presente.

Em um segundo momento desta pesquisa, chegou ao conhecimento do autor o conceito de cicloestacionariedade, fato que o motivou a buscar descrições mais detalhadas sobre a cavitação (descrições não só fenomenológicas, mas físicas, detalhando os mecanismos de formação das cavidades e os processos de erosão). Esta busca por sua vez teve como finalidade possibilitar a modelagem de uma turbina hidráulica com cavitação erosiva como um processo cicloestacionário, mesmo que esta modelagem fosse limitada e simplificada. A modelagem, que foi mostrada no capítulo 2,

infelizmente se tornou complexa em demasiado para implementar em ASIC a metodologia desenvolvida no capítulo 3, necessária para estimação da agressividade da cavitação. A detecção, identificação e estimação da agressividade da cavitação em tempo real (propriamente dito) também é tecnicamente inviável, pois envolve reamostragem angular com interpolação e numerosas FFTs. Não obstante, exceto pelo processamento em "tempo real propriamente dito", o escopo deste projeto de pesquisa foi alcançado através do desenvolvimento de um *software* aplicativo específico que implementa a metodologia desenvolvida no capítulo 3, e de um *hardware* dedicado à execução do *software* no próprio local da turbina em análise, de forma automática (processamento *online*).

A execução do projeto então seguiu em múltiplas frentes de trabalho: a implementação prática da metodologia de processamento dos sinais em *software*, a aquisição de sinais reais de turbinas, a elaboração de um sintetizador de sinais a fim de verificar a performance do *software* (principalmente diante da dificuldade em obter sinais reais que fossem comprovadamente produzidos por determinado tipo de cavitação e com determinado nível de agressividade) e a confecção de um *hardware* dedicado (algo análogo a uma pequena placa de circuito integrado com um ASIC, que seria de baixo consumo e baixo custo).

4.1 PRODUÇÃO POR *SOFTWARE* DE SINAIS SINTÉTICOS DE TESTE

A fim de testar as capacidades do *software* aplicativo identificador e estimador da agressividade da cavitação, e em vista da falta de sinais reais cuja cavitação originadora fosse bem qualificada em termos de tipo e agressividade, um simulador de sinais baseado na modelagem exposta no capítulo 2 foi criado. Na realidade, o simulador faz parte de uma *suite* de *softwares,* que realizam o pré-processamento, o processamento e a extração de informações dos resultados, e que serão descritos na secção 4.3.

O objetivo do simulador é produzir sinais que sejam os equivalentes sintéticos aos obtidos com os sensores acelerômetros reais combinados a condicionadores de sinal e conversores analógico/digital (A/D). A simulação deve portanto, produzir sinais

amostrados no domínio do tempo, e para isso o efeito *droop speed control* deve também ser simulado. Adicionalmente, os sinais de posição angular instantânea produzidos pelo gerador elétrico e pelo tacômetro ótico devem ser produzidos. Além dos SVI decorrentes de cada tipo de cavitação, é importante também que o simulador possa produzir ruídos aditivos, que permitem verificar quão sensível é a metodologia desenvolvida a sinais ruidosos (incluindo ruído branco gaussiano e ruído tonal). Finalmente, é fundamental que o simulador produza sinais de modo que a potência de cada SVI e de cada tipo diferente de ruído seja conhecida. Também é importante que grau de modulação presente em cada SVI seja conhecido, pois aí estão as informações principais sobre a agressividade da cavitação.

4.1.1 A plataforma em que o sintetizador foi desenvolvido

A escolha da plataforma de desenvolvimento do sintetizador foi simplesmente a mesma utilizada na elaboração dos outros softwares da *suite*, pois é conveniente ter no mesmo equipamento que realiza as análises, um meio de produzir sinais de testes. Ademais, o *hardware* que consegue executar os softwares principais de análise da *suite*, certamente consegue executar o sintetizador, por ser computacionalmente bem menos intensivo.

Todos os softwares da *suite* foram desenvolvidos como aplicativos de linha de comando sem interface gráfica, pois o *hardware* dedicado não deve possuir monitor. Todas as entradas de dados são feitas por arquivos em disco e linhas de comando, e as saídas de dados são também em arquivos de dados e de texto contendo relatórios. Finalmente, seguindo as mesmas diretrizes aplicadas no desenvolvimento dos outros softwares da *suite*, o sintetizador foi desenvolvido em linguagem C++ a fim de ser portável a outras plataformas, e de empregar bibliotecas de código fonte aberto, também portáveis a inúmeras plataformas nos dias atuais. As listagens dos códigos fonte de todos os softwares desenvolvidos e arquivos importantes podem ser encontradas no Apêndice E – Código fonte dos softwares desenvolvidos.

4.1.2 Configuração do sintetizador

A configuração de todos os parâmetros do sintetizador de sinais é feita a partir de um arquivo texto .ini. Nele estão informações sobre: O conjunto TG (número de lâminas do rotor e palhetas do distribuidor, número de polos magnéticos e frequência nominal gerada e o *droop speed control*), os sinais a serem sintetizados (número de sensores, a taxa de amostragem e o número de revoluções do eixo a serem simuladas), os tipos de cavitação presentes (características da excitação acústica e da transmissão mecânica para os tipos de cavitação em bolhas e em nuvem), as flutuações de pressão na turbina (a RSI, ondas estacionárias na caixa voluta e derramamento de vórtices de Von Kárman com *lock-in*) e as características dos ruídos aditivos (potência, frequência ou cor e densidade espectral de potência).

4.1.3 Simulação do *droop speed control* e das revoluções do eixo

Sabendo o número de polos do gerador e a frequência nominal da tensão gerada, é possível determinar o período de revolução nominal do eixo da turbina. De acordo com o número de revoluções a serem simuladas e a taxa de amostragem desejada, a duração total dos sinais sintetizados pode ser determinada e com isso, a memória necessária já alocada. Um conjunto de sinais sintéticos é composto por dois sinais vibracionais, um sinal do tacômetro ótico simulado, e um sinal para a tensão alternada gerada, totalizando 4 canais.

O *droop speed control* é simulado através da produção de um sinal aleatório limitado em banda, e que é a flutuação da velocidade angular instantânea do eixo $\Delta\omega(t)$. Na prática, tal sinal é conseguido com um gerador de ruído branco gaussiano (função *gsl_ran_gaussian* da biblioteca G*nu Scientific Library***)** e um filtro de resposta ao impulso infinita (IIR) que limita a banda do sinal em 1 Hz. O sinal produzido também sofre normalização em amplitude, para garantir que o desvio de frequência máximo seja atingido. A posição angular instantânea do rotor pode então ser determinada por:

$$\theta(t) = \omega t + \Delta\omega(t) \tag{31}$$

Efetivamente, $\theta(t)$ é um acumulador que vai recebendo incrementos de $(\omega+\Delta\omega(t))\cdot T_s$, onde ω é a velocidade angular de revolução do eixo nominal e t é o tempo já discretizado ($t=nT_s$), T_s é o inverso da taxa de amostragem e n é o número da amostra.

Paralelamente à geração do sinal sintético $\theta(t)$, as portadoras vibracionais aleatórias $c_B(t)$ e $c_C(t)$ são produzidas com um gerador de ruído branco gaussiano e filtros passa faixa IIR. No caso, $n_B(t)$ e $n_C(t)$ são sinais aleatórios produzidos pela mesma função *gsl_ran_gaussian*, e os filtros IIR correspondentes às respostas ao impulso $h_B(t)$ e $h_C(t)$ são configurados pelo arquivo .ini. Após as filtragens, os sinais vibracionais estacionários (e temporais) estão prontos para serem modulados.

4.1.4 Produção das componentes modulantes

Tendo já preparado o sinal $\theta(t)$, é possível já produzir os sinais modulantes $m_B(t)$ e $m_C(t)$, lembrando que estes sinais são séries de Fourier em função da posição angular. Os coeficientes das séries são lidos do arquivo de configuração e os sinais são computados:

$$m_B(t)=1+\sum_{n=1}^{N} B_n\cos(n\,\theta(t))+A_n\sin(n\,\theta(t)) \tag{32}$$

$$m_C(t)=1+\sum_{n=1}^{N} C_n\cos(n\,\theta(t))+D_n\sin(n\,\theta(t)) \tag{33}$$

Nas equações (32) e (33), A_N, B_N, C_N e D_N são os coeficientes dos senos e cossenos lidos do arquivo de configuração, para a componente de ordem de máquina de número n. Os sinais $m_B(t)$ e $m_C(t)$ são sempre não negativos devido à adição do 1 (os valores dos coeficientes devem ser cuidadosamente escolhidos para tal), e a parcela oscilatória é, do ponto de vista de um sinal no domínio do tempo, aleatoriamente modulada em frequência.

4.1.5 Modulação em amplitude

A modulação em amplitude, como convencionalmente encontrada na literatura especializada, é realizada pela multiplicação direta das componentes modulante e portadora. No caso deste sintetizador de sinais, é conveniente que a componente modulante provoque variações não na amplitude, mas na potência, que é a informação sobre a agressividade da cavitação que o sinal transporta. Deste modo, a modulação é feita por uma função não linear ligeiramente modificada em relação ao usual, conforme citado nas secções 2.4.1 e 2.4.2, equações (9) e (12):

$$f_{B,C}(m_{B,C}(\cdot), c_{B,C}(\cdot)) = \sqrt{m_{B,C}(\cdot)} \cdot c_{B,C}(\cdot) \tag{34}$$

A mesma função é usada para a cavitação em bolhas e em nuvem, e a raiz quadrada é introduzida para o sinal modulante influenciar linearmente na potência instantânea do SVI. Se outra função fosse usada, o resultado seria a produção de outras componentes de ordens de máquina múltiplas (harmônicas) que não constam no arquivo de configuração.

Os SVIs sintéticos para a cavitação em bolhas e para a cavitação em nuvem são então respectivamente:

$$\hat{x}_{SVI-B}(t) = \sqrt{m_B(\theta(t))} \cdot c_B(t) \tag{35}$$

$$\hat{x}_{SVI-C}(t) = \sqrt{m_C(\theta(t))} \cdot c_C(t) \tag{36}$$

Ambos sinais são produzidos diretamente no domínio do tempo, e incluem os efeitos do *droop speed control*. O acento circunflexo denota que os sinais são sintéticos. Finalmente, uma componente de ruído é criada, incluindo dois ruídos tonais e um ruído branco gaussiano, que aqui é representado por *WGN*(*t*):

$$\hat{\eta}(t) = \frac{2P_W}{F_s} WGN(t) + \sqrt{2P_1}\cos(2\pi f_1 t) + \sqrt{2P_2}\cos(2\pi f_2 t) \tag{37}$$

O ruído *WGN*(*t*) tem variância igual a 1, e P_W é a potência média desejada do ruído, que se espalha na faixa de frequências que vai de zero à metade da frequência de

amostragem do sinal, F_s. As variáveis P_1 e P_2 e f_1 e f_2 são respectivamente as potências e as frequências dos ruídos tonais. O fator 2 que multiplica P_1 e P_2 foi estrategicamente adicionado, para que as potências médias dos ruídos tonais sejam P_1 e P_2. Os dois SVIs sintéticos e a componente de ruído são combinados aditivamente, para formar um sinal vibracional sintético:

$$\hat{x}(t) = \hat{x}_{SVI-B}(t) + \hat{x}_{SVI-C}(t) + \hat{\eta}(t) \tag{38}$$

4.1.6 Síntese dos sinais de sincronismo

O sinal senoidal produzido pelo gerador elétrico pode ser sintetizado a partir de $\theta(t)$, e sabendo que o número de polos magnéticos é P, ele é igual a $cos(P \cdot \theta(t)/2)$. Já para a simulação de um tacômetro ótico, o sinal $\theta(t)$ é processado por uma função que imita o funcionamento de um *Schmitt trigger* seguido de um monoestável, resultando em pulsos retangulares que coincidem com as transições positivas de $\theta(t)$.

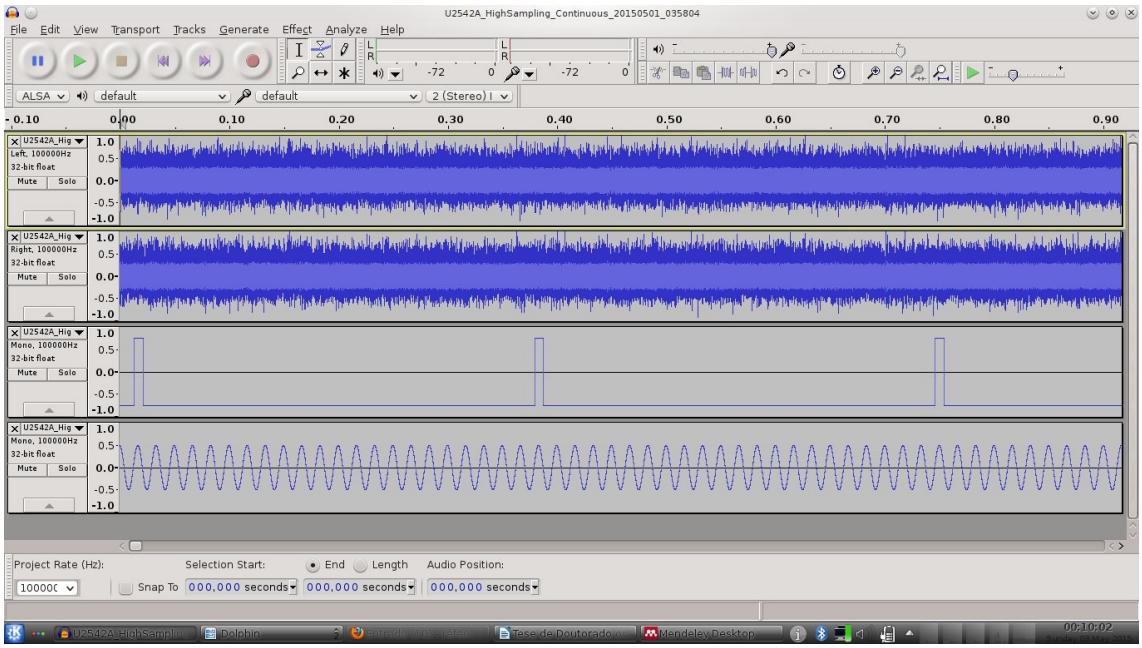


Figura 35 - Exemplo de conjunto de sinais produzido pelo sintetizador.

Fonte: Elaboração própria com *freeware* Audacity 2.0.5 Linux.

4.1.7 Reunião do conjunto de sinais em arquivo

Dois sinais vibracionais, o sinal simulado do tacômetro e o sinal simulado do gerador de tensão alternada são gravados juntos em um único arquivo binário, com resolução de 32 *bits* por amostra e formato .wav multicanal (figura 35). No cabeçalho no arquivo .wav são gravadas informações como nomes dos sinais, data da síntese, ganhos dos canais, etc.

Juntamente com o arquivo .wav, um outro arquivo texto com o mesmo nome do arquivo .wav, mas com a extensão .ini é produzido. Este arquivo texto contém informações sobre os sinais produzidos, e funciona durante as etapas de pré-processamento e processamento como um "prontuário" do arquivo de dados. Todas as configurações de processamento dos sinais, bem como todos os processamentos que o arquivo .wav sofrer, serão registrados neste arquivo .ini.

4.2 AQUISIÇÃO DE SINAIS DE TURBINAS REAIS DO SIN

Três conjuntos de sinais reais foram gravados de dois conjuntos TG (funcionando com a mesma água, idênticos e conectados ao SIN), nomeadamente UG01 e UG02. A única diferença entre UG01 e UG02, e que foi usada como base de comparação dos resultados, foi o ponto de operação em que estavam no momento da gravação: o conjunto TG UG01 estava configurado para geração fixa de 48 MW (67% de sua capacidade nominal) e UG02 estava somente girando em vazio, produzindo 0 MW, mas com o enrolamento de campo energizado.

Ambos UG01 e UG02 têm capacidade nominal de P_{nom}=72 MW, e trabalham com vazões nominais de Q_{nom}=123 m³/s, e nível nominal de reservatório de entrada H_{nom}=67 m. As turbinas possuem v=24 palhetas diretrizes em seus distribuidores e b=11 lâminas nos rotores, e a frequência nominal de revolução dos eixos é α_f=2,72 Hz (163,6 r.p.m.). A frequência nominal da tensão gerada é f_{nom}=60 Hz, e portanto, pode-se calcular o número de polos magnéticos P dos geradores pela fórmula:

$$P = 2\frac{f_{nom}}{\alpha_f} = 2\frac{60}{2{,}72} \simeq 44 \tag{39}$$

Obviamente o número de polos deve ser inteiro, e a frequência nominal de revolução dos eixos é na verdade 120/44 ou 30/11 Hz. Ambas turbinas de UG01 e UG02 tinham seus eixos girando sincronizados, pois ambos UG01 e UG02 estavam ligados ao SIN. Adicionalmente, ambas as turbinas já estavam operando há mais de 50.000 horas sem manutenção corretiva, e apresentavam aproximadamente a mesma perda de massa. Desde o comissionamento em 1969, as turbinas têm históricos de níveis de agressividade de cavitação notavelmente baixos.

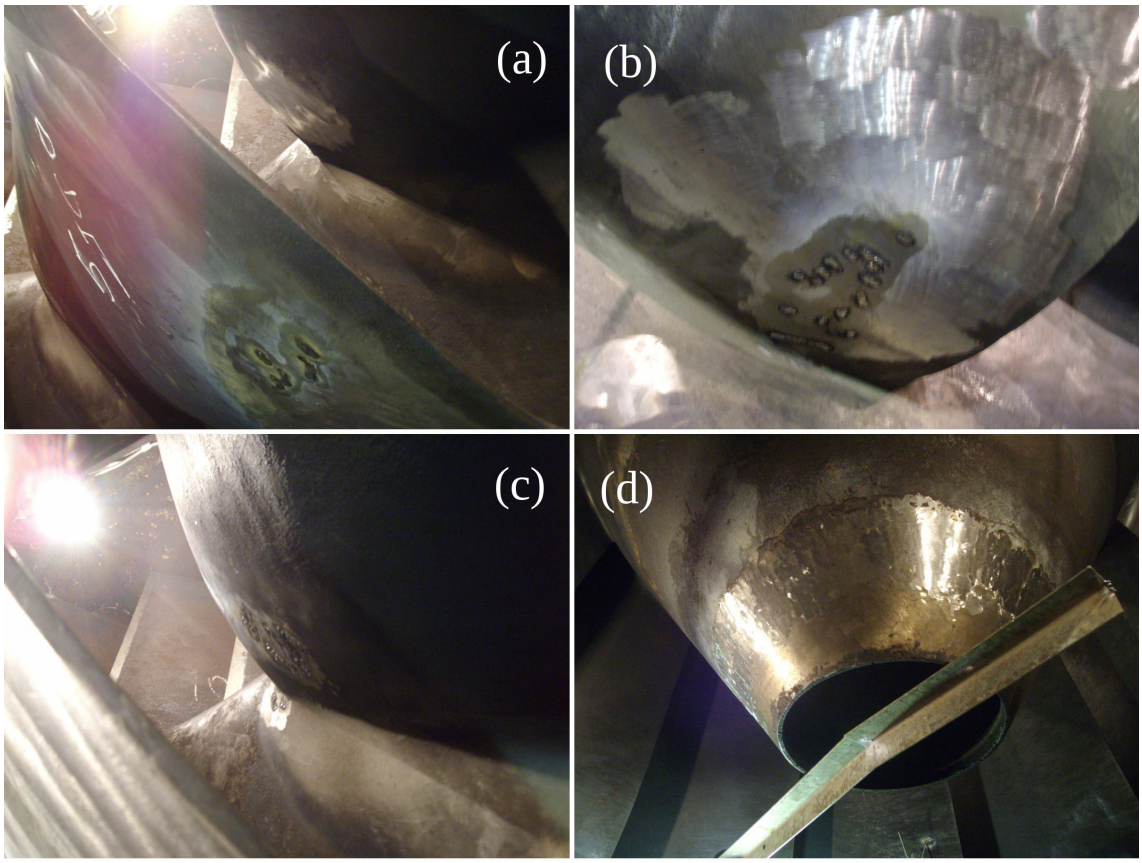


Figura 36 - Visualização da erosão durante a parada de manutenção corretiva. Em (a): visão das áreas atingidas pela erosão, visto do tubo de descarga. Em (b): erosão no bordo de fuga, característica da cavitação em bolhas. Em (c): erosão no bordo de ataque, característica da cavitação em nuvem. Em (d): erosão no cubo do rotor, característica da cavitação em vórtice no tubo de descarga.

Fonte: produção própria.

Cada uma das turbinas consumiu apenas 40 kg de eletrodos de solda durante a parada de manutenção, e alguns padrões característicos de erosão puderam ser

visualmente identificados naquela ocasião, sendo eles: a erosão causada pela cavitação em bolhas itinerantes, em nuvem e em vórtice no tubo de descarga (figura 36).

Os sinais vibracionais foram captados por dois acelerômetros *piezoelétricos* de eletrônica integrada (*Integrated Electronic Piezoelectric*, ou IEPE) apropriados para a captação de vibrações de alta frequência. Os acelerômetros, de modelo 9700A, foram fabricados pela *Rockwell Automation* e, quando montados, têm frequência de ressonância típica de 85 kHz.

Para a resolução da posição angular, trechos de fita adesiva de aproximadamente 40 cm foram colados nos eixos de UG01 e UG02, sendo 20 cm de fita branca e 20 cm de fita preta. Um tacômetro ótico feito com um LED infravermelho e um fotodiodo compatível foi colocado próximo aos eixos, sem tocá-los, e produziu um pulso elétrico a cada revolução (figura 37).

Figura 37 - Tacômetro ótico posicionado junto ao eixo da turbina (centro). Ao fundo é possível ver um acelerômetro montado na tampa da turbina. À esquerda, o condicionador e o banco de baterias.

Fonte**:** produção própria.

Como medida complementar, as tensões elétricas alternadas produzidas pelos geradores de UG01 e UG02 foram reduzidas por meios de transformadores, ao valor de 2 V pico a pico.

Os sinais provenientes dos acelerômetros foram tratados por um condicionador analógico de sinais, que realizou a filtragem *anti-aliasing* e aplicou ganho de voltagem 27. Tanto os sensores acelerômetros quanto o tacômetro ótico foram alimentados pelo condicionador de sinais, que por sua vez empregou um banco de baterias de PbO_2//Pb. O emprego da alimentação por baterias, bem como a montagem do condicionador realizada em caixa metálica e o uso de cabos blindados, foram medidas tomadas para

minimizar interferências eletromagnéticas, tanto induzidas nos cabos e placas de circuito, como provenientes da rede elétrica.

Os quatro sinais (2 vibracionais, 1 do tacômetro e 1 do gerador) compondo um conjunto de sinais reais foram amostrados a 100.000 amostras por segundo por canal, e gravados em um único arquivo da dados com a extensão .bin, por um gravador de dados digitais modelo U2525A, fabricado pela *Agilent Technologies*, e um computador portátil comum. A conversão A/D foi feita em 16 *bits* por amostra. Juntamente com cada arquivo com extensão .bin, um arquivo de texto é produzido com as informações de configuração da gravação. No total foram gravados três conjuntos de sinais reais, sendo dois provenientes da turbina de UG01 e um da turbina de UG02.

4.2.1 Procedimentos para a aquisição não invasiva

Uma das exigências delimitadas pelo escopo deste projeto de pesquisa é que o método seja não invasivo, e não perturbe a operação da máquina a ser testada. Ademais, tanto UG01 e UG02 estavam conectadas ao SIN no momento da gravação, não cabendo ao autor alterar as condições de operação. As fotos da figura 36 foram tiradas de forma oportuna, do interior das turbinas durante uma parada já programada para manutenção, não caracterizando procedimento invasivo.

Figura 38 - Instalação dos sensores para a aquisição do conjunto de sinais número 1. À esquerda: o acelerômetro e sua base de montagem sobre o munhão. Á direita: sobre a tampa da turbina.

Fonte: produção própria.

Os acelerômetros IEPE modelo 9700A têm um parafuso para serem fixados ao ponto desejado de captação das vibrações, e a medição não invasiva não permite

modificações de qualquer espécie nas peças da turbina. A solução encontrada, e que produz resultados sub-ótimos, foi produzir bases de montagem em aço cuidadosamente usinadas e com a rosca adequada ao parafuso dos acelerômetros, para serem coladas nos pontos de medição. A cola usada foi à base de cianoacrilato, após cuidadosa limpeza das superfícies a serem coladas.

No primeiro conjunto de sinais adquirido de UG01, o primeiro acelerômetro (acelerômetro 1 conectado ao canal 0) foi instalado sobre o munhão de número 5, e o acelerômetro 2 (conectado ao canal 1) foi instalado na tampa da turbina, bem próximo ao seu eixo. Na literatura, há autores que indicam que um dos locais preferenciais de captação das vibrações induzidas pela cavitação é o mancal principal da turbina, próximo ao rotor. Entretanto, no caso de UG01 e UG02 este mancal estava inacessível, e o ponto mais próximo onde poderia ser instalado um acelerômetro era a referida tampa (figura 38).

Durante a aquisição do segundo conjunto de sinais de UG01, o acelerômetro 2 foi movido para o munhão de número 4. O acelerômetro 1 permaneceu instalado no munhão 5 (figura 39).

Figura 39 - Instalação dos acelerômetros para a gravação do conjunto de sinais número 2.

Fonte: produção própria.

Finalmente, para o conjunto de sinais adquirido da turbina de UG02 (conjunto 3), ambos acelerômetros foram instalados sobre munhões, sendo que o acelerômetro 1

foi instalado sobre o munhão de número 4, e o acelerômetro 2 foi instalado sobre o munhão de número 5. Para os três conjuntos de sinais, foram gravadas milhares de revoluções do eixo das máquinas, em aproximadamente 15 minutos em cada conjunto de sinais. A tabela 2 sumariza as principais diferenças entre as condições de gravação dos três conjuntos de sinais reais.

Tabela 2 - Diferenças principais entre os conjuntos de sinais reais adquiridos.

Conjunto de sinais	Conjunto TG (Potência Gerada)	Local do Acelerômetro 1	Local do Acelerômetro 2	Duração da Gravação
1	UG01 (48 MW)	Munhão 5	Tampa da turbina	920 s
2	UG01 (48 MW)	Munhão 5	Munhão 4	875 s
3	UG02[a] (0 MW)	Munhão 4	Munhão 5	973 s

[a]O conjunto TG UG02 estava girando em vazio, porém sincronizado com a rede elétrica.

4.3 IMPLEMENTAÇÃO EM *SOFTWARE* DA METODOLOGIA PROPOSTA

As etapas de pré-processamento, processamento e extração de informações das matrizes CMS foram implementadas em softwares diferentes, visando uma maior flexibilidade de uso dos *softwares* que compõem a *suite,* bem como tornar possível a execução com alocações de menores quantidades de memória RAM.

4.3.1 O conversor de arquivos binários *bin2wav*

Um *software* auxiliar foi desenvolvido com a finalidade de converter os arquivos de sinais reais com a extensão .bin, para o formato .wav multicanal, que é o formato que o restante da *suite* consegue processar. O formato .wav multicanal também permite que os arquivos sejam visualizados ou editados com editores *freeware*.

O programa *bin2wav* lê os dados binários do arquivo .bin gerado pelo gravador U2525A, e também lê as informações no arquivo texto gerado pelo gravador, e produz um arquivo .wav multicanal com exatamente os mesmos dados gravados no arquivo

original. As informações obtidas do arquivo texto gerado pelo gravador U2525A são inseridas no cabeçalho do arquivo .wav, de modo que todas as informações sobre as condições em que a gravação foi realizada, bem como os próprios dados gravados, ficam armazenados em um único arquivo.

Adicionalmente, o programa bin2wav produz um arquivo .ini, o arquivo "prontuário", já com algumas opções *default* de processamento preenchidas. Se for necessária alguma alteração nas etapas de pré-processamento e processamento, o arquivo pode ser editado em um editor de textos comum.

4.3.2 O *software* de reamostragem angular *cot-tsa*

A etapa de pré-processamento é executada por um único programa, chamado *cot-tsa*. O programa inicialmente realiza a reamostragem angular pelo método *computer order tracking*, e produz um novo arquivo com o mesmo nome original, mas com "-COT" adicionado ao nome.

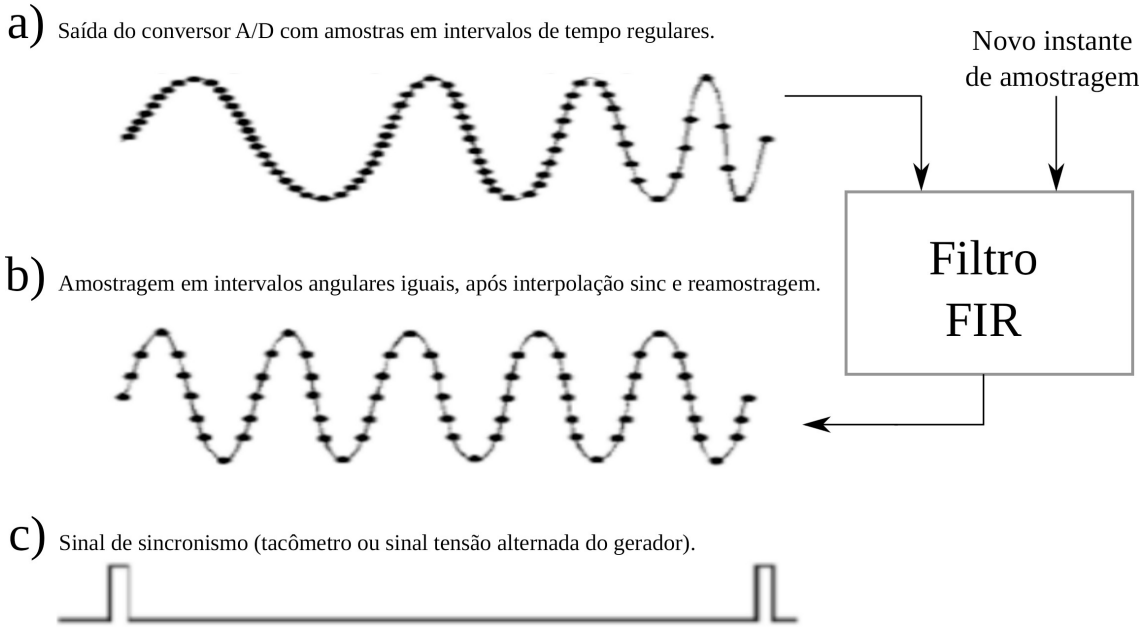


Figura 40 - Reamostragem angular com filtro FIR.

Fonte: Elaboração própria com *freeware* Inkscape v_0.91 for Linux.

A figura 40 mostra como o *computer order tracking* é realizado com o emprego de um filtro FIR (*finite impulse response*). Um filtro FIR, cuja resposta ao impulso é

uma aproximação da função *sinc,* é usado para produzir uma "interpretação analógica" (SMITH, GOSSETT, 1984) do sinal original (figura 40a), que foi amostrado em intervalos regulares de tempo. O processamento é baseado em sintetizar um filtro FIR que seja o mais próximo do filtro passa baixas ideal, cuja resposta seria teoricamente uma função *sinc*. Entretanto, a função *sinc* não é limitada no tempo, e o comprimento da resposta impulsiva do filtro FIR é selecionado para produzir uma interpolação garantindo uma certa relação sinal/ruído e uma certa banda passante no sinal "analógico em tempo contínuo" reconstruído. O sinal em "tempo contínuo" é então reamostrado em instantes de tempo que correspondem ao intervalos regularmente espaçados no domínio angular (figura 40b).

Na prática, uma biblioteca de código fonte aberto (*libsamplerate*) realiza esta reamostragem em segmentos correspondentes a uma revolução do eixo (como na figura 40, letra c), ou em segmentos menores, correspondentes a um ciclo senoidal de tensão alternada gerada. A biblioteca é configurada para realizar a reamostragem com a melhor qualidade possível, sendo garantida uma relação sinal/ruído de 97 dB no pior caso, e uma banda passante correspondente a 97% da frequência de Nyquist do sinal original (pois está configurada para fazer sempre uma sobreamostragem). A biblioteca é capaz de sintetizar o filtro FIR de acordo com a relação de reamostragem, mesmo que esta seja variante com o tempo, de modo totalmente transparente ao *software* aplicativo.

Os quatro canais do arquivo original são reamostrados, e gravados no novo arquivo com o "-COT" adicionado ao nome. O arquivo .ini "prontuário" é atualizado com a nova taxa de amostragem, e a quantidade de amostras L por revolução.

Além do arquivo com os sinais reamostrados, mais dois arquivos são produzidos: um contendo um conjunto de sinais resíduo (e que tem adicionado "-COT-residual" ao nome), e o outro contendo um conjunto de sinais determinísticos (que tem adicionado "-COT-periodic" ao nome). Desta maneira, é possível processar ou meramente visualizar somente os termos determinístico e aleatório, ou o sinal completo. A decomposição determinístico/periódico é realizada através do estimador SA, já descrito na secção 3.2. No total, para cada arquivo original de dados processado, são produzidos 3 arquivos .wav maiores, pois a taxa de amostragem aumenta. O custo computacional desta etapa, apesar de ser um pré-processamento, é bastante elevado,

tornando o tempo gasto para a reamostragem uma parcela considerável (superior a 50%) do tempo total de processamento dos sinais.

4.3.3 O *software* de processamento *spectrogram*

Os arquivos com os conjuntos de sinais já pré-processados são lidos pelo *software* chamado *spectrogram*, que inicialmente o divide em segmentos de *N* amostras (somente os canais de sinais vibracionais são processados por este programa). O parâmetro *N*, que é o número de amostras da janela de análises empregada para computar o AIPS, bem como o tipo de janela empregada (Hanning, Hamming, retangular, triangular, etc) é lido do arquivo "prontuário" .ini.

Cada segmento de *N* amostras é multiplicado amostra por amostra pela janela de análises selecionada, e os resultados são processados por uma outra biblioteca de código fonte aberto: a FFTW (*the Fastest Fourier Transform from the West*, que implementa o algoritmo de Cooley-Tuker). Esta biblioteca calcula a FFT do segmento de sinal, e os resultados vão sendo armazenados, já com os valores dos módulos elevados ao quadrado, em colunas da matriz IPS, na memória do computador.

A seguir, a matriz IPS é dividida em *K* (número de revoluções) submatrizes e a média, elemento a elemento, é computada, resultando na matriz AIPS. Opcionalmente, o programa *spectrogram* aceita um *flag* na linha de comando indicando que esta média não deve ser realizada, e a matriz IPS é processada no lugar da AIPS. Este *flag* foi adicionado com a intenção de investigar a existência da cavitação em vórtice no tubo de descarga, e raramente é usado na prática. Após a matriz AIPS computada, esta é salva em arquivo binário, e adicionalmente, um gráfico é produzido e também gravado em disco, no formato TGA (*Truevision "targa" Graphics Adapter*). Para maior comodidade na leitura de informações do gráfico, duas escalas são construídas no eixo das abcissas: uma para a *posição angular* do rotor, e a outra para o *tempo* decorrido desde o início da revolução. Todos os gráficos produzidos pelo programa são elaborados com a ajuda da biblioteca MathGL, também de código fonte aberto.

Continuando com o processamento, agora cada linha da matriz AIPS (ou IPS) é alimentada novamente à biblioteca FFTW, e os resultados armazenados como linhas da

nova matriz CMS. Novamente, esta matriz complexa é gravada em arquivo binário, e o gráfico contendo os módulos de seus elementos é elaborado e gravado em disco. Duas escalas são construídas no eixo das abcissas: uma escala de ordens de máquina, e uma escala de frequências cíclicas. O eixo das ordenadas permanece como no AIPS, com somente uma escala de frequências espectrais.

Finalmente, a matriz CMC é calculada a partir da matriz CMS, conforme descrito pela equação (21), e mais dois novos gráficos são produzidos: um com a estimativa do *power spectrum* (PS) do sinal (os módulos dos elementos da coluna correspondendo à ordem de máquina de número zero da matriz CMS), e o outro contendo os módulos dos elementos da matriz CMC, que também é salva em arquivo binário.

Como informação adicional, o software elabora um gráfico que mostra a potência total do sinal discriminada por ordens de máquina (esta informação é conseguida a partir da matriz CMC, acumulando-se separadamente os módulos de todos os elementos de cada coluna). Este gráfico é denominado *Cyclic Power Spectrum* (CPS), e serve como um resultado análogo ao que seria obtido com o processamento DEMON.

Os gráficos contendo o CPS e o PS são plotados como gráficos 2D comuns. Já os gráficos contendo o AIPS, a CMS e a CMC são plotados como uma imagem em tons de cinza, sendo que um pixel branco corresponde a um elemento da matriz com módulo zero, e um pixel totalmente preto corresponde ao elemento da matriz com o maior módulo. Desta forma, apesar de não haver escala logarítmica envolvida, a escala linear de tons de cinza sempre se adapta automaticamente aos dados da matriz a ser plotada.

Após todos os gráficos serem elaborados e, juntamente com as matrizes de dados, serem gravados em disco, o programa libera a memória alocada e encerra a sua execução. Para cada sinal vibracional, 3 arquivos binários contendo as matrizes AIPS, CMS e CMC são gravados, juntamente com 5 gráficos (AIPS, CMS, PS, CMC e CPS). Um extenso relatório de texto é produzido no terminal durante a execução, indicando o progresso de cada etapa do processamento, e os parâmetros usados. Este software foi otimizado para ser usado em processadores com múltiplos núcleos, explorando ao máximo o aumento na velocidade de execução devido ao processamento em paralelo de

múltiplas FFTs.

4.3.4 O *software* de extração de informações *detectcorrelations*

O *software* que é executado por último é chamado *detectcorrelations,* e inicia sua execução carregando as matrizes CMS na memória RAM. As colunas correspondentes às ordens de máquina de números *v* e *b* são tomadas como referências de distribuição espectral (somente os módulos dos elementos são considerados), e o método baseado no coeficiente de correlação de Pearson é aplicado, conforme descrito na secção 3.5. Os valores para as estimativas da agressividade da cavitação em bolhas itinerantes e em nuvem, tanto em termos absolutos quanto em termos relativos, são mostrados no terminal. Além de P_B, P_C, $P_{B\%}$ e $P_{C\%}$, a potência total do sinal (parcela estacionária) é também mostrada, e os cinco valores podem ser usados para diagnosticar a agressividade da cavitação. Os gráficos produzidos na etapa anterior podem revelar informações extras através de observação visual, e servem como auxiliares no diagnóstico.

4.4 CONFECÇÃO DO *HARDWARE* DEDICADO

Para possibilitar a execução automatizada dos *softwares in locus*, um *hardware* dedicado foi desenvolvido. A função deste *hardware* é reunir em um só equipamento todos os circuitos necessários para a alimentação dos sensores, condicionamento dos sinais, conversão A/D, gravação e processamento dos dados, bem como uma interface de comunicações com a sala de controles (por meio de um barramento serial ou *Ethernet*). Este equipamento então, imita o que era o escopo inicial deste projeto de pesquisa: um equipamento completo baseado em um ASIC.

Para reunir todas as funções citadas, o autor decidiu usar um computador compacto de baixo consumo de energia, e uma placa de áudio USB projetada especialmente para esta aplicação. Esta placa de áudio USB foi construída em primeiro lugar, e usa um módulo *USBstreamer*, fabricado pela empresa miniDSP (figura 41).

O módulo *USBstreamer* permite a conexão de até 4 conversores A/D estéreo pelos seus barramentos padrão I2S, totalizando 8 canais com resoluções de até 24 *bits* por amostra e 192.000 amostras por segundo em cada canal, e é visto pelo sistema operacional como um dispositivo USB de classe “áudio 2.0”.

Figura 41 - Módulo USBstreamer.

Fonte: <http://www.minidsp.com>

Uma placa de circuito impresso com 4 circuitos integrados PCM1804 (conversores A/D estéreo da *Texas Instruments* de 24 bits e 192.000 amostras por segundo, e entradas diferenciais) foi confeccionada e conectada ao módulo *USBstreamer*. Ainda nesta placa de circuito impresso, 4 amplificadores operacionais, modelo AD8138 da *Analog Devices*, com entradas e saídas diferenciais foram adicionados para realizar o condicionamento dos sinais (filtragem e amplificação). Finalmente, 4 fontes de corrente constante, montadas com os circuitos integrados LM234, fornecem a alimentação a até 4 acelerômetros. O módulo *USBstreamer* fornece os sinais de sincronismo, e garante que todos os canais são amostrados simultaneamente.

Esta placa e o módulo *USBstreamer* são alimentados por um banco de baterias, e juntas foram encerradas em uma caixa metálica, constituindo a placa de áudio com interface USB. Os detalhes de construção desta placa de áudio podem ser encontrados no Apêndice F – Esquemas elétricos e layouts.

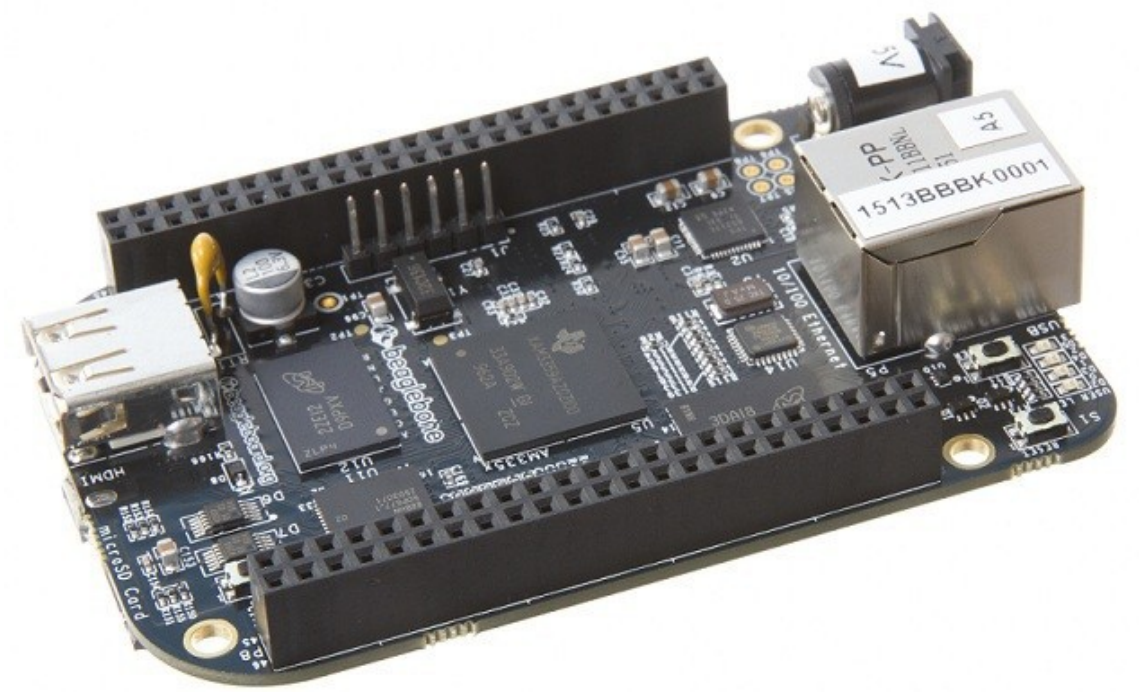


Figura 42 - Computador embarcado *Beaglebone Black*, do tamanho de um cartão de crédito e com alimentação de 5V x 2A. O computador possui conexão à rede Ethernet, porta serial e portas USB.

Fonte:<http://www.farnellnewark.com.br>.

A escolha inicial para o computador compacto e de baixo custo foi um computador embarcado modelo *Beaglebone Black*, produzido pela *Texas Instruments*, e que é capaz de executar o sistema operacional Linux *kernel* 3.8 em um processador ARM7 e 512 MB de memória RAM (figura 42). Os *softwares* da *suite* são compilados usando o compilador GCC, e portanto, podem ser executados no *Beaglebone Black.* Entretanto, devido a um *bug* no *kernel* 3.8, a gravação de sinais de dispositivos USB classe "áudio 2.0" era sempre corrompida, devido a sobrecarga de *buffers*.

O *Beaglebone Black* foi substituído por um computador mini-ITX (*Information Technology eXtended*), encerrado em um gabinete compacto, com 4 GB de memória RAM e processador de dois núcleos da família x86. Para tornar o computador resistente a vibrações, um disco rígido de estado sólido foi usado, e o sistema operacional Linux instalado. A placa de áudio USB foi conectada e testada com sucesso, não sendo observada nenhuma anomalia no funcionamento do conjunto. A *suite* de *softwares* foi compilada e executada no próprio computador dedicado (figura 43).

Figura 43 - Equipamento final desenvolvido.

Fonte: Produção própria.

5 RESULTADOS E DISCUSSÕES

Após a metodologia ter sido desenvolvida e implementada como uma *suite* de *softwares*, ela necessitou ser validada, e seu desempenho na extração de informações sobre a agressividade da cavitação avaliado. Para esta avaliação, um conjunto de sinais sintéticos foi produzido. Os sinais reais adquiridos com o gravador U2525A também foram processados, e diagnósticos comparativos entre os tipos de cavitação (e suas agressividades) encontrados em UG01 e UG02 foram elaborados. Alguns sinais interferentes, presentes em ambos conjuntos TG, foram identificados. Adicionalmente, algumas discussões sobre o *droop speed control* observado em UG01 e UG02, e a efetividade da sua correção com a reamostragem angular serão apresentadas.

5.1 PRODUÇÃO DE UM CONJUNTO DE SINAIS SINTÉTICOS

Um conjunto de sinais sintéticos foi produzido pelo *software* sintetizador, com as seguintes especificações: 900 revoluções do eixo com *droop speed control* de 1%, rotor com b=11 lâminas, distribuidor com v=24 palhetas, gerador elétrico com P=44 polos magnéticos e frequência nominal da tensão alternada 60 Hz. Estas configurações passadas ao *software* sintetizador foram escolhidas por serem semelhantes às características de UG01 e UG02. Foi configurada também a mesma taxa de amostragem usada para as gravações dos sinais reais: 100.000 amostras por segundo.

Os dois sinais vibracionais do conjunto sintético foram produzidos com configurações idênticas, sendo as componentes modulantes $m_B(\theta)$ e $m_C(\theta)$ descritas pelas seguintes séries de Fourier:

$$m_B(\theta) = 1 + \cos(11\theta) - 0{,}25\cos(33\theta) + 0{,}25\cos(55\theta) \tag{40}$$

$$m_C(\theta) = 1 + \cos(24\theta) \tag{41}$$

Novamente, estas ordens de máquina foram escolhidas por serem as mesmas ordens de máquina observadas em alguns dos sinais reais. As amplitudes, entretanto, foram arbitradas com o intuito de gerar um sinal com índice de modulação de 100%, o que é normalmente observado em sinais reais (intermitência). Isto também garantiu que

os sinais modulantes fossem sempre não negativos.

As portadoras vibracionais foram sintetizadas como respostas de filtros IIR passa-faixa ao ruído branco gaussiano. O ruído branco simula as excitações acústicas $n_B(t)$ e $n_C(t)$ (cavitações em bolha itinerantes e em nuvem), sendo sintetizado um ruído branco com densidade espectral de potência de 0,625 μV²/Hz (este valor foi o maior possível que evita o *clipping* excessivo de *outliers*, devido à faixa dinâmica limitada a ±1,0). A potência total de cada uma das excitações acústicas, considerando uma banda de 0 a 50 kHz, é portanto 31,25 mV².

Ambos os filtros IIR $h_B(t)$ e $h_C(t)$ que simulam as transmissões mecânicas são filtros de ordem 4. O filtro que simula a transmissão para a cavitação em bolhas é um passa faixas que só permite a passagem de frequências entre 5 e 20 kHz. Já o filtro correspondente para a cavitação em nuvem permite a passagem em uma faixa mais larga, e de frequências espectrais superiores: de 15 a 35 kHz. Estes filtros foram escolhidos desta forma para, em primeiro lugar, permitir a diferenciação entre as vibrações produzidas por cada cavitação, e em segundo lugar, para simular o efeito *harmonic cascading*, apesar de este efeito ocorrer na excitação acústica e não na transmissão mecânica.

As portadoras vibracionais sem modulação $c_B(t)$ e $c_C(t)$ têm portanto, potência média de 10,187 mV² e 13,221 mV², e estes valores são exatamente as *potências médias* dos SVIs sintéticos, pois a modulação não altera a potência média, somente a variação ao longo do tempo. Nota-se que estes seriam respectivamente os valores verdadeiros de P_B e P_C, caso os sinais não fossem sintéticos.

Com a modulação, o SVI sintético da cavitação em bolhas itinerantes $\hat{x}_{SVI-B}(t)$ tem a sua *potência instantânea média* variada em ±100% do valor da sua *potência média*, devido aos coeficientes da série de Fourier na equação (40). O mesmo ocorre com $\hat{x}_{SVI-C}(t)$ (equação 41).

Ambos $\hat{x}_{SVI-B}(t)$ e $\hat{x}_{SVI-C}(t)$ são somados a um ruído branco gaussiano com potência média total de 5 mV² (na faixa entre 0 e 50 kHz), e o resultado compõe o sinal sintético vibracional do canal 0, $\hat{x}_1(t)$. O sinal vibracional do canal 1, $\hat{x}_2(t)$, é produzido de forma idêntica, exceto que são adicionados dois ruídos tonais à

composição de $\hat{x}_1(t)$. Os ruídos tonais estão nas frequências 30 e 40 kHz, e cada um tem potência de 25 mV². Este conjunto de sinais é o mesmo retratado na figura 35.

5.2 RESULTADOS OBTIDOS COM OS SINAIS SINTÉTICOS

O conjunto de sinais sintéticos foi processado pela *suite* de *softwares*, visando a validação da metodologia, e os três conjuntos de sinais reais foram processados pela mesma *suite* para investigar as diferenças entre UG01 e UG02 no momento das gravações. Em todos os casos, janelas de análise do tipo *Hanning* com comprimento de 512 amostras e fração de *overlap* de 75% foram usadas. A reamostragem angular também elevou a taxa de amostragem original de 100.000 amostras por segundo para a taxa de amostragem angular equivalente a 122.880 amostras por segundo (45.056 amostras por revolução). Em todos os casos, o sinal alternado do gerador foi usado como sinal de sincronismo (*tracking* da ordem de máquina de número 22).

5.2.1 Validação da metodologia desenvolvida

O conjunto de sinais sintéticos foi inicialmente reamostrado no domínio angular, e o software *cot-tsa* detectou as flutuações na velocidade de rotação do eixo, e um gráfico foi elaborado (figura 44).

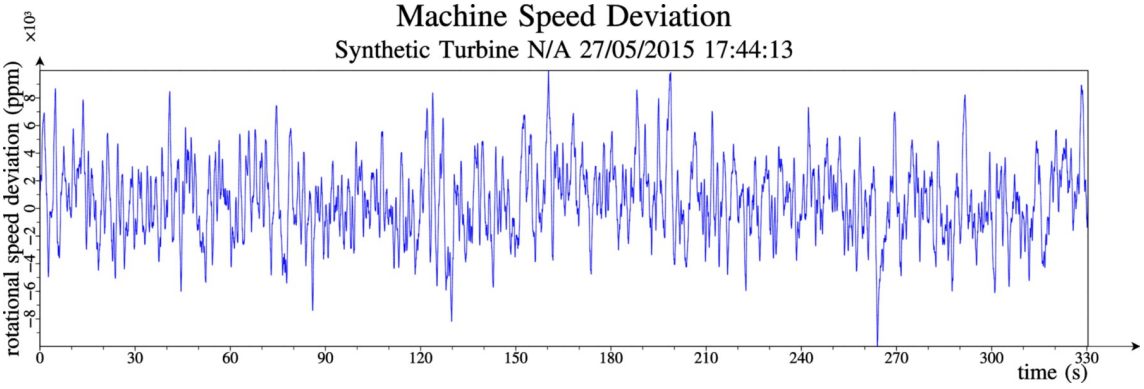


Figura 44 - Flutuações de velocidade da turbina simulada, em 1% do valor nominal.

Fonte: Elaboração própria do *software cot-tsa*.

A decomposição periódico/residual foi realizada, totalizando 3 arquivos de saída

da etapa de pré-processamento (um contendo os sinais vibracionais no domínio angular $\hat{x}_1(\theta)$ e $\hat{x}_2(\theta)$, um contendo seus termos periódicos $\hat{x}_{p1}(\theta)$ e $\hat{x}_{p2}(\theta)$, e o último contendo os termos residuais $\hat{x}_{r1}(\theta)$ e $\hat{x}_{r2}(\theta)$). Os termos residuais foram processados pelo software *spectrogram*, que inicialmente produziu os gráficos com as matrizes AIPS.

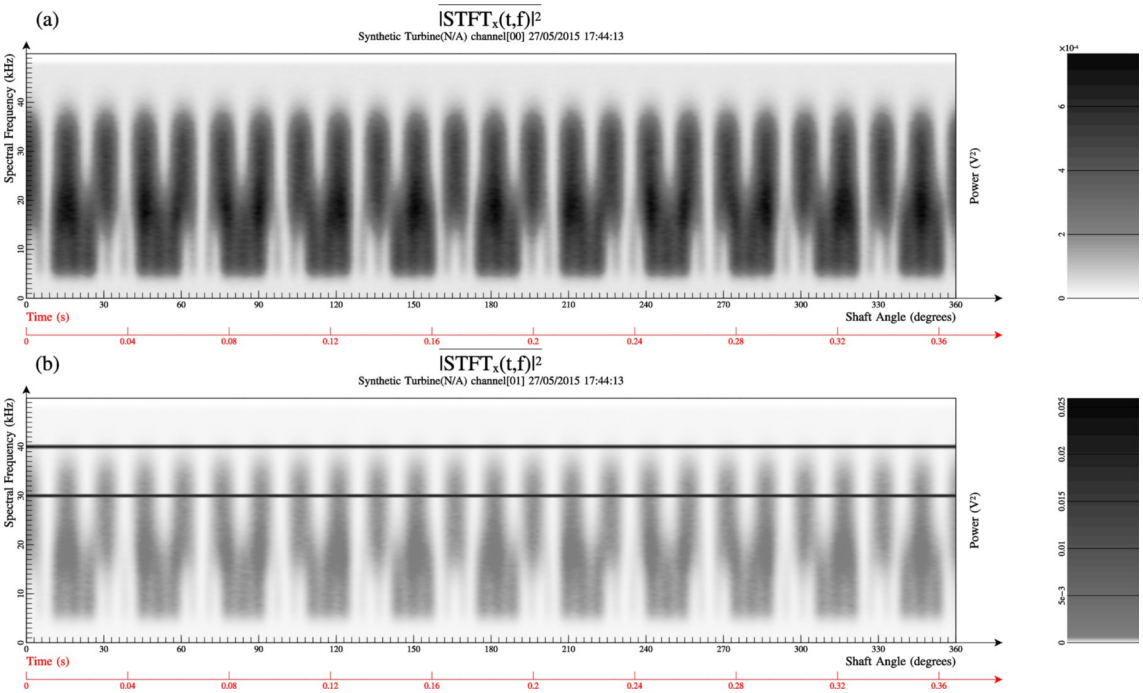


Figura 45 - AIPS calculado sobre sinal sintético, mostrando as flutuações de potência entre 5 e 35 kHz. Em (a): Os SVIs da cavitação em bolhas e em nuvem e ruído branco gaussiano aditivo. Em (b): Adição de mais dois termos de ruído tonais em 30 e 40 kHz.

Fonte: Elaboração própria, *software spectrogram*.

Na figura 45a, é possível ver as flutuações de potência do sinal na faixa entre 5 e 35 kHz aproximadamente, caracterizando as componentes cicloestacionárias do sinal, e em torno de 20 kHz há uma sobreposição dos padrões periódicos, de forma que a detecção visual de algum padrão de repetição se torna impraticável. Já na figura 45b, os dois termos de ruídos tonais são claramente visíveis, e a escala de potências (à direita) mudou para poder acomodar o sinal completo. Como resultado, as componentes cicloestacionárias acabaram por aparecer mais apagadas.

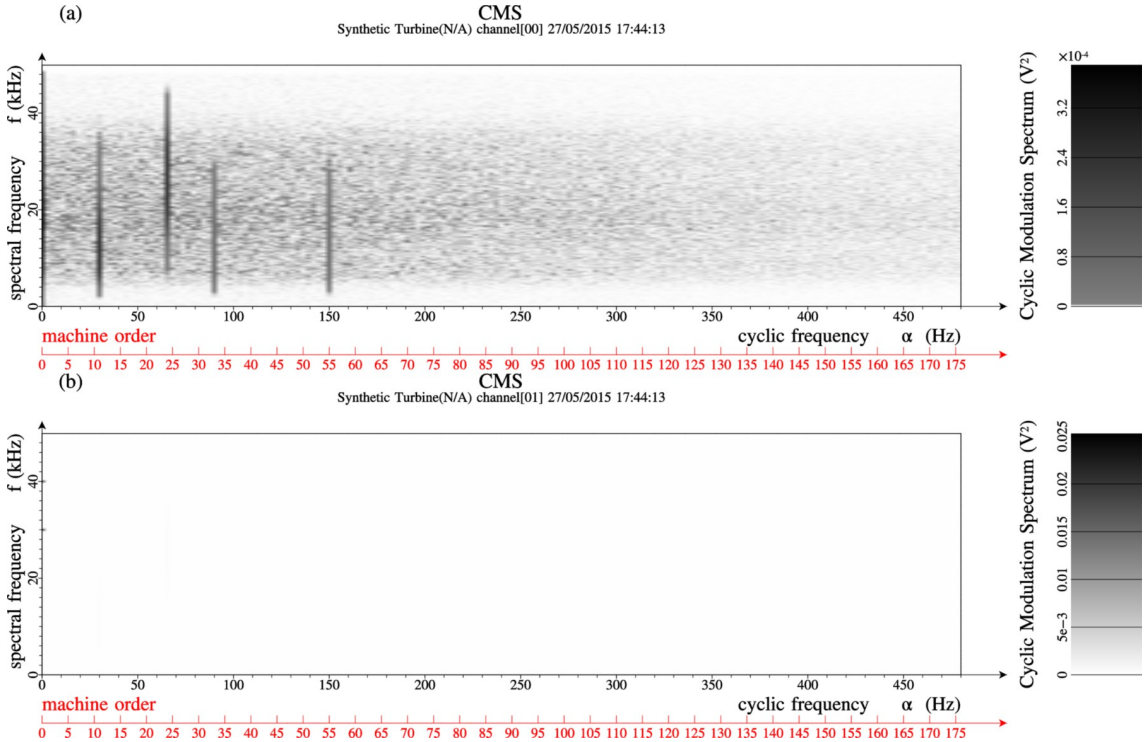


Figura 46 - Imagens formadas pelo módulo dos elementos das matrizes CMS, para o conjunto de sinais sintéticos. Os ruídos tonais em (b) causaram o apagamento total da imagem, só restando dois pontos sobre o eixo das frequências espectrais (30 e 40 kHz).

Fonte: Elaboração própria, *software spectrogram*.

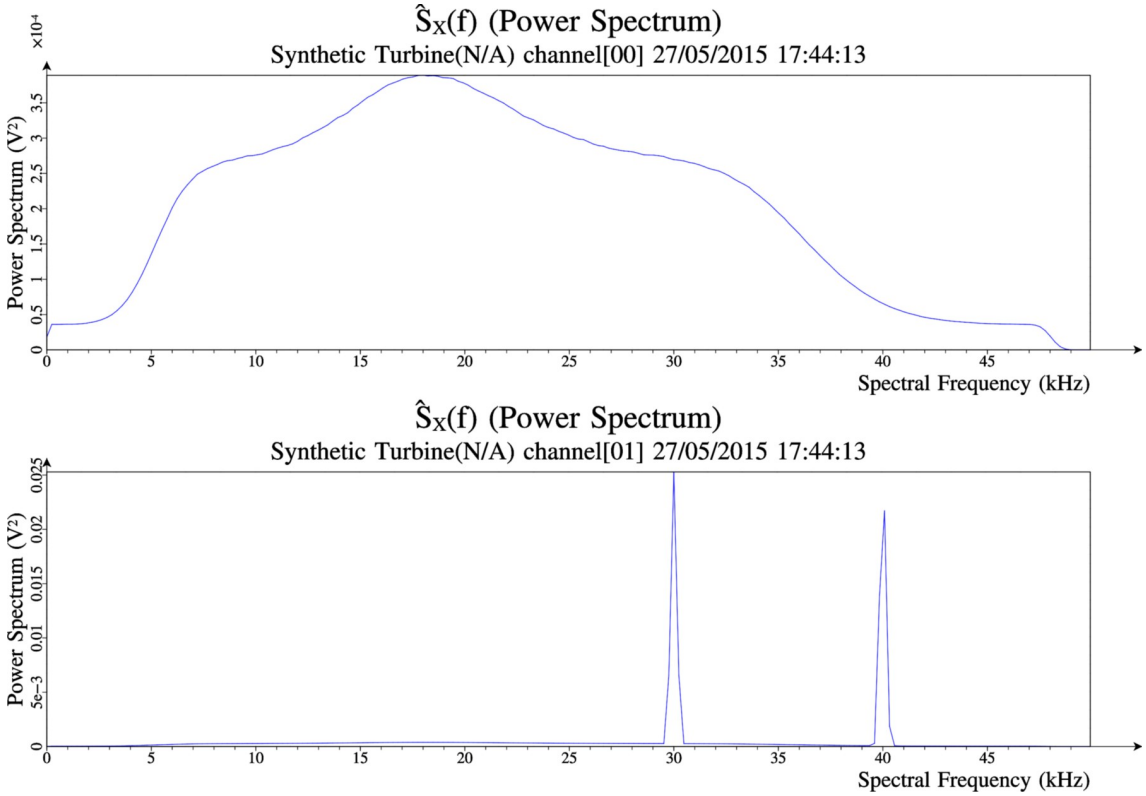


Figura 47 - Espectros de potência (PS) dos sinais sintéticos.

Fonte: Elaboração própria, *software spectrogram*.

Na figura 46a, as três componentes de ordem de máquina produzidas pela cavitação em bolhas, e a componente de ordem de máquina produzida pela cavitação em nuvem são visíveis como linhas verticais, e distinguíveis entre si. Adicionalmente, é possível ver a componente estacionária plotada sobre o eixo das frequências espectrais, que é a estimativa do espectro em potência do sinal (figura 47). Com a adição dos ruídos tonais (figura 46b), a imagem é apagada, pois a maioria da potência do sinal está concentrada em somente dois pontos sobre o eixo das frequências espectrais. A matriz CMS fica intacta na memória do computador, apenas a sua visualização em forma de imagem que é inapropriada quando há fortes componentes estacionárias e/ou periódicas.

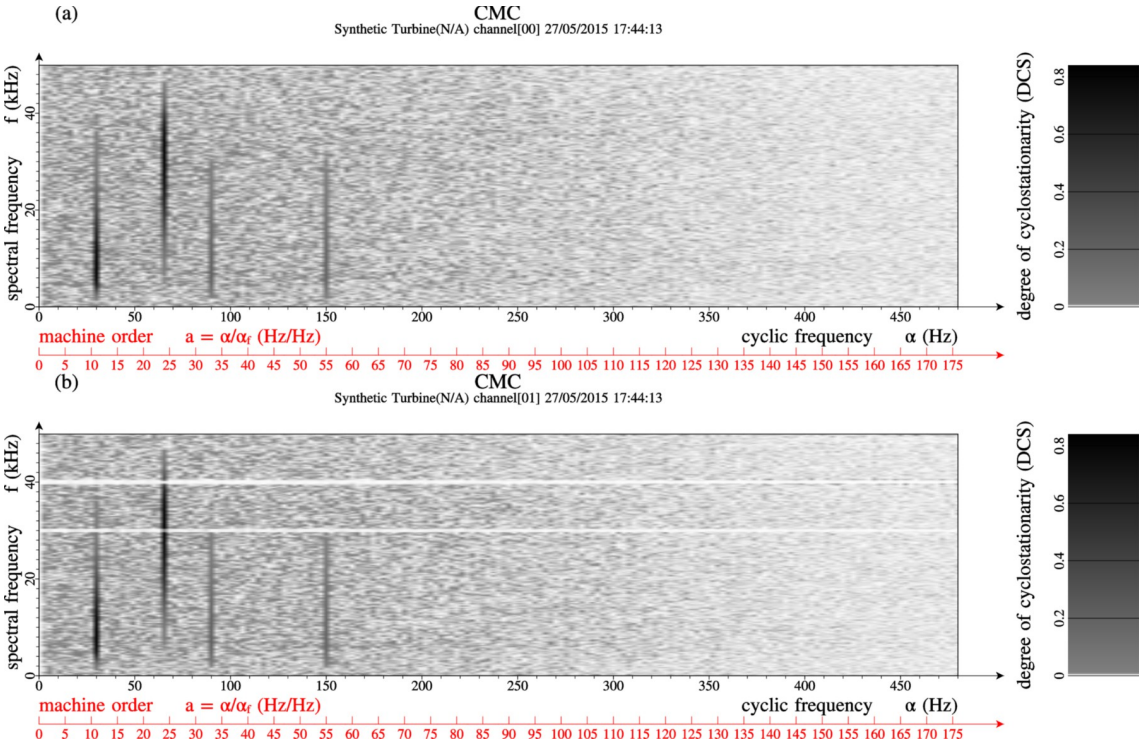


Figura 48 - Imagens formadas pelos módulos dos elementos das matrizes CMC do conjunto de sinais sintéticos.

Fonte: Elaboração própria, *software spectrogram*.

As matrizes CMC foram calculadas a partir das CMSs e PSs, e os resultados podem ser vistos na figura 48. Nota-se claramente que a influência dos termos de ruído tonais é mínima, somente causando o apagamento das linhas correspondentes às frequências espectrais. Ambas as escalas de grau de cicloestacionariedade estão com o mesmo valor de fundo de escala, diferentemente dos gráficos da figura 46. O valor de fundo de escala neste caso deveria ser 1 (100%), que é o maior grau de

cicloestacionariedade produzido no sinal sintético (devido à cavitação em nuvem). Entretanto, o gráfico apresenta um valor de fundo de escala ligeiramente menor, pois a adição do ruído branco gaussiano, e a parcela estacionária do SVI da cavitação em bolhas itinerantes contribuem para uma diminuição do grau de cicloestacionariedade.

5.2.1.1 Discussão sobre a agressividade da cavitação simulada

Após o conjunto de sinais sintéticos ter sido preprocessado e processado, o software *detectcorrelations* realizou a extração, nas matrizes CMS, das informações sobre a agressividade da cavitação simulada. Os resultados das medições de P_B, P_C e da potência total do sinais estão na tabela 3.

Tabela 3 - Resultados obtidos com o conjunto de sinais sintéticos.

Canal (sinal)	P_B(mV²) (% do total)	P_C(mV²) (% do total)	Potência Total (mV²)
$\hat{x}_1(t)$ (canal 0)	9,93494 (35,21%)	12,83301 (45,48%)	28,21804
$\hat{x}_2(t)$ (canal 1)	9,90310 (12,66%)	12,81262 (16,38%)	78,23155

Tabela 4 - Componentes usadas na produção do conjunto de sinais sintéticos.

Componente	SVI da cavitação em bolhas $\hat{x}_{SVI-B}(t)$	SVI da cavitação em nuvem $\hat{x}_{SVI-C}(t)$	Ruído branco gaussiano aditivo	Cada componente de ruído tonal
Potência da componente	10,191 mV²	13,216 mV²	5,0 mV²	25,0 mV²

Lembrando que o conjunto de sinais sintéticos foi produzido com as componentes sumarizadas na tabela 4 (vide secção 5.1), e lembrando que o *valor verdadeiro convencional* (VVC) de P_B é a potência média de $\hat{x}_{SVI-B}(t)$, e o VVC de P_C é a potência média de $\hat{x}_{SVI-C}(t)$, temos então os seguintes VVCs:

$$\widetilde{P_B} = 10{,}191\, mV^2 \tag{42}$$

$$\widetilde{P_C} = 13{,}216\, mV^2 \tag{43}$$

Nas equações 42 e 43, o til denota o valor verdadeiro convencional. Também, o VVC para a potência total de $\hat{x}_1(t)$ é 28,408 mV² e para $\hat{x}_2(t)$, 78,408 mV².

A medição dos valores de P_B, P_C e da potência total, para o sinal $\hat{x}_1(t)$ através da metodologia desenvolvida, apresentou *erros de medição relativos* de -2,51%; -2,89% e -0,67% respectivamente. Já para o sinal $\hat{x}_2(t)$, os erros relativos são respectivamente -2,82%, -3,05% e -0,22%. De imediato, é possível concluir que, para as condições de aquisição de dados, todos os *erros de medição relativos* foram inferiores a 4% (ou seja, esta é a acurácia na medição da grandeza agressividade da cavitação, se o equipamento for entendido como um instrumento de medição que acabou de ser calibrado usando um sinal padrão sintético, na impossibilidade de usar um sinal padrão real ou um outro instrumento de medição como padrão de calibração). Para as incertezas de medição, há teoricamente um limite inferior determinado pelo ruído de estimação no cálculo da matriz CMS (ANTONI, HANSON, 2010, 2012), o qual é inversamente proporcional ao número total de janelas de análise, usada na etapa de cálculo da matriz IPS. O número de janelas de análise pode ser aumentado através de um aumento na fração de *overlap*, mas a fração de *overlap* de 75% já foi escolhida para a janela tipo Hanning, pois este valor resulta em planura perfeita em amplitude e também em potência.

A segunda conclusão que pode ser tirada vem de uma observação do fato que todos os erros relativos são negativos, significando que a estimação das potências das componentes de ordem de máquina está sempre retornando valores ligeiramente abaixo dos valores esperados. O motivo é de fácil entendimento, se a etapa de cálculo da matriz AIPS for enxergada como se o sinal fosse processado por um *comb filter* com vários canais de largura de banda Δf. Fica evidente que a variação na potência média instantânea do sinal já filtrado, e que passa por uma banda Δf, não pode acontecer em frequências muito superiores a Δf. Em outras palavras, o cálculo da matriz AIPS através da aplicação da STFT com uma resolução em frequência espectral Δf funciona como um filtro passa baixas no domínio da frequência cíclica, sendo que quanto maior a

frequência cíclica, ou melhor, a ordem de máquina, maior o erro introduzido. Para valores baixos de número de ordem de máquina, o erro introduzido é bem pequeno, mas para 88, o erro introduzido chega a aproximadamente 33% da potência medida. O software *spectrogram* calcula os fatores de correção para as ordens de máquina de número 88 e múltiplos somente. Este efeito de filtragem que a STFT produz, apesar de introduzir um pequeno erro na medição de P_B e P_C e parecer indesejável, também evita o *aliasing* no domínio da frequência cíclica.

5.3 RESULTADOS OBTIDOS COM OS SINAIS REAIS

Os três conjuntos de sinais, provenientes de UG01 e UG02, foram processados pela suíte de *softwares*, e os resultados das estimativas de P_B, P_C, $P_{B\%}$ e $P_{C\%}$ estão sumarizados na tabela 5. Na computação das matrizes AIPS, as linhas correspondentes às frequências espectrais abaixo de 5 kHz foram anuladas, pois esta faixa é demasiadamente contaminada com ruídos mecânicos da própria máquina.

Tabela 5 - Estimadores computados

Conjunto de sinais	Acelerômetro (local)	P_B (V²)	$P_{B\%}$	P_C (V²)	$P_{C\%}$	Potência total (V²)
1 (UG01)	1 (munhão 5)	22,9 μ	3,85%	9,10 μ	1,53%	594,3 μ
	2 (tampa)	31,6 μ	6,99%	7,34 μ	1,62%	452,7 μ
2 (UG01)	1 (munhão 5)	21,1 μ	4,80%	7,89 μ	1,80%	438,5 μ
	2 (munhão 4)	21,0 μ	1,17%	7,69 μ	0,42%	1792,8 μ
3 (UG02)	1 (munhão 4)	0,72 μ	0,05%	0,78 μ	0,06%	1288,7 μ
	2 (munhão 5)	0,77 μ	0,45%	0,53 μ	0,31%	168,8 μ

5.3.1 Diagnóstico da cavitação em UG01

5.3.1.1 Conjunto de sinais 1

O primeiro conjunto de sinais reais ao ser processado pelo software *spectrogram*, produziu como resultados os gráficos mostrados nas figuras 49, 50 e 51.

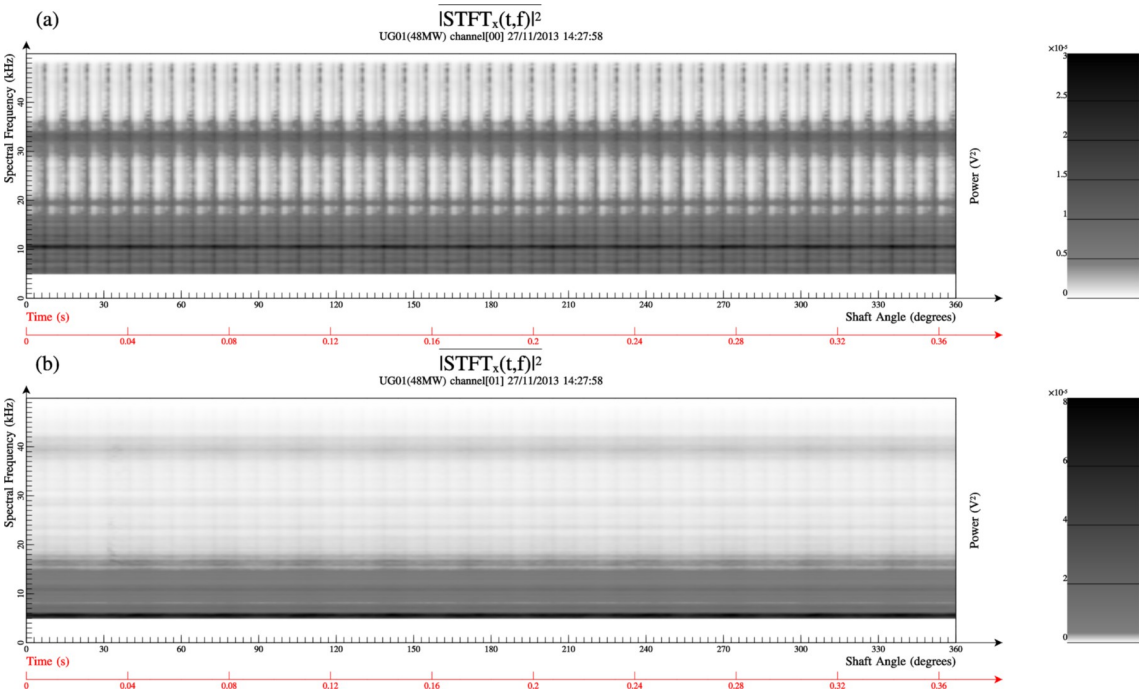


Figura 49 - Imagens das matrizes AIPS computadas no primeiro conjunto de sinais.

Fonte: Elaboração própria, *software spectrogram*.

As imagens formadas pelos AIPS mostram padrões periódicos de variação da potência em função da posição angular, para largas faixas de frequência espectral, caracterizando sinais aleatórios modulados em amplitude. Na figura 49a, as linhas são muito mais nítidas que na figura 49b, significando um grau de modulação muito maior. Também é notável na figura 49b uma linha horizontal próxima à frequência espectral 5 kHz, indicando que a maior parte da potência está concentrada nas frequências inferiores. Estas diferenças existem em razão das localizações bem diferentes entre os dois sensores, e, isto indica (visualmente) que a captação de sinais cicloestacionários é mais favorável quando o sensor é instalado sobre o munhão.

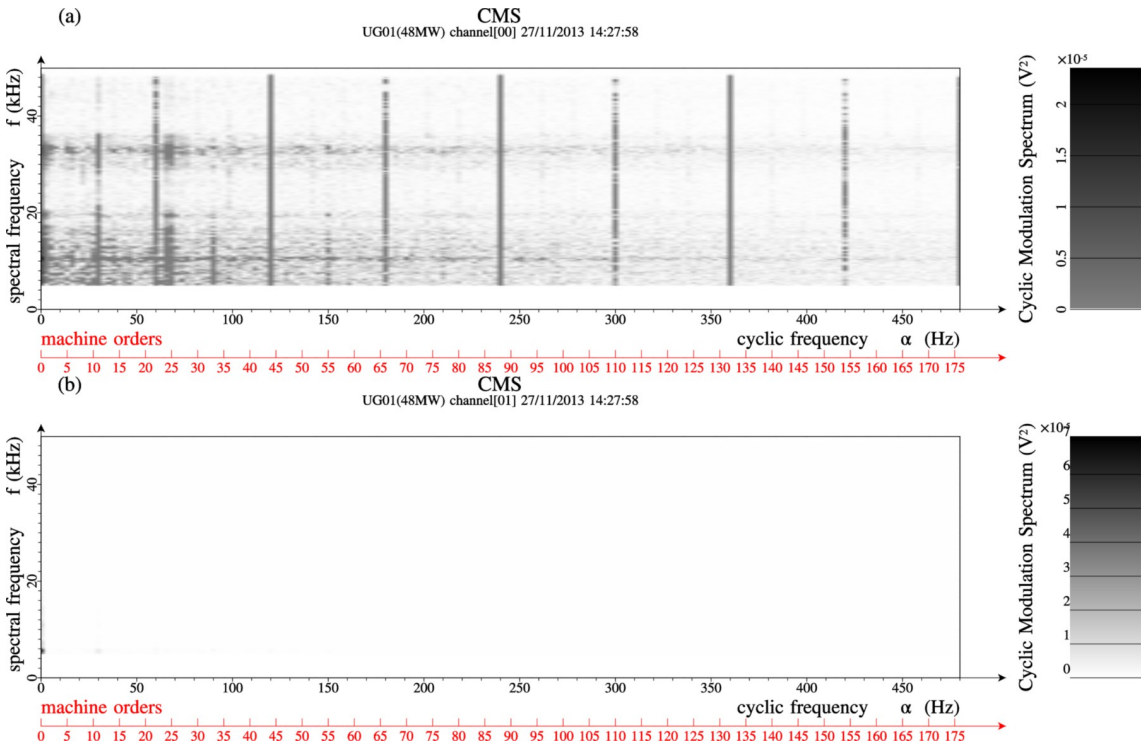


Figura 50 - Imagens das matrizes CMS computadas no primeiro conjunto de sinais.

Fonte: Elaboração própria, *software spectrogram*.

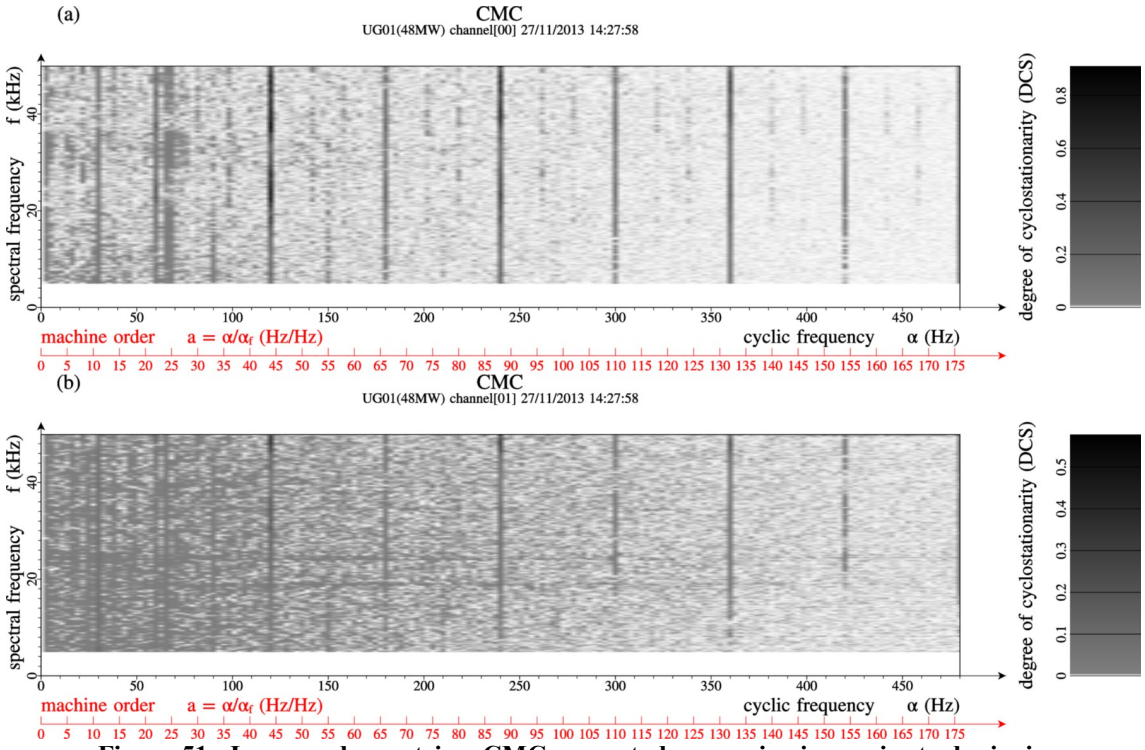


Figura 51 - Imagens das matrizes CMC computadas no primeiro conjunto de sinais.

Fonte: Elaboração própria, *software spectrogram*.

O gráfico formado pela matriz CMS da figura 50a mostra linhas *descontínuas* nas ordens de máquina de número 11, 33 e 55 (frequências cíclicas 30, 90 e 150 Hz), e uma linha extremamente fraca na ordem de máquina 77 (210 Hz). Apesar de serem linhas descontínuas com padrões aleatórios, os mesmos padrões se repetem, indicando que são todas componentes produzidas pelo mesmo tipo de cavitação, que no caso é identificado como cavitação em bolhas itinerantes. Adicionalmente, há também uma linha descontínua na ordem de máquina de número 24, evidenciando cavitação em nuvem. Esta linha descontínua é na verdade uma linha dupla, de aparência destacada das demais, ocupando a ordem de máquina de número 25 também. Esta é uma forte evidência que o derramamento de vórtices de Von Kármán está interferindo com a cavitação em nuvem. Finalmente, a informação mais marcante que o gráfico revela é a presença de fortes componentes estranhas nas ordens de máquina múltiplas de 22 (60 Hz e harmônicas) e ainda mais fortes nas ordens de máquina múltiplas de 44 (120 Hz e harmônicas). Visualmente, estas componentes parecem ter similaridade espectral, e isso sugere que elas sejam produzidas pelo mesmo fenômeno. A figura 50b não revela nenhuma informação, exceto por uma componente de ruído tonal presente em torno da frequência espectral 5 kHz, que causou o apagamento do gráfico inteiro. De fato, a instalação do sensor na tampa da turbina favoreceu a captação de vibrações de frequências inferiores a 5 kHz, provenientes da própria máquina.

A imagem formada pela matriz CMC na figura 51b revela informações invisíveis na imagem da CMS equivalente, pois diferentemente da CMS, a CMC rejeita o ruído tonal. Nota-se que a linha na ordem de máquina de número 24 não é mais uma linha dupla (como na figura 51a, e sim uma linha simples e bem fraca), significando que a instalação do sensor na tampa da turbina desfavoreceu a detecção da cavitação em nuvem e da interferência dos vórtices de Von Kármán. A cavitação em bolhas itinerantes entretanto, teve a sua detecção favorecida. As linhas correspondentes da tabela 5 mostram valores que corroboram com estas conclusões, pois nota-se que, apesar de a potência total do sinal do sensor na tampa ser menor que a potência do sinal proveniente do sensor no munhão, a proporção da potência detectada para o SVI da cavitação em bolhas itinerantes é bem maior.

5.3.1.2 Conjunto de sinais 2

O segundo conjunto de sinais reais ao ser processado pelo software *spectrogram*, produziu como resultados os gráficos mostrados nas figuras 52, 53 e 54.

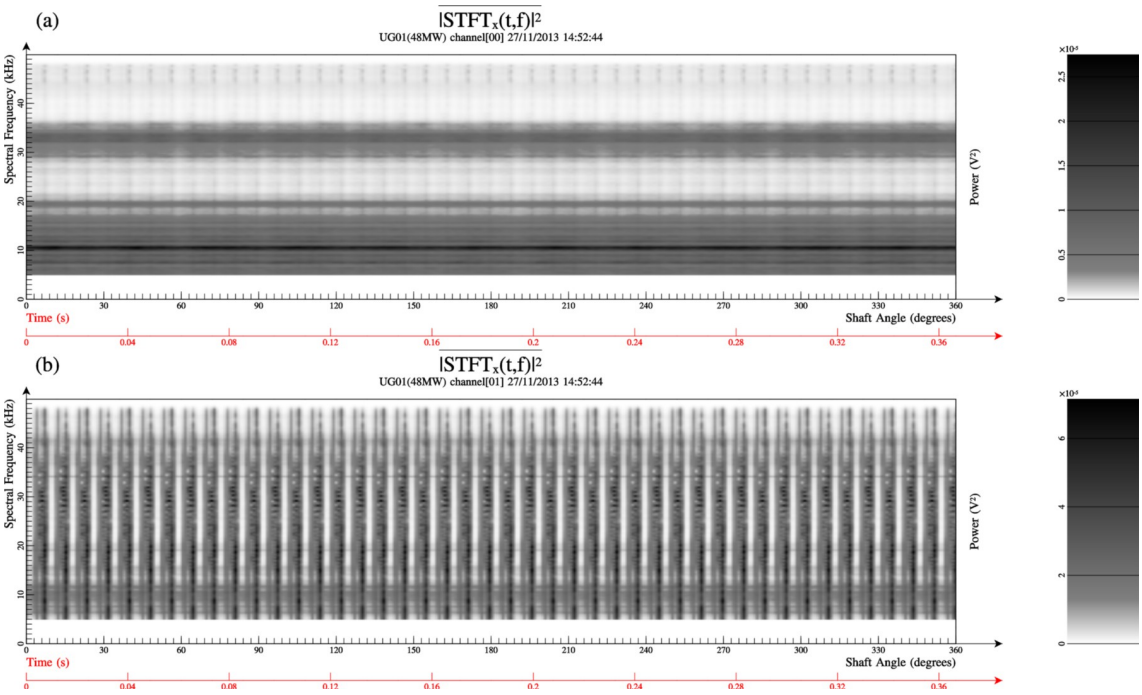


Figura 52 - Imagens das matrizes AIPS computadas no segundo conjunto de sinais.

Fonte: Elaboração própria, *software spectrogram*.

Na figura 52a, a imagem do AIPS para o sinal do acelerômetro 1 apresenta grande semelhança com a calculada para o sinal do mesmo acelerômetro no primeiro conjunto de sinais. Ao comparar as linhas correspondentes na tabela 5, nota-se que houve uma variação muito pequena, o que indica que houve poucas variações nos sinais vibracionais do acelerômetro 1, entre as duas gravações. Já a figura 52b mostra padrões periódicos de variação de potência muito mais nítidos que todos os outros sinais.

Na figura 53a, as mesmas linhas descontínuas nas ordens de máquina 11, 33, 55 (fraca) e 77 (muito fraca) estão presentes. Adicionalmente, uma linha com similaridade espectral é visível na ordem de máquina 22.

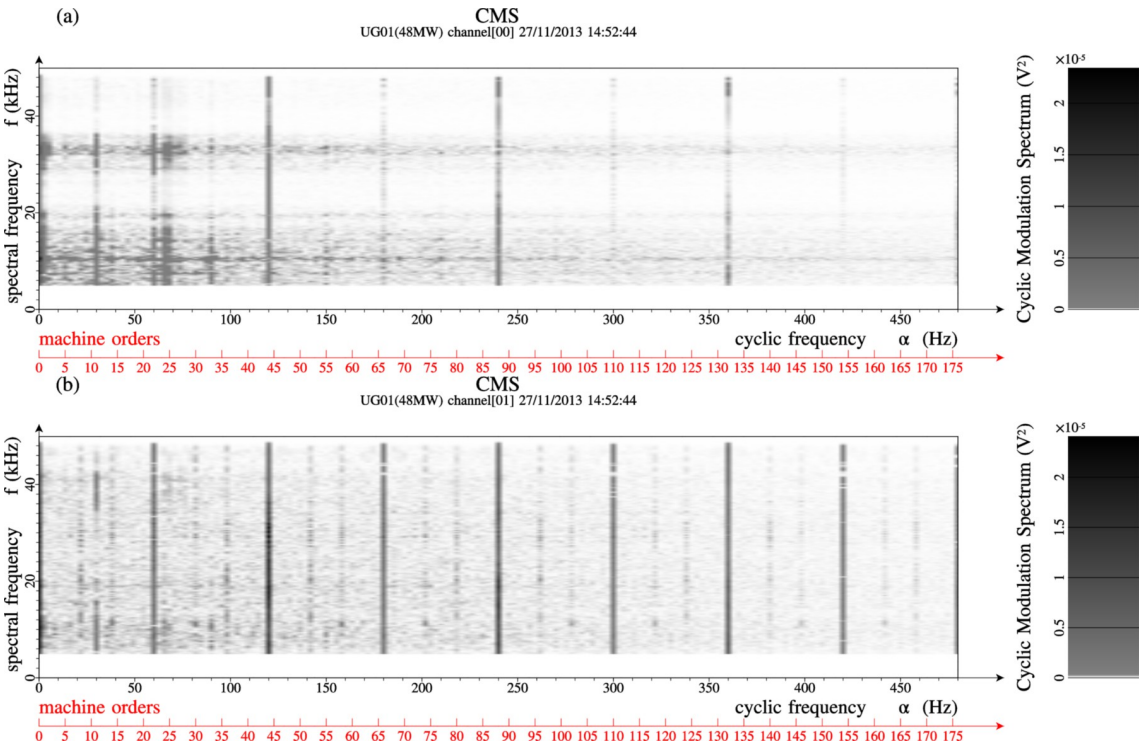


Figura 53 - Imagens das matrizes CMS computadas no segundo conjunto de sinais.

Fonte: Elaboração própria, *software spectrogram*.

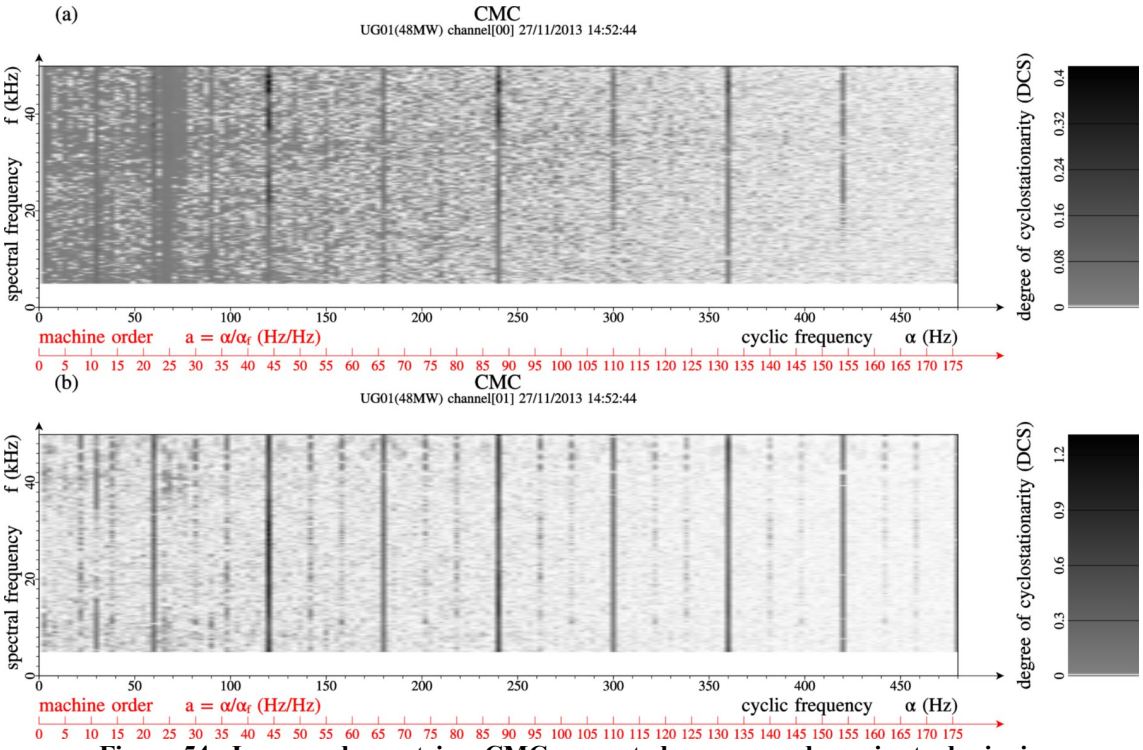


Figura 54 - Imagens das matrizes CMC computadas no segundo conjunto de sinais.

Fonte: Elaboração própria, *software spectrogram*.

As mesmas componentes de interferência presentes no conjunto de sinais 1, nas ordens de máquina múltiplas de 44 (120 Hz), estão visíveis na figura 53a, com a diferença que as ordens de máquina múltiplas ímpares de 22 (frequências cíclicas 180 Hz, 300 Hz, ...) apresentam componentes relativamente bem mais fracas. A cavitação em nuvem associada ao derramamento de vórtices de Von Kármán aparece da mesma forma que nos resultados do primeiro conjunto de sinais (ordens de máquina de número 24 e 25).

Na figura 53b (acelerômetro 2), as componentes de interferência são muito mais fortes, causando o apagamento de quase todas as linhas dos SVIs das cavitações. A linha correspondente na tabela 5 mostra valores estimados de P_B e P_C que são próximos aos valores para os sinais do acelerômetro 1, tanto no primeiro conjunto de sinais quanto no segundo (obtidos do munhão 5). Entretanto, a potência total do sinal é substancialmente maior (1792,8 μV^2), indicando a forte contribuição das componentes de interferência, o que causa diminuição significativa em ambos $P_{B\%}$ e $P_{C\%}$. Não obstante, os valores de P_B e P_C sofreram pouca variação. A figura 54 mostra as mesmas informações obtidas da figura 53.

5.3.2 Diagnóstico da cavitação em UG02

O terceiro conjunto de sinais reais ao ser processado pelo software *spectrogram*, produziu como resultados os gráficos mostrados nas figuras 55, 56 e 57. A figura 55a mostra os mesmos padrões de variações periódicas de potência dos casos anteriores, mas na figura 55b, estes padrões estão ausentes, parecendo ser um sinal estacionário. Na figura 56a, a imagem formada pela matriz CMS foi totalmente apagada devido ao ruído tonal presente em aproximadamente 46 kHz. Já a figura 56b não mostra linhas nas ordens de máquina que são características da cavitação em bolhas itinerantes, e da cavitação em nuvem. Nem mesmo as linhas dos sinais interferentes estão visíveis.

As imagens formadas pelas matrizes CMC na figura 57 mostram, para o primeiro acelerômetro, somente as componentes cicloestacionárias de interferência, e para o segundo acelerômetro, ausência de linhas representando quaisquer componentes cicloestacionárias.

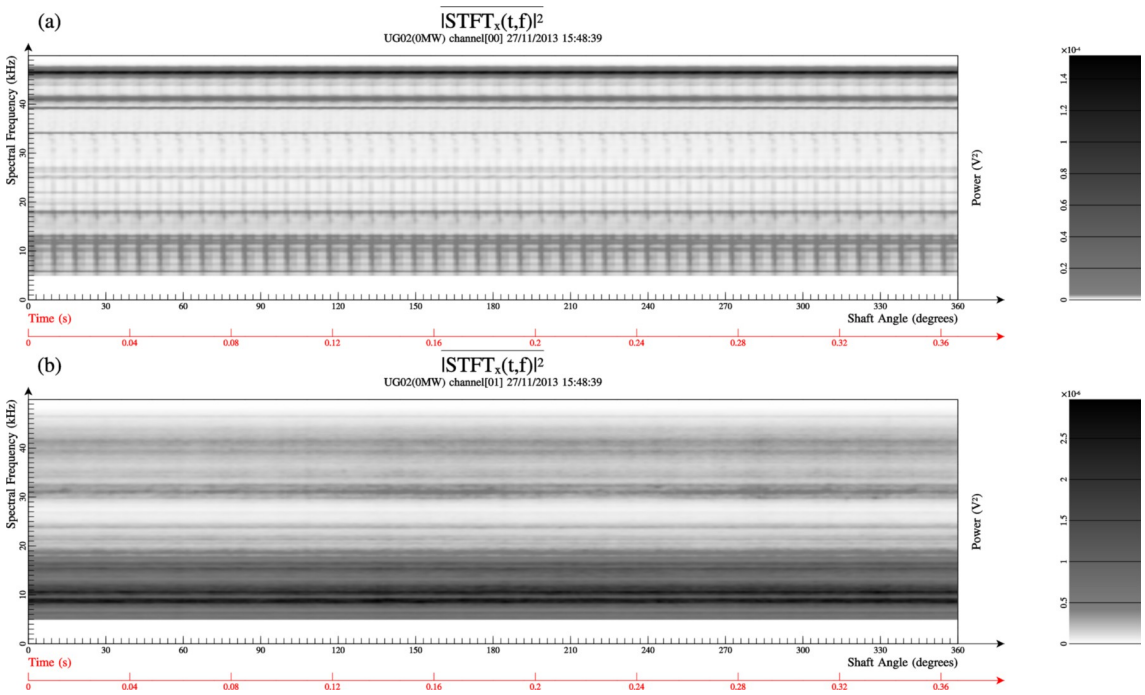


Figura 55 - Imagens das matrizes AIPS computadas no terceiro conjunto de sinais.

Fonte: Elaboração própria, *software spectrogram*.

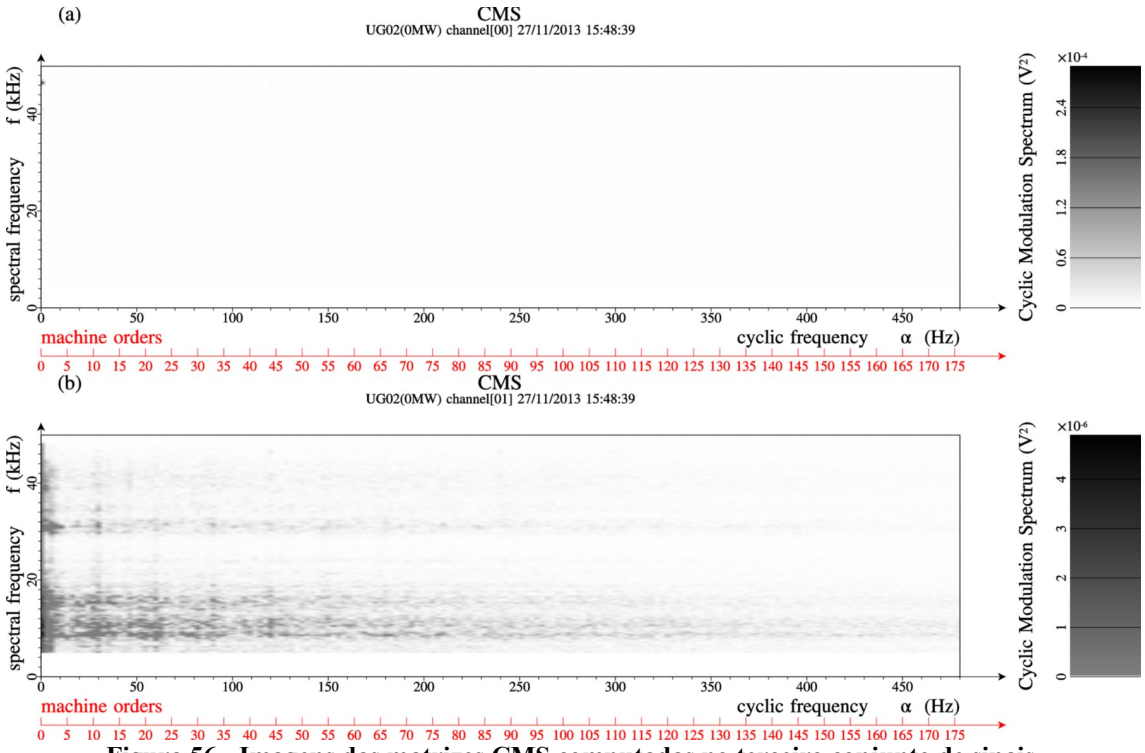


Figura 56 - Imagens das matrizes CMS computadas no terceiro conjunto de sinais.

Fonte: Elaboração própria, *software spectrogram*.

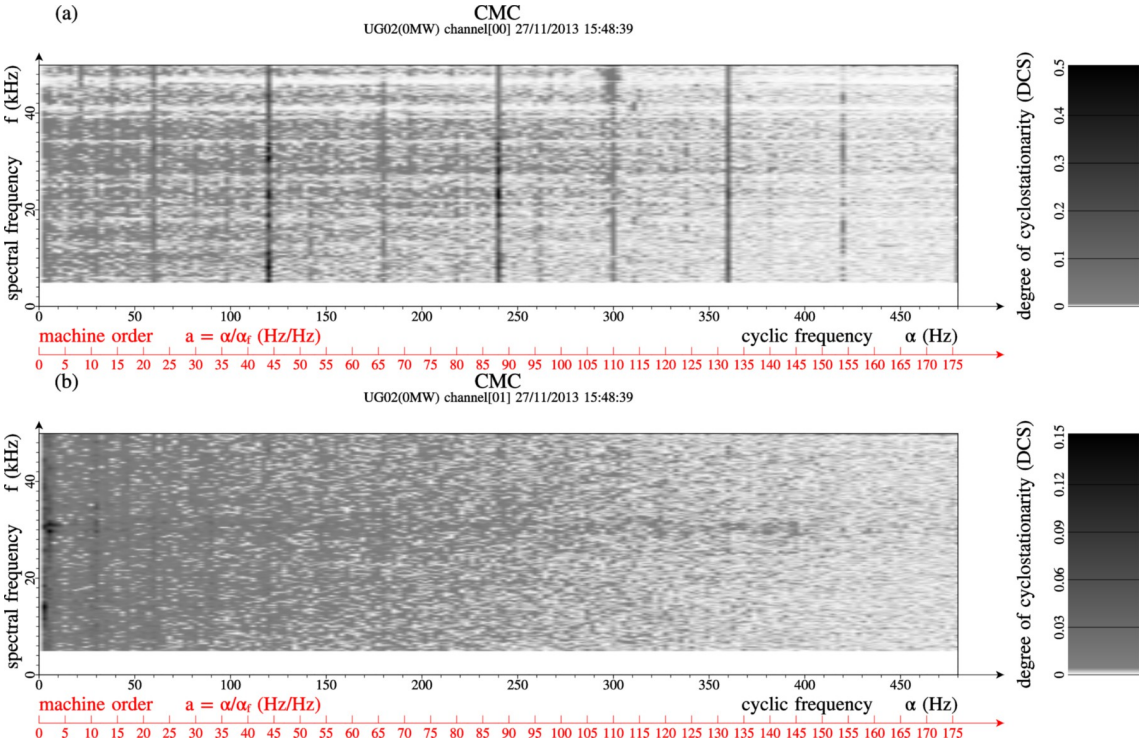


Figura 57 - Imagens das matrizes CMC computadas no terceiro conjunto de sinais.

Fonte: Elaboração própria, *software spectrogram*.

As linhas correspondentes da tabela 5 revelam para os sinais de ambos acelerômetros, valores muito mais baixos de P_B e P_C do que os obtidos nos sinais provenientes de UG01 (Estão pelo menos uma ordem de grandeza abaixo). A potência total entretanto, é a segunda mais alta, fazendo $P_{B\%}$ e $P_{C\%}$ cair significativamente (ambos são <0,1%). Conclui-se que, para o primeiro acelerômetro, tanto a imagem formada pela CMC quanto os estimadores computados na tabela 5, indicam ausência de cavitação em nuvem, em bolhas itinerantes e em vórtice de Von Kármán. Os sinais do segundo acelerômetro não apresentam componentes de interferência, nem componentes características de SVIs, e a CMC revela um sinal com baixo grau de cicloestacionariedade, o que corrobora o diagnóstico de ausência de cavitação durante a gravação.

5.3.3 Interferências encontradas nas duas turbinas

Ambas as turbinas de UG01 e UG02 apresentaram fortes componentes de ruído

nas ordens de máquina de número 22 e múltiplos, com especial atenção nas componentes de ordens de máquina múltiplas de 44, que são ainda mais intensas. Estas componentes são sem dúvida, sinais cicloestacionários de segunda ordem puros, pois se fossem de primeira ordem ou se fossem sinais estacionários, o processamento iria discriminá-los.

Esta interferência inclusive se sobrepõe a algumas componentes de ordem de máquina produzidas pela cavitação em bolhas itinerantes, tornando as suas estimativas de agressividade tendenciosas em algum grau. O fato desses sinais interferentes serem bem mais intensos que os SVIs das cavitações, e de ainda não ser possível eliminá-los por meio da metodologia proposta, despertou o interesse do autor, que investigou a natureza dessas interferências. A investigação começou levando em conta que 44 é exatamente o número de polos magnéticos dos geradores de UG01 e UG02, e portanto, as componentes de ruído são moduladas na frequência da rede elétrica (60 Hz ou ordem de máquina 22) e na *frequência de passagem dos polos magnéticos* (120 Hz ou ordem de máquina 44).

O *ruído Barkhausen* é um ruído magnético, e é produzido toda vez que um material ferromagnético é magnetizado ou desmagnetizado (figura 58), mas pode ser produzido também, quando o material é submetido a alguma tensão mecânica. Os materiais magnéticos dos geradores de UG01 e UG02 são compostos por *domínios magnéticos*, e a magnetização do material não é gradual e suave. Os domínios magnéticos se reorientam de acordo com o campo magnetizante aplicado em pequenos "saltos", até que o material esteja totalmente magnetizado, e portanto, saturado.

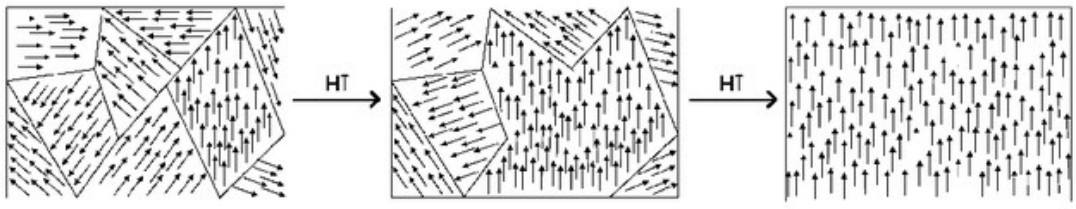


Figura 58 - Orientação dos domínios magnéticos (dipolos) em resposta a um campo externo magnetizante crescente.

Fonte: <http://en.wikipedia.org/wiki/Magnetic_domain> Acesso em 09/05/2015.

Se o material for colocado no interior de uma bobina, um pulso de tensão é

produzido a cada reorientação de um domínio magnético individual. Os pulsos de tensão são gerados aleatoriamente, e compõem um ruído com características estatísticas muito semelhantes ao ruído acústico produzido pela cavitação hidrodinâmica. Ao material magnético ser saturado, o ruído cessa, só sendo produzido quando o campo magnetizante inverte a polaridade, obrigando todos os domínios magnéticos a se reorientarem de forma desordenada novamente (figura 59). É possível notar que a potência do ruído Barkhausen é máxima em instantes que sucedem as transições de polaridade do campo magnetizante.

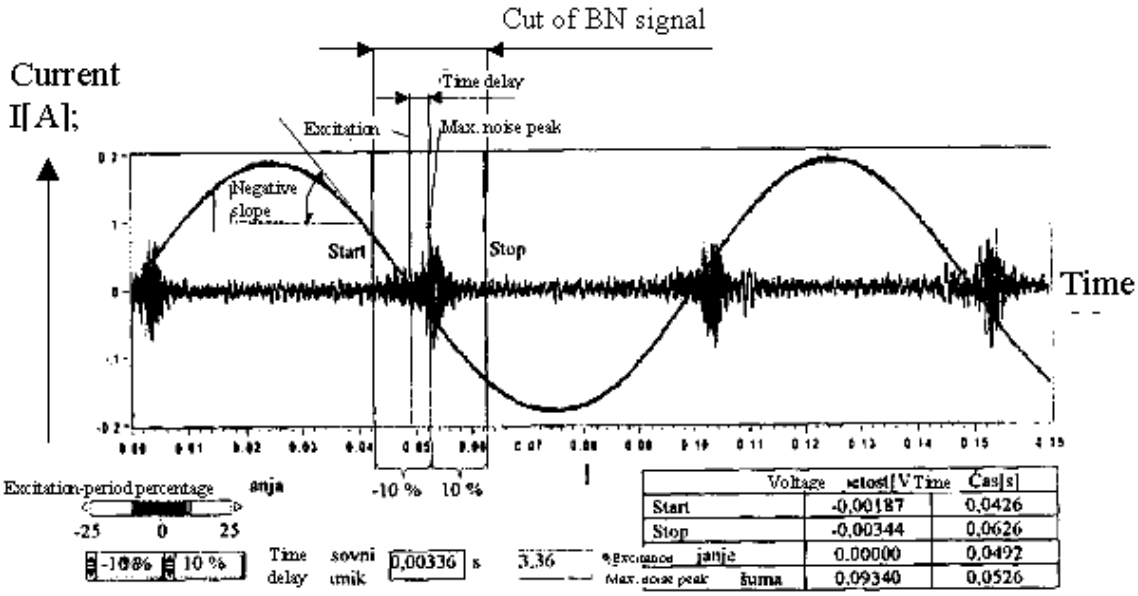


Figura 59 - Ruído Barkhausen produzido por transições na corrente que produz o campo magnetizante em uma amostra ferromagnética.

Fonte: (GRUM *et al.*, 2000)

A magnetização dos materiais magnéticos também causa um aumento ou diminuição das suas dimensões em alguns sentidos somente (fenômeno chamado *magnetostrição*). Esta variação nas dimensões é também devido a uma reorganização dos domínios magnéticos, que acontece de forma desorganizada. Desta forma, a variação de dimensões do corpo magnético não é suave, mas sujeita a degraus aleatórios, o que acaba por induzir ruído mecânico (AUGUSTYNIAK, VANNES, 1987).

Ambos os ruídos mecânico e elétrico podem interferir na gravação de sinais. O ruído mecânico produz interferência direta, produzindo vibrações aleatórias que

propagam do gerador através do eixo ao rotor da turbina e finalmente até os acelerômetros. O ruído elétrico pode ser visto nas gravações se for dada amplificação necessária na tensão alternada vinda do gerador, que foi usada como sinal de sincronismo. Entretanto, os sensores e condicionadores de sinais foram alimentados por baterias, para prevenir contaminações vindas da rede elétrica. Também, nota-se nos resultados que algumas posições em que os sensores estão instalados são muito mais suscetíveis à captação do ruído Barkhausen, o que caracteriza locais onde a transmissão mecânica da vibração interferente é mais favorável. Alternativamente, no terceiro conjunto de sinais há um sinal onde as interferências estão completamente ausentes, o que corrobora a hipótese do ruído Barkhausen se manifestar em forma de vibrações mecânicas e não elétricas ou magnéticas.

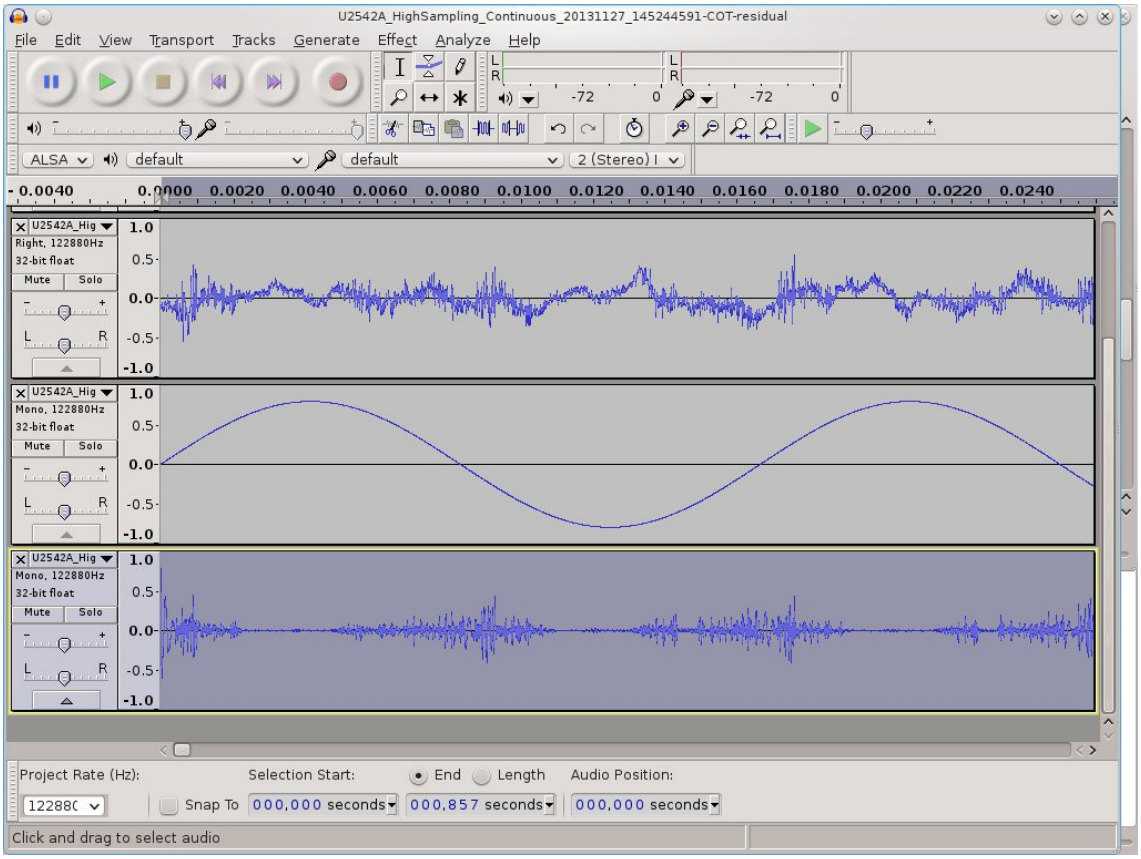


Figura 60 - Ruído Barkhausen no sinal vibracional resídual gravado na UG01 (acima) e no resíduo da tensão alternada do gerador (abaixo). O sinal do meio é uma senóide gerada matematicamente para fins de comparação da posição angular, e é igual à componente determinística do sinal de sincronismo. Os sinais não estão na mesma escala de amplitudes.

Fonte: Elaboração própria com *freeware* Audacity 2.0.5 Linux.

A distinção entre o ruído Barkhausen e os SVIs das cavitações só foi possível

pois, para as turbinas de UG01 e UG02, a cavitação existe em nível limitado ou incipiente, e portanto intermitente (produzindo sinais aleatórios *esporádicos*). Já o ruído Barkhausen *sempre* ocorre em determinadas posições angulares do eixo do rotor (figura 60). A visualização dos sinais residuais representados no domínio do tempo mostra a contaminação por ruído Barkhausen nos sinais vibracionais e também no sinal de sincronismo.

5.3.4 A acurácia da reamostragem angular

A literatura especializada em processamentos de sinais mecânicos, que está a disposição do autor, relata amiúde sobre a influência da reamostragem angular na qualidade dos resultados obtidos. Nenhuma amostragem angular direta é perfeita, e tampouco é a reamostragem angular. A turbina pode sofrer alterações na velocidade angular entre dois pontos de referência angulares para a reamostragem, que vai tentar interpolar de forma imperfeita a velocidade angular instantânea entre estes dois pontos. Ademais, a interpolação dos sinais vibracionais e de sincronismo com a função *sinc* introduz ruído adicional e uma leve redução na banda passante.

Idealmente, a reamostragem angular perfeita permitiria uma decomposição determinística/aleatória dos sinais perfeita. Nos sinais residuais não deveria haver componentes periódicas. Entretanto, o autor observou um fato curioso ao amplificar o sinal residual da tensão alternada do gerador, pois lá havia uma componente periódica, que apareceu como uma sucessão de picos agudos alternadamente positivos e negativos (sinal inferior na figura 61). Os picos positivos inicialmente coincidiam com as transições positivas de uma sinal senoidal de 60 Hz sintetizado, mas ao verificar outros intervalos da *mesma gravação*, o autor percebeu o oposto: os picos positivos do sinal resíduo coincidiam com as transições negativas do sinal senoidal sintético. O autor então procurou e de fato encontrou alguns pontos em que há uma mudança de fase abrupta nesta componente “residual aleatória”, conforme pode ser visto na figura 61, no sinal inferior. O pulso no centro inicia como se fosse um pico negativo, mas muda abruptamente para um pico positivo. O autor desconhece, entretanto, a causa destes picos alternados no sinal residual, mas assume que pode ser algo até mesmo exterior à

usina de geração de energia. A verdadeira causa obviamente, contribui com as imperfeições no sincronismo e reamostragem angular, e portanto afeta a qualidade dos resultados obtidos.

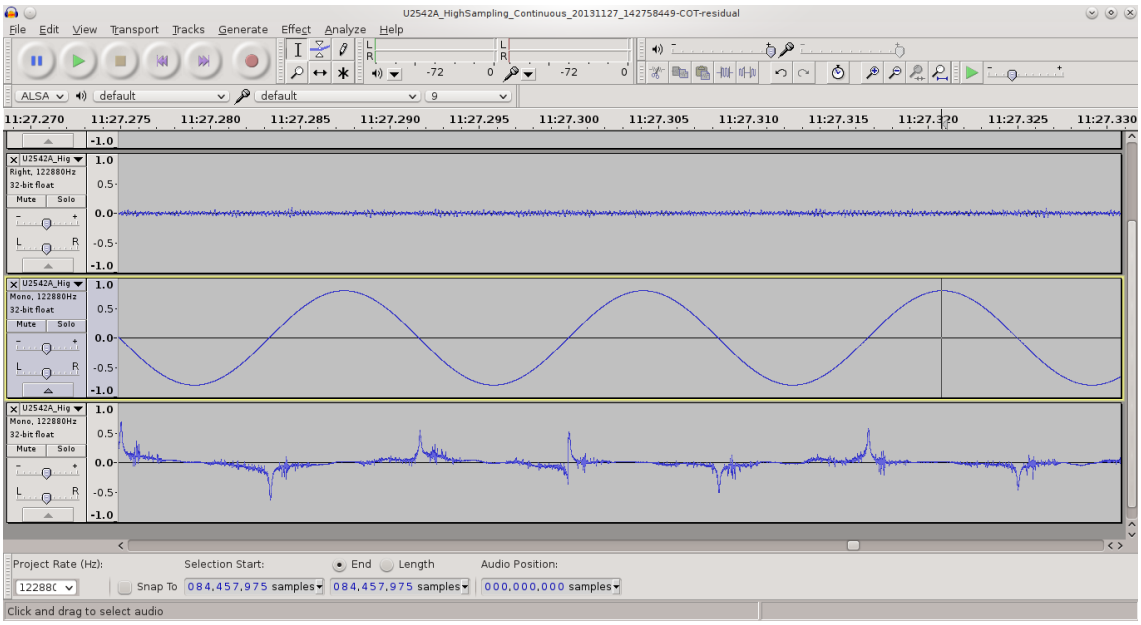


Figura 61 - Não idealidades da reamostragem angular no primeiro conjunto de sinais devido à instabilidades na tensão alternada do gerador.

Fonte: Elaboração própria com *freeware* Audacity 2.0.5 Linux.

5.3.5 Cavitação em vórtice no tubo de descarga

O autor empregou a metodologia desenvolvida, mas ao invés de computar a matriz CMS a partir da matriz AIPS, o cálculo foi feito a partir da matriz IPS (sem a média de milhares de revoluções). Isto foi feito como uma tentativa de detectar a cavitação em vórtice no tubo de descarga, já que a figura 36d mostra uma evidência da presença deste tipo de cavitação, já que o cubo do rotor estava sendo restaurado. Os resultados entretanto, não mostraram qualquer início deste tipo de cavitação, pois nenhuma componente cicloestacionária em ordem de máquina fracionária (segundo a literatura, de 0,25 a 0,40) foi detectada. Na figura 62 estão as imagens formadas por duas matrizes CMC computadas em sinais vibracionais provenientes de UG01 (letra a) e UG02 (letra b). A escala de ordens de máquina vai de zero até um, e há uma boa resolução em ordens de máquina neste intervalo (intervalos de ordem de máquina de milésimos). Apesar disso, nenhuma linha vertical, característica de sinais

cicloestacionários de segunda ordem, pode ser vista.

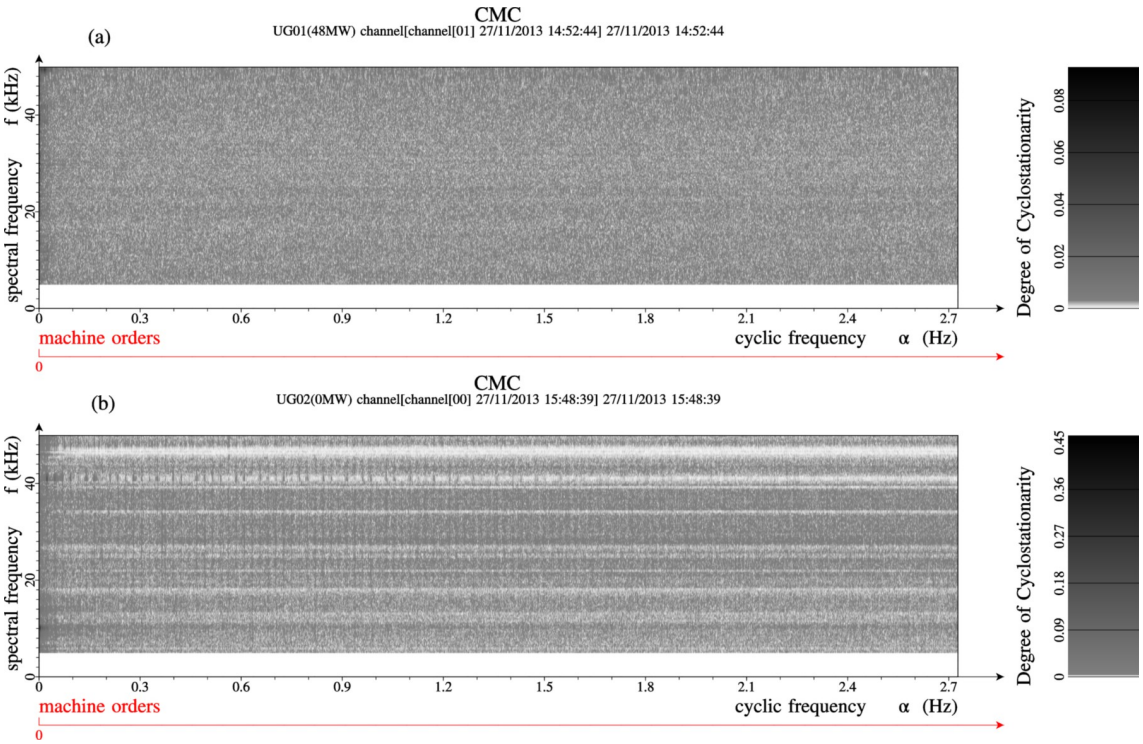


Figura 62 - Investigação da presença da cavitação em vórtice no tubo de descarga.

Fonte: Elaboração própria, *software spectrogram*.

Pode-se concluir que, apesar de o processamento não ter detectado a cavitação no tubo de descarga e haver evidências físicas que comprovam a presença da mesma, não significa que o processamento está incorreto, pois a cavitação no tubo de descarga pode ter existido e causado erosão em *intervalos anteriores à gravação*, nas 50.000 horas de operação sem manutenção corretiva. Ademais, o cubo do rotor é ligado à uma válvula de injeção de ar (o eixo da máquina é oco e funciona como tubulação para o ar) que apresentou defeitos em períodos anteriores ao da gravação. A injeção de ar é uma medida eficaz para mitigar os efeitos erosivos da cavitação no tubo de descarga (ARNDT *et al.*, 1995, 1993), e segundo relatos orais, "quando a válvula de injeção de ar está defeituosa, a turbina cavita *um pouco* mais".

6 CONCLUSÕES

A metodologia desenvolvida para a detecção, identificação e estimação da agressividade da cavitação consiste em modelar os sinais vibracionais induzidos pelos diferentes tipos de cavitação como *processos cicloestacionários com relação à posição angular* do eixo da turbina. A cicloestacionariedade foi encarada como uma característica vantajosa de fato (e não como um empecilho), e que possibilitou o emprego de ferramentas de análise cicloestacionárias. A escolha *default*, que é o emprego de ferramentas de análises de processos estacionários, se mostrou inapropriada. A teoria da cicloestacionariedade surgiu há mais de meio século, e inicialmente foi empregada para processamento de sinais de telecomunicações. Atualmente, esta teoria tem mostrado ser um valioso recurso para o processamento de sinais mecânicos provenientes de máquinas rotativas.

Apesar de as descrições sobre o fenômeno de cavitação erosiva em turbinas serem relativamente escassas, e em sua maioria fenomenológicas, o autor propôs um modelo simplificado para os sinais *vibracionais* possíveis de serem captados externamente com acelerômetros e sensores de emissão acústica. Não foram encontrados modelos na literatura descrevendo os sinais vibracionais produzidos pela cavitação em turbinas, mas foi encontrado um modelo semelhante, para sinais *acústicos* produzidos por hélices de barcos de superfície devido à cavitação (que ocorre em nível desenvolvido, e não intermitente como nas turbinas). O mecanismo que produz a modulação destes sinais acústicos por sua vez, nada tem a haver com a RSI da máquina, afinal, um barco não é uma máquina hidráulica (existe interação entre a hélice e o casco, mas esta também não é a causadora da modulação neste caso), e portanto não possui rotor nem estator.

O modelo simplificado para os SVI da cavitação foi implementado em *software*, dando origem a um sintetizador de sinais, e estes sinais foram usados na validação com sucesso da metodologia proposta para detecção e estimação da agressividade da cavitação. Não obstante, no momento da conclusão desta obra o modelo simplificado pode ser considerável elegível a um aprimoramento, principalmente no que se refere à produção de SVIs que sejam *intermitentes*. O autor neste momento trabalha no sentido deste aprimoramento. A utilização do modelo simplificado, entretanto, não invalida a metodologia, pois ao computar a matriz AIPS de inúmeras revoluções do eixo, ambos

sinais (intermitentes ou não) acabam convergindo para resultados similares.

6.1 ASIC VERSUS DSP

O processamento dos SVIs propostos de forma analógica em um ASIC não é tecnicamente viável. Não obstante, o autor desenvolveu uma metodologia que pode ser implementada em *software.* Um *hardware,* composto por um microcomputador de baixo consumo e uma placa de aquisição USB, foi desenvolvido especialmente para executar o *software* de forma contínua no chão de fábrica, e produzir informações que servem para: construir históricos de estimadores computados em função de perdas de massa *reais,* checar a eficácia de uma manutenção corretiva da erosão baseando-se nos estimadores computados antes e depois, detectar condições de operação em que a turbina está mais sujeita a cavitação erosiva, investigar causas da cavitação erosiva, meramente gravar sinais para estudos posteriores, etc.

Os históricos de estimadores da agressividade da cavitação, que podem ser construídos com a metodologia proposta, podem constituir valioso recurso para a investigação da perda de massa, e servir de base para a construção de um estimador da quantidade de massa perdida em função da agressividade da cavitação. Desta forma, pode-se planejar as paradas de manutenção das unidades geradoras de modo otimizado.

6.2 APRIMORAMENTO DO PROCESSAMENTO DEMON

Atualmente, a metodologia que é mais amplamente empregada para a detecção e identificação da cavitação erosiva em turbinas é o processamento DEMON, que na literatura aparece com algumas variações não essenciais (variações no método de detecção de envoltória, na filtragem dos sinais, etc). O processamento DEMON inclusive é empregado para detecção e classificação de barcos, através da *assinatura acústica* produzida pela cavitação de suas hélices. O custo computacional do processamento DEMON, quando implementado em *software* é muito mais baixo que o custo da metodologia proposta. O processamento DEMON também pode ser implementado de forma analógica (com limitações adicionais), em um ASIC.

Entretanto, o processamento DEMON, além de ser uma metodologia empírica, não serve para a estimação da agressividade da cavitação em turbinas hidráulicas. As principais desvantagens do processamento DEMON são: a filtragem passa faixa dos sinais, constituindo grande perda de informações sobre a cavitação, pois os SVIs ocupam invariavelmente faixas largas de frequência espectral; a necessidade do conhecimento prévio da faixa de frequências espectrais que o(s) SVI(s) se sobressai(em) ou a sintonia do filtro passa faixas por um operador humano; e a não distinção de componentes cicloestacionárias geradas pelo mesmo fenômeno através da similaridade espectral. A figura 63 mostra, a título de ilustração, o cálculo da distribuição da potência em componentes cíclicas (ou de ordem de máquina), ou C*yclic Power Spectrum* (CPS).

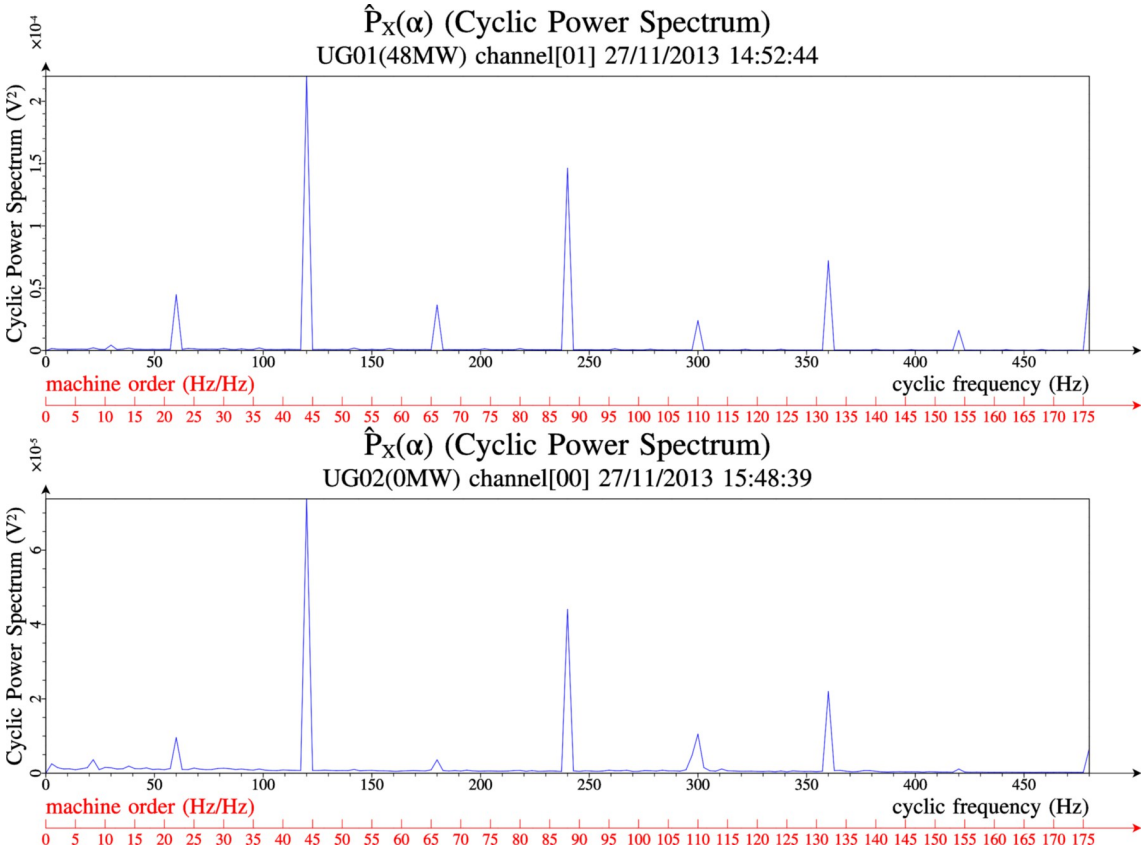


Figura 63 - Resultados do cáculo da distribuição de potência em componentes cíclicas (de ordem de máquina) para os sinais de UG01 e UG02.

Fonte: Elaboração própria, *software spectrogram*.

Os gráficos da figura 63 foram elaborados a partir das matrizes CMS correspondentes, através da acumulação das potências em cada coluna (ordem de máquina). Este procedimento é similar ao processamento DEMON no que se refere à

incapacidade de distinguir quais componentes de ordem de máquina são produzidas por cada mecanismo de cavitação (ou mecanismo interferente). Não há de se falar em similaridade espectral, pois não há informação sobre a composição em frequências espectrais dos sinais, havendo somente informações sobre a composição em frequências cíclicas. Não obstante, este procedimento não é empírico como o processamento DEMON. Nota-se que não há grande diferença entre os resultados obtidos em UG01 (com cavitação) e UG02 (sem cavitação). Ademais, os picos de potência que são evidentes, na verdade, são produzidos pelo ruído Barkhausen e pela magnetostrição, mas podem ser erroneamente interpretados como componentes produzidas pela cavitação. Mais detalhes são discutidos na secção 6.4.

A metodologia proposta a partir da modelagem cicloestacionária é um aprimoramento do processamento DEMON, e que não é empírico, mas baseado em tratamentos estatísticos dos SVIs. As desvantagens inerentes do processamento DEMON são eliminadas com a metodologia proposta, permitindo no futuro a implementação de sistemas de monitoramento bem mais elaborados e totalmente automáticos. Um operador humano não é necessário para sintonizar filtros de seleção de bandas, pois este nem existe na metodologia proposta, o que significa que não há descarte de componentes importantes dos SVIs, e também que os padrões de sinais vibracionais produzidos podem variar (como é muito comum) sem a necessidade de ressintonizar o filtro.

Uma das vantagens mais evidentes de se empregar a detecção e identificação baseada na CMC computada a partir do AIPS é a forte rejeição de ruídos estacionários e tonais. As turbinas que forneceram sinais para este trabalho estão entre as turbinas que menos sofrem danos com a cavitação erosiva no país (segundo uma lista que o autor dispõe, estas turbinas são as que menos sofrem com cavitação, mas a lista não é completa). O processamento DEMON por outro lado, além de não fornecer estimativas da agressividade da cavitação, é muito mais sensível a ruídos tonais e estacionários.

6.3 COMPARAÇÃO COM A ANÁLISE VIBROACÚSTICA MULTIDIMENSIONAL

A outra metodologia empregada para diagnosticar cavitação erosiva em turbinas foi proposta por B. Bajic, e é empregada comercialmente pela empresa *Korto Cavitation Services* (http://www.korto.com), especializada em inspeção de cavitação erosiva em turbinas. A análise multidimensional vibroacústica consegue fornecer resultados muito detalhados dos padrões de cavitação encontrados em uma turbina, como por exemplo, que a cavitação de determinado tipo X só ocorre quando há interação entre a lâmina de número Y e a palheta de número Z. Esta metodologia consegue detectar a cavitação em níveis incipientes, muito antes que haja perda de massa. Trata-se de uma metodologia de altíssimo custo, e que envolve dezenas de sensores de emissão acústica, também relativamente caros. Ademais, a turbina testada deve necessariamente ser retirada da operação normal, e aquisições de dados que duram até 24 horas (com a velocidade angular da turbina sendo *controlada* e a potência gerada variada de 0 a 100% da nominal) devem ser feitas, para processamento *posterior*.

A análise multidimensional vibroacústica é incomparável à metodologia desenvolvida neste trabalho. De fato, os resultados obtidos com a primeira são incomparavelmente melhores e mais detalhados que os obtidos com a metodologia deste trabalho. Entretanto, a análise vibroacústica não é apropriada para o monitoramento *on-line* e não invasivo de turbinas, haja vista que é necessário retirar a turbina de operação por até 24 horas.

A metodologia desenvolvida evolui no sentido de ser de baixo custo de implementação, de operação bem mais simples que a análise multidimensional (eventualmente dispensando mão-de-obra muito especializada), e inerentemente não invasiva e que não interfere com o funcionamento da unidade geradora.

Obviamente, a metodologia desenvolvida, assim como a análise multidimensional vibroacústica, tem como objetivo processar o SVI na sua íntegra, sem filtragens de seleção de banda como no processamento DEMON. A análise multidimensional vibroacústica é empregada com sensores de emissão acústica, que captam sinais em uma faixa de frequências que vai de 0 a 1 MHz. O autor não dispõe de

sensores de emissão acústica, dispondo somente de dois acelerômetros de alta frequência, o que limitou severamente a banda passante dos sinais captados. Nada impede, entretanto, que a metodologia seja empregada com sensores mais apropriados, como os de emissão acústica.

6.4 LIMITAÇÕES DA METODOLOGIA DESENVOLVIDA

Toda metodologia tem suas limitações, e a metodologia desenvolvida neste trabalho não é exceção. As limitações começam com imperfeições na reamostragem angular que afetam diretamente a qualidade dos resultados; incluem o número de janelas de análise usadas a cada revolução do eixo e a fração de *overlap*, que afeta o ruído de estimação dos elementos da matriz CMS e portanto CMC também; o número de revoluções gravadas também tem influência nos resultados, etc. Todas essas limitações citadas podem ser, em maior ou menor grau, minimizadas através de aperfeiçoamentos futuros e aumento no poder computacional.

A maior limitação desta metodologia, entretanto, está na impossibilidade de estimar a potência total de um SVI cicloestacionário em presença de sinais interferentes também cicloestacionários com pelo menos uma componente de ordem de máquina de mesmo número que o SVI. Neste caso, há uma sobreposição de componentes (como ocorrido com o ruído Barkhausen, em uma infeliz coincidência nas turbinas reais de teste, onde o número de polos magnéticos é um múltiplo do número de lâminas do rotor), e parte do SVI fica mascarada pela interferência, tornando a estimação tendenciosa. Felizmente, a metodologia permite detectar esta sobreposição, e com isso o sinal interferente não é processado como se fosse um SVI. Por outro lado, o processamento DEMON não permite a detecção desta sobreposição, e por isso, com ele sozinho é impossível distinguir um SVI de um sinal interferente cicloestacionário na mesma ordem de máquina. Curiosamente, foi observado em um artigo científico (ESCALER *et al.*, 2006a), referência na literatura que trata da detecção de cavitação em turbinas, que os autores empregaram dois protótipos, nomeadamente *Francis Prototype I e II,* em que as turbinas têm o número de palhetas no distribuidor v e o número de polos magnéticos do gerador P iguais (P=v=24). Apesar de os autores terem detectado e

identificado a cavitação em nuvem em uma das turbinas protótipo com o processamento DEMON, esta situação peculiar constitui um risco aos resultados da detecção pelo processamento DEMON. A detecção pela metodologia proposta neste trabalho não ficaria comprometida, somente estimação da agressividade da cavitação seria afetada.

REFERÊNCIAS BIBLIOGRÁFICAS

AITBOUZIAD, Y. **Physical Modelling of Leading Edge Cavitation : Computational Methodologies and Application To Hydraulic Machinery**. Lausanne, Switzerland: École Polytechnique Fédérale de Lausanne, 2005.

ANDRE, H., DAHER, Z., ANTONI, J. *et al.* "Comparison between angular sampling and angular re- sampling methods applied on the vibration monitoring of a gear meshing in non stationary conditions " **In: anais do congresso Proceedings of ISMA2010 Including USD2010** 2010

ANTONI, J. "Cyclostationarity by examples." **Mechanical Systems and Signal Processing**, v. 23, n. 4, p. 987–1036, maio 2009.

ANTONI, J., ELTABACH, M. "A KIS solution for high fidelity interpolation and resampling of signals." **Mechanical Systems and Signal Processing**, v. 35, n. 1-2, p. 127–136, fev. 2013.

ANTONI, J., HANSON, D. "Detection of Propeller Noise under Low SNR with the Cyclic Modulation Coherence (CMC)" **In: anais do congresso 10ème Congrès Français d'Acoustique** Lyon, France: Société Française d'Acoustique, 2010

ANTONI, J., HANSON, D. "Detection of Surface Ships From Interception of Cyclostationary Signature With the Cyclic Modulation Coherence." **IEEE Journal of Oceanic Engineering**, v. 37, n. 3, p. 478–493, jul. 2012.

ARNDT, R. E. A., ELLIS, C. R., PAUL, S. "Preliminary Investigation of the Use of Air Injection to Mitigate Cavitation Erosion." **Journal of Fluids Engineering**, v. 117, n. 3, p. 498, 1995.

ARNDT, R., PAUL, S., ELLIS, C. "Investigation of the Use of Air Injection to Mitigate Cavitation Erosion." n. 343, 1993.

AUGUSTYNIAK, B., VANNES, A. "Relation Between Mechanical Barkhausen Noise and Magnetomechanical Internal Friction." **Le Journal de Physique Colloques**, v. 48, n. C8, p. C8–407–C8–412, dez. 1987.

AUSONI, P., FARHAT, M., ESCALER, X. *et al.* "Cavitation Influence on von Kármán Vortex Shedding and Induced Hydrofoil Vibrations." **Journal of Fluids**

Engineering, v. 129, n. 8, p. 966, 2007.

BAJIC, B. “Multidimensional Diagnostics of Turbine Cavitation.” **Journal of Fluids Engineering**, v. 124, n. 4, p. 943, 2002.

BAJIC, B. “Methods for vibro-acoustic diagnostics of turbine cavitation.” **Journal of Hydraulic Research**, v. 41, n. 1, p. 87–96, jan. 2003.

BALDASSARRE, A., DE LUCIA, M., NESI, P. “Real-Time Detection of Cavitation for Hydraulic Turbomachines.” **Real-Time Imaging**, v. 4, n. 6, p. 403–416, dez. 1998.

BERCHICHE, N. “A cavitation erosion model for ductile materials.” **Journal of Fluids Engineering**, v. 124, n. 3, p. 601, 2002.

BORGHESANI, P., PENNACCHI, P., RANDALL, R. B. *et al.* “Order tracking for discrete-random separation in variable speed conditions.” **Mechanical Systems and Signal Processing**, v. 30, p. 1–22, jul. 2012.

BOURDON, A., ANDRÉ, H., RÉMOND, D. “Introducing angularly periodic disturbances in dynamic models of rotating systems under non-stationary conditions.” **Mechanical Systems and Signal Processing**, v. 44, n. 1-2, p. 60–71, fev. 2014.

BRENNEN, C. E. **Cavitation and Bubble Dynamics**. New York: Oxford University Press, 1995.

CALAINHO, J., HORTA, C. “Cavitação em turbinas hidráulicas do tipo Francis e Kaplan no Brasil” **In: anais do congresso Anais do XV Seminário Nacional de Produção e Transmissão de Energia Elétrica** Foz do Iguaçu - Parana - Brazil: 1999

CAPDESSUS, C., ANTONI, J. “Speed Transform, a New Time-Varying Frequency Analysis Technique.” **Advances in Condition Monitoring of Machinery in Non-Stationary Operations**, p. 1–14, 2014.

ČARIJA, Z., MRŠA, Z., FUĆAK, S. “Validation of Francis water turbine CFD simulations.” **Strojarstvo**, v. 50, n. 1, p. 5–14, 2008.

CARLTON, J. **Marine propellers and propulsion**. 2nd. ed. Burlington, USA: Elsevier Inc., 2007.

CHAHINE, G. L. “Nuclei effects on cavitation inception and noise” **In: anais**

do congresso 25 th Symposium on Naval Hydrodynamics 2004

CHEN, P., TANIGUCHI, M., TOYOTA, T. *et al.* “Fault diagnosis method for machinery in unsteady operating condition by instantaneous power spectrum and genetic programming.” **Mechanical Systems and Signal Processing**, v. 19, n. 1, p. 175–194, 2005.

CHOI, J., JAYAPRAKASH, A., CHAHINE, G. “Scaling of cavitation erosion progression with cavitation intensity and cavitation source.” **Wear**, v. 278, n. March, p. 53–61, 2012.

CLINE, R., GERMANN, J., MCSTRAW, B. **Facilities Instructions, Standards, and Techniques.** Denver, Colorado: [s.n.].

D’ELIA, G., DELVECCHIO, S. “Combining blind separation and cyclostationary techniques for monitoring distributed wear in gearbox rolling bearings” **In: anais do congresso Proc. of Surveillance Methods and Diagnostic Techniques** Compiègne, France: 2011

DE, M. K., HAMMITT, F. G. “New Method for Monitoring and Correlating Cavitation Noise to Erosion Capability.” **Journal of Fluids Engineering**, v. 104, n. 4, p. 434, 1982.

DRTINA, P., SALLABERGER, M. “Hydraulic turbines—basic principles and state-of-the-art computational fluid dynamics applications” **In: anais do congresso Proceedings of the Institution of Mechanical Engineers** 1999

EL BADAOUI, M., BONNARDOT, F. “Impact of the non-uniform angular sampling on mechanical signals.” **Mechanical Systems and Signal Processing**, v. 44, n. 1-2, p. 199–210, fev. 2014.

ELOY, C. “Optimal Strouhal number for swimming animals.” **Journal of Fluids and Structures**, v. 30, p. 205–218, abr. 2012.

ESCALER, X., EGUSQUIZA, E., FARHAT, M. *et al.* “Cavitation erosion prediction in hydro turbines from onboard vibrations” **In: anais do congresso 22nd IAHR Symposium on Hydraulic Machinery and Systems** Stockholm, Swenden: 2004

ESCALER, X., EGUSQUIZA, E., FARHAT, M. *et al.* “Detection of cavitation in hydraulic turbines.” **Mechanical Systems and Signal Processing**, v. 20, n. 4, p. 983–1007, maio 2006a.

ESCALER, X., EKANGER, J. V., FRANCKE, H. H. *et al.* “Detection of Draft Tube Surge and Erosive Blade Cavitation in a Full-Scale Francis Turbine.” **Journal of Fluids Engineering**, v. 137, n. 1, p. 011103, 10 set. 2014.

ESCALER, X., FARHAT, M., AUSONI, P. *et al.* “Cavitation monitoring of hydroturbines: tests in a francis turbine model” **In: anais do congresso 6th International Symposium on Cavitation** Wageningen, Netherlands: 2006b

FARHAT, M., BOURDON, P. “Improving hydro turbine profitability by monitoring cavitation aggressiveness” **In: anais do congresso CEA Electricity ’99 Conference and Exposition** Vancouver, Canada: 1999

FRANC, J.-P., MICHEL, J.-M. **Fundamentals of cavitation**. Dordrecht: Kluwer Academic Publishers, 2004. v. 76

GARDNER, W. “The spectral correlation theory of cyclostationary time-series.” **Signal processing**, v. 11, p. 13–36, 1986.

GENTIL, V. **Corrosão**. Rio de Janeiro: LTC, 1982.

GRUM, J., PECNIK, B., FEFER, D. *et al.* “Barkhausen noise at deep drawing steel after various degrees of cold plastic deformation” **In: anais do congresso 15th World conference on non-destructive testing** Rome: 2000

GUBRAN, A. A., SINHA, J. K. “Shaft instantaneous angular speed for blade vibration in rotating machine.” **Mechanical Systems and Signal Processing**, v. 44, n. 1-2, p. 47–59, fev. 2014.

HALLANDER, J. **Influence of acoustic interaction between cavities that generate cavitation noise**. Göteborg, Sweden: Chalmers University of Technology, 2002.

HAMMITT, F. G., DE, M. K. “Cavitation damage prediction.” **Wear**, v. 52, n. 2, p. 243–262, fev. 1979.

HEINZEL, G. “Spectrum and spectral density estimation by the Discrete Fourier

transform (DFT), including a comprehensive list of window functions and some new flat-top windows .” p. 1–84, 2002.

KECK, H., SICK, M. “Thirty years of numerical flow simulation in hydraulic turbomachines.” **Acta Mechanica**, v. 201, n. 1-4, p. 211–229, 2 set. 2008.

KHURANA, S., NAVTEJ, SINGH, H. “Effect of cavitation on hydraulic turbines- A review.” **International Journal of Current Engineering and Technology**, v. 2, n. 1, p. 172–177, 2012.

KIM, J.-H., SHIN, H.-C. “Application of the ALE technique for underwater explosion analysis of a submarine liquefied oxygen tank.” **Ocean Engineering**, v. 35, n. 8-9, p. 812–822, jun. 2008.

KNAPP, R. T., DAILY, J. W., HAMMITT, F. G. **Cavitation**. 1st. ed. New York: McGraw-Hill, 1970.

KUMAR, S., BRENNEN, C. “Harmonic cascading in bubble clouds.” 1992.

KUMAR, S., BRENNEN, C. E. “Some nonlinear interactive effects in bubbly clouds.” **Journal of Fluid Mechanics**, v. 253, n. -1, p. 565, 26 abr. 2006.

LUCKY, M. **Global Hydropower Installed Capacity and Use Increase | Vital Signs Online**. Disponível em: <http://vitalsigns.worldwatch.org/vs-trend/global-hydropower-installed-capacity-and-use-increase>. Acesso em: 25 fev. 2013.

MØRCH, K. A. “Cavitation Nuclei: Experiments and Theory.” **Journal of Hydrodynamics, Ser. B**, v. 21, n. 2, p. 176–189, abr. 2009.

MOROZOV, V. P. “Cavitation noise as a train of sound pulses generated at random times.” **SOVIET PHYSICS ACOUSTICS**, n. 14, p. 361–365, 1969.

NAMKUNG, M., YOST, W. T., UTRATA, D. *et al.* “Effects of microstructure of ferromagnetic alloys on magnetoacoustic emission.” **IEEE 1988 Ultrasonics Symposium Proceedings.**, p. 1013–1016, 1988.

NENNEMANN, B. “Kaplan turbine blade and discharge ring cavitation prediction using unsteady CFD” **In: anais do congresso 2nd IAHR International Meeting of the Workgroup on Cavitation and Dynamic Problems in Hydraulic Machinery and Systems** Timisoara, Romania: 2007

NICOLET, C., RUCHONNET, N., ALLIGNÉ, S. *et al.* “Hydroacoustic simulation of rotor-stator interaction in resonance conditions in Francis pump-turbine.” **IOP Conference Series: Earth and Environmental Science**, v. 12, p. 012005, 1 ago. 2010.

PETKOVŠEK, M., DULAR, M. “Simultaneous observation of cavitation structures and cavitation erosion.” **Wear**, v. 300, n. 1-2, p. 55–64, mar. 2013.

QIAN, Z., YANG, J., HUAI, W. “Numerical simulation and analysis of pressure pulsation in francis hydraulic turbine with air admission.” **Journal of Hydrodynamics, Ser. B**, v. 19, n. 4, p. 467–472, ago. 2007.

RAABE, J. **Hydro power: the design, use, and function of hydromechanical, hydraulic, and electrical equipment**. [s.l.] VDI-Verlag, 1985.

REISMAN, G. E., WANG, Y.-C., BRENNEN, C. E. “Observations of shock waves in cloud cavitation.” **Journal of Fluid Mechanics**, v. 355, n. June, p. 255–283, 25 jan. 1998.

RUCHONNET, N., NICOLET, C., AVELLAN, F. “Hydroacoustic Modeling of Rotor Stator Interaction in Francis Pump-Turbine” **In: anais do congresso IAHR Int. Meeting of WG on Cavitation and Dynamic Problems in Hydraulic Machinery and Systems** Barcelona: 2006a

RUCHONNET, N., NICOLET, C., AVELLAN, F. “One-Dimensional Modeling of Rotor Stator Interaction in Francis Pump-Turbine” **In: anais do congresso 23rd IAHR Symposium on Cavitation and Dynamic Problems in Hydraulic Machinery and Systems** Yokohama - Japan: 2006b

SMITH, J. O., GOSSETT, P. “A flexible sampling-rate conversion method” **In: anais do congresso ICASSP ’84. IEEE International Conference on Acoustics, Speech, and Signal Processing** Institute of Electrical and Electronics Engineers, 1984

STINEBRING, D. R., BILLET, M. L., LINDAU, J. W. *et al.* **DEVELOPED CAVITATION- CAVITY DYNAMICS**. [s.l.] Ft. Belvoir Defense Technical Information Center, 2001.

URBANEK, J., BARSZCZ, T., ANTONI, J. “A two-step procedure for estimation of instantaneous rotational speed with large fluctuations.” **Mechanical**

Systems and Signal Processing, v. 38, n. 1, p. 96–102, jul. 2013.

VIVIER, L. **Turbines hydrauliques et leur régulation, théorie, construction, utilisation**. Paris: A. Michel, 1966.

VOKURKA, K. "Power spectrum of the periodic group pulse process." **Kybernetika**, v. 16, n. 5, p. 462–471, 1980.

VOKURKA, K. "Cavitation noise modeling and analyzing" **In: anais do congresso Forum Acusticum Sevilla 2002** Sevilla, Spain: 2002

VOKURKA, K., BUOGO, S. "A minimum hydrophone bandwidth for undistorted cavitation noise measurement." **The 76th Acoustic seminar**, p. 99–108, 2008.

WILCZYŃSKI, L. "Stochastic Modeling of Cavitation Erosion Based on the Concept of Resonant Energy Absorption" **In: anais do congresso Fifth International Symposium on Cavitation** Osaka, Japão: 2003

WILLIAM DUNCAN, J. **Facilities Instructions, Standards, and Techniques.** Denver, Colorado: [s.n.].

ZHANG, Y. J., LI, S. C., HAMMITT, F. G. "Statistical investigation of bubble collapse and cavitation erosion effect." **Wear**, v. 133, n. 2, p. 257–265, out. 1989.

ZOBEIRI, A. **Investigations of time dependent flow phenomena in a turbine and a pump-turbine of Francis type: rotor-stator interactions and precessing vortex rope**. [s.l.] ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2009.

ZOBEIRI, A., KUENY, J., FARHAT, M. *et al.* "Pump-turbine Rotor-Stator Interactions in Generating Mode : Pressure Fluctuation in Distributor Channel" **In: anais do congresso 23rd IAHR Symposium** Yokohama - Japan: Ecole polytechnique fédérale de Lausanne Switzerland, 2006

GLOSSÁRIO

Bordo de ataque – exprime o ponto por onde o fluido se encontra com uma superfície sólida, e o escoamento se separa em duas partes, uma parte fluindo pela face de pressão, e a outra pela face de sucção.

Bordo de fuga – exprime o ponto por onde o fluido se separa de uma superfície sólida, e o escoamento que foi separado pelo bordo de ataque se une novamente.

Deslocação – é um defeito ou irregularidade na estrutura cristalina. A presença de deslocações, e a mobilidade das mesmas, influenciam fortemente muitas propriedades dos materiais.

Emissão acústica – é um método de ensaio não destrutivo, baseado na detecção de ondas transientes geradas pelo processo de degradação do material, quando submetido a tensões mecânicas.

Encruamento – é o fenômeno modificador da estrutura cristalina dos metais e ligas, em que a deformação plástica é realizada abaixo da temperatura de recristalização, e causa o endurecimento e consequente resistência do metal. Também denominado trabalho a frio.

Erro de Medição Relativo – Quociente entre o *erro da medição* e o valor verdadeiro da mensuranda. O erro de medição é a diferença algébrica entre o resultado de uma medição e o valor verdadeiro da mensuranda. Uma vez que o valor verdadeiro não pode ser determinado, utiliza-se, na prática, um valor verdadeiro convencional.

Esteira de turbulência – é a perturbação turbulenta deixada por um corpo sólido em um escoamento, à sua jusante.

Fadiga de contato – é um fenômeno a partir do qual uma trinca será nucleada e se propagará pelo material até o desprendimento de uma lasca da superfície, devido a uma interação entre uma inclusão na microestrutura e as tensões de cisalhamento provocadas pelo contato com um corpo.

Jusante – em hidráulica, é todo ponto ou seção de escoamento compreendido entre o ponto referencial e a foz de um curso d'água. Oriundo do latim *jusum*, é um substantivo feminino que também significa "para o lado da foz" ou "água abaixo".

Onda de choque – é uma onda caracterizada por ser um distúrbio em propagação onde propriedades como velocidade, pressão, temperatura ou densidade variam de maneira abrupta e quase descontínua.

Piezoeletricidade – é a capacidade de alguns cristais gerarem tensão elétrica por resposta a uma pressão mecânica.

Produtos de intermodulação - ou distorção de intermodulação (IMD) é a modulação em amplitude de sinais que contêm duas ou mais frequências diferentes, num sistema com não linearidades.

Sistema Interligado Nacional (SIN) – O Sistema Interligado Nacional é formado pelas empresas de geração e transmissão das regiões Sul, Sudeste, Centro-Oeste, Nordeste e parte da região Norte.

Valor Verdadeiro Convencional (VVC) – Valor atribuído a uma grandeza por um acordo, para um dado propósito. Geralmente considera-se que um valor convencional está associado a uma incerteza de medição convenientemente baixa, que pode ser nula.

Vórtice – escoamento giratório onde as linhas de corrente apresentam um padrão circular ou espiral. São movimentos espirais ao redor de um centro de rotação.

APÊNDICE A – CAVITAÇÃO

Diferentes níveis de cavitação

Basicamente, a definição *simplificada* de cavitação é baseada na pressão de vapor da água, de modo que se a pressão cai abaixo da mesma, há a formação de cavidades cheias de vapor e inevitavelmente implosões posteriores. Baixas pressões e altas velocidades de escoamento favorecem o aparecimento da cavitação. A tendência que um escoamento tem de produzir cavidades de vapor é indicada pelo *número de cavitação* σ, definido na equação 2. Quanto menor o valor de σ, maior a tendência do escoamento cavitar. Na situação limítrofe entre o escoamento não cavitante e cavitante, o número de cavitação é σ_i, indicando que a cavitação está em nível *incipiente*.

Para valores de σ um pouco menores que σ_i, o número de cavidades produzidas aumenta, mas a cavitação é dita estar em um nível *limitado*. Neste caso, não há muita interação entre as cavidades individuais, e a cavitação é ainda intermitente.

Para valores de σ muito menores que σ_i, as cavidades produzidas se tornam tão próximas no tempo e no espaço que passam a interagir entre si, dando origem a padrões de cavitação mais complexos, casos em que há a chamada *cavitação desenvolvida* ou mesmo a *supercavitação*, em casos extremos.

A cavitação desenvolvida difere da cavitação incipiente (ou limitada) não só pelo seu σ, mas por esta ser composta de eventos esporádicos e menos permanentes, capazes de causar erosão, enquanto aquela tem uma permanência maior, perturbando mais fortemente o escoamento, e provocando funcionamento instável de turbinas hidráulicas além de também causar erosão e outros danos.

Caracterização de um *nucleus*

Para a caracterização de um *nucleus*, inicialmente consideramos uma microbolha esférica, contendo gás não condensável e vapor d'água dentro de uma massa líquida (água) em repouso.

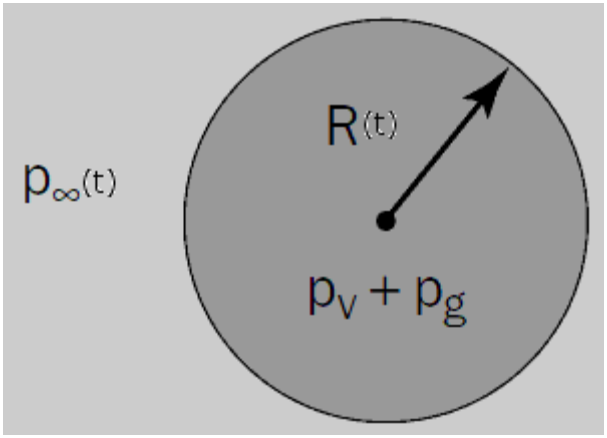


Figura 64 - Condições iniciais de análise de um nucleus em equilíbrio.

Fonte: (FRANC, MICHEL, 2004).

As pressões parciais do gás e do vapor são representadas por p_g e p_v respectivamente. O líquido é considerado capaz de suportar pressões inferiores à sua pressão de vapor sem produzir novas bolhas, pois está em seu estado metastável, e a bolha, objeto desta análise, é o ponto fraco susceptível à ruptura. Seja $R(t)$ o raio da microbolha, considerado ser pequeno o suficiente para que as diferenças de pressão hidrostáticas entre o seu ponto mais baixo e seu ponto mais alto sejam desprezíveis em relação à pressão correspondente à tensão superficial. Como a bolha em análise é microscópica por definição, automaticamente estas condições estão satisfeitas. Seja $p_\infty(t)$ a pressão externa a bolha, considerada uniforme em todo o líquido circundante (figura 64).

A condição para que o *nucleus* esteja em equilíbrio com o meio líquido é que a pressão externa seja igual à soma das pressões parciais do gás e do vapor, junto com a contribuição da tensão superficial (que sempre vai contribuir para a redução do raio e está representada por S), conforme a equação 44.

$$p_\infty(t) = p_g + p_v - \frac{2S}{R(t)} \tag{44}$$

Nas condições iniciais $(t=0)$ então, assumindo que o raio inicial é R_0 , e a pressão inicial do gás é p_{g0} a equação se torna:

$$p_{\infty 0} = p_{g0} + p_v - \frac{2S}{R_0} \tag{45}$$

Mas, supondo que a pressão $p_\infty(t)$ varia lentamente a ponto de manter o equilíbrio mecânico e permitir trocas de calor entre o gás e o líquido, mas não tão lentamente a ponto de permitir que uma quantidade considerável de gás seja trocada com o líquido externo por meio da difusão, temos então uma transformação isotérmica (a pressão do gás varia inversamente com o volume) e a massa do gás é considerada constante. Podemos reescrever a equação 45 como:

$$p_{\infty 0} = p_{g0}\left(\frac{R_0}{R(t)}\right)^3 + p_v - \frac{2S}{R(t)} \tag{46}$$

A pressão de vapor não muda, pois é uma constante para a temperatura em questão e a conversão líquido/vapor e vapor/líquido é muito rápida, levando um tempo desprezível. Como a pressão interna (que tenta aumentar o volume da bolha) sempre age contra o efeito da tensão superficial (que tenta diminuir o volume) temos, para a função $p_\infty(R)$, cujo gráfico pode ser visto na figura 65, sempre um ponto de mínimo cujas coordenadas podem ser obtidas por:

$$R_c = R_0\sqrt{\frac{3p_{g0}}{2S/R_0}} \tag{47}$$

$$p_c = p_v - \frac{4S}{3R_c} \tag{48}$$

Ainda na figura 65 está representado o lugar geométrico dos pontos de mínimo em função do raio inicial de um *nucleus* em particular, além da função $p_\infty(R)$ plotada para mais um *nucleus* distinto com raio inicial diferente do anterior. Podemos então tirar importantes conclusões desta análise simplificada. A primeira é que, o comportamento de cada *nucleus* depende de seu raio inicial, em determinada pressão. A segunda é que, temos uma pressão mínima que o conjunto suporta, e se, partindo das

condições iniciais formos abaixando gradualmente a pressão, ao atingir a pressão p_c , (que é sempre menor que a pressão de vapor do líquido), o *nucleus* cresce ilimitadamente e perde a estabilidade (em outras palavras, se torna instável e cresce explosivamente).

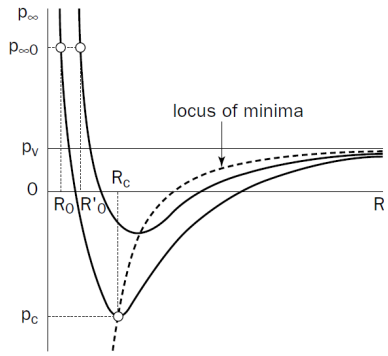


Figura 65 - Relação entre a pressão externa e o raio de um *nucleus*. Estão representados dois *nuclei* com raios iniciais diferentes sob a mesma pressão. Notar o lugar geométrico dos mínimos da função.

Fonte: (FRANC, MICHEL, 2004).

A terceira conclusão que podemos tirar dessa análise, é que a linha tracejada, ou lugar dos mínimos, separa as condições instáveis das estáveis, daí p_c e R_c serem denominados respectivamente *pressão crítica* e *raio crítico* de um *nucleus*. O *static delay* depende do raio inicial do *nucleus*. Quanto menor o *nucleus,* maior o *static delay*, ou seja, mais difícil de cavitar, como pode ser visto na equação 48.

Um *nucleus* é portanto caracterizado pelas suas condições iniciais, ou seja, a quantidade de gás e seu raio e pressão inicial. Estas condições pode ser resumidas na pressão crítica de um *nucleus*, ou ainda em seu raio crítico, e ambos os parâmetros são redundantes.

Finalmente, percebemos olhando as equações 44,47 e 48 que um *nucleus* só tem alguma estabilidade devido justamente à massa de gás não condensável presente. Se fizermos $p_{g0}=0$ então não há nenhuma solução finita para as equações, de modo que a área instável da figura 64 começaria em $R(t)=0$, ou seja, todo o gráfico é instável. Não existe bolha preenchida puramente com vapor que seja estável. Ademais, quanto mais gás presente, mais o *nucleus* suporta crescer de forma estável, sem explodir e quanto menos gás, mais tensão o *nucleus* suporta, porém cresce menos e explode mais violentamente. Nota-se que esta análise nos remete a uma melhor explicação sobre o

static delay e à baixa repetibilidade dos experimentos com cavitação, pois agora vemos que a sua ocorrência e a forma de como a cavitação evolui, depende não só da pressão de vapor, que é fixa para um determinado fluido em uma temperatura constante, mas de uma população de *nuclei*, que pode mudar devido a inúmeros fatores.

Equações que regem o comportamento de um *nucleus*

A análise do comportamento dinâmico de um *nucleus* de modo a deduzir uma equação diferencial parte de alguns pressupostos: a gravidade é desprezada, a quantidade de gás no *nucleus* é constante, o líquido é incompressível, qualquer troca de calor é negligenciada (transformações adiabáticas), a inércia também é negligenciada e a bolha está preenchida com vapor com pressão parcial igual à pressão de vapor do líquido na temperatura em questão.

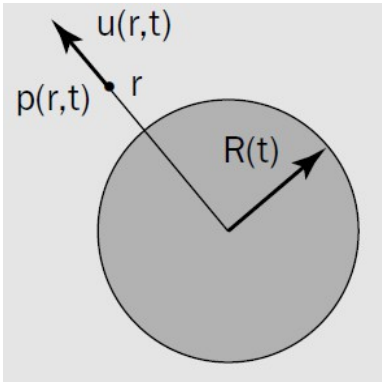


Figura 66 - Modelagem inicial de um *nucleus*.
Fonte: (FRANC, MICHEL, 2004).

Definimos então, como visto na figura 66, que o raio instantâneo da bolha esférica é $R(t)$, a pressão instantânea a uma distância r do centro da bolha é $p(r,t)$ e $u(r,t)$ é a velocidade radial desenvolvida nas partículas do líquido $(r \geq R(t))$ devido à evolução da bolha.

A velocidade $u(R(t),t)$ na interface da bolha é igual a $\dot{R}$ (derivada em função do tempo), considerando a troca de massa pela interface desprezível.

No caso de um líquido com coeficiente de viscosidade cinemática μ , temos a

tensão (pressão) normal sobre a superfície da bolha descrita pela equação 49:

$$t_{rr}(R(t),t) = -p(R(t),t) + 2\mu \left.\frac{\partial u}{\partial r}\right|_{r=R} \tag{49}$$

E o balanço é dado pela soma das pressões de vapor, pressão parcial do gás não condensável e contribuição da tensão superficial:

$$-t_{rr}(R(t),t) = p_v + p_g(t) - \frac{2S}{R(t)} \tag{50}$$

Considerando uma transformação adiabática do gás, onde γ é a razão entre as capacidades caloríficas do gás e do vapor, temos então:

$$p_g(t) = p_{g0}\left[\frac{R_0}{R(t)}\right]^{3\gamma} \tag{51}$$

Reescrevendo com base nas equações 49,50 e 51 temos:

$$p(R(t),t) = p_v + p_{g0}\left[\frac{R_0}{R(t)}\right]^{3\gamma} - \frac{2S}{R(t)} + 2\mu \left.\frac{\partial u}{\partial r}\right|_{r=R} \tag{52}$$

Ou seja, a pressão externa de excitação é transmitida para a interface e equilibra a soma das pressões parciais do vapor e do gás, adicionadas da contribuição da tensão superficial e da viscosidade. Distante da bolha assume-se que a velocidade do fluido tende a zero $(u(\infty,t)\to 0)$ e que a pressão à longa distância é conhecida $(p(\infty,t) = p_\infty(t))$. E para as condições iniciais em equilíbrio, assume-se que $\dot{R}(0) = 0$ de modo que a equação 45 seja satisfeita.

Equação de Rayleigh-Plesset

Devido à simetria esférica, o fluxo é irrotacional. A equação de conservação de massa para um líquido incompressível obriga o campo de velocidades ter divergente nulo, portanto:

$$u(r,t)=\dot{R}\frac{R^2}{r^2} \tag{53}$$

E neste caso particular, o termo referente à viscosidade da equação de Navier-Stokes é igual a zero. Portanto, a equação de momento se torna:

$$\frac{\partial u}{\partial t}+u\frac{\partial u}{\partial r}=-\frac{1}{\rho}\frac{\partial p}{\partial r} \tag{54}$$

Substituindo a equação 53 em 54, conclui-se que:

$$\ddot{R}\frac{R^2}{r^2}+2\dot{R}^2\left[\frac{R}{r^2}-\frac{R^4}{r^5}\right]=-\frac{1}{\rho}\frac{\partial p}{\partial r} \tag{55}$$

E integrando ambos os lados com relação à distancia r e considerando as condições de contorno no infinito, temos o equivalente à equação de Bernoulli:

$$\frac{p(r,t)-p_\infty(t)}{\rho}=\ddot{R}\frac{R^2}{r}+2\dot{R}^2\left[\frac{R}{r}-\frac{R^4}{4r^4}\right] \tag{56}$$

Na interface $(r=R)$, a equação se reduz a:

$$\frac{p(r,t)-p_\infty(t)}{\rho}=\ddot{R}R+\frac{3}{2}\dot{R}^2 \tag{57}$$

E levando em conta a equação 52, e que na interface:

$$\left.\frac{\partial u}{\partial r}\right|_{r=R}=-2\frac{\dot{R}}{R} \tag{58}$$

Finalmente chegamos à dedução da equação 59 conhecida como *equação de Rayleigh-Plesset* e que nos permite calcular a evolução do raio $R(t)$ em função de uma dada perturbação na pressão $p_\infty(t)$. Esta equação diferencial é não linear devido aos termos de caráter inercial (que são dominantes), e serve para calcular como é o crescimento explosivo de um *nucleus*, bem como a sua implosão. O termo da viscosidade toma papel secundário durante o colapso.

$$\rho\left[R\ddot{R}+\frac{3}{2}\dot{R}^2\right]=p_v-p_\infty(t)+p_{g0}\left(\frac{R_0}{R(t)}\right)^{3\gamma}-\frac{2S}{R(t)}+4\mu\frac{\dot{R}}{R(t)} \tag{59}$$

O equilíbrio de um *nucleus*

Para estudar o comportamento de um *nucleus* durante sua fase de crescimento, devemos partir da equação de Rayleigh-Plesset e, considerando inicialmente as condições de equilíbrio com raio inicial R_0 e a pressão aplicada em um ponto distante $p_{\infty 0}$, vamos aplicar uma pressão $p_\infty \leqslant p_{\infty 0}$ a fim de estimular o *nucleus* a crescer, lembrando que a nova pressão aplicada pode ser maior ou menor que a pressão de vapor. Isto é equivalente a um degrau de abaixamento de pressão, onde a mudança brusca pode ter várias consequências distintas, como será mostrado a seguir.

O primeiro termo entre colchetes na equação 59 pode, com auxílio de manipulações matemáticas como a regra da cadeia, e com a finalidade de eliminar a derivada segunda do raio, ser reescrito como:

$$R\ddot{R}+\frac{3}{2}\dot{R}^2=\frac{1}{2\dot{R}R^2}\frac{d}{dt}\left[\dot{R}^2R^3\right] \tag{60}$$

E então, a equação de Rayleigh-Plesset pode ser reescrita e integrada analiticamente, levando em conta uma transformação adiabática do gás não condensável, haja vista que esperamos que o *nucleus* se expanda até um raio bastante grande, resultando em:

$$\rho\left[\frac{1}{2\dot{R}R^2}\frac{d}{dt}\left[\dot{R}^2R^3\right]\right]=p_v-p_\infty(t)+p_{g0}\left(\frac{R_0}{R(t)}\right)^{3\gamma}-\frac{2S}{R(t)}+4\mu\frac{\dot{R}}{R(t)} \tag{61}$$

A equação diferencial apresentada é não linear e do tipo $\dot{R}^2=f(R)$, e a sua análise é baseada no estudo do sinal de $f(.)$ e na existência de suas raízes, levando em conta que só pode haver solução real com movimentação se $f(R)>0$. Notar que, no caso particular de $p_\infty=p_{\infty 0}$ então $R=R_0$ e a equação se iguala a zero do lado direito, obrigando uma raiz dupla em $\dot{R}=0$ e $\ddot{R}=0$, sendo que para $R\neq R_0$ (ou seja, na

vizinhança de R_0), por critérios de estabilidade $f(R)$ tem de ser negativo. Não há mudança no raio, e a equação se resume a equação 45.

O crescimento de um *nucleus*

Considerando agora um real abaixamento de pressão tal que $p_\infty < p_{\infty 0}$, sendo a nova pressão maior ou menor que a pressão de vapor, então $f(R)$ tem uma raiz $R_1 > R_0$ e o sinal de $f(R)$ muda tanto em R_1 como em R_0 , e para valores intermediários $f(R)$ tem valor positivo. Então o raio tem a sua excursão limitada entre R_0 e R_1 e como não há dissipação considerada na equação (viscosidade), então surgem oscilações periódicas. As características das oscilações dependem fortemente da relação R_1/R_0 . Se a relação é grande, as oscilações tornam-se altamente não lineares e parecem mais com uma sucessão de colapsos e explosões. A não linearidade dos *nuclei* desempenha papel muito importante no comportamento dos mesmos sob excitação de uma pressão externa periódica, causando até mesmo respostas não periódicas, ou repostas em harmônicas da frequência natural. A não linearidade também leva os *nuclei* a adotarem comportamentos caóticos, com bifurcações e outras características tais que acabam por tornar o ruído gerado pela cavitação algo bem complexo.

Alternativamente, se $R_1 \approx R_0$ então as oscilações são aproximadamente lineares e com uma frequência que depende do raio inicial, da pressão inicial do gás e da tensão superficial. A bolha se torna um oscilador com frequência fundamental f_0:

$$f_0 = \frac{1}{2\pi R_0}\sqrt{\frac{1}{\rho}\left[3\gamma p_{g0} - \frac{2S}{R_0}\right]} \tag{62}$$

Valores de frequência natural tipicamente encontrados para um *nucleus* de 10 μm de raio na água e pressão aplicada de 1 atm ficam em torno de 340 kHz. Para um *nucleus* com raio de 1 μm, a frequência sobe para 4.7 MHz.

Já no caso de $R_1 \gg R_0$ de modo que a equação não tenha raiz maior que R_0 , não há mudança de sinal de $\dot{R}$ e o *nucleus* cresce ilimitadamente. Pode-se calcular a velocidade terminal da interface neste crescimento explosivo:

$$\dot{R}_{\infty}=\sqrt{\frac{2}{3}\frac{p_v-p_{\infty}}{\rho}} \tag{63}$$

Valores muito próximos são observados experimentalmente.

A implosão de um *nucleus*

Para estudar a implosão de um *nucleus*, devemos partir do pressuposto que o mesmo não tem gás não condensável em seu interior, e simplificando ainda mais, podemos desconsiderar a contribuição da viscosidade e da tensão superficial. A pressão inicialmente aplicada fora do *nucleus* é igual a pressão de vapor, e portanto, o *nucleus* está em equilíbrio instável, de acordo com a equação 45.

Após o instante $t=0$ a pressão aplicada fora do *nucleus* aumenta para um valor $p_{\infty}>p_v$ o que resulta no colapso do *nucleus* em um tempo característico τ chamado de *tempo de Rayleigh*, ou *Rayleigh time*.

Como desconsideraremos a existência de gás condensável $(p_{g0}=0)$ e de efeitos da tensão superficial $S=0$ então a equação 59 ao ser integrada se reduz a:

$$\rho\dot{R}^2R^3=-\frac{2}{3}(p_{\infty}-p_v)\left(R^3-R_0^3\right) \tag{64}$$

Estamos interessados somente em $\dot{R}$ com valores negativos, pois trata-se de uma implosão. Daí selecionamos somente a raiz negativa:

$$\frac{dR}{dt}=-\sqrt{\frac{2}{3}\frac{p_{\infty}-p_v}{\rho}\left[\frac{R_0^3}{R^3}-1\right]} \tag{65}$$

Onde podemos ver que o raio tende a zero e a velocidade radial cresce ilimitadamente, e é maior quanto menor o raio. Calculando através de uma integração numérica (a integral de *dt*) da equação acima, seguida de uma aproximação, podemos encontrar o tempo de Rayleigh:

$$\tau \simeq 0.915\, R_0 \sqrt{\frac{\rho}{p_\infty - p_v}} \tag{66}$$

Esta relação está em boa aproximação com valores observados experimentalmente para raios desde 1 μm até 1 m e, pode-se estimar a velocidade da interface nos últimos estágios da implosão pela aproximação da equação 65, levando em conta que $R \to 0$:

$$\dot{R} \simeq -\sqrt{\frac{2}{3}\frac{p_\infty - p_v}{\rho}\left[\frac{R_0^3}{R^3}\right]} \simeq 0.747\,\frac{R_0}{\tau}\left[\frac{R_0}{R}\right]^{\frac{3}{2}} \tag{67}$$

As equações 66 e 67 se resolvidas para o caso de um *nucleus* imerso na água com de raio inicial de 1 cm e pressão aplicada de aproximadamente uma atmosfera, retornam o tempo de implosão de 1 ms e velocidade final de mais de 720 m/s, o que chega a ser metade da velocidade de propagação do som na água.

Obviamente, esta é uma boa aproximação que nos mostra como uma bolha se comporta, e serve para termos ideia da natureza das implosões. Velocidades finais tão elevadas sugerem que há a formação de ondas de choque, algo realmente observado na prática, lembrando que neste caso, a compressibilidade do fluido não deve ser desprezada. Outros efeitos como a tensão superficial contribuiriam para reduzir ainda mais o tempo de Rayleigh e aumentar a geração de ondas de choque. O gás não condensável confere ao *nucleus* certa elasticidade, fazendo possivelmente a implosão ricochetear e dar início a um novo ciclo de explosão/implosão.

Quando um *nucleus* se aproxima de um hidrofólio ou de uma pá de um rotor de turbina, considera-se que o seu tempo de trânsito (o *nucleus* seguindo uma linha de fluxo) é grande o suficiente para permitir troca de calor entre o gás não condensável e o líquido/vapor circundante, portanto, é válida a aproximação isotérmica do crescimento estável e lento de um *nucleus*. Por outro lado, se o *nucleus* encontra um ponto na linha de fluxo com pressão inferior à sua pressão crítica, se instabiliza e cresce explosivamente, pouco importando para a sua evolução o quanto de gás não condensável possuía inicialmente e seu volume original, bem como a transferência de calor. A transformação adiabática para esses casos é uma aproximação adequada.

Como uma extensão importante de nossa análise, podemos derivar em relação ao

tempo mais uma vez a equação 65, obtendo $\ddot{R}$:

$$\ddot{R} = -\frac{p_\infty - p_v}{\rho}\frac{R_0^3}{R^4} \tag{68}$$

E substituindo $\dot{R}$ e $\ddot{R}$ na equação 56 podemos calcular o campo de pressões $p(r,t)$ em função da pressão de vapor, pressão inicial de excitação e relação entre o raio instantâneo e o raio inicial. A dedução das fórmulas está detalhada em uma das referências bibliográficas (FRANC, MICHEL, 2004) de modo que é possível encontrar o valor de máximo da pressão gerada por uma implosão. É mostrado que, apesar de ser uma aproximação que só leva em conta fatores inerciais, modelos mais complexos exibem comportamento parecido e, valores típicos de pressões encontradas nas proximidades de um *nucleus* que implode na água com diminuição de raio em 20 vezes e pressão atmosférica de excitação giram em torno de 1200 atmosferas.

APÊNDICE B – TURBINAS HIDRÁULICAS

Turbinas de impulso e reação

Turbinas de impulso funcionam alterando a direção de um jato de fluido em alta velocidade, sendo que a segunda lei de Newton descreve a transferência de energia do fluido para o rotor. Em geral, neste tipo de turbina há um conjunto de bocais fixos (que formam o estator) que convertem energia potencial, a pressão da água, em energia cinética, daí a formação dos jatos de fluido de alta velocidade. Um exemplo clássico é a turbina do tipo *Pelton*.

Turbinas de reação têm seu mecanismo de transferência de energia do fluido para o rotor descrito pela terceira lei de Newton. Este tipo de turbina desenvolve torque no rotor através de uma reação à pressão ou massa do fluido. Neste caso, há uma diferença de pressão entre um lado e outro do rotor, haja vista que o mesmo funciona como um conjunto de bocais móveis. O estator tem a função de dirigir o fluido ao rotor.

As turbinas usadas atualmente para a produção de energia elétrica são quase que na totalidade turbinas de reação. Neste tipo de turbina, além de termos o rotor e o estator, que funciona como distribuidor/diretor do fluxo de água, temos o invólucro (caixa voluta), que mantém o fluxo contido na unidade, e o tubo de descarga, que mantém a sucção na saída da unidade. Alguns tipos de turbinas que têm este tipo de construção são as turbinas to tipo *Francis* e *Kaplan*, que constituem a maioria absoluta das turbinas usadas em geração hidroelétrica.

Turbinas Francis e Kaplan

Na turbina tipo Francis (figura 67) temos um invólucro em espiral, ou caixa voluta, por onde a água entra. Após transferir parte de sua energia mecânica para o rotor, a água sai pelo tubo de descarga, ligado ao centro do caracol, na mesma direção do eixo do rotor. Este tipo de turbina tem uma eficiência típica de 90 % e opera com velocidades típicas de 83 a 1000 rpm e em geral é montada na posição de eixo vertical, com o

gerador acima, para evitar que a água chegue até o mesmo.

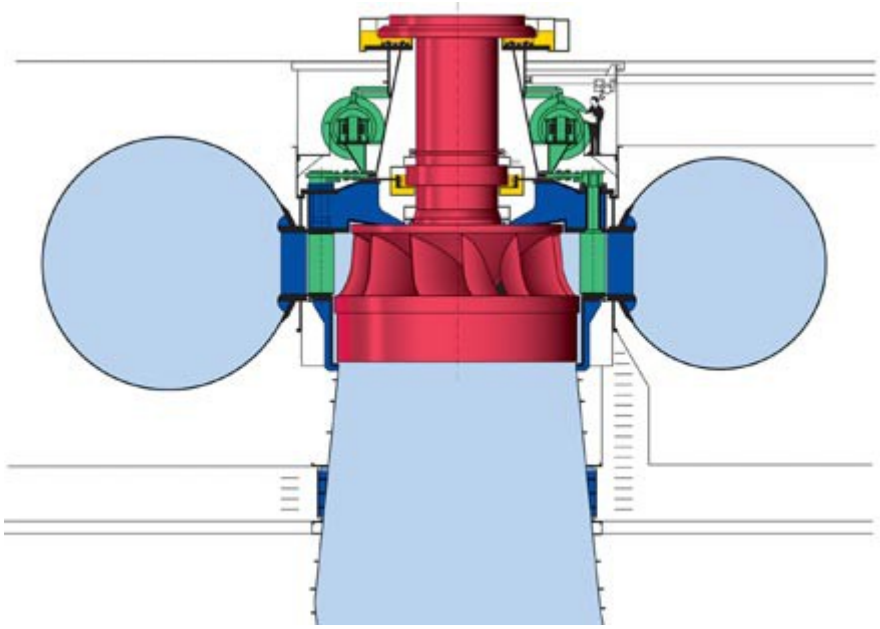


Figura 67 - Visão artística de uma turbina Francis vista em corte vertical, com o rotor em vermelho. A água entra pelas laterais e sai por baixo.

Fonte:<http://upload.wikimedia.org/wikipedia/commons/5/5f/M_vs_francis_schnitt_1_zoom.jpg>. Acesso em 25/02/2015.

A fim de poder adequar o regime de escoamento da água à necessidade instantânea de geração de potência, essas turbinas dispõem de um distribuidor com palhetas guias móveis (em verde na figura 67), também chamadas de *guide vanes* ou *wicket gate*. Tais palhetas dirigem o fluxo de água de forma conveniente ao rotor (em vermelho) e suas aberturas são alteradas de acordo com a necessidade de maior ou menor potência a ser gerada.

Como esta turbina é do tipo de reação e seu rendimento é elevado, idealmente a água sai com baixa velocidade (baixa energia cinética) e baixa pressão (baixa energia potencial). Esta turbina também pode ter a sua operação revertida, e daí passar a funcionar como uma bomba.

Um outro tipo também bastante comum de turbina empregada na geração de energia hidroelétrica, e que surgiu como uma evolução das turbinas tipo Francis, é a turbina *Kaplan*. A sua invenção pelo professor Viktor Kaplan em 1913 permitiu a geração de energia em locais com colunas d'água/pressões bem menores que as necessárias em uma turbina Francis. Hoje em dia essas turbinas são largamente

empregadas em todo o mundo, pois através de um sistema de pás do rotor ajustáveis, conseguem se adaptar a uma faixa bastante larga de vazões e níveis de água. Este tipo de turbina teve, na época de sua invenção e desenvolvimento em 1922, sérios problemas com a cavitação.

A turbina Kaplan tem sua construção muito semelhante à turbina Francis, com o mesmo invólucro em formato de caracol e o mesmo tubo de descarga, porém com rotor completamente diferente (figura 68). Neste caso, o rotor se parece com um *propeller* (uma hélice de navio), e o ângulo de ataque das pás é ajustável. Desta forma, tanto o rotor quanto o estator são ajustáveis, de modo que a turbina se ajusta bem às diversas condições de operação, mantendo em geral eficiência superior a 90 %. Este tipo de turbina é mais adequado para locais com pequenos desníveis e grandes vazões.

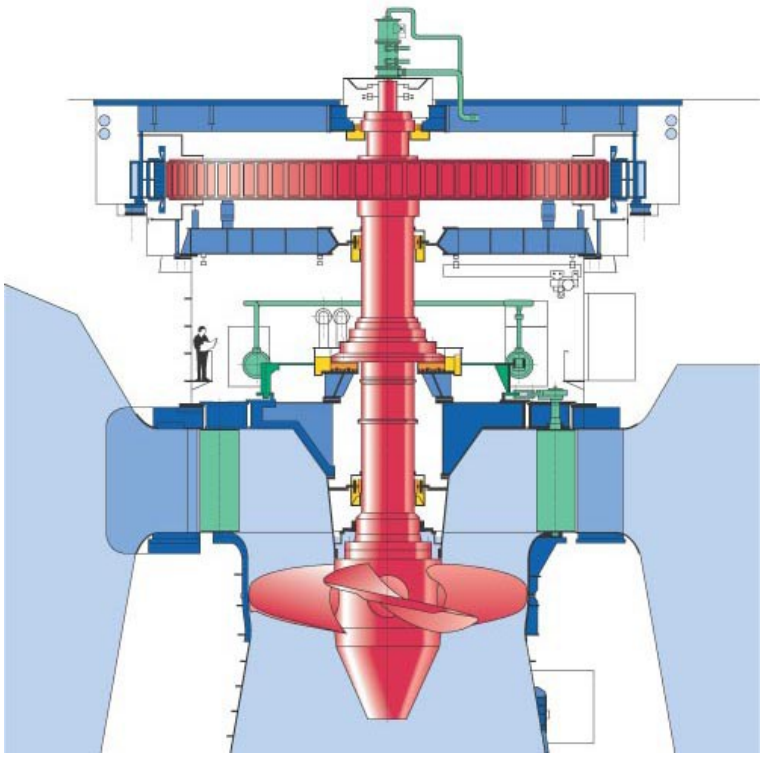


Figura 68 - Visão artística de uma turbina tipo Kaplan vista em corte vertical.

Fonte: <http://upload.wikimedia.org/wikipedia/commons/1/16/S_vs_kaplan_schnitt_1_zoom.jpg>. Acesso em 26/02/2013.

Há ainda outros tipos de turbinas empregados para geração hidroelétrica, porém, são muito pouco empregados, e constituem quase sempre variações dos tipos Kaplan e Francis, de modo que não serão abordados em detalhes.

Os diferentes tipos de turbinas também diferem nas faixas de pressão *H* (nível de

reservatório) e vazão Q possíveis para operação. O ponto de operação de uma turbina determina a potência mecânica produzida, e depende também da eficiência η da máquina:

$$P = \rho g Q H \eta \tag{69}$$

Onde g é a gravidade e ρ é a densidade da água. O fabricantes de turbinas limitam as faixas de pressão e vazão para mitigar o aparecimento da cavitação, e dependendo do tipo de turbina, há uma *faixa de operação apropriada* e um *ponto de operação ótimo*. Diferenças entre as faixas de operação apropriadas para diversos tipos de turbinas podem ser vistas na figura 69.

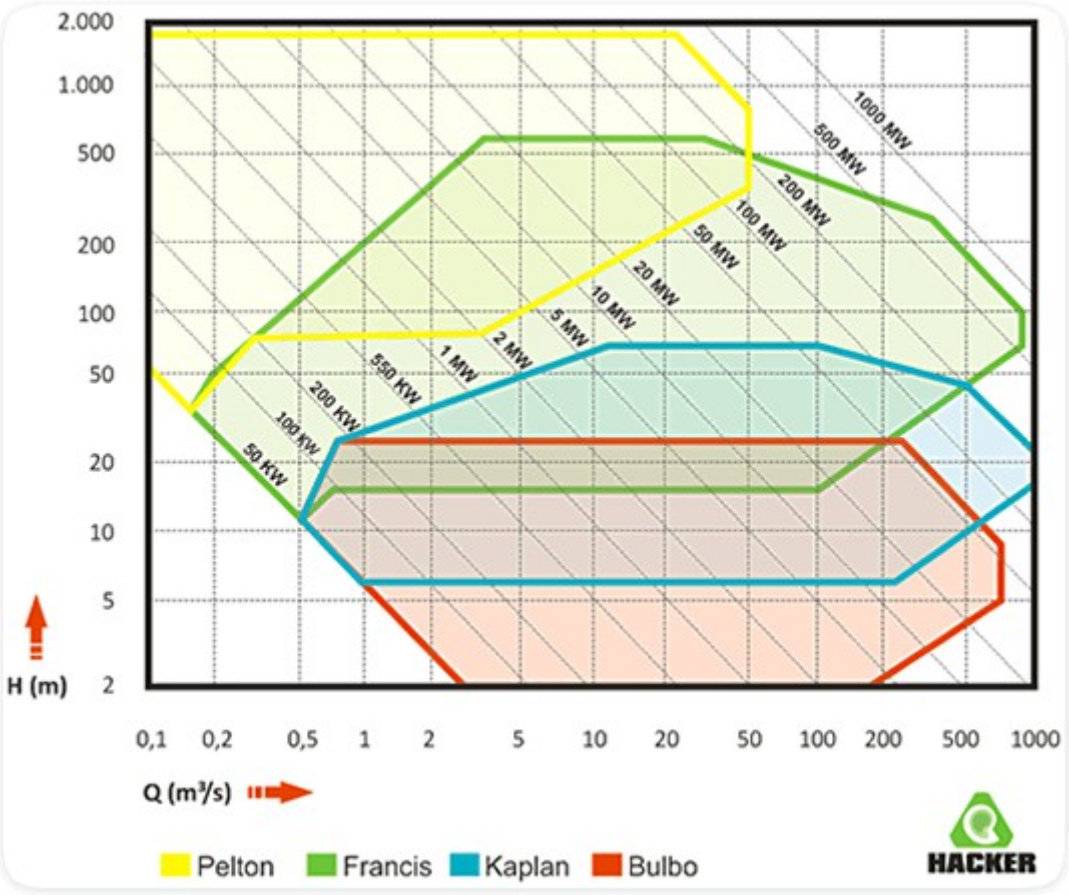


Figura 69 - Características operacionais dos tipos de turbina hidráulicas.

Fonte:<http://www.hacker.ind.br/produtos_turbinas_hidraulicas.php> Acesso em 03/05/2015.

APÊNDICE C – EROSÃO

Mecanismos de concentração de energia

Os principais mecanismos de concentração de energia devido à cavitação são: a produção de *ondas de choque* conforme descrito no Apêndice A – Cavitação, e a formação de microjatos. As altas temperaturas e pressões produzidas no interior de uma bolha que implode, estando esta bolha longe de qualquer superfície sólida, dão origem às ondas de choque, e o colapso é possui simetria esférica.

Se a implosão ocorre próxima a algum corpo sólido, a implosão deixa de ter simetria esférica, e surgem os microjatos. Em 1971, PLESSET e CHAPMAN calcularam numericamente a evolução temporal do colapso de um *nucleus* próximo a uma parede plana sólida, e excitado por uma pressão constante p_∞. Várias distâncias entre a parede e o *nucleus* foram empregadas para o cálculo e o *nucleus* é assumido conter somente vapor, com a tensão superficial desprezada. O método utilizado envolve primeiro encontrar o campo potencial de velocidades usando algum algorítimo numérico e depois a velocidade da interface pode ser deduzida deste campo potencial, pois é seu gradiente.

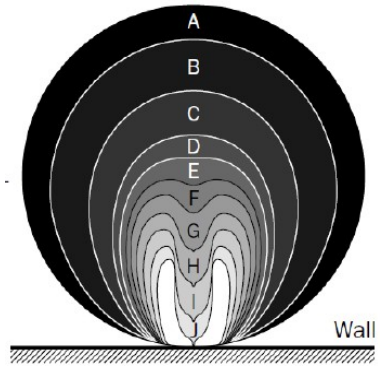


Figura 70 - Evolução da implosão de um nucleus próximo a uma parede sólida, em diversos instantes (A a J).

Fonte: (FRANC, MICHEL, 2004)

O resultado mais significativo desta simulação numérica, para este estudo sobre

cavitação em turbinas, é que a presença da parede sólida nas proximidades ou mesmo tocando a interface interfere no campo de pressões de tal forma que o colapso não é mais esférico, mas forma um *jato reentrante* sempre em direção à parede sólida (figuras 70 e 71). A porção da interface que está mais próxima da parede é comprimida e se torna mais achatada, enquanto a porção de interface mais distante da parede é acelerada e atinge velocidades finais muito altas (da mesma ordem da velocidade do som na água), atingindo com violência a parede sólida. Este mecanismo é atualmente bastante aceito como um dos principais responsáveis pela erosão devido à cavitação.

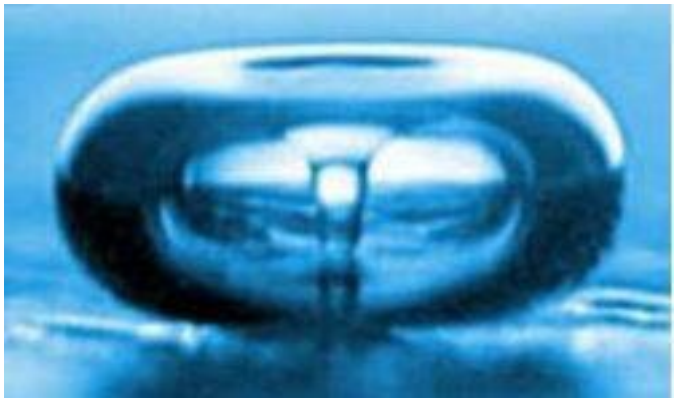


Figura 71 - Microfotografia de um nucleus adjacente a uma parede sólida no momento exato da implosão e formação de microjato reentrante.

Fonte:<http://www.shockwavetherapy.ca/about_eswt.htm>. Acesso em 02/03/2015.

Notamos então que, um objeto sólido nas proximidades de uma população de *nuclei* instáveis sofre um verdadeiro "bombardeio" por mini cargas de profundidade. Este comportamento de explosões subaquáticas já era bem conhecido pelos engenheiros da II Guerra Mundial, que criaram armas anti-submarino (figuras 72 e 73).

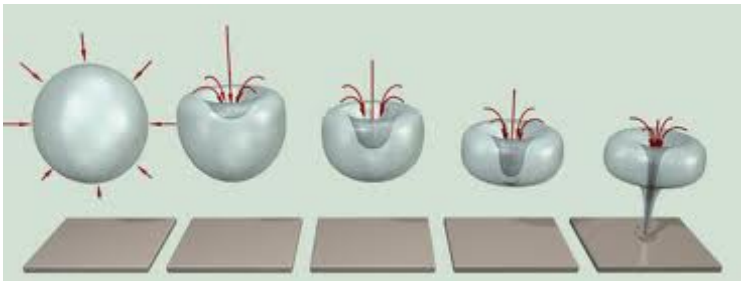


Figura 72 - Evolução da implosão e formação de microjato atingindo uma parede sólida próxima.

Fonte: <http://eswt.net/biological-mechanism>. Acesso em 02/03/2013.

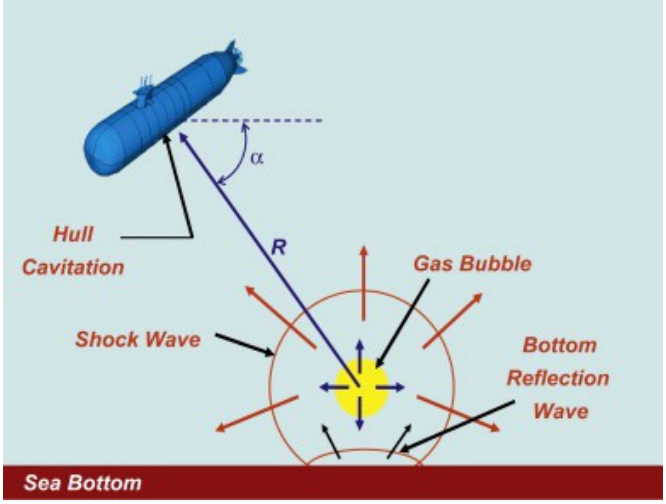


Figura 73 - Produção de ondas de choque e consequente destruição por fadiga de corpos próximos.

Fonte: (KIM, SHIN, 2008).

Como uma alternativa do processo de colapso próximo a uma superfície sólida, o toroide remanescente pode ainda se partir e dar origem a inúmeras bolhas filhas, que produzirão por sua vez novos colapsos. Neste caso, a formação de bolhas menores cria novos "osciladores" de frequências naturais mais altas, deslocando a emissão acústica para frequências superiores.

Variações na taxa de perda de massa devido à erosão

Uma característica marcante da erosão por cavitação é a variação da taxa de erosão (em termos de perda de massa) com o passar do tempo, mantendo-se constante a agressividade da cavitação. É possível identificar diferentes estágios do material, que correspondem a diferentes taxas distintas de perdas de massa.

Inicialmente há um período, denominado de *período de incubação*, onde não se observa perda de massa alguma, pois o material é dito "virgem" e os impactos sofridos e consequentes deformações microscópicas não se sobrepuseram entre si. Os únicos efeitos perceptíveis neste período são microscópicos. O próprio corpo de prova neste período serve, através de uma análise microscópica, como transdutor, com a finalidade

de se medir a agressividade da cavitação, pois o número de deformações é o principal indicador.

Após o período de incubação, segue o *período de aceleração*, onde a perda de massa por unidade de tempo aumenta até um determinado limite. Nesta fase as deformações microscópicas começam a se sobrepor, causando fraturas por fadiga do material.A taxa de perda de massa chega então a um valor máximo, e o corpo metálico chega ao *período de estado estacionário*, onde permanece perdendo massa em ritmo constante por um longo período, até que em testes de muito longa duração, chega-se ao *período de atenuação*, onde a taxa de perda de massa cai. Na figura 74 é mostrado um gráfico representando o comportamento típico da erosão por cavitação. Uma maneira alternativa de entender a evolução da erosão é através do fluxo de energia, pois, a energia produzida pelos colapsos das cavidades vai sendo gradualmente armazenada (energia potencial) no retículo cristalino do corpo metálico (endurecimento), até o limite onde supera a energia de ligação entre duas partículas, dando origem a uma fissura microscópica (WILCZYŃSKI, 2003).

É óbvio que os valores de duração dos períodos citados e das taxas de perda de massa dependem dos materiais empregados e dos equipamentos usados nos ensaios de cavitação/erosão, entretanto, experimentalmente observa-se que o produto da taxa de perda de massa máxima (no estado estacionário) pela duração do período de incubação é aproximadamente constante para um determinado equipamento de ensaios de erosão.

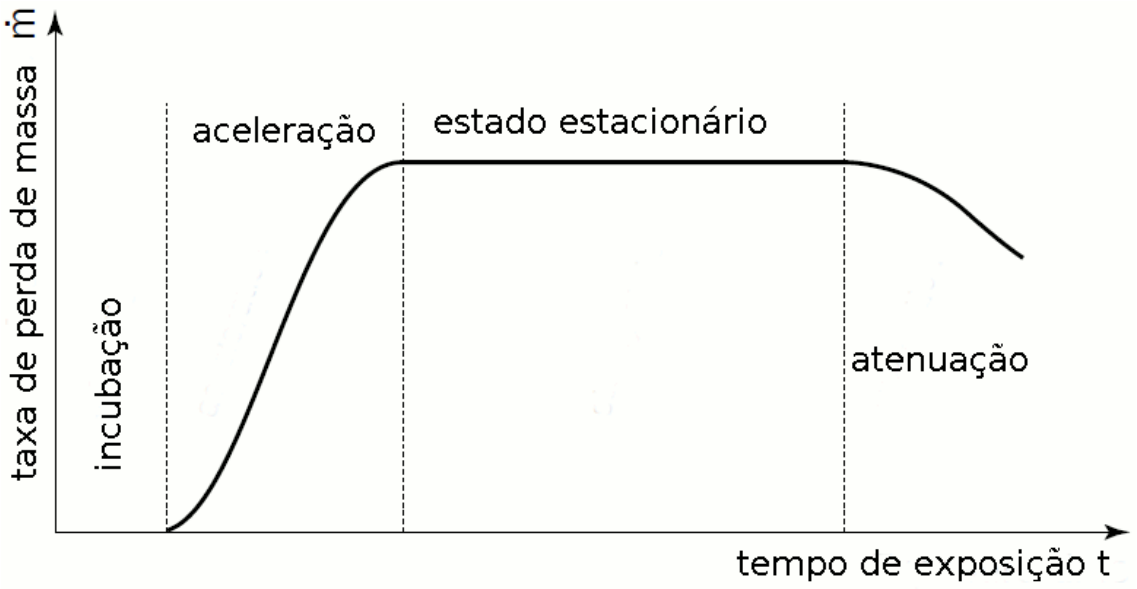


Figura 74 - Evolução típica da erosão por cavitação, mantendo-se a agressividade constante. O tempo de incubação e a taxa máxima atingida são inversamente proporcionais.

Fonte: Adaptado de (FRANC, MICHEL, 2004).

Para um escoamento em velocidade constante, a taxa de erosão segue a mesma tendência da emissão acústica: Ao diminuir o número de cavitação até seu valor incipiente, a erosão aparece e começa a aumentar até a condição de cavitação desenvolvida, onde a taxa de erosão cai, provavelmente em decorrência da produção de um menor número de cavidades de vapor.

A quantidade de gás não condensável dissolvido no fluido, mais especificamente o teor de ar na água é também de grande influência na erosão, pois uma maior parcela de gás não condensável torna as cavidades de vapor menos instáveis e portanto, os golpes contra as paredes metálicas dos corpos expostos à cavitação são de intensidade menor. De fato, a injeção de ar em algumas turbinas hidráulicas pode reduzir ou mesmo mitigar a erosão (ARNDT *et al.*, 1995; QIAN *et al.*, 2007; WILLIAM DUNCAN, 2000).

APÊNDICE D – RUÍDO BARKHAUSEN E MAGNETOSTRIÇÃO

O *efeito Barkhausen* é o nome dado ao ruído produzido em um indutor, quando o material magnético em seu interior sofre mudança no campo magnetizante. Este efeito é causado por mudanças bruscas no tamanho do *domínios magnéticos*, que são regiões dentro de um material magnético em que a magnetização assume uma direção uniforme. Isto significa que dentro de um dado domínio magnético, todos os dipolos magnéticos estão com a mesma orientação. Quando um material ferromagnético é posto em um campo magnético de intensidade crescente (suavemente), a densidade do fluxo magnético não varia suavemente, mas com pequenos saltos, denominados *saltos de Barkhausen* (figura 75).

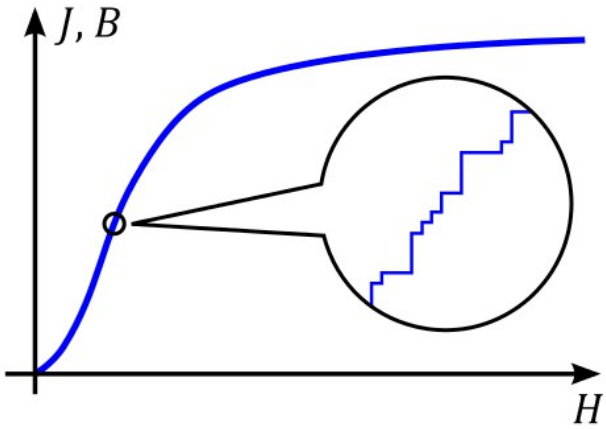


Figura 75 - Gráfico mostrando a relação entre a magnetização (J) ou densidade de fluxo magnético (B) e o campo magnetizante (H). Notar que há saltos de Barkhausen.

Fonte:<http://en.wikipedia.org/wiki/Barkhausen_effect#/media/File:Barkhausen_jumps.svg> Acesso em 15/05/2015.

Estes saltos na magnetização produzem pulsos aleatórios de voltagem nos terminais do indutor, que soam como um ruído. Efeitos similares podem ser obtidos também aplicando somente tensões mecânicas na amostra de material ferromagnético. Este efeito é usado largamente para avaliações não destrutivas das propriedades mecânicas dos materiais, por exemplo, na inspeção de oleodutos de aço.

Ao mesmo tempo que o ruído elétrico é produzido nos terminais do indutor, ondas mecânicas (elásticas) de alta frequência (até 2 MHz) são produzidas, devido ao movimento das paredes dos domínios magnéticos enquanto são reorientados. Esta onda elástica produzida é chamada de *emissão magnetoacústica* (EMA), também conhecida

por *efeito Barkhausen acústico* (NAMKUNG *et al.*, 1988). Esta emissão acústica é gerada pelos saltos Barkhausen, que provocam deformações localizadas, devido ao fenômeno da magnetostrição e mudanças de volume devido à magnetização. As interações entre as paredes dos domínios magnéticos e os defeitos na estrutura cristalina do material ferromagnético (deslocações) também ajudam a produzir ondas elásticas, que em certas condições são assimétricas.

A *magnetostrição* é uma propriedade dos materiais ferromagnéticos, e consiste na alteração da forma ou das dimensões dos corpos ferromagnéticos quando submetidos à magnetização. Este efeito causa perdas devido à fricção (que causa aquecimento) entre os domínios magnéticos (perdas por histerese magnética no núcleo). Este efeito também é o responsável pela emissão sonora conhecida como *ruído hum* em transformadores e máquinas elétricas de corrente alternada e alta potência.

APÊNDICE E – CÓDIGO FONTE DOS *SOFTWARES* DESENVOLVIDOS

Arquivo **Makefile**:

```
#configura o diretório com os fontes .c e .cc:
SRCDIR=src
#configura o diretório com os arquivos headers:
INCDIR=include
#configura o diretório onde ficarão os objetos .o intermediários:
OBJDIR=obj
#configura o diretório onde ficarão os executáveis finais
EXEDIR=bin
#configura o diretório de recursos (fontes true type, etc...)
RESDIR=resource

#define os nomes dos executáveis que serão gerados
EXEC=bin2wav turbinesoundsimulator spectrogram computerordertracking cot-tsa detectcorrelations syncHOS

###################################
#Variáveis Globais de Compilação:
###################################
#define o compilador C/C++:
CC=gcc
#define o compilador obrigatório C++:
CXX=g++
#define e stripper
STRIP=strip
#define os flags de compilação C/C++:
CFLAGS=-W -Wall -O2 -I$(INCDIR) -fopenmp
#define os flags de compilação C++:
CXXFLAGS=$(CFLAGS)
#define os flags de linkagem
LDFLAGS=


#varre o diretório dos fontes e encontra todos os arquivos .c e .cc e define-os como fontes:
SRC = $(notdir $(wildcard $(SRCDIR)/*.c) $(wildcard $(SRCDIR)/*.cc))
#varre o diretório corrente e encontra todos os arquivos .h e define-os como headers:
HDR=$(notdir $(wildcard $(INCDIR)/*.h))
#varre o diretório corrente e encontra todos os arquivos .TTF e define-os como fontes true type:
TTF=$(notdir $(wildcard $(RESDIR)/*.TTF))
#substitui na variável SRC os nomes com .c e .cc para .o
OBJ=$(addsuffix .o,$(basename $(SRC)))
#OBJ=$(addprefix obj/,$(addsuffix .o,$(basename $(SRC))))
#configura os diretórios que serão usados para buscar os prerequisitos:
vpath %.cc $(SRCDIR)
vpath %.c  $(SRCDIR)
vpath %.o  $(OBJDIR)

#Targets:
all: $(EXEC)

stripped: all
	$(STRIP) $(EXEC)

analyzer: analyzer.o inifile.o stft.o cms.o tsa.o grpout.o wavfile.o signal.o window.o
	$(CC) -o $@ $^ $(LDFLAGS) $(shell Magick++-config --libs) -lsndfile -lm -lfftw3 -lstdc++ -lsamplerate

analyzer2: analyzer2.o inifile.o stft.o cms.o wavfile.o signal.o window.o grpGL.o tsa.o grpout.o
	$(CC) -o $@ $^ $(LDFLAGS) -lmgl -liniparser -lstdc++ -lm -lfftw3 -lsndfile -lsamplerate $(shell Magick++-
config --libs)

bin2wav: bin2wav.o
	$(CC) -o $@ $^ $(LDFLAGS) -lsndfile -liniparser

spectrogram: spectrogram.o matrices.o signal.o wavfile.o grpGL.o STFT-iSTFT.o CMS-iCMS.o CMC-iCMC.o
	$(CC) -o $@ $^ $(LDFLAGS) -lm -liniparser -lfftw3 -lsndfile -lmgl -lstdc++ -lgomp

turbinesoundsimulator: turbinesoundsimulator.o wavfile.o signal.o
	$(CC) -o $@ $^ $(LDFLAGS) -lgsl -lgslcblas -lm  -liniparser -lsndfile -lstdc++

computerordertracking: computerordertracking.o signal.o wavfile.o grpGL.o
	$(CC) -o $@ $^ $(LDFLAGS) -lm -liniparser -lsndfile -lstdc++ -lsamplerate -lmgl

cot-tsa: cot-tsa.o signal.o wavfile.o grpGL.o
	$(CC) -o $@ $^ $(LDFLAGS) -lm -liniparser -lsndfile -lstdc++ -lsamplerate -lmgl

detectcorrelations: detectcorrelations.o matrices.o grpGL.o wavfile.o
	$(CC) -o $@ $^ $(LDFLAGS) -lm -liniparser -lsndfile -lstdc++ -lmgl

syncHOS: syncHOS.o signal.o wavfile.o grpGL.o
	$(CC) -o $@ $^ $(LDFLAGS) -lm -liniparser -lfftw3 -lsndfile -lmgl -lstdc++ -lgomp


#ensina o make a fazer um arquivo .o a partir de um .c:
#.c.o:
# compilador C++ flags C++ -o nome_da_regra -c primeira_dependência:
%.o : %.c
	$(CXX) $(CXXFLAGS) -o $@ -c $<
```

```
#ensina o make a fazer um arquivo .o a partir de um .cc:
%.o : %.cc
# compilador C/C++ flags C/C++ -o nome_da_regra -c primeira_dependência
	$(CC) $(CFLAGS) -o $@ -c $<


.PHONY: clean

clean:
	rm -f $(OBJ)
	rm -f $(EXEC)

backup:
	zip backup.zip $(SRC) $(HDR) $(TTF) Makefile *.ini Readme.txt show_fonts imagick_type_gen
generate_spectrogram.sh

install: all
	cp $(EXEC) /usr/bin

enviro:
	@echo $(EXEC)
	@echo $(SRC)
	@echo $(HDR)
	@echo $(OBJ)
	@echo $(TTF)
	@echo $(VPATH)
```

Arquivo **bin2wav.c**:

```
#define _BSD_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sndfile.h>
extern "C"
{
#include <iniparser.h>
}

short* samplebuffer = NULL;
unsigned char* bytebuffer = NULL;

unsigned int binfilesize;
char binfilename[255];
char inifilename[255];
FILE* binfile = NULL;

char infofilename[255];
FILE* infofile = NULL;

unsigned int numchannels = 0;
unsigned int samplingrate = 0;

unsigned int numframes = 0;

char wavfilename[255];
SNDFILE* sf;
SF_INFO info;

unsigned int i; //contador
char tmpstr[255]; //string auxiliar


char channelnames[512];
char dateinfo[512];
char rangeinfo[512];


int main(int argc,char** argv)
{
//primeiro checa se o número de argumentos está correto:
if(argc!=2)
  {
  printf("Usage: %s <bin file name prefix>\r\n",argv[0]);
  exit(EXIT_FAILURE);
  }

//depois checa se o arquivo _info.txt e o arquivo .bin estão presentes:
strcpy(binfilename,argv[1]);
strcat(binfilename,".bin");
if(!(binfile = fopen(binfilename,"rb")))
  {
  printf("File %s does not exist.\r\n",binfilename);
  exit(EXIT_FAILURE);
  }

//descobre o tamanho do arquivo .bin indo até seu final e depois retornando ao início:
```

```
fseek(binfile, 0L, SEEK_END);
binfilesize = ftell(binfile);
printf("Binary file size: %u bytes.\r\n",binfilesize);
rewind(binfile);

//cria um buffer de memoria RAM exatamente do tamanho do arquivo a ser lido:
bytebuffer = (unsigned char*)malloc(binfilesize*sizeof(char));
if(bytebuffer==NULL)
  {
  printf("Memory error. Couldn't allocate memory.\r\n");
  exit(EXIT_FAILURE);
  }

//le completamente o arquivo e coloca em um buffer de memória RAM:
i = fread(bytebuffer,1,binfilesize,binfile);
if(i!=binfilesize) { printf("Read error.\r\n"); exit(EXIT_FAILURE); };
printf("Binary file totally loaded in RAM.\r\n");

//fecha o arquivo
fclose(binfile);

strcpy(infofilename,argv[1]);
strcat(infofilename,"_info.txt");
if(!(infofile = fopen(infofilename,"rt")))
  {
  printf("File %s does not exist.\r\n",infofilename);
  exit(EXIT_FAILURE);
  }

rewind(infofile);
while(!feof(infofile))
  {
  fgets(tmpstr,255,infofile);
  if(strstr(tmpstr,"SamplingRate"))
    {
    samplingrate=atoi(strrchr(tmpstr,';')+1);
    printf("Sampling Rate: %u Samples/s.\r\n",samplingrate);
    }
  if(strstr(tmpstr,"Channel;"))
    {
    numchannels=atoi(strrchr(tmpstr,';')+1);
    printf("Number of channels: %u\r\n",numchannels);
    }
  if(strstr(tmpstr,"Channel Name;"))
    {
    strcpy(channelnames,tmpstr);
    *strrchr(channelnames,'\n')=0;
    *strrchr(channelnames,'\r')=0;
    }
  if(strstr(tmpstr,"Range;"))
    {
    strcpy(rangeinfo,tmpstr);
    *strrchr(rangeinfo,'\n')=0;
    *strrchr(rangeinfo,'\r')=0;
    }
  if(strstr(tmpstr,":"))
    {
    strcpy(dateinfo,tmpstr);
    *strrchr(dateinfo,'\n')=0;
    *strrchr(dateinfo,'\r')=0;
    }
  }
fclose(infofile);

if(samplingrate==0) { printf("Couldn't detect sampling rate.\r\n"); exit(EXIT_FAILURE); }
if(numchannels==0) { printf("Couldn't detect number of channels.\r\n"); exit(EXIT_FAILURE); }

printf("\r\nExtracted info:\r\n");
printf("Date: %s\r\n",dateinfo);
printf("%s\r\n",channelnames);
printf("%s\r\n\n",rangeinfo);

//cria o arquivo com o nome automaticamente gerado
strcpy(wavfilename,argv[1]);
strcat(wavfilename,".wav");
//configura a taxa de amostragem
info.samplerate=samplingrate;
//configura o numero de canais
info.channels=numchannels;
//configura o formato como WAV microsoft little endian
info.format=(SF_FORMAT_WAV|SF_FORMAT_PCM_16|SF_ENDIAN_LITTLE);

if(!sf_format_check(&info)) { printf("Invalid format.\r\n"); exit(EXIT_FAILURE); }

//abre o arquivo wav para escrita
sf = sf_open(wavfilename,SFM_WRITE,&info);

sf_set_string(sf,SF_STR_TITLE,channelnames);
//sf_set_string(sf,SF_STR_COPYRIGHT,this->metadata_copyright.c_str());
//sf_set_string(sf,SF_STR_SOFTWARE,this->metadata_software.c_str());
//sf_set_string(sf,SF_STR_ARTIST,this->metadata_artist.c_str());
sf_set_string(sf,SF_STR_COMMENT,rangeinfo);
sf_set_string(sf,SF_STR_DATE,dateinfo);
//sf_set_string(sf,SF_STR_ALBUM,this->metadata_album.c_str());
//sf_set_string(sf,SF_STR_LICENSE,this->metadata_license.c_str());
```

```
//sf_set_string(sf,SF_STR_TRACKNUMBER,this->metadata_tracknumber.c_str());
//sf_set_string(sf,SF_STR_GENRE,this->metadata_genre.c_str());

//calcula o número de frames:
numframes = binfilesize/(2*numchannels);
printf("Frames to be written: %u\r\n",numframes);

//calcula a duração em segundos do arquivo:
printf("Duration: %5.1f seconds.\r\n",numframes*1.0/samplingrate);

//prepara um ponteiro para o buffer onde estão os dados a serem escritos no arquivo wav
samplebuffer = (short*)bytebuffer;

//escreve no arquivo wav
printf("Written %u frames.\r\n",(unsigned int)sf_writef_short(sf,samplebuffer,numframes));

if(sf_error(sf)!=0) printf("Error: %s\r\n",sf_strerror(sf));

//fecha o arquivo wav
sf_close(sf);

//libera o buffer no final de tudo:
free(bytebuffer);

//fazer o arquivo .ini automaticamente aqui.
strcpy(inifilename,argv[1]);
strcat(inifilename,".ini");
FILE* fout = fopen(inifilename,"wt");
fprintf(fout,"[Global]\r\n\n[Turbine]\r\n\n[Generator]\r\n\n[Experiment]\r\n\n[Sensors]\r\n\n[Optical
Encoder]\r\n\n[Electric Grid]\r\n\n[COT]\r\n\n[TSA]\r\n\n[STFT]\r\n\n[CMS]\r\n\n");
fclose(fout);

dictionary* description_file = iniparser_load(inifilename);

iniparser_set(description_file,"Global:Version","1.0");
iniparser_set(description_file,"Turbine:Name","Synthetic Turbine");
iniparser_set(description_file,"Turbine:Location","Fill Location");
iniparser_set(description_file,"Turbine:Type","Fill Type");
iniparser_set(description_file,"Turbine:Guide Vanes Number","Fill Guide Vanes");
iniparser_set(description_file,"Turbine:Runner Blades Number","Fill Runner Blades");
iniparser_set(description_file,"Generator:Number Of Poles","Fill Generator Poles");
iniparser_set(description_file,"Generator:Nominal Frequency","Fill Nominal Frequency");
iniparser_set(description_file,"Experiment:Power Setting","Fill Power Setting");
iniparser_set(description_file,"Experiment:Guide Vane Openning","Fill Vane Openning");
iniparser_set(description_file,"Sensors:Number of Accelerometers","2");
for(unsigned int i=0;i<2;i++)
   {
   char buffer[180];
   sprintf(buffer,"Sensors:Accelerometer %02u Channel",i+1);
   char buf[80];
   sprintf(buf,"%u",i);
   iniparser_set(description_file,buffer,buf);
   sprintf(buffer,"Sensors:Accelerometer %02u Location",i+1);
   iniparser_set(description_file,buffer,"N/A");
   }
iniparser_set(description_file,"Sensors:Number of Acoustic Emission Sensors","0");
for(unsigned int i=0;i<2;i++)
   {
   char buffer[180];
   sprintf(buffer,"Sensors:Acoustic Emission Sensor %02u Channel",i+1);
   char buf[80];
   sprintf(buf,"%u",i);
   iniparser_set(description_file,buffer,buf);
   sprintf(buffer,"Sensors:Acoustic Emission Sensor %02u Location",i+1);
   iniparser_set(description_file,buffer,"N/A");
   }
iniparser_set(description_file,"Optical Encoder:Optical Encoder Channel","2");
iniparser_set(description_file,"Optical Encoder:Pre-process","No");
iniparser_set(description_file,"Optical Encoder:Histeresis High Level","+0.1");
iniparser_set(description_file,"Optical Encoder:Histeresis Low Level","-0.1");
iniparser_set(description_file,"Optical Encoder:Perform TSA","No");
iniparser_set(description_file,"Electric Grid:Electric Grid Channel","3");
iniparser_set(description_file,"Electric Grid:Pre-process","No");
iniparser_set(description_file,"Electric Grid:Perform TSA","No");
iniparser_set(description_file,"COT:Use Resampling","Yes");
iniparser_set(description_file,"COT:Synchronism Channel","3");
iniparser_set(description_file,"TSA:Perform TSA Analysis","No");
iniparser_set(description_file,"TSA:Use Resampling","No");
iniparser_set(description_file,"TSA:Synchronism Channel","2");
iniparser_set(description_file,"TSA:Invert Graphics","No");
iniparser_set(description_file,"STFT:Perform STFT Analysis","Yes");
iniparser_set(description_file,"STFT:STFT Window Type","Hanning");
iniparser_set(description_file,"STFT:Window Size","512");
iniparser_set(description_file,"STFT:Window Overlap Index","2");
iniparser_set(description_file,"STFT:Generate Graphic","No");
iniparser_set(description_file,"STFT:Scaled Graphic Output","No");
iniparser_set(description_file,"STFT:Invert Graphic","Yes");
iniparser_set(description_file,"STFT:Use RMS","No");
iniparser_set(description_file,"STFT:Use PWR","Yes");
iniparser_set(description_file,"STFT:Use Density","Yes");
iniparser_set(description_file,"STFT:HighPass Cutoff Frequency","5000");
iniparser_set(description_file,"CMS:Perform CMS Analysis","Yes");
iniparser_set(description_file,"CMS:Generate Graphic","No");
iniparser_set(description_file,"CMS:Scaled Graphic Output","No");
iniparser_set(description_file,"CMS:Invert Graphic","No");
```

```
fout = fopen(inifilename,"wt");
iniparser_dump_ini(description_file,fout);
fclose(fout);
iniparser_freedict(description_file);

return(0);
}
```

Arquivo **cot-tsa.cc**:

```
//faz resampling do sinal de entrada, garantindo que o sinal resultante tenha sempre o mesmo número de
//amostras a cada volta da turbina. Faz também o número de amostras por volta ser sempre potência de 2

/* constantes necessárias */

const double PI=3.14159265358979311599796 3;

/* includes necessários */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
extern "C"
{
#include <iniparser.h>
}
#include <vector>
#include <samplerate.h>
#include <sndfile.h>
#include "wavfile.h"
#include "matrices.h"
#include "signal.h"
#include "grpGL.h"


/* Variáveis Globais */
/
**********************************************************************************************************
*****/
string turbinename;
string turbinepower;
string turbinetype;
char NAstring[80]="N/A";
double turbinenominalperiod;      //o período nominal de rotação da turbina em segundos
unsigned int nguidevanes;         //número de palhetas do distribuidor
unsigned int nrunnerblades;       //número de pás do rotor
unsigned int ngeneratorpoles;     //número de polos do gerador de tensão
double generatornominalfrequency; //a frequência nominal da energia elétrica gerada
double turbinenominalrpm;         //a velocidade nominal de rotação da turbina em RPM
double turbinenominalfrequency;   //a velocidade nominal de rotação da turbina em rotações por segundo ou Hz
double blade_passing_frequency;   //a frequência de passagem das pás
double vane_passing_frequency;    //a frequência de passagem das palhetas

//variaveis globais de configuração do arquivo de dados .WAV:
string wavfilename;                                 //o nome do arquivo no sistema de arquivos
unsigned int num_channels;                          //o número de canais distintos que há no arquivo WAV
unsigned long int num_frames;                       //o número de frames (amostras) que há no arquivo WAV
unsigned int samplingrate;                          //a taxa de amostragem em amostras por segundo
double duration;                                    //a duração em segundos do arquivo

unsigned int num_accelerometers;                    //número de canais que armazenam sinais provenientes de
acelerômetros
vector <unsigned int> channel_accelerometer;    //um vetor de inteiros com os números dos canais dos acelerômetros
vector <string> location_accelerometer;             //um vetor de strings com as localizações de cada um dos
acelerômetros
unsigned int num_AE_sensors;                        //o número de sensores de emissão acústica
vector <unsigned int> channel_AE_sensor;        //um vetor com os números dos canais dos sensores de emissão
acústica
vector <string> location_AE_sensor;             //um vetor de strings com as localizações dos de cada um dos senores
de emissão acústica

unsigned int channel_optical_encoder;               //o canal do encoder optico
unsigned int channel_electric_grid;                 //o canal que grava a forma de onda da rede elétrica

unsigned int total_signal_info_channels;            //o número total de canais com informação vibroacustica
unsigned int total_signal_sync_channels;            //o número total de canais com informação de sincronismo
vector <unsigned int> total_signal_info_channel;//um vetor com os números dos canais de acelerômetros
vector <string> total_signal_location_sensor;    //um vetor de strings com as localizações de todos os sensores de
sinal vibroacustico


unsigned int synchronism_channel; //guarda o número do canal que vai servir para sincronizar os sinais
double histeresis_high; //os níveis de histerese para serem usados no smith trigger
double histeresis_low;

double optical_frequency;
double electric_frequency;
unsigned int ncycles;
```

```
vector <unsigned long int> cycle_start_point;

double max_cycle_lenght;
double min_cycle_lenght;
unsigned int new_samplingrate;
unsigned int new_samples_per_cycle;
unsigned int nominal_samples_per_turn;
unsigned int window_lenght; //o comprmento da janela que está configurado para fazer o espectrograma

//Início do programa principal:
/
*****************************************************************************************************
*****/
int main(int argc,char** argv)
{
         /*************************** Inicialização ***************************************************/
         //primeiro checa se o número de argumentos está correto:
         if(argc!=2)   {  printf("Usage: %s <inputfile without .wav>\r\n",argv[0]);  exit(EXIT_FAILURE);   }

         //carrega o arquivo txt contendo a descrição do que há no arquivo .wav e suas análises:
         dictionary* description_file = iniparser_load((string(argv[1])+".ini").c_str());
         if(!description_file) {  /*printf("File %s not found. Aborting.",(string(argv[1])+".ini").c_str()); */
exit(EXIT_FAILURE);   }

         nguidevanes = iniparser_getint(description_file,"Turbine:Guide Vanes Number",0);
         nrunnerblades = iniparser_getint(description_file,"Turbine:Runner Blades Number",0);
         ngeneratorpoles = iniparser_getint(description_file,"Generator:Number Of Poles",0);
         generatornominalfrequency = iniparser_getdouble(description_file,"Generator:Nominal Frequency",0.0);
         turbinename = iniparser_getstring(description_file,"Turbine:Name",NAstring);
         turbinetype = iniparser_getstring(description_file,"Turbine:Type",NAstring);
         turbinepower = iniparser_getstring(description_file,"Experiment:Power Setting",NAstring);

         //acelerômetros:
         num_accelerometers = iniparser_getint(description_file,"Sensors:Number of Accelerometers",0);

         for(unsigned int i=0;i<num_accelerometers;i++)
            {
            char tmpstr[256];
            sprintf(tmpstr,"Accelerometer %02u Channel",i+1);

channel_accelerometer.push_back(iniparser_getint(description_file,string("Sensors:"+string(tmpstr)).c_str(),0));
            sprintf(tmpstr,"Accelerometer %02u Location",i+1);

location_accelerometer.push_back(iniparser_getstring(description_file,string("Sensors:"+string(tmpstr)).c_str(),0));
            }

         //sensores de emissão acústica:
         num_AE_sensors = iniparser_getint(description_file,"Sensors:Number of Acoustic Emission Sensors",0);
         for(unsigned int i=0;i<num_AE_sensors;i++)
            {
            char tmpstr[256];
            sprintf(tmpstr,"Acoustic Emission Sensor %02u Channel",i+1);

channel_AE_sensor.push_back(iniparser_getint(description_file,string("Sensors:"+string(tmpstr)).c_str(),0));
            sprintf(tmpstr,"Acoustic Emission Sensor %02u Location",i+1);

location_AE_sensor.push_back(iniparser_getstring(description_file,string("Sensors:"+string(tmpstr)).c_str(),0));
            }

         //encoder ótico:
         channel_optical_encoder = iniparser_getint(description_file,"Optical Encoder:Optical Encoder Channel",0);

         //sincronismo pela rede elétrica:
         channel_electric_grid = iniparser_getint(description_file,"Electric Grid:Electric Grid Channel",0);

         //qual será o sinal usado para sincronismo
         synchronism_channel = iniparser_getint(description_file,"COT:Synchronism Channel",0);

         //lê os valores dos níveis de histerese do smith trigger
         histeresis_high=iniparser_getdouble(description_file,"Optical Encoder:histeresis high level",0.0);
         histeresis_low=iniparser_getdouble(description_file,"Optical Encoder:histeresis low level",0.0);

         window_lenght=iniparser_getint(description_file,"STFT:Window Size",0);

         iniparser_freedict(description_file);
         /*************************** Fim da Inicialização ********************************************/

         //algumas variaveis dá pra calcular o valor
         turbinenominalrpm = 120.0*generatornominalfrequency/ngeneratorpoles;
         turbinenominalfrequency = turbinenominalrpm/60.0;
         turbinenominalperiod = 1.0/turbinenominalfrequency;
         blade_passing_frequency = turbinenominalfrequency*nrunnerblades;
         vane_passing_frequency = turbinenominalfrequency*nguidevanes;

         //imprime algumas informações sobre o conjunto analizado:
         printf("\r\n************************************** ANALYSIS CONFIGURATION
**********************************\r\n");
         printf("Generator:\r\n");
         printf("Poles: %u  Nominal Frequency: %4.1f Hz. -> ",ngeneratorpoles,generatornominalfrequency);
         printf("Nominal Speed: %5.1f rpm.\r\n\n",turbinenominalrpm);
         printf("Turbine: %s type %s. Power Setting:
%s\r\n",turbinename.c_str(),turbinetype.c_str(),turbinepower.c_str());
         printf("Runner Blades: %u  Guide Vanes: %u.\r\n",nrunnerblades,nguidevanes);
         printf("Nominal Frequency: %9.7f Hz. Nominal Period: %f
seconds.\r\n",turbinenominalfrequency,turbinenominalperiod);
```

```
         printf("Blade Passing Frequency: %9.7f Hz.  Guide Vane Passing Frequency: %9.7f
Hz.\r\n",blade_passing_frequency,vane_passing_frequency);

         /************************************ Arquivo de dados de entrada
***********************************************/
         printf("\r\n********************************************** INPUT FILE
*********************************************\r\n");

         //imprime o nome do arquivo que será processado
         printf("\r\n");
         wavfilename=string(argv[1])+".wav";
         printf("Processing file: %s\r\n",wavfilename.c_str());
         WavFile inputfile(wavfilename.c_str()); //cria um objeto da classe WavFile
         num_channels = inputfile.get_num_channels();
         samplingrate = inputfile.get_samplingrate();
         nominal_samples_per_turn = samplingrate/(turbinenominalfrequency);
         printf("WAV File: %u channels. Samplingrate: %u samples/second. ",num_channels,samplingrate);
         //calcula a duração em segundos do arquivo:
         num_frames = inputfile.get_num_samples();
         duration = inputfile.get_duration();
         printf("Duration: %8.4f seconds (%u complete turns).\r\n",duration,(unsigned
int)floor(duration*turbinenominalfrequency));
         printf("Datarate: %u bauds. ",inputfile.get_bits_per_second());
         printf("Bits per sample: %u bits. ",inputfile.get_bits_per_sample());
         printf("File size: %7.1f MBytes.\r\n",(double)inputfile.get_filesize()/(1024*1024));
         printf("Recording date: %s\r\n",inputfile.get_metadata_date());

         printf("\r\nChannel Mapping:\r\n");
         for(unsigned int i=0; i<num_channels; i++)
            {
            printf("Channel %02u -> %s. ",i,inputfile.get_channel_name(i));
            if(i==channel_optical_encoder) printf("(Sync Optical)");
            if(i==channel_electric_grid) printf("(Sync Electric)");
            if((i!=channel_electric_grid)&&(i!=channel_optical_encoder)) printf("(Vibroacoustic)");
            printf("\r\n");
            }

         printf("\r\n************************************ PROCESSING STRATEGY
*****************************************\r\n");

         total_signal_info_channels = num_accelerometers + num_AE_sensors;
         printf("Total Signal Avaliable Channels: %u - ",total_signal_info_channels);
         total_signal_sync_channels = 0;
         if(channel_optical_encoder<num_channels) total_signal_sync_channels++;
         if(channel_electric_grid<num_channels) total_signal_sync_channels++;
         printf("Total Sync Avaliable Channels: %u\r\n",total_signal_sync_channels);
         if(total_signal_info_channels+total_signal_sync_channels>num_channels)
           {
           printf("Something is wrong! Declared vibroacoustic signal channels: %u  - Declared sync signal channels:
%u ",total_signal_info_channels,total_signal_sync_channels);
           printf("Your .WAV actual number of channels is %u. \r\nReview your analyzer.ini file.\r\n",num_channels);
           exit(EXIT_FAILURE);
           }

if((synchronism_channel!=channel_optical_encoder)&&(synchronism_channel!=channel_electric_grid))
  {
  printf("You should choose a synchronism channel to be used as reference. Review your .ini file. Aborting.");
  exit(EXIT_FAILURE);
  }

//carrega somente o canal que será usado para fazer o sincronismo e resampling.
printf("Loading synchronism channel...%s\r\n",inputfile.get_channel_name(synchronism_channel)); fflush(stdout);
electric_signal* synchro = new electric_signal(wavfilename.c_str(),synchronism_channel);
printf("\r\n");
synchro->removeDC();
synchro->maximize();
vector <double> positive_transition_points; //um vetor que vai armazenar os instantes exatos das transições
positivas

if(synchronism_channel==channel_optical_encoder) //se o sincronismo for pelo encoder ótico
  {
  positive_transition_points.clear(); //limpa o vetor inicialmente;
  printf("Performing Smith Trigger [%4.2f %4.2f]...",histeresis_low,histeresis_high);
  synchro->SmithTrigger(histeresis_high,histeresis_low); printf("Done.\r\n"); fflush(stdout);
  optical_frequency = synchro->MeasureDigitalFrequency();
  ncycles = synchro->CountCycles();
  unsigned long int* tmp = synchro->CyclesStartPoints();
  cycle_start_point.resize(ncycles+1);
  for(unsigned int i=0;i<=ncycles;i++) { positive_transition_points.push_back(tmp[i]); };
  free(tmp);
  ncycles = positive_transition_points.size()-1;
  printf("Detected %u cycles. Optical  frequency: %f Hz. (%f
Hz)\r\n",ncycles,optical_frequency,optical_frequency*(ngeneratorpoles/2));
  }

if(synchronism_channel==channel_electric_grid) //se o sincronismo for pelo gerador elétrico
  {
  //calcula o valor nominal da duração de um ciclo senoidal em número de amostras:
  unsigned int nominal_cyle_num_samples = (synchro->get_samplingrate()/generatornominalfrequency);

  //primeiro conta o número de transições positivas:
  printf("Counting positive transitions...\r\n"); fflush(stdout);
  positive_transition_points.clear(); //limpa o vetor inicialmente;
  for(long unsigned int i=0;i<synchro->get_num_samples()-1;i++) //varre o sinal de sincronismo inteiro
           if(((*synchro)[i]<0.0)&&((*synchro)[i+1]>=0.0)) //se encontrar uma transição positiva
```

```
                    {
                    if(positive_transition_points.empty())
                      {
                      positive_transition_points.push_back((double)(i)-((*synchro)[i]/((*synchro)[i+1]-(*synchro)
[i])));
                      }
                    else if((i-positive_transition_points.back())>(nominal_cyle_num_samples/1.8))
                              positive_transition_points.push_back((double)(i)-((*synchro)[i]/((*synchro)[i+1]-
(*synchro)[i])));
                    }
  ncycles=positive_transition_points.size()-1;
  electric_frequency = (double)ncycles/(synchro->get_sample_duration()*(positive_transition_points.back()-
positive_transition_points.front())); //calcula a frequencia segundo a rede elétrica
  electric_frequency/=(ngeneratorpoles/2.0); //corrige
  printf("Detected %u cycles. Electric frequency: %f Hz. (%f
Hz)\r\n\n",ncycles,electric_frequency,electric_frequency*(ngeneratorpoles/2));
  }

  {
  double* data = (double*)malloc(sizeof(double)*positive_transition_points.size());

  if(synchronism_channel==channel_electric_grid)
    for(unsigned int i=0;i<positive_transition_points.size()-1;i++)
         data[i]=(((double)samplingrate/(positive_transition_points[i+1]-
positive_transition_points[i])/generatornominalfrequency)-1.0)*1000000.0;

  if(synchronism_channel==channel_optical_encoder)
    for(unsigned int i=0;i<positive_transition_points.size()-1;i++)
         data[i]=((((double)samplingrate/(positive_transition_points[i+1]-
positive_transition_points[i]))/turbinenominalfrequency)-1.0)*1000000.0;

  for(unsigned int i=0;i<positive_transition_points.size()-1;i++)
          if((data[i]<-10000)||(data[i]>10000))
          printf("%u %f %f %f\r\n",i,data[i],positive_transition_points[i+1],positive_transition_points[i]);

  GL2D* graPH = new GL2D(3740,984);
  graPH->SetMainTitle("Machine Speed Deviation");
  graPH->SetSubTitle(turbinename+" "+turbinepower+" "+synchro->get_date());
  graPH->SetXrange(0.0,duration);
  graPH->SetYscaleTitle("rotational speed deviation (ppm)");
  graPH->SetXscaleTitle("time (s)");
  graPH->SaveToFile(string(argv[1])+"-Machine_Speed_Deviation");
  graPH->showGraph();
  graPH->PlotSignal(data,positive_transition_points.size()-1,"");
  delete graPH;
  free(data);
  }
delete(synchro); //o sinal de sincronismo não é mais necessário, pois agora
//o vetor positive_transition_points já armazena com precisão as transições positivas

//começa a fazer o estudo das flutuações de frequência:
max_cycle_lenght=positive_transition_points[1]-positive_transition_points[0];
min_cycle_lenght=positive_transition_points[1]-positive_transition_points[0];

for(unsigned int i=0;i<positive_transition_points.size()-1;i++)
   {
   double cycle_lenght;
   cycle_lenght=positive_transition_points[i+1]-positive_transition_points[i];
   if(synchronism_channel==channel_optical_encoder)
     {
     if(cycle_lenght<0.8*nominal_samples_per_turn) printf("Warning: cycle too short at %8.3fs (cycle %u)\r\n",
(double)positive_transition_points[i]/samplingrate,i);
     if(cycle_lenght>1.2*nominal_samples_per_turn) printf("Warning: cycle too long at %8.3fs (cycle %u)\r\n",
(double)positive_transition_points[i]/samplingrate,i);
     }
   if(synchronism_channel==channel_electric_grid)
     {
     if(cycle_lenght<(1.6*nominal_samples_per_turn/ngeneratorpoles)) printf("Warning: cycle too short at %8.3fs
(cycle %u)\r\n",(double)positive_transition_points[i]/samplingrate,i);
     if(cycle_lenght>(2.4*nominal_samples_per_turn/ngeneratorpoles)) printf("Warning: cycle too long at %8.3fs
(cycle %u)\r\n",(double)positive_transition_points[i]/samplingrate,i);
     }
   if(max_cycle_lenght<cycle_lenght) max_cycle_lenght=cycle_lenght;
   if(min_cycle_lenght>cycle_lenght) min_cycle_lenght=cycle_lenght;
   }
printf("Longest cycle has %9.3f samples. Shortest cycle has %9.3f samples.\r\n",max_cycle_lenght,min_cycle_lenght);

//calcula qual será o novo comprimento do ciclo:
new_samples_per_cycle = (1+floor(max_cycle_lenght/window_lenght))*window_lenght;
new_samplingrate = new_samples_per_cycle*turbinenominalfrequency;
if(synchronism_channel==channel_electric_grid) //se a correção for feita a cada ciclo de 60Hz
  {
  //corrige multiplicando a taxa de amostragem pela metade de polos deo gerador
  new_samplingrate = new_samples_per_cycle*generatornominalfrequency;
  printf("Resampling to equalize all cycles to %u samples (%9.2f samples/s or %8.2f
samples/turn)...\r\n",new_samples_per_cycle,(double)new_samples_per_cycle*generatornominalfrequency,
(double)new_samples_per_cycle*ngeneratorpoles/2);
  }
if(synchronism_channel==channel_optical_encoder) printf("Resampling to equalize all cycles to %u samples (%9.2f
samples/s)...\r\n",new_samples_per_cycle,new_samples_per_cycle*turbinenominalfrequency);
/*-------------------------------------------------------------------------------------------------------------*/

/*****************************USANDO A FULL API DA LIBSAMPLERATE*****************************/
/************************************ APROPRIADA PARA TAXA DE REAMOSTRAGEM VARIÁVEL ************/
/*********************************Fazendo a reamostragem multicanal ***********************/
```

```
//sem carregar os sinais todos de uma vez pra memória pois fica pesado
//demais e não é qualquer computador que vai conseguir executar

//aloca dois buffers, um para entrada de dados diretamente do arquivo original
//e o outro para a saída de dados para o arquivo novo e reamostrado
//já associa estes buffers com os arquivos .WAV:

float* inputsamples = (float*)malloc(2*sizeof(float)*max_cycle_lenght*num_channels);
float* outputsamples = (float*)malloc(sizeof(float)*new_samples_per_cycle*num_channels);

SNDFILE *in_sf; //são dois descritores (ponteiros) para arquivos, da biblioteca sndfile
SNDFILE *out_sf;

SF_INFO in_info; //são duas struct de configuração e informação da biblioteca sndfile
SF_INFO out_info;

//abre o arquivo in_sf para leitura:
in_info.format = 0;
in_sf = sf_open(wavfilename.c_str(),SFM_READ,&in_info);

//abre o arquivo out_sf para escrita:
out_info.samplerate=new_samplingrate;
out_info.format=(SF_FORMAT_WAV|SF_FORMAT_DOUBLE);
out_info.channels=num_channels;
out_sf = sf_open((char*)string(string(argv[1])+"-COT.wav").c_str(),SFM_WRITE,&out_info);

//copia o metadata:
for(unsigned int i=SF_STR_FIRST;i<=SF_STR_LAST;i++)
   {
   sf_set_string(out_sf,i,sf_get_string(in_sf,i));
   };

//configura a libsamplerate
SRC_DATA config_libsamplerate;
int error;
SRC_STATE* state_libsamplerate = src_new(SRC_SINC_BEST_QUALITY,num_channels,&error);

unsigned long int written_frames;
written_frames=0;

//descarta os primeros frames que não fazem parte de nenhum segmento inteiro:
sf_readf_float(in_sf,inputsamples,ceil(positive_transition_points.front())); //tudo que vem antes da primeira
transição é descartado

for(unsigned int i=0;i<positive_transition_points.size()-1;i++) //varre os segmentos detectados um por um
         {
         //lê o segmento e armazena no inputbuffers
         //lê o segmento diretamente do arquivo original
         unsigned int read_frames = sf_readf_float(in_sf,inputsamples,floor(positive_transition_points[i+1])-
floor(positive_transition_points[i]));
         //configura o pointer para os frames de entrada:
         config_libsamplerate.data_in = inputsamples;
         //o número de frames de entrada é igual a diferença do início do ciclo seguinte menos o ciclo corrente
         config_libsamplerate.input_frames = read_frames;
         //configura o pointer para os frames de saída:
         config_libsamplerate.data_out = outputsamples;
         //o novo número de frames por segmento já foi calculado:
         config_libsamplerate.output_frames = new_samples_per_cycle;
         //a taxa de reamostragem atualizada para o segmento é calculada:
         config_libsamplerate.src_ratio = (double)new_samples_per_cycle/(positive_transition_points[i+1]-
positive_transition_points[i]);
         //diz que ainda não é o ultimo chunk de dados a serem alimentados:
         config_libsamplerate.end_of_input = 0;

         //processa:
         src_process(state_libsamplerate,&config_libsamplerate);
         if(error!=0) { printf("Error at segment %u: %s\r\n",i,src_strerror(error)); }


         //escreve o segmento reamostrado no arquivo de saída:
         if(sf_writef_float(out_sf,outputsamples,config_libsamplerate.output_frames_gen)!
=config_libsamplerate.output_frames_gen) printf("problema!\r\n");
         written_frames+=config_libsamplerate.output_frames_gen;
         printf("\rResampling segment %u...%5.1f%%",i,(100.0*i)/(ncycles)); fflush(stdout);
         }
printf("\r\n");
//empurra mais amostras para a reamostragem para poder recuperar as amostras que ficam presas no delay
for(unsigned int i=0;i<2*max_cycle_lenght*num_channels;i++) inputsamples[i]=0.0;
sf_readf_float(in_sf,inputsamples,max_cycle_lenght);
config_libsamplerate.end_of_input = 1; //é o ultimo segmento
src_process(state_libsamplerate,&config_libsamplerate);
if(error!=0) { printf("Error at final: %s\r\n",src_strerror(error)); }
written_frames+=sf_writef_float(out_sf,outputsamples,ncycles*new_samples_per_cycle-written_frames);
if(written_frames!=ncycles*new_samples_per_cycle) printf("PROBLEMA NA REAMOSTRAGEM!\r\n ");
printf("Written %lu frames in file %s\r\n",written_frames,(char*)string(string(argv[1])+"-COT.wav").c_str());

sf_close(out_sf); //fecha o arquivo de saída reamostrado
sf_close(in_sf); //fecha o arquivo de entrada original
src_delete(state_libsamplerate);
free(inputsamples); //desaloca os buffers de entrada e saída
free(outputsamples);

//carrega o arquivo txt contendo a descrição do que há no arquivo .wav e suas análises:
description_file = iniparser_load((string(argv[1])+".ini").c_str());
```

```
char buffer[100];
sprintf(buffer,"%u",samplingrate);
iniparser_set(description_file,"COT:Original Samplingrate",buffer);
sprintf(buffer,"%u",new_samplingrate);
iniparser_set(description_file,"COT:New Samplingrate",buffer);
if(synchronism_channel==channel_electric_grid) new_samples_per_cycle*=(ngeneratorpoles/2);
sprintf(buffer,"%u",new_samples_per_cycle);
iniparser_set(description_file,"COT:Samples per turn",buffer);

FILE* fout = fopen((string(argv[1])+".ini").c_str(),"wt");
iniparser_dump_ini(description_file,fout);
fclose(fout);

iniparser_freedict(description_file);

/*****************************************************************************************************/
/*********************************AQUI COMEÇA A SEPARAÇÃO*********************************************/
/* A parte cicloestacionária de primeira ordem (periódica) vai ser calculada pela TSA*****************/
printf("Performing cyclostationary separation by Time(Angle) Sample Averaging...\r\n");

//aloca buffers para fazer as médias
//pode ser um buffer somente, pois não há necessidade de desfazer o interleaving:
printf("Allocating buffers...\r\n");
double* meanbuffer = (double*)malloc(sizeof(double)*num_channels*new_samples_per_cycle);
double* periodicbuffer = (double*)malloc(sizeof(double)*num_channels*new_samples_per_cycle);
double* residualbuffer = (double*)malloc(sizeof(double)*num_channels*new_samples_per_cycle);
double* receivebuffer = (double*)malloc(sizeof(double)*num_channels*new_samples_per_cycle);
//zera o buffer, pois ele será um acumulador:
for(unsigned int i=0;i<num_channels*new_samples_per_cycle;i++) meanbuffer[i]=0.0;

//configura a libsndfile para um arquivo de leitura, o original:
SNDFILE *original_sf;
SF_INFO original_info;

//e configura a libsndfile para dois arquivos de escrita, um é a parte periodica e o outro é a parte residual
SNDFILE *periodic_sf;
SF_INFO periodic_info;
SNDFILE *residual_sf;
SF_INFO residual_info;

//abre o arquivo de entrada original (já com a COT):
printf("Opening raw input signal file...\r\n");
original_info.format = 0;
original_sf = sf_open((char*)string(string(argv[1])+"-COT.wav").c_str(),SFM_READ,&original_info);

//abre os arquivos de saída para escrita:
printf("Opening output periodic (CS1) and residual (CS2+)+noise files...\r\n");
periodic_info.samplerate=new_samplingrate;
periodic_info.format=(SF_FORMAT_WAV|SF_FORMAT_DOUBLE);
periodic_info.channels=num_channels;
periodic_sf = sf_open((char*)string(string(argv[1])+"-COT-periodic.wav").c_str(),SFM_WRITE,&periodic_info);
residual_info.samplerate=new_samplingrate;
residual_info.format=(SF_FORMAT_WAV|SF_FORMAT_DOUBLE);
residual_info.channels=num_channels;
residual_sf = sf_open((char*)string(string(argv[1])+"-COT-residual.wav").c_str(),SFM_WRITE,&residual_info);

//copia o metadata:
for(unsigned int i=SF_STR_FIRST;i<=SF_STR_LAST;i++)
   {
   sf_set_string(periodic_sf,i,sf_get_string(original_sf,i));
   sf_set_string(residual_sf,i,sf_get_string(original_sf,i));
   };

//calcula o número de voltas inteiras que a turbina realizou:
unsigned int n_turns = original_info.frames/new_samples_per_cycle; //retorna o numero de frames no arquivo
//acumula no buffer a soma das voltas
printf("Averaging %u turns...\r\n",n_turns);
for(unsigned int i=0;i<n_turns;i++)
   {
   if(sf_readf_double(original_sf,receivebuffer,new_samples_per_cycle)!=new_samples_per_cycle) printf("Problem in
cycle %u!!!!",i);
   printf("\rAccumulating turn %u...",i); fflush(stdout);
   for(unsigned int j=0;j<num_channels*new_samples_per_cycle;j++) meanbuffer[j]+=receivebuffer[j];
   }
//divide todos pelo número de voltas, toma a média:
printf("\r\nAveraging..."); fflush(stdout);
for(unsigned int i=0;i<num_channels*new_samples_per_cycle;i++) meanbuffer[i]/=n_turns;
printf("Done.\r\n");
sf_seek(original_sf,0,SEEK_SET); //volta ao começo do arquivo

printf("Separating...");
unsigned int read_frames;
unsigned int turn_counter=0;
do
{
read_frames = sf_readf_double(original_sf,receivebuffer,new_samples_per_cycle);
for(unsigned int i=0;i<num_channels*read_frames;i++)
   {
   periodicbuffer[i]=meanbuffer[i];
   residualbuffer[i]=receivebuffer[i]-meanbuffer[i];
   }
sf_writef_double(periodic_sf,periodicbuffer,read_frames);
sf_writef_double(residual_sf,residualbuffer,read_frames);
turn_counter++;
printf("\rSeparating...turn %u (%5.1f%%)",turn_counter,(100.0*turn_counter)/n_turns);
```

```
}while(read_frames!=0);
printf("\rSeparating...Done.            \r\n");

printf("Closing Files...\r\n");
//fecha os arquivos e libera a memória do buffer
sf_close(original_sf);
sf_close(periodic_sf);
sf_close(residual_sf);
printf("Freeing memory...\r\n");
free(meanbuffer);
free(periodicbuffer);
free(residualbuffer);
free(receivebuffer);
printf("Pre-processing complete!\r\n");
return(EXIT_SUCCESS);
}
```

Arquivo **spectrogram.cc**:

```
/* Produz o espectrograma do sinal de audio em .WAV */
/* e gera como resultado uma matriz de complexos em arquivo */

/* constantes necessárias */

const double PI = 3.14159265358979311599796 3;

/* includes necessários */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
extern "C" {
#include <iniparser.h>
}
#include <vector>
#include "wavfile.h"
#include "matrices.h"
#include "signal.h"
#include "grpGL.h"
#include "STFT-iSTFT.h"
#include "CMS-iCMS.h"
#include "CMC-iCMC.h"

/* Variáveis Globais */
/
**********************************************************************************************************************
*****/
string turbinename;
string turbinepower;
string turbinetype;
char NAstring[80] = "N/A";
double turbinenominalperiod; //o período nominal de rotação da turbina em segundos
unsigned int nguidevanes;         //número de palhetas do distribuidor
unsigned int nrunnerblades;       //número de pás do rotor
unsigned int ngeneratorpoles;     //número de polos do gerador de tensão
double generatornominalfrequency; //a frequência nominal da energia elétrica gerada
double turbinenominalrpm;    //a velocidade nominal de rotação da turbina em RPM
double turbinenominalfrequency; //a velocidade nominal de rotação da turbina em rotações por segundo ou Hz
double blade_passing_frequency;   //a frequência de passagem das pás
double vane_passing_frequency;    //a frequência de passagem das palhetas

//variaveis globais de configuração do arquivo de dados .WAV:
string wavfilename;                  //o nome do arquivo no sistema de arquivos
unsigned int num_channels;  //o número de canais distintos que há no arquivo WAV
unsigned long int num_frames; //o número de frames (amostras) que há no arquivo WAV
unsigned int samplingrate;        //a taxa de amostragem em amostras por segundo
double duration;                                 //a duração em segundos do arquivo

unsigned int num_accelerometers; //número de canais que armazenam sinais provenientes de acelerômetros
vector<unsigned int> channel_accelerometer; //um vetor de inteiros com os números dos canais dos acelerômetros
vector<string> location_accelerometer; //um vetor de strings com as localizações de cada um dos acelerômetros
unsigned int num_AE_sensors;          //o número de sensores de emissão acústica
vector<unsigned int> channel_AE_sensor; //um vetor com os números dos canais dos sensores de emissão acústica
vector<string> location_AE_sensor; //um vetor de strings com as localizações dos de cada um dos senores de emissão
acústica

unsigned int channel_optical_encoder;                     //o canal do encoder optico
unsigned int channel_electric_grid; //o canal que grava a forma de onda da rede elétrica

unsigned int total_signal_info_channels;//o número total de canais com informação vibroacustica
unsigned int total_signal_sync_channels;//o número total de canais com informação de sincronismo
vector<unsigned int> total_signal_info_channel;  //um vetor com os números dos canais de acelerômetros
vector<string> total_signal_location_sensor;//um vetor de strings com as localizações de todos os sensores de sinal
vibroacustico

//variaveis globais da análise STFT, CMS e CMC:
double stft_nominal_period;
double cms_nominal_frequency;
string window_name;
unsigned int window_size;
unsigned int overlap_index;
```

```
vector<electric_signal> channel; //um vetor de objetos do tipo signal, representando todos os sinais de entrada

unsigned int samples_per_turn;
unsigned int original_samplingrate;
bool calculate_average;
bool use_RMS_on_STFT;
bool use_PWR_on_STFT;
bool use_Modulus_on_STFT=false;
bool dump_overlapped_windows;
bool use_density;
double highpassfrequency;
bool draw_inverted;

bool process_residue;
bool process_periodic;

//Início do programa principal:
/
**********************************************************************************************************************
*****/
int main(int argc, char** argv) {
        /***************************** Inicialização ******************************************************/
//primeiro checa se o número de argumentos está correto:
        if (argc < 2) {
                printf("Usage: %s <inputfile without .wav> [optional flags]\r\n",
                                        argv[0]);
                exit(EXIT_FAILURE);
        }

        calculate_average = true;
        dump_overlapped_windows = false;
        process_periodic=false;
        process_residue=false;
        for (int i = 1; i < argc; i++)
                if (strcmp(argv[i], "-na") == 0)
                        calculate_average = false;
        for (int i = 1; i < argc; i++)
                if (strcmp(argv[i], "-dow") == 0)
                        dump_overlapped_windows = true;
        for (int i = 1; i < argc; i++)
                if (strcmp(argv[i], "-periodic") == 0)
                                        process_periodic = true;
        for (int i = 1; i < argc; i++)
                if (strcmp(argv[i], "-residue") == 0)
                                                process_residue = true;

//carrega o arquivo txt contendo a descrição do que há no arquivo .wav e suas análises:
        dictionary* description_file = iniparser_load((string(argv[1]) + ".ini").c_str());
        if (!description_file) { /*printf("File %s not found. Aborting.",(string(argv[1])+".ini").c_str()); */
                exit(EXIT_FAILURE);
        }

        nguidevanes = iniparser_getint(description_file,
                        "Turbine:Guide Vanes Number", 0);
        nrunnerblades = iniparser_getint(description_file,
                        "Turbine:Runner Blades Number", 0);
        ngeneratorpoles = iniparser_getint(description_file,
                        "Generator:Number Of Poles", 0);
        generatornominalfrequency = iniparser_getdouble(description_file,
                        "Generator:Nominal Frequency", 0.0);
        turbinename = iniparser_getstring(description_file, "Turbine:Name",
                        NAstring);
        turbinetype = iniparser_getstring(description_file, "Turbine:Type",
                        NAstring);
        turbinepower = iniparser_getstring(description_file,
                        "Experiment:Power Setting", NAstring);

//acelerômetros:
        num_accelerometers = iniparser_getint(description_file,"Sensors:Number of Accelerometers", 0);

        for (unsigned int i = 0; i < num_accelerometers; i++) {
                char tmpstr[256];
                sprintf(tmpstr, "Accelerometer %02u Channel", i + 1);
                channel_accelerometer.push_back(
                                        iniparser_getint(description_file,
                                                        string("Sensors:" + string(tmpstr)).c_str(), 0));
                sprintf(tmpstr, "Accelerometer %02u Location", i + 1);
                location_accelerometer.push_back(
                                        iniparser_getstring(description_file,
                                                        string("Sensors:" + string(tmpstr)).c_str(), 0));
        }

//sensores de emissão acústica:
        num_AE_sensors = iniparser_getint(description_file,
                        "Sensors:Number of Acoustic Emission Sensors", 0);
        for (unsigned int i = 0; i < num_AE_sensors; i++) {
                char tmpstr[256];
                sprintf(tmpstr, "Acoustic Emission Sensor %02u Channel", i + 1);
                channel_AE_sensor.push_back(
                                        iniparser_getint(description_file,
                                                        string("Sensors:" + string(tmpstr)).c_str(), 0));
                sprintf(tmpstr, "Acoustic Emission Sensor %02u Location", i + 1);
                location_AE_sensor.push_back(
                                        iniparser_getstring(description_file,
```

```
                                                        string("Sensors:" + string(tmpstr)).c_str(), 0));
        }

//encoder ótico:
        channel_optical_encoder = iniparser_getint(description_file,
                            "Optical Encoder:Optical Encoder Channel", 0);

//sincronismo pela rede elétrica:
        channel_electric_grid = iniparser_getint(description_file,"Electric Grid:Electric Grid Channel", 0);

        window_name = iniparser_getstring(description_file,"STFT:STFT Window Type",NAstring);
        window_size = iniparser_getint(description_file,"STFT:Window Size", 0);
        overlap_index = iniparser_getint(description_file,"STFT:Window Overlap Index", 0);

        samples_per_turn = iniparser_getint(description_file,"COT:Samples Per Turn", 0);

        draw_inverted = iniparser_getboolean(description_file,"STFT:Invert Graphic", false);

        use_RMS_on_STFT = iniparser_getboolean(description_file,"STFT:Use RMS",false);
        use_PWR_on_STFT = iniparser_getboolean(description_file,"STFT:Use PWR",false);
        if((!use_RMS_on_STFT)&&(!use_PWR_on_STFT))
                  {
                  use_Modulus_on_STFT = true;
                  printf("Warning: Automatic selection: Spectrogram calculus based on Modulus operator.\r\n");
                  }

        use_density = iniparser_getboolean(description_file,"STFT:Use Density",true);
    highpassfrequency=iniparser_getdouble(description_file,"STFT:HighPass Cutoff Frequency",0.0);
        original_samplingrate = iniparser_getint(description_file,"COT:Original Samplingrate",0);

        iniparser_freedict(description_file);
        /***************************** Fim da Inicialização ***************************************************/

//algumas variaveis dá pra calcular o valor
        turbinenominalrpm = 120.0 * generatornominalfrequency / ngeneratorpoles;
        turbinenominalfrequency = turbinenominalrpm / 60.0;
        turbinenominalperiod = 1.0 / turbinenominalfrequency;
        blade_passing_frequency = turbinenominalfrequency * nrunnerblades;
        vane_passing_frequency = turbinenominalfrequency * nguidevanes;

//imprime algumas informações sobre o conjunto analizado:
        printf(
                            "\r\n*************************************** ANALYSIS CONFIGURATION
************************************\r\n");
        printf("Generator:\r\n");
        printf("Poles: %u  Nominal Frequency: %4.1f Hz. -> ", ngeneratorpoles,
                            generatornominalfrequency);
        printf("Nominal Speed: %5.1f rpm.\r\n\n", turbinenominalrpm);
        printf("Turbine: %s type %s. Power Setting: %s\r\n", turbinename.c_str(),
                            turbinetype.c_str(), turbinepower.c_str());
        printf("Runner Blades: %u  Guide Vanes: %u.\r\n", nrunnerblades,
                            nguidevanes);
        printf("Nominal Frequency: %9.7f Hz. Nominal Period: %f seconds.\r\n",
                            turbinenominalfrequency, turbinenominalperiod);
        printf(
                            "Blade Passing Frequency: %9.7f Hz.  Guide Vane Passing Frequency: %9.7f Hz.\r\n",
                            blade_passing_frequency, vane_passing_frequency);

        /************************************* Arquivo de dados de entrada
*************************************************/
        printf(
                            "\r\n********************************************** INPUT FILE
*******************************************\r\n");

//imprime o nome do arquivo que será processado
        printf("\r\n");
        wavfilename = string(argv[1]) + "-COT.wav";
        if(process_periodic) {
                                  wavfilename = string(argv[1]) + "-COT-periodic.wav";
                                  strcat(argv[1],"-P");
                             }

        if(process_residue) {
                                 wavfilename = string(argv[1]) + "-COT-residual.wav";
                                 strcat(argv[1],"-R");
                            }

        printf("Processing file: %s\r\n", wavfilename.c_str());
        WavFile inputfile(wavfilename.c_str()); //cria um objeto da classe WavFile
        num_channels = inputfile.get_num_channels();
        samplingrate = inputfile.get_samplingrate();
        printf("WAV File: %u channels. Samplingrate: %u samples/second. ",
                            num_channels, samplingrate);
//calcula a duração em segundos do arquivo:
        num_frames = inputfile.get_num_samples();
        duration = inputfile.get_duration();
        printf("Duration: %8.4f seconds.\r\n", duration);
        printf("Datarate: %u bauds. ", inputfile.get_bits_per_second());
        printf("Bits per sample: %u bits. ", inputfile.get_bits_per_sample());
        printf("File size: %7.1f MBytes.\r\n",
                            (double) inputfile.get_filesize() / (1024 * 1024));
        printf("Recording date: %s\r\n", inputfile.get_metadata_date());

        printf("\r\nChannel Mapping:\r\n");
        for (unsigned int i = 0; i < num_channels; i++) {
```

```
                    printf("Channel %02u -> %s. ", i, inputfile.get_channel_name(i));
                    if (i == channel_optical_encoder)
                              printf("(Sync Optical)");
                    if (i == channel_electric_grid)
                              printf("(Sync Electric)");
                    if ((i != channel_electric_grid) && (i != channel_optical_encoder))
                              printf("(Vibroacoustic)");
                    printf("\r\n");
          }

          printf("\r\n************************************ PROCESSING STRATEGY
*****************************************\r\n");

          total_signal_info_channels = num_accelerometers + num_AE_sensors;
          printf("Total Signal Avaliable Channels: %u - ",total_signal_info_channels);
          total_signal_sync_channels = 0;
          if (channel_optical_encoder < num_channels) total_signal_sync_channels++;
          if (channel_electric_grid < num_channels) total_signal_sync_channels++;
          printf("Total Sync Avaliable Channels: %u\r\n", total_signal_sync_channels);
          if (total_signal_info_channels + total_signal_sync_channels> num_channels)
             {
             printf("Something is wrong! Declared vibroacoustic signal channels: %u  - Declared sync signal channels:
%u ",total_signal_info_channels, total_signal_sync_channels);
             printf("Your .WAV actual number of channels is %u. \r\nReview your analyzer.ini
file.\r\n",num_channels);
        exit(EXIT_FAILURE);
             }

//aloca memória para receber os canais
          channel.resize(total_signal_info_channels);

//neste ponto, o programa já tem um vetor de sinais, mas não vou carregá-los todos de uma vez
//para não saturar a memória

//inicia o pre-processamento:
/************************************ pré-processamento dos sinais dos canais
*******************************************/
printf("\r\n***************************************** PROCESSING SIGNALS
*************************************\r\n");

if (samples_per_turn == 0)
   {
   printf("\r\nYour .ini file don't have COT:Samples Per Turn Field! Possibly your .wav file is NOT
synchronized.\r\n");
   exit(EXIT_FAILURE);
   };

if (original_samplingrate == 0)
   {
   printf("\r\nYour .ini file don't have COT:Original Samplingrate! Possibly your .wav file is NOT
synchronized.\r\n");
   exit(EXIT_FAILURE);
   };

//para todos os canais com sinais vibroacústicos:
          for (unsigned int i = 0; i < num_channels; i++)   if ((i != channel_electric_grid) && (i !=
channel_optical_encoder))
                    {
                    printf("Loading Channel %u (%s)...\r\n",i,inputfile.get_channel_name(i));
                    channel[i] = electric_signal(wavfilename.c_str(),i);
                    printf("Done.\r\n");
                    fflush(stdout);

                    printf("\r\nProcessing Channel %u:\r\n", i);
                    STFT_Computer STFT_Calculator;
                    STFT_Calculator.setup_window(window_name, window_size,overlap_index);
                    //configura a taxa de amostragem original para haver o corte superior depois
                    STFT_Calculator.set_original_samplingrate(original_samplingrate);
                    //isso é usado para debug
                    if (dump_overlapped_windows)
                       {
                       electric_signal test = STFT_Calculator.Get_Overlapped_Windows();
                       test.SaveToDisk("Overlapped Windows");
                       }
                    unsigned int windows_per_turn = samples_per_turn/STFT_Calculator.get_window_size();
                    unsigned int steps_per_turn = samples_per_turn/STFT_Calculator.get_step_size();
                    printf("Using Window type %s with %u samples, %u samples step.\r\n",window_name.c_str(),
STFT_Calculator.get_window_size(),STFT_Calculator.get_step_size());
                    printf("Samples per turn: %u. Entire windows per turn: %u (%u steps per
turn).\r\n",samples_per_turn,windows_per_turn,steps_per_turn);

                    //calcula a Short Time Fourier Transform:
                    Matrix STFT;
                    if (!calculate_average) STFT = STFT_Calculator.Compute_STFT(channel[i]);
                    else
                              {
                              if(use_RMS_on_STFT) STFT =
STFT_Calculator.Compute_RMS_STFT(channel[i],samples_per_turn);
                              if(use_PWR_on_STFT) STFT =
STFT_Calculator.Compute_PWR_STFT(channel[i],samples_per_turn);
                              if(use_Modulus_on_STFT) STFT =
STFT_Calculator.Compute_Mean_STFT(channel[i],samples_per_turn);
                              }

                    {
```

```
                    unsigned int lower_row;
                    lower_row = round(highpassfrequency/STFT_Calculator.get_freq_resolution());
                    printf("Supressing all power below %5.1f
Hz...\r\n",lower_row*STFT_Calculator.get_freq_resolution());
                    for(unsigned int row=0;row<lower_row;row++)
                              for(unsigned int column=0;column<STFT.GetNumColumns();column++) STFT[row][column]=0.0;
                    }

                    /* usado para debug, força a matriz STFT com valores falsos: */
          /*********************************************************************************************/
                    /*for(unsigned int row=0;row<STFT.GetNumRows();row++)
                                        for(unsigned int column=0;column<STFT.GetNumColumns();column++)
                                                  {
                                                  STFT[row][column]=0.0;
                                                  if(row==125)
                                                      {
                                                            STFT[row][column]=0.5

           +0.3*cos(2*PI*11*(double(column)/STFT.GetNumColumns()-1))

           +0.4*sin(2*PI*11*(double(column)/STFT.GetNumColumns()-1));
                                                      }

                                                  if(row==100)
                                                      {
                                                      STFT[row][column]=0.5

           +0.3*cos(2*PI*24*(double(column)/STFT.GetNumColumns()-1))

           +0.4*sin(2*PI*24*(double(column)/STFT.GetNumColumns()-1));
                                                            }
                                                  }
*/
                    /*********************************************************************************************/

                    if(use_density)
                      {
                      printf("Converting to Density...\r\n");
                      printf("ENBW (Effective Noise Bandwidth) is %7.2f Hz.\r\n",STFT_Calculator.get_ENBW());
                      if(use_PWR_on_STFT)
                        {
                              for(unsigned int row=0;row<STFT.GetNumRows();row++)
                                        for(unsigned int column=0;column<STFT.GetNumColumns();column++)
                                                  STFT[row][column]/=STFT_Calculator.get_ENBW();
                        }
                      else
                        {
                              for(unsigned int row=0;row<STFT.GetNumRows();row++)
                                 for(unsigned int column=0;column<STFT.GetNumColumns();column++)
                                                  STFT[row][column]/=sqrt(STFT_Calculator.get_ENBW());
                        }
                      }

                    //salva o resultado da STFT em um arquivo .dat (a matriz)
                    char buffer[200];
                    sprintf(buffer,"%02u",i);
                    if(calculate_average) STFT.SaveToFile(string(argv[1])+"-STFT-channel "+string(buffer)+".dat");
                    else STFT.SaveToFile(string(argv[1])+"-STFT-na-channel "+string(buffer)+".dat");
                    printf("Channel %u Done.\r\n\n", i);

                    //STFT.PrintOut();

                    sprintf(buffer, "channel[%02u] %s", i, channel[i].get_date());

                    //desaloca o sinal que acabou de ser processado (libera memória)
                    channel[i] = electric_signal();

                    //plota um gráfico imagem do espectrograma
                     if(calculate_average)
                       {
                       GLImg* graPHImg = new GLImg(3740,984);
                       graPHImg->set_inverted(draw_inverted);
                       if (!calculate_average) graPHImg->SetMainTitle("|STFT_x(t,f)|^2");
                       else
                          {
                                 if (use_Modulus_on_STFT) graPHImg->SetMainTitle("\\a{|STFT_x(t,f)|}");
                                 if (use_RMS_on_STFT) graPHImg->SetMainTitle("\\big{\\sqrt{_{\\a{|STFT_x(t,f)|
^2}}}}\n");

                                 if (use_PWR_on_STFT)
                                                  {//graPHImg->SetMainTitle("\\a{|STFT_x(t,f)|^2}\n");
                                                  if(i==0) graPHImg->SetMainTitle("\n\\big{(a)
\\a{|STFT_x(t,f)|^2}                                                  }");
                                                     if(i==1) graPHImg-
>SetMainTitle("\n\\big{(b)                                           \\a{|STFT_x(t,f)|^2}
}");
                                                  }

                          }
                       graPHImg->SetSubTitle(turbinename+"("+turbinepower+") "+string(buffer));
                       if (calculate_average)
                                 {
                                 graPHImg->SetXrange(0.0, 360.0);
                                 graPHImg->SetXscaleticks(30.0, 9);
                                 }
                       else
```

```
                                {
                                graPHImg->SetXrange(0.0,STFT_Calculator.get_time_limit());
                                graPHImg->SetXscaleTitle("Time (s)");
                                }
                        graPHImg->SetYscaleticks(10000.0, 9);
                        graPHImg->SetYrange(0.0, STFT_Calculator.get_freq_limit());
                        if (calculate_average)
                                {
                                graPHImg->SetXNormalizationFactor(360.0 / turbinenominalperiod);
                                graPHImg->SetNormalizedXscaleTitle("Time (s)");
                                graPHImg->SetXscaleTitle("Shaft Angle (degrees)");
                                }
                        graPHImg->SetYscaleTitle("Spectral Frequency (kHz)");

                        if(use_density)
                          {
                                if (use_RMS_on_STFT)         graPHImg->SetZscaleTitle("Linear Spectral Density
(V_{RMS}/\\sqrt{Hz})");
                                if (use_Modulus_on_STFT) graPHImg->SetZscaleTitle("Average Voltage Spectral Desity
(V/\\sqrt{Hz})");
                                if (use_PWR_on_STFT)         graPHImg->SetZscaleTitle("Power Spectral Density
(V^2/Hz)");
                          }
                        else
                          {
                                if (use_RMS_on_STFT)         graPHImg->SetZscaleTitle("RMS Voltage (V_{RMS})");
                                if (use_Modulus_on_STFT) graPHImg->SetZscaleTitle("Average Voltage (V)");
                                if (use_PWR_on_STFT)         graPHImg->SetZscaleTitle("Power (V^2)");
                          }

                        sprintf(buffer, "%02u", i);
                        if (!calculate_average) graPHImg->SaveToFile(string(argv[1]) + "-spectrogram-channel "+
string(buffer));
                                else
                                   {
                                        if (use_Modulus_on_STFT) graPHImg->SaveToFile(string(argv[1]) + "-AVG-
spectrogram-channel "+ string(buffer));
                                        if (use_RMS_on_STFT) graPHImg->SaveToFile(string(argv[1]) + "-RMS-
spectrogram-channel "+ string(buffer));
                                        if (use_PWR_on_STFT) graPHImg->SaveToFile(string(argv[1]) + "-PWR-
spectrogram-channel "+ string(buffer));
                                   }

                        printf("Rendering Graphic Output...");   fflush(stdout);
                    graPHImg->PlotRealPartOfComplexSignal((double**) STFT.GetData(),STFT.GetNumColumns(),
STFT.GetNumRows());
                        printf("Done\r\n\n");
                        delete graPHImg;
                        }

                    //cria um objeto calulador de CMS:
                    CMS_Computer CMS_Calculator;
                    //Configura-o:
                    CMS_Calculator.set_signal_name(inputfile.get_channel_name(i));
                    CMS_Calculator.set_experience_name(turbinename + "(" + turbinepower + ") channel["+
string(buffer) + "] "+ inputfile.get_metadata_date());
                    if(calculate_average) CMS_Calculator.set_time_limit((double) samples_per_turn / samplingrate);
                    else CMS_Calculator.set_time_limit(STFT_Calculator.get_time_limit());
                    CMS_Calculator.set_spectral_freq_limit(STFT_Calculator.get_freq_limit());

                    //calcula a CMS e grava ela por cima da STFT:
                    STFT = CMS_Calculator.Compute_CMS(STFT);


                    //calcula a NENBW (normalized effective noise bandwidth)
                    double NENBW_out=(STFT_Calculator.get_ENBW()/STFT_Calculator.get_freq_resolution());

                    {
                    double stationary_power=0.0;
                    for(unsigned int row=0;row<STFT.GetNumRows();row++)
                              stationary_power+=cabs(STFT[row][0]);

                    //é necessário dividir pelo NENBW (normalized effective noise bandwidth)
                    //pois o vazamento espectral acaba aumentado a potência aparente do sinal...
                    //na hora de plotar, aparece OK, e os picos têm valores certos.
                    //mas na hora de contabilizar, dá sempre errado pelo mesmo fator.
                    printf("Stationary Power: %9.8f V²\r\n",stationary_power/NENBW_out);
                    }

                    //antes de salvar a matriz CMS, divide ela toda pelo NENBW:
                    for(unsigned int row=0;row<STFT.GetNumRows();row++)
                       for(unsigned int column=0;column<STFT.GetNumColumns();column++)
                                {
                                STFT[row][column]/=NENBW_out;
                                }

                    //salva o resultado da CMS em um arquivo .dat (a matriz)
                    sprintf(buffer, "%02u", i);
                    STFT.SaveToFile(string(argv[1]) + "-CMS-channel " + string(buffer)+ ".dat");
                    printf("Channel %u Done.\r\n\n", i);

                    //depois de salvar a matriz CMS, multiplica ela toda pelo NENBW:
                    for(unsigned int row=0;row<STFT.GetNumRows();row++)
                       for(unsigned int column=0;column<STFT.GetNumColumns();column++)
                                {
```

```
                                STFT[row][column]*=NENBW_out;
                                }

                        //plota a matriz CMS
                        if (calculate_average)
                           {
                           GLImg* graPHImg = new GLImg(3740,984);
                           graPHImg->set_inverted(draw_inverted);
                           //graPHImg->SetMainTitle("CMS");
                           if(i==0) graPHImg->SetMainTitle("\n\\big{(a)
CMS                                                        }");
                           if(i==1) graPHImg->SetMainTitle("\n\\big{(b)
CMS                                                        }");

                           graPHImg->SetSubTitle(CMS_Calculator.get_experience_name());
                           graPHImg->SetXrange(0.0,CMS_Calculator.get_cyclic_freq_limit());
                           //graPHImg->SetXscaleticks(30.0,9);
                           graPHImg->SetYscaleticks(20000.0, 9);
                           graPHImg->SetYrange(0.0,
                           CMS_Calculator.get_spectral_freq_limit());
                           graPHImg->SetXNormalizationFactor(turbinenominalfrequency);
                     graPHImg->SetNormalizedXscaleTitle("machine orders");
                           graPHImg->SetXscaleTitle("cyclic frequency     \\alpha   (Hz)");
                           graPHImg->SetYscaleTitle("spectral frequency     f (kHz)");

                           if(use_density)
                                    {
                                    if (use_Modulus_on_STFT) graPHImg->SetZscaleTitle("Cyclic Modulation Spectrum
(V/\\sqrt{Hz})");
                                    if (use_RMS_on_STFT) graPHImg->SetZscaleTitle("Cyclic Modulation Spectrum
(V_{RMS}/\\sqrt{Hz})");
                                    if (use_PWR_on_STFT) graPHImg->SetZscaleTitle("Cyclic Modulation Spectrum
(V^2/Hz)");
                                    }
                           else
                                    {
                                    if (use_Modulus_on_STFT) graPHImg->SetZscaleTitle("Cyclic Modulation Spectrum
(V)");
                                    if (use_RMS_on_STFT) graPHImg->SetZscaleTitle("Cyclic Modulation Spectrum
(V_{RMS})");
                                    if (use_PWR_on_STFT) graPHImg->SetZscaleTitle("Cyclic Modulation Spectrum (V^2)");
                                    }

                           sprintf(buffer, "%02u", i);
                           if (!calculate_average) graPHImg->SaveToFile(string(argv[1]) + "-CMS-channel " +
string(buffer));
                                             else
                                                {
                                                      if (use_Modulus_on_STFT) graPHImg->SaveToFile(string(argv[1]) +
"-AVG-CMS-channel " + string(buffer));
                                                      if (use_RMS_on_STFT) graPHImg->SaveToFile(string(argv[1]) + "-
RMS-CMS-channel " + string(buffer));
                                                      if (use_PWR_on_STFT) graPHImg->SaveToFile(string(argv[1]) + "-
PWR-CMS-channel " + string(buffer));
                                                }

                           printf("Rendering Graphic Output...");  fflush(stdout);
                           graPHImg->PlotModulusOfComplexSignal((double**) STFT.GetData(),STFT.GetNumColumns(),
STFT.GetNumRows());
                           printf("Done\r\n\n");
                           delete graPHImg;
                           }

                        if (!calculate_average) //se não é para calcular a média, então plota a CMS pra ver o draft tube
swirl
                           {
                           Matrix DtCMS = Matrix(STFT.GetNumRows(),num_frames/samples_per_turn);

                           //copia somente as primeiras num_turns colunas da matriz CMS para a nova matriz de plotar
                           for(unsigned int k=0;k<num_frames/samples_per_turn;k++)
                              for(unsigned int j=0;j<STFT.GetNumRows();j++)
                                 {DtCMS.SetElement(j,k,STFT.GetElement(j,k));}

                           GLImg* graPHImg = new GLImg(3740,984);
                           graPHImg->set_inverted(draw_inverted);
                           if(i==0) graPHImg->SetMainTitle("\n\\big{(a)
CMS                                                        }");
                           if(i==1) graPHImg->SetMainTitle("\n\\big{(b)
CMS                                                        }");
                           graPHImg->SetSubTitle(CMS_Calculator.get_experience_name());
                           graPHImg->SetXrange(0.0,turbinenominalfrequency);
                           graPHImg->SetYscaleticks(20000.0, 9);
                           graPHImg->SetYrange(0.0,CMS_Calculator.get_spectral_freq_limit());
                           graPHImg->SetXNormalizationFactor(turbinenominalfrequency);
                           graPHImg->SetNormalizedXscaleTitle("machine orders");
                           graPHImg->SetXscaleTitle("cyclic frequency     \\alpha   (Hz)");
                           graPHImg->SetYscaleTitle("spectral frequency     f (kHz)");

                           graPHImg->SetZscaleTitle("Cyclic Modulation Spectrum (V^2)");
                           sprintf(buffer, "%02u", i);
                           graPHImg->SaveToFile(string(argv[1]) + "-PWR-DT-CMS-channel " + string(buffer));

                           printf("Rendering Graphic Output...");  fflush(stdout);
                           graPHImg->PlotModulusOfComplexSignal((double**) DtCMS.GetData(),DtCMS.GetNumColumns(),
DtCMS.GetNumRows());
```

```
                    printf("Done\r\n\n");
                    delete graPHImg;
                    }

                            //cria um objeto calulador de CMC:
                    CMC_Computer CMC_Calculator;
                    //Configura-o:
                    CMC_Calculator.set_signal_name(CMS_Calculator.get_signal_name());
                    CMC_Calculator.set_experience_name(CMS_Calculator.get_experience_name());
                    CMC_Calculator.set_cyclic_freq_limit(CMS_Calculator.get_cyclic_freq_limit());
                    CMC_Calculator.set_spectral_freq_limit(CMS_Calculator.get_spectral_freq_limit());
                    //plota a PSD:
                    {
                    //extrai a PSD:
              double* PSD = (double*)malloc(sizeof(double)*STFT.GetNumRows());
              for(register unsigned int j=0;j<STFT.GetNumRows();j++) PSD[j]=cabs(STFT[j][0]);
                    GL2D* graPH = new GL2D(3740,984);
                    if(use_density)
                              {
                              if(use_Modulus_on_STFT) graPH->SetMainTitle("\\hat{S}_{X}(f) (Spectral Density)");
                              if(use_PWR_on_STFT) graPH->SetMainTitle("\\hat{S}_{X}(f) (Power Spectral Density)");
                              if(use_RMS_on_STFT) graPH->SetMainTitle("\\hat{S}_{X}(f) (Linear Spectral Density)");
                              }
                    else
                              {
                              if(use_Modulus_on_STFT) graPH->SetMainTitle("\\hat{S}_{X}(f) (Spectrum)");
                              if(use_PWR_on_STFT) graPH->SetMainTitle("\\hat{S}_{X}(f) (Power Spectrum)");
                              if(use_RMS_on_STFT) graPH->SetMainTitle("\\hat{S}_{X}(f) (Linear Spectrum)");
                              }
                    graPH->SetSubTitle(CMS_Calculator.get_experience_name());
                    graPH->SetXrange(0.0, CMS_Calculator.get_spectral_freq_limit()/1000.0);
                    if(use_density)
                              {
                              if(use_Modulus_on_STFT) graPH->SetYscaleTitle("Average Spectral Density
(V/\\sqrt{Hz})");
                              if(use_PWR_on_STFT) graPH->SetYscaleTitle("Power Spectral Density (V^2/Hz)");
                              if(use_RMS_on_STFT) graPH->SetYscaleTitle("Linear Spectral Density
(V_{RMS}/\\sqrt{Hz})");
                              }
                    else
                              {
                              if(use_Modulus_on_STFT) graPH->SetYscaleTitle("Average Spectrum (V)");
                              if(use_PWR_on_STFT) graPH->SetYscaleTitle("Power Spectrum (V^2)");
                              if(use_RMS_on_STFT) graPH->SetYscaleTitle("Linear Spectrum (V_{RMS})");
                              }

                    graPH->SetXscaleTitle("Spectral Frequency (kHz)");
                    if(use_Modulus_on_STFT) graPH->SaveToFile(string(argv[1]) + "-AVG-PSD-channel " +
string(buffer));
                    if(use_RMS_on_STFT) graPH->SaveToFile(string(argv[1]) + "-RMS-PSD-channel " + string(buffer));
                    if(use_PWR_on_STFT) graPH->SaveToFile(string(argv[1]) + "-PWR-PSD-channel " + string(buffer));
                    graPH->PlotSignal(PSD,STFT.GetNumRows(),"");
                    delete graPH;
                    free(PSD);
                    }
                    //plota a Cyclic Power Spectrum
                    {
              double* CPS = (double*)malloc(sizeof(double)*STFT.GetNumColumns()); //aloca um vetor de doubles
              //para cada coluna da matriz STFT:
              for(register unsigned int j=0;j<STFT.GetNumColumns();j++)
                    {
                    CPS[j]=0.0; //zera o vetor inicialmente, para usá-lo como acumulador
                    for(register unsigned int i=0;i<STFT.GetNumRows();i++) CPS[j]+=cabs(STFT[i][j]); //soma todos os
valores das colunas
                    if(use_density) CPS[j]*=STFT_Calculator.get_ENBW(); //se ele usou a densidade de potência,
corrige pois é uma distribuição discreta de potências
                    }
              //anula a componente de potência estacionária:
              CPS[0]=0.0;
              //estimador do Cyclic Power Spectrum Calculado
              GL2D* graPH = new GL2D(3740,984);
              if(use_Modulus_on_STFT) graPH->SetMainTitle("\\hat{P}_{X}(\\alpha) (Cyclic Spectrum)");
              if(use_PWR_on_STFT) graPH->SetMainTitle("\\hat{P}_{X}(\\alpha) (Cyclic Power Spectrum)");
              if(use_RMS_on_STFT) graPH->SetMainTitle("\\hat{P}_{X}(\\alpha) (Cyclic Linear Spectrum)");
              graPH->SetSubTitle(CMS_Calculator.get_experience_name());
              graPH->SetXrange(0.0, CMS_Calculator.get_cyclic_freq_limit());
              graPH->SetXNormalizationFactor(turbinenominalfrequency);
              graPH->SetNormalizedXscaleTitle("machine orders (Hz/Hz)");
              if(use_Modulus_on_STFT) graPH->SetYscaleTitle("Cyclic Average Spectrum (V)");
              if(use_PWR_on_STFT) graPH->SetYscaleTitle("Cyclic Power Spectrum (V^2)");
              if(use_RMS_on_STFT) graPH->SetYscaleTitle("Cyclic Linear Spectrum (V_{RMS})");
              graPH->SetXscaleTitle("cyclic frequency (Hz)");
              if(use_Modulus_on_STFT) graPH->SaveToFile(string(argv[1]) + "-AVG-CPS-channel " + string(buffer));
              if(use_RMS_on_STFT) graPH->SaveToFile(string(argv[1]) + "-RMS-CPS-channel " + string(buffer));
              if(use_PWR_on_STFT) graPH->SaveToFile(string(argv[1]) + "-PWR-CPS-channel " + string(buffer));
              graPH->PlotSignal(CPS,STFT.GetNumColumns(),"");
              delete graPH;
              free(CPS);
                    }

                    //*****************************************************************************************

                    //anula o nível DC da matriz STFT (STFT[0][0] é o nível DC dos níveis DC do sinal);
                    //for (unsigned int row=0;row<10;row++) STFT[row][0]=0.0;
```

```
                    //calcula a CMC e grava ela por cima da CMS:
                    STFT = CMC_Calculator.Compute_CMC(STFT);
          //salva o resultado da CMS em um arquivo .dat (a matriz)
                    sprintf(buffer,"%02u",i);
                    STFT.SaveToFile(string(argv[1]) + "-CMC-channel " + string(buffer)+ ".dat");
                    printf("Channel %u Done.\r\n\n", i);

                    if (calculate_average) //se a média dos espectrogramas é calculada, então elabora um gráfico:
                       {
                       GLImg* graPHImg = new GLImg(3740,984);
                       graPHImg->set_inverted(draw_inverted);
                       if (!calculate_average) graPHImg->SetMainTitle("CMC");
                          else
                              {
                              if (use_Modulus_on_STFT) graPHImg->SetMainTitle("CMC (Modulus)");
                                          if (use_RMS_on_STFT) graPHImg->SetMainTitle("CMC (Linear)");
                                          //if (use_PWR_on_STFT) graPHImg->SetMainTitle("CMC (Power)");
                                          if (use_PWR_on_STFT)
                                                  {
                                                  if(i==0) graPHImg->SetMainTitle("\n\\big{(a)
CMC                                                       }");
                                                  if(i==1) graPHImg->SetMainTitle("\n\\big{(b)
CMC                                                       }");
                                                  }
                                          }

                       graPHImg->SetSubTitle(CMC_Calculator.get_experience_name());
                       graPHImg->SetXrange(0.0,CMC_Calculator.get_cyclic_freq_limit());
                       //graPHImg->SetXscaleticks(30.0,9);
                       graPHImg->SetYscaleticks(20000.0, 9);
                       graPHImg->SetYrange(0.0,CMC_Calculator.get_spectral_freq_limit());
                       graPHImg->SetXNormalizationFactor(turbinenominalfrequency);
                       graPHImg->SetNormalizedXscaleTitle("ordem de maquina     a = \\alpha/\\alpha_f (Hz/Hz)");
                       graPHImg->SetXscaleTitle("frequencia ciclica     \\alpha  (Hz)");
                       graPHImg->SetYscaleTitle("frequencia espectral     f (kHz)");
                       graPHImg->SetZscaleTitle("grau de cicloestacionariedade (DCS)");

                       sprintf(buffer, "%02u", i);
                       if (!calculate_average) graPHImg->SaveToFile(string(argv[1]) + "-CMC-channel " +
string(buffer));
                                          else
                                    {
                                          if (use_Modulus_on_STFT) graPHImg->SaveToFile(string(argv[1]) + "-AVG-CMC-
channel " + string(buffer));
                                          if (use_RMS_on_STFT) graPHImg->SaveToFile(string(argv[1]) + "-RMS-CMC-
channel " + string(buffer));
                                          if (use_PWR_on_STFT) graPHImg->SaveToFile(string(argv[1]) + "-PWR-CMC-
channel " + string(buffer));
                                    }

                       printf("Rendering Graphic Output..."); fflush(stdout);
                       graPHImg->PlotModulusOfComplexSignal((double**) STFT.GetData(),STFT.GetNumColumns(),
STFT.GetNumRows());
                       printf("Done\r\n\n");
                       delete graPHImg;
                       }

                    if (!calculate_average) //se não é para calcular a média, então plota a CMS pra ver o draft tube
swirl
                        {
                        Matrix DtCMS = Matrix(STFT.GetNumRows(),num_frames/samples_per_turn);
                       //copia somente as primeiras num_turns colunas da matriz CMS para a nova matriz de plotar
                       for(unsigned int k=0;k<num_frames/samples_per_turn;k++)
                 for(unsigned int j=0;j<STFT.GetNumRows();j++)
                                    {DtCMS.SetElement(j,k,STFT.GetElement(j,k));}

                       GLImg* graPHImg = new GLImg(3740,984);
                       graPHImg->set_inverted(draw_inverted);
                       if(i==0) graPHImg->SetMainTitle("\n\\big{(a)
CMC                                                    }");
                       if(i==1) graPHImg->SetMainTitle("\n\\big{(b)
CMC                                                    }");
                       graPHImg->SetSubTitle(CMS_Calculator.get_experience_name());
                       graPHImg->SetXrange(0.0,turbinenominalfrequency);
                       graPHImg->SetYscaleticks(20000.0, 9);
                       graPHImg->SetYrange(0.0,CMS_Calculator.get_spectral_freq_limit());
                       graPHImg->SetXNormalizationFactor(turbinenominalfrequency);
                       graPHImg->SetNormalizedXscaleTitle("machine orders");
                       graPHImg->SetXscaleTitle("cyclic frequency     \\alpha   (Hz)");
                       graPHImg->SetYscaleTitle("spectral frequency     f (kHz)");
                       graPHImg->SetZscaleTitle("Degree of Cyclostationarity (DCS)");
                       sprintf(buffer, "%02u", i);
                       graPHImg->SaveToFile(string(argv[1]) + "-PWR-DT-CMC-channel " + string(buffer));
                       printf("Rendering Graphic Output...");  fflush(stdout);
                       graPHImg->PlotModulusOfComplexSignal((double**) DtCMS.GetData(),DtCMS.GetNumColumns(),
DtCMS.GetNumRows());
                       printf("Done\r\n\n");
                       delete graPHImg;
                       }


           printf("Correction Factors:\r\n");
           for(unsigned int alpha=0;alpha<4;alpha++) printf("Kw(%5.1f Hz)=%f
",alpha*CMC_Calculator.get_spectral_freq_resolution(),1.0/STFT_Calculator.get_Kw()[alpha]);
           printf("\r\n");
```

```
printf("\r\n*****************************************************************************************************
\r\n");
                    } //fim do for(unsigned int i=0;i<num_channels;i++) if((i!=channel_electric_grid)&&(i!
=channel_optical_encoder))

}
```

Arquivo **turbinesoundsimulator.cc**:

```
#include <stdio.h>
#include <time.h>
extern "C"
{
#include <iniparser.h>
}
#include <vector>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <math.h>
#include "signal.h"
#include "wavfile.h"

/* constantes globais necessárias  */

const double PI=3.14159265358979311599796 3;


/********************************** Variáveis Globais *********************************************/

string outputfilename;

unsigned int nguidevanes;
unsigned int nrunnerblades;
unsigned int ngeneratorpoles;
double generatornominalfrequency;

unsigned int naccelerometers;
unsigned int nacousticemissionsensors;
vector <electric_signal> accelerometer_channel;
vector <electric_signal> acoustic_emission_channel;

unsigned int samplerate;
unsigned int turbineturns;

unsigned int nframes;
unsigned int frames_per_turn;
double turbinenominalfrequency;
double turbinenominalperiod;
double duration;

//variáveis para implementação dos filtros da cloud cavitation e da travelling bubble cavitation
string cloud_carrier_type;
bool cloud_enabled;
unsigned int cloud_filter_order;
double cloud_filter_gain;
double* cloud_filter_numerator;
double* cloud_filter_denominator;
IIRFilter cloud_filter;

string bubble_carrier_type;
bool bubble_enabled;
unsigned int bubble_filter_order;
double bubble_filter_gain;
double* bubble_filter_numerator;
double* bubble_filter_denominator;
IIRFilter bubble_filter;

//criar vetor de doubles para as amplitudes do envelope
unsigned int bubble_harmonics;
unsigned int cloud_harmonics;
double* bubble_harmonic_cosine;
double* bubble_harmonic_sine;
double* cloud_harmonic_cosine;
double* cloud_harmonic_sine;

char DefaultCarrier[80]="White Noise";

//modelagem do droop speed control
double generator_speed_percentual_MaxDev;

//ruidos de interferência
bool stationary_noise_enabled;
bool tonal_noise_enabled;
double stationary_noise_power_density;
double tonal_noise_01_freq;
double tonal_noise_01_power;
double tonal_noise_02_freq;
double tonal_noise_02_power;
```

```
double bubble_VK_freq;   //armazena a frequência de derramamento de vórtices Von Kármán
double bubble_VK_index;  //o índice de modulação da cavitação Von Kármán sobre a Travelling bubble cavitation
double bubble_DT_freq;   //o mesmo para o Draft Tube Swirl
double bubble_DT_index;

double cloud_VK_freq;   //armazena a frequência de derramamento de vórtices Von Kármán
double cloud_VK_index;  //o índice de modulação da cavitação Von Kármán sobre a Cloud Cavitation
double cloud_DT_freq;   //o mesmo para o Draft Tube Swirl
double cloud_DT_index;

/*****************************************************************************************/


/********************************** Protótipos das funcões ***********************************/


/*****************************************************************************************/

//Início do programa principal:
/
*************************************************************************************************************************
*****/
int main(int argc,char** argv)
{
if(argc>2) exit(EXIT_FAILURE);
//carrega o arquivo txt contendo a descrição do que há no arquivo .wav e suas análises:
dictionary* ini_file = iniparser_load("turbinesoundsimulator.ini");
if(!ini_file) {  printf("File %s not found. Aborting.",(string(argv[0])+".ini").c_str());  exit(EXIT_FAILURE);   }

nguidevanes = iniparser_getint(ini_file,"Turbine:Guide Vanes Number",0);
nrunnerblades = iniparser_getint(ini_file,"Turbine:Runner Blades Number",0);
ngeneratorpoles = iniparser_getint(ini_file,"Generator:Number Of Poles",0);
generatornominalfrequency = iniparser_getdouble(ini_file,"Generator:Nominal Frequency",0.0);
generator_speed_percentual_MaxDev = iniparser_getdouble(ini_file,"Generator:Speed Percentual MaxDev",0.0);

samplerate = iniparser_getint(ini_file,"Syntesized Signal:Samplerate",0);
turbineturns = iniparser_getint(ini_file,"Syntesized Signal:TurbineTurns",0);
naccelerometers = iniparser_getint(ini_file,"Sensors:Number of Accelerometers",0);
nacousticemissionsensors = iniparser_getint(ini_file,"Sensors:Number of Acoustic Emission Sensors",0);

bubble_carrier_type = iniparser_getstring(ini_file,"Travelling Bubble Cavitation:Carrier Type",DefaultCarrier);
bubble_filter_order = iniparser_getint(ini_file,"Travelling Bubble Cavitation:IIR Filter Order",0);
bubble_filter_gain = iniparser_getdouble(ini_file,"Travelling Bubble Cavitation:IIR Filter Gain",1.0);
bubble_filter_numerator = (double*)malloc(sizeof(double)*(bubble_filter_order+1));
bubble_filter_denominator = (double*)malloc(sizeof(double)*(bubble_filter_order+1));
for(unsigned int i=0;i<=bubble_filter_order;i++)
   {
   char buffer[100];
   sprintf(buffer,"Travelling Bubble Cavitation:IIR Filter Numerator Coeficient %02u",i);
   bubble_filter_numerator[i]=iniparser_getdouble(ini_file,buffer,0.0);
   sprintf(buffer,"Travelling Bubble Cavitation:IIR Filter Denominator Coeficient %02u",i);
   bubble_filter_denominator[i]=iniparser_getdouble(ini_file,buffer,0.0);
   }

cloud_carrier_type = iniparser_getstring(ini_file,"Cloud Cavitation:Carrier Type",DefaultCarrier);
cloud_filter_order = iniparser_getint(ini_file,"Cloud Cavitation:IIR Filter Order",0);
cloud_filter_gain = iniparser_getdouble(ini_file,"Cloud Cavitation:IIR Filter Gain",1.0);
cloud_filter_numerator = (double*)malloc(sizeof(double)*(cloud_filter_order+1));
cloud_filter_denominator = (double*)malloc(sizeof(double)*(cloud_filter_order+1));
for(unsigned int i=0;i<=cloud_filter_order;i++)
   {
   char buffer[100];
   sprintf(buffer,"Cloud Cavitation:IIR Filter Numerator Coeficient %02u",i);
   cloud_filter_numerator[i]=iniparser_getdouble(ini_file,buffer,0.0);
   sprintf(buffer,"Cloud Cavitation:IIR Filter Denominator Coeficient %02u",i);
   cloud_filter_denominator[i]=iniparser_getdouble(ini_file,buffer,0.0);
   }

//incializa os filtros:
bubble_filter.set_coeficients(bubble_filter_numerator,bubble_filter_order,bubble_filter_denominator,bubble_filter_or
der,1.0/bubble_filter_gain);
cloud_filter.set_coeficients(cloud_filter_numerator,cloud_filter_order,cloud_filter_denominator,cloud_filter_order,1
.0/cloud_filter_gain);

//lê do arquivo .ini as amplitudes das componentes dos envelopes:
bubble_harmonics=iniparser_getint(ini_file,"Travelling Bubble Cavitation:Harmonics",0);
cloud_harmonics=iniparser_getint(ini_file,"Cloud Cavitation:Harmonics",0);
bubble_harmonic_cosine = (double*)malloc(sizeof(double)*(1+bubble_harmonics));
bubble_harmonic_sine = (double*)malloc(sizeof(double)*(1+bubble_harmonics));
cloud_harmonic_cosine = (double*)malloc(sizeof(double)*(1+cloud_harmonics));
cloud_harmonic_sine = (double*)malloc(sizeof(double)*(1+cloud_harmonics));

for(unsigned int i=0;i<=bubble_harmonics;i++)
   {
   char buffer[100];
   sprintf(buffer,"Travelling Bubble Cavitation:Harmonic %03u Cosine Amplitude",i);
   bubble_harmonic_cosine[i]=iniparser_getdouble(ini_file,buffer,0.0);
   sprintf(buffer,"Travelling Bubble Cavitation:Harmonic %03u Sine Amplitude",i);
   bubble_harmonic_sine[i]=iniparser_getdouble(ini_file,buffer,0.0);
   }

for(unsigned int i=0;i<=cloud_harmonics;i++)
   {
   char buffer[100];
```

```
   sprintf(buffer,"Cloud Cavitation:Harmonic %03u Cosine Amplitude",i);
   cloud_harmonic_cosine[i]=iniparser_getdouble(ini_file,buffer,0.0);
   sprintf(buffer,"Cloud Cavitation:Harmonic %03u Sine Amplitude",i);
   cloud_harmonic_sine[i]=iniparser_getdouble(ini_file,buffer,0.0);
   }

bubble_enabled=iniparser_getboolean(ini_file,"Travelling Bubble Cavitation:Enabled",false);
cloud_enabled=iniparser_getboolean(ini_file,"Cloud Cavitation:Enabled",false);

stationary_noise_enabled=iniparser_getboolean(ini_file,"Stationary Noise:Enabled",false);
tonal_noise_enabled=iniparser_getboolean(ini_file,"Tonal Noise:Enabled",false);

stationary_noise_power_density = iniparser_getdouble(ini_file,"Stationary Noise:Power Density",0.0);

tonal_noise_01_freq = iniparser_getdouble(ini_file,"Tonal Noise:Frequency 01",0.0);
tonal_noise_01_power = iniparser_getdouble(ini_file,"Tonal Noise:Power 01",0.0);
tonal_noise_02_freq = iniparser_getdouble(ini_file,"Tonal Noise:Frequency 02",0.0);
tonal_noise_02_power = iniparser_getdouble(ini_file,"Tonal Noise:Power 02",0.0);

bubble_VK_freq = iniparser_getdouble(ini_file,"Travelling Bubble Cavitation:Von Karman Reduced Frequency",1.0);
bubble_VK_index = iniparser_getdouble(ini_file,"Travelling Bubble Cavitation:Von Karman Modulation Index",0.0);
bubble_DT_freq = iniparser_getdouble(ini_file,"Travelling Bubble Cavitation:Draft Tube Swirl Reduced
Frequency",1.0);
bubble_DT_index = iniparser_getdouble(ini_file,"Travelling Bubble Cavitation:Draft Tube Swirl Modulation
Index",0.0);
cloud_VK_freq = iniparser_getdouble(ini_file,"Cloud Cavitation:Von Karman Reduced Frequency",1.0);
cloud_VK_index = iniparser_getdouble(ini_file,"Cloud Cavitation:Von Karman Modulation Index",0.0);
cloud_DT_freq = iniparser_getdouble(ini_file,"Cloud Cavitation:Draft Tube Swirl Reduced Frequency",1.0);
cloud_DT_index = iniparser_getdouble(ini_file,"Cloud Cavitation:Draft Tube Swirl Modulation Index",0.0);

//fim da inicialização
iniparser_freedict(ini_file);

//a esta altura, já se sabe o nome do arquivo, o número de canais e quais canais serão os de sinais e os de
//sincronismo. Já dá para calcular a duração dos sinais e o número de amostras totais.

turbinenominalfrequency = (2*generatornominalfrequency)/ngeneratorpoles;
turbinenominalperiod = ngeneratorpoles/(2*generatornominalfrequency);
frames_per_turn = samplerate*turbinenominalperiod;
nframes=turbineturns*frames_per_turn;

/*****************************************************************************/
//mostra algumas informações parciais:
if((naccelerometers!=0)&&(nacousticemissionsensors!=0))
{ printf("Nothing to synth! Review your %s file.",argv[0]); }
printf("Syntesizing signal for experiment:");
if(naccelerometers!=0) printf(" %u accelerometers",naccelerometers);
if(nacousticemissionsensors!=0) printf(" %u AE sensors",naccelerometers);
printf(" 1 electric generator and 1 optical encoder.\r\n");
printf("Electric Generator: %6.3f Hz. %u poles. ",generatornominalfrequency,ngeneratorpoles);
printf("Sampling rate: %u samples/second or %u samples/turn.\r\n",samplerate,frames_per_turn);
printf("Syntesizing %u turns.\r\n",turbineturns);
printf("Travelling Bubble Cavitation Filter Order: %02u. Basic Carrier:
%s\r\n",bubble_filter_order,bubble_carrier_type.c_str());
printf("Cloud Cavitation Filter Order: %02u. Basic Carrier:%s\r\n",cloud_filter_order,cloud_carrier_type.c_str());
/*****************************************************************************/

//reserva memória
printf("\r\n\nAllocating Memory...");
accelerometer_channel.resize(naccelerometers);
acoustic_emission_channel.resize(nacousticemissionsensors);
electric_signal optical_encoder_channel(nframes,samplerate,"Optical Encoder");
electric_signal electric_generator_channel(nframes,samplerate,"Electric Generator");
printf("Done.\r\n");

printf("Creating empty output signals...\r\n"); fflush(stdout);
//cria os sinais inicialmente vazios:
for(unsigned int i=0;i<naccelerometers;i++)
   {
   char buffer[80];
   sprintf(buffer,"Acelerômetro %02u",i);
   accelerometer_channel[i]=electric_signal(nframes,samplerate,buffer);
   printf("Created %s\r\n",buffer); fflush(stdout);
   }
for(unsigned int i=0;i<nacousticemissionsensors;i++)
   {
   char buffer[80];
   sprintf(buffer,"AE sensor %02u",i);
   acoustic_emission_channel[i]=electric_signal(nframes,samplerate,buffer);
   printf("Created %s\r\n",buffer); fflush(stdout);
   }

/* Estes sinais servem de auditoria: */
/*****************************************************************************************************/
electric_signal* tonal_noise_1 = new electric_signal[nacousticemissionsensors+naccelerometers]();
electric_signal* tonal_noise_2 = new electric_signal[nacousticemissionsensors+naccelerometers]();
electric_signal* random_noise = new electric_signal[nacousticemissionsensors+naccelerometers]();
electric_signal* cloud_SOI = new electric_signal[nacousticemissionsensors+naccelerometers]();
electric_signal* bubble_SOI = new electric_signal[nacousticemissionsensors+naccelerometers]();

for(unsigned int i=0; i<nacousticemissionsensors+naccelerometers; i++)
   {
   char buffer[80];
   sprintf(buffer," channel %02u",i);
   tonal_noise_1[i]=electric_signal(nframes,samplerate,string(string("Tonal noise 1")+string(buffer)).c_str());
```

```
   tonal_noise_2[i]=electric_signal(nframes,samplerate,string(string("Tonal noise 2")+string(buffer)).c_str());
   random_noise[i]=electric_signal(nframes,samplerate,string(string("Random noise")+string(buffer)).c_str());
   cloud_SOI[i]=electric_signal(nframes,samplerate,string(string("Cloud SOI")+string(buffer)).c_str());
   bubble_SOI[i]=electric_signal(nframes,samplerate,string(string("Bubble SOI")+string(buffer)).c_str());
   printf("Created tonal and random noises empty signals for channel %u\r\n",i);
   }

/*********************************************************************************************/

//cria um vetor de ponteiros de sinais, para poder futuramente gerenciar melhor e salvar todos em um arquivo só:
electric_signal** signalpointer=
(electric_signal**)malloc(sizeof(electric_signal*)*(2+naccelerometers+nacousticemissionsensors));
electric_signal** p = signalpointer;
for(unsigned int i=0;i<naccelerometers;i++) { *p = &(accelerometer_channel[i]); p++; };
for(unsigned int i=0;i<nacousticemissionsensors;i++) { *p = &(acoustic_emission_channel[i]); p++; };
*p = &optical_encoder_channel; p++;
*p = &electric_generator_channel;
printf("Done.\r\n\n");

//prepara o gerador de números aleatórios:
const gsl_rng_type * T;
gsl_rng * r;
T = gsl_rng_default;
r = gsl_rng_alloc (T);

printf("Processing: Travelling Bubble Cavitation:\r\n");
printf("Generating basic noise type: %s\r\n",bubble_carrier_type.c_str());
electric_signal bubble_carrier = *signalpointer[0];
if(bubble_carrier_type.compare("White Noise")==0)
  {
  //preenche com ruído branco gaussiano de desvio padrão igual a 0.25 (variância 0.0625 V²).
  for(unsigned int j=0;j<bubble_carrier.get_num_samples();j++) bubble_carrier[j]=gsl_ran_gaussian(r,0.25);
  }
//aplica a filtragem IIR
printf("Applying filter:\r\n\n");
bubble_filter.printout();
bubble_filter.apply(&bubble_carrier);
printf("Stationary carrier for bubble cavitation has average power of %9.8f
V²\r\n",bubble_carrier.MeasureAveragePower());
printf("Done.\r\n\n");

printf("Processing: Cloud Cavitation:\r\n");
printf("Generating basic noise type: %s\r\n",cloud_carrier_type.c_str());
electric_signal cloud_carrier = *signalpointer[0];
if(cloud_carrier_type.compare("White Noise")==0)
  {
  //preenche com ruído branco gaussiano de desvio padrão igual a 0.25 (variância 0.0625 V²).
  for(unsigned int j=0;j<cloud_carrier.get_num_samples();j++) cloud_carrier[j]=gsl_ran_gaussian(r,0.25);
  }
//aplica a filtragem IIR
printf("Applying filter:\r\n\n");
cloud_filter.printout();
cloud_filter.apply(&cloud_carrier);
printf("Stationary carrier for cloud cavitation has average power of %9.8f
V²\r\n",cloud_carrier.MeasureAveragePower());
printf("Done.\r\n");

printf("Processing: Droop Speed Control:\r\n");
printf("Generating White Gaussian Noise\r\n");
electric_signal generator_speed_deviation=*signalpointer[0];
for(unsigned int j=0;j<generator_speed_deviation.get_num_samples();j++)
generator_speed_deviation[j]=gsl_ran_gaussian(r,0.1);
double numerator[3]={1.0,2.0,1.0};
double denominator[3]={1.0,1.9999537200,-0.9999537210};
printf("\r\nSimulating Inertia...\r\n");
IIRFilter inercia_filter;
inercia_filter.set_coeficients(numerator,2,denominator,2,1.0/3.735190612e+09);
inercia_filter.reset_state();
inercia_filter.printout();
inercia_filter.apply(&generator_speed_deviation);
generator_speed_deviation.maximize();
generator_speed_deviation.apply_gain(generator_speed_percentual_MaxDev/100.0);
printf("Done.\r\n"); fflush(stdout);

//generator_speed_deviation.SaveToDisk("test");
printf("Generating RSI...\r\n\n");
printf("Bubble Harmonics: %u\r\n",bubble_harmonics);
for(unsigned int i=0;i<=bubble_harmonics;i++) if((bubble_harmonic_cosine[i]!=0.0)||(bubble_harmonic_sine[i]!=0.0))
         printf("%u = [%7.3f %7.3f]\r\n",i,bubble_harmonic_cosine[i],bubble_harmonic_sine[i]);
printf("\r\n");

printf("Cloud Harmonics: %u\r\n",cloud_harmonics);
for(unsigned int i=0;i<=cloud_harmonics;i++) if((cloud_harmonic_cosine[i]!=0.0)||(cloud_harmonic_sine[i]!=0.0))
         printf("%u = [%7.3f %7.3f]\r\n",i,cloud_harmonic_cosine[i],cloud_harmonic_sine[i]);
printf("\r\n\n");
printf("Done RSI\r\n\n");

if(cloud_VK_index!=0.0) printf("Von Kármán modulation in Cloud Cavitation detected: %f order and %f%%
index.\r\n",cloud_VK_freq,cloud_VK_index*100.0);

printf("Modulating..."); fflush(stdout);
double angle=0.0; //a posição angular do rotor, é a mesma para todos os sinais
//para cada uma das amostras
for(unsigned int j=0;j<signalpointer[0]->get_num_samples();j++)
   {
```

```
   printf("\rModulating...%5.1f%%",100.0*(double)j/signalpointer[0]->get_num_samples()); fflush(stdout);

   //a posição angular instantânea do rotor é a integral da velocidade angular em relação ao tempo:
   angle+=2*PI*(turbinenominalfrequency*(1.0+generator_speed_deviation[j]))/samplerate;

   //para cada um dos sinais
         for(unsigned int i=0;i<nacousticemissionsensors+naccelerometers+2;i++) //para todos os canais
            {

            //começa com os termos constantes para fazer a modulação em amplitude COM portadora:
            //os termos constantes são iguais a 0.5 (para não saturar)
            double bubble_RSI=0.5;
            if(!bubble_enabled) bubble_RSI=0.0;
            double cloud_RSI=0.5;
            if(!cloud_enabled) cloud_RSI=0.0;

            //calcula os termos da série de Fourier
            //de modo que fique 0.5 + 0.5*A*cos + 0.5*B*sin + ...
            if(bubble_enabled)
       for(unsigned int k=0;k<bubble_harmonics;k++) if((bubble_harmonic_cosine[k]!=0.0)||(bubble_harmonic_sine[k]!
=0.0))
            {
            bubble_RSI+=(bubble_harmonic_cosine[k]*0.5*cos(angle*k)+bubble_harmonic_sine[k]*0.5*sin(angle*k));
            }

            //calcula os termos da série de Fourier
            //de modo que fique 0.5 + 0.5*A*cos + 0.5*B*sin + ...
            if(cloud_enabled)
       for(unsigned int k=0;k<cloud_harmonics;k++) if((cloud_harmonic_cosine[k]!=0.0)||(cloud_harmonic_sine[k]!
=0.0))
            {
            cloud_RSI+=(cloud_harmonic_cosine[k]*0.5*cos(angle*k)+cloud_harmonic_sine[k]*0.5*sin(angle*k));
            }


       //para todos os sinais de aceleração e/ou emissão acústica:
            if(i<nacousticemissionsensors+naccelerometers)
                     {
                     double WSSNoise = 0.0;
                     if(stationary_noise_enabled) WSSNoise =
gsl_ran_gaussian(r,sqrt(stationary_noise_power_density*samplerate/2));

                     /* tira uma cópia do SOI da cloud cavitation */
                     cloud_SOI[i][j]= sqrt(cloud_RSI)*cloud_carrier[j];
                     /* tira uma cópia do SOI da bubble cavitation */
                     bubble_SOI[i][j]=sqrt(bubble_RSI)*bubble_carrier[j];
                     /* tira uma cópia do ruído aleatório estacionário */
                     random_noise[i][j]=WSSNoise;
                     /* tira uma cópia do tonal noise 1 */
                     if(tonal_noise_enabled) tonal_noise_1[i]
[j]=sqrt(2.0*tonal_noise_01_power)*cos(tonal_noise_01_freq*2*PI*j/samplerate);
                     /* tira uma cópia do tonal noise 2 */
                     if(tonal_noise_enabled) tonal_noise_2[i]
[j]=sqrt(2.0*tonal_noise_02_power)*cos(tonal_noise_02_freq*2*PI*j/samplerate);

                     /*finalmente combina todos os sinais aditivamente*/
                     (*signalpointer[i])[j] = cloud_SOI[i][j]+bubble_SOI[i][j]+random_noise[i][j]+tonal_noise_1[i]
[j]+tonal_noise_2[i][j];
                     }

       //para o sinal do encoder ótico e do gerador elétrico, faz uma cossenoide baseada no ângulo instantâneo
            if(i==nacousticemissionsensors+naccelerometers) (*signalpointer[i])[j] =
0.5*cos(angle*ngeneratorpoles/2);
            if(i==nacousticemissionsensors+naccelerometers+1) (*signalpointer[i])[j] = (*signalpointer[i-1])[j];
            }
   }
//printf("\r\nSmith trigger:\r\n");
optical_encoder_channel.SmithTrigger(0.1,-0.1);
//printf("\r\ndivide freq\r\n");
optical_encoder_channel.DivideFrequency(ngeneratorpoles/2);
//printf("\r\nmonoestable\r\n");
//optical_encoder_channel.monoestable(true,100);
printf("\rModulating...Done.            \r\n");

/*********************************************************************************************/
//Calcula a estimativa de cada potência modulada pelas cavitações:
double percent_of_bubble=0.0;
double percent_of_cloud=0.0;
for(unsigned int i=0;i<bubble_harmonics;i++)
{percent_of_bubble+=sqrt(pow(bubble_harmonic_sine[i],2.0)+pow(bubble_harmonic_cosine[i],2.0)); };
for(unsigned int i=0;i<cloud_harmonics;i++)
{percent_of_cloud+=sqrt(pow(cloud_harmonic_sine[i],2.0)+pow(cloud_harmonic_cosine[i],2.0)); };


/*Apresenta os resultados da auditoria: */
printf("\r\n\nMeasured Powers for the components:\r\n\n");
for(unsigned int i=0;i<nacousticemissionsensors+naccelerometers;i++)
   {
   printf("For channel %u:\r\n",i);
   printf("    Bubble SOI Average Power: %9.8f V²    Modulated: %9.8f V² (%5.1f%
%)\r\n",bubble_SOI[i].MeasureAveragePower(),bubble_SOI[i].MeasureAveragePower()*percent_of_bubble,percent_of_bubble*
100.0);
   printf("    Cloud  SOI Average Power: %9.8f V²    Modulated: %9.8f V² (%5.1f%
%)\r\n",cloud_SOI[i].MeasureAveragePower(),cloud_SOI[i].MeasureAveragePower()*percent_of_cloud,percent_of_cloud*100)
;
```

```
   printf("    Tonal Noise 1 Average Power: %9.8f V²\r\n",tonal_noise_1[i].MeasureAveragePower());
   printf("    Tonal Noise 2 Average Power: %9.8f V²\r\n",tonal_noise_2[i].MeasureAveragePower());
   printf("    Stationary Random Noise Average Power: %9.8f V²\r\n",random_noise[i].MeasureAveragePower());
   printf("\r\n\n");
   }
delete [] bubble_SOI;
delete [] cloud_SOI;
delete [] tonal_noise_1;
delete [] tonal_noise_2;
delete [] random_noise;
/*****************************************************************************************************/

//gera o nome do arquivo baseado na data e hora corrente
printf("Saving Results to file...\r\n"); fflush(stdout);
char buffer[800];
time_t rawtime;
struct tm * timeinfo;
time(&rawtime);
timeinfo=localtime(&rawtime);
strftime(buffer,800,"%d/%m/%Y %H:%M:%S",timeinfo);
//gera a data e grava dentro do arwquivo WAV, no campo de metadata
for(unsigned int i=0;i<accelerometers+nacousticemissionsensors+2;i++) (*signalpointer[i]).set_date(buffer);
strftime (buffer,80,"U2542A_HighSampling_Continuous_%Y%m%d_%H%M%S",timeinfo);
electric_signal::SaveSignalSetToDisk(buffer,signalpointer,accelerometers+nacousticemissionsensors+2);
free(signalpointer);

//fazer o arquivo .ini automaticamente aqui.
strftime (buffer,80,"U2542A_HighSampling_Continuous_%Y%m%d_%H%M%S.ini",timeinfo);
FILE* fout = fopen(buffer,"wt");
fprintf(fout,"[Global]\r\n\n[Turbine]\r\n\n[Generator]\r\n\n[Experiment]\r\n\n[Sensors]\r\n\n[Optical
Encoder]\r\n\n[Electric Grid]\r\n\n[COT]\r\n\n[TSA]\r\n\n[STFT]\r\n\n[CMS]\r\n\n");
fclose(fout);

dictionary* description_file = iniparser_load(buffer);

iniparser_set(description_file,"Global:Version","1.0");
iniparser_set(description_file,"Turbine:Name","Synthetic Turbine");
strftime (buffer,80,"Home, %H:%M:%S %d/%m/%Y",timeinfo);
iniparser_set(description_file,"Turbine:Location",buffer);
iniparser_set(description_file,"Turbine:Type","Francis");
sprintf(buffer,"%u",nguidevanes);
iniparser_set(description_file,"Turbine:Guide Vanes Number",buffer);
sprintf(buffer,"%u",nrunnerblades);
iniparser_set(description_file,"Turbine:Runner Blades Number",buffer);
sprintf(buffer,"%u",ngeneratorpoles);
iniparser_set(description_file,"Generator:Number Of Poles",buffer);
sprintf(buffer,"%4.1f",generatornominalfrequency);
iniparser_set(description_file,"Generator:Nominal Frequency",buffer);
iniparser_set(description_file,"Experiment:Power Setting","N/A");
iniparser_set(description_file,"Experiment:Guide Vane Openning","N/A");
sprintf(buffer,"%u",accelerometers);
iniparser_set(description_file,"Sensors:Number of Accelerometers",buffer);
for(unsigned int i=0;i<accelerometers;i++)
   {
   sprintf(buffer,"Sensors:Accelerometer %02u Channel",i+1);
   char buf[80];
   sprintf(buf,"%u",i);
   iniparser_set(description_file,buffer,buf);
   sprintf(buffer,"Sensors:Accelerometer %02u Location",i+1);
   iniparser_set(description_file,buffer,"N/A");
   }
sprintf(buffer,"%u",nacousticemissionsensors);
iniparser_set(description_file,"Sensors:Number of Acoustic Emission Sensors",buffer);
for(unsigned int i=0;i<nacousticemissionsensors;i++)
   {
   sprintf(buffer,"Sensors:Acoustic Emission Sensor %02u Channel",i+1);
   char buf[80];
   sprintf(buf,"%u",i);
   iniparser_set(description_file,buffer,buf);
   sprintf(buffer,"Sensors:Acoustic Emission Sensor %02u Location",i+1);
   iniparser_set(description_file,buffer,"N/A");
   }
sprintf(buffer,"%u",nacousticemissionsensors+naccelerometers);
iniparser_set(description_file,"Optical Encoder:Optical Encoder Channel",buffer);
iniparser_set(description_file,"Optical Encoder:Pre-process","No");
iniparser_set(description_file,"Optical Encoder:Histeresis High Level","+0.1");
iniparser_set(description_file,"Optical Encoder:Histeresis Low Level","-0.1");
iniparser_set(description_file,"Optical Encoder:Perform TSA","No");
sprintf(buffer,"%u",nacousticemissionsensors+naccelerometers+1);
iniparser_set(description_file,"Electric Grid:Electric Grid Channel",buffer);
iniparser_set(description_file,"Electric Grid:Pre-process","No");
iniparser_set(description_file,"Electric Grid:Perform TSA","No");
iniparser_set(description_file,"COT:Use Resampling","Yes");
sprintf(buffer,"%u",nacousticemissionsensors+naccelerometers+1);
iniparser_set(description_file,"COT:Synchronism Channel",buffer);
iniparser_set(description_file,"TSA:Perform TSA Analysis","No");
iniparser_set(description_file,"TSA:Use Resampling","No");
sprintf(buffer,"%u",nacousticemissionsensors+naccelerometers);
iniparser_set(description_file,"TSA:Synchronism Channel",buffer);
iniparser_set(description_file,"TSA:Invert Graphics","No");
iniparser_set(description_file,"STFT:Perform STFT Analysis","Yes");
iniparser_set(description_file,"STFT:STFT Window Type","Hanning");
iniparser_set(description_file,"STFT:Window Size","512");
iniparser_set(description_file,"STFT:Window Overlap Index","2");
iniparser_set(description_file,"STFT:Generate Graphic","No");
```

```
iniparser_set(description_file,"STFT:Scaled Graphic Output","No");
iniparser_set(description_file,"STFT:Invert Graphic","Yes");
iniparser_set(description_file,"STFT:Use RMS","No");
iniparser_set(description_file,"STFT:Use PWR","Yes");
iniparser_set(description_file,"STFT:Use Density","No");
iniparser_set(description_file,"STFT:HighPass Cutoff Frequency","5000");
iniparser_set(description_file,"CMS:Perform CMS Analysis","Yes");
iniparser_set(description_file,"CMS:Generate Graphic","No");
iniparser_set(description_file,"CMS:Scaled Graphic Output","No");
iniparser_set(description_file,"CMS:Invert Graphic","No");

strftime (buffer,80,"U2542A_HighSampling_Continuous_%Y%m%d_%H%M%S.ini",timeinfo);
fout = fopen(buffer,"wt");
iniparser_dump_ini(description_file,fout);
fclose(fout);
iniparser_freedict(description_file);

printf("Freeing Memory...");
while(!accelerometer_channel.empty()) accelerometer_channel.pop_back();
while(!acoustic_emission_channel.empty()) acoustic_emission_channel.pop_back();
free(bubble_filter_numerator);
free(bubble_filter_denominator);
free(cloud_filter_numerator);
free(cloud_filter_denominator);
free(bubble_harmonic_cosine);
free(bubble_harmonic_sine);
free(cloud_harmonic_cosine);
free(cloud_harmonic_sine);
printf("Done\r\n");
return 0;
}
```

Arquivo **detectcorrelations.cc**:

```
#include<stdlib.h>
#include<stdio.h>
#include <string.h>
#include <math.h>
extern "C" {
#include <iniparser.h>
}
#include "matrices.h"
#include "grpGL.h"
#include <vector>
#include <dirent.h>
#include <wavfile.h>

/*****************************Variáveis globais*************************************************/

string basefilename;
string inifilename;
DIR* directory_path;
dirent* directory_entry;
vector <string> datafile;
unsigned int nguidevanes;         //número de palhetas do distribuidor
unsigned int nrunnerblades;       //número de pás do rotor
unsigned int ngeneratorpoles;     //número de polos do gerador de tensão
double turbinenominalfrequency;
double generatornominalfrequency;
bool was_used_density;
Matrix datamatrix;
unsigned int samples_per_turn;
unsigned long int n_samples;
unsigned int n_turns;
bool calculate_average;
double corr_threshold;


int main(int argc,char** argv)
{
/*************************************************************************************************/
printf("\r\nExtractor by Rafael Linhares Marinho (28/11/2014).\r\n\n");
/***************************** Inicialização *****************************************************/
//primeiro checa se o número de argumentos está correto:
if (argc < 2)
   {
   printf("Usage: %s <inputfile without .wav> [optional flags]\r\n",argv[0]);
   exit(EXIT_FAILURE);
   }

calculate_average = true;
for (int i = 1; i < argc; i++)
         if (strcmp(argv[i], "-na") == 0) calculate_average = false;

corr_threshold = 0.6;
for (int i = 1; i < argc; i++)
         if (strncmp(argv[i], "-t=",3) == 0)
            {
            printf("Using threshold to Pearson coefficient: ");
            argv[i][0]=' '; argv[i][1]=' '; argv[i][2]=' ';
```

```
       corr_threshold=atof(argv[i]);
       printf("%f\r\n",corr_threshold);
            }

basefilename=string(argv[1]);
inifilename=basefilename+".ini";
printf("Processing files:\r\n");
directory_path = opendir(".");
datafile.clear();
if(directory_path)
  {
  while(true)
       {
       directory_entry = readdir(directory_path);
       if(directory_entry==NULL) break;
       string filename = string(directory_entry->d_name);
       //aqui vem a filtragem dos nomes dos arquivos
       if((filename.length()-filename.rfind(".dat"))!=4) continue; //se não termina em ".dat", pula
       if(filename.find(basefilename)!=0) continue; //se o nome não começa pelo basefilename, pula
       if(filename.find("CM")==string::npos) continue; //se o nome do arquivo não tiver "CM", pula
       //fim da filtragem
       datafile.push_back(filename);
       }
  closedir(directory_path);
  }
else
  {
  printf("No files to process ? Are you sure ?\r\n Aborting.");
  exit(EXIT_FAILURE);
  }
//lista de arquivos a serem processados pronta
for(unsigned int i=0;i<datafile.size();i++) printf("%s\r\n",datafile[i].c_str());
printf("\r\n\n");

/*****************************************************************************************************/
//carrega o arquivo txt contendo a descrição do experimento:
dictionary* description_file = iniparser_load(inifilename.c_str());
if (!description_file)
    { /*printf("File %s not found. Aborting.",(string(argv[1])+".ini").c_str()); */
         exit(EXIT_FAILURE);
         }

nguidevanes = iniparser_getint(description_file,"Turbine:Guide Vanes Number", 0);
nrunnerblades = iniparser_getint(description_file,"Turbine:Runner Blades Number", 0);
ngeneratorpoles = iniparser_getint(description_file,"Generator:Number Of Poles", 0);
generatornominalfrequency = iniparser_getdouble(description_file,"Generator:Nominal Frequency", 0.0);
turbinenominalfrequency= 2.0*generatornominalfrequency/ngeneratorpoles;
was_used_density = iniparser_getboolean(description_file,"STFT:Use Density",true);
samples_per_turn = iniparser_getint(description_file,"cot:samples per turn",0);

iniparser_freedict(description_file);
/*****************************************************************************************************/

WavFile* wavfile = new WavFile(string(basefilename+"-COT.wav").c_str());

n_samples = wavfile->get_num_samples();
delete wavfile;
n_turns = n_samples/samples_per_turn;
printf("Detected %u complete turns.\r\n",n_turns);

for(unsigned int i=0;i<datafile.size();i++) //para cada um dos arquivos de dados encontrados
   {
   printf("\r\n\nProcessing file %s...\r\n",datafile[i].c_str());
   datamatrix.LoadFromFile(datafile[i].c_str());
   //datamatrix.PrintOut();
   printf("Matrix file %s loaded with %u rows and %u
columns.\r\n",datafile[i].c_str(),datamatrix.GetNumRows(),datamatrix.GetNumColumns());

   //inicio do processamento da matriz
   unsigned int ncolumns = datamatrix.GetNumColumns();
   unsigned int nrows = datamatrix.GetNumRows();
   bool is_CMC = false;
   bool is_CMS = false;
   if(datafile[i].find("CMS")!=string::npos) is_CMS = true;
   if(datafile[i].find("CMC")!=string::npos) is_CMC = true;
   if((!is_CMS)&&(!is_CMC)) { printf("Unknown file. I don't know what it is.\r\n"); exit(EXIT_FAILURE);}

   if(is_CMS) //se é CMS, os elementos da matriz têm significado de potência
     {
          //double total_power = 0.0;
          double stationary_power = 0.0;
          double bubble_power = 0.0;
          double cloud_power = 0.0;

          double sum_x;
          double sum_y;
          double sum_xy;
          double sum_x_squared;
          double sum_y_squared;


          double* cps = (double*)malloc(sizeof(double)*ncolumns);
          double* corr= (double*)malloc(sizeof(double)*ncolumns);
          for(unsigned int column=0;column<ncolumns;column++) { cps[column]=0.0; corr[column]=0.0; };
```

```
          //acha a soma de todas as potências da coluna 0 (parte estacionária):
          for(unsigned int row=0;row<nrows;row++) stationary_power+=cabs(datamatrix[row][0]);
          printf("Total stationary Power: %e V²\r\n",stationary_power);

          //para a bubble cavitation, a coluna de referência é a do número de lâminas do rotor.
          sum_x=0.0;
          sum_x_squared=0.0;
          for(unsigned int row=0;row<nrows;row++)
             {
                   if(calculate_average)
                     {
                     sum_x+=cabs(datamatrix[row][nrunnerblades]);
                     sum_x_squared+=pow(cabs(datamatrix[row][nrunnerblades]),2.0);
                     }
                   else
                     {
                     sum_x+=cabs(datamatrix[row][nrunnerblades*n_turns]);
                     sum_x_squared+=pow(cabs(datamatrix[row][nrunnerblades*n_turns]),2.0);
                     }
                   }

     for(unsigned int column=0;column<ncolumns;column++) //para todas as colunas:
        {
         //calcula o somatório de y, y ao quadrado e xy
         sum_y=0.0;
         sum_xy=0.0;
         sum_y_squared=0.0;
         for(unsigned int row=0;row<nrows;row++)
           {
          sum_y+=cabs(datamatrix[row][column]);
          sum_y_squared+=pow(cabs(datamatrix[row][column]),2.0);
          if(calculate_average) sum_xy+=cabs(datamatrix[row][column])*cabs(datamatrix[row][nrunnerblades]);
          else sum_xy+=cabs(datamatrix[row][column])*cabs(datamatrix[row][nrunnerblades*n_turns]);
           }
         //finalmente aplica a fórmula para o coeficiente de correlação de Pearson
        corr[column]=(nrows*sum_xy-sum_x*sum_y)/(sqrt((nrows*sum_x_squared-sum_x*sum_x)*(nrows*sum_y_squared-
sum_y*sum_y)));
         //printf("column[%u]: sum_x=%f sum_y=%f sum_xy=%f sum_x_squared=%f sum_y_squared=
%f\r\n",column,sum_x,sum_y,sum_xy,sum_x_squared,sum_y_squared);
         //aproveita para calcular a potência total em cada frequencia cíclica:
         if((corr[column]>corr_threshold)&&(column!=0)) cps[column]=sum_y; else cps[column]=0.0;
        // if(column==0) cps[0]=sum_y;
        // if(column!=0) printf("%u %9.6f%%\r\n",column,cps[column]*1000.0/cps[0]);
         }

     cps[0]=0.0;
     {
     GL2D* graPH = new GL2D(3740,984);
     graPH->SetMainTitle("\\hat{P}_{X}(\\alpha) (Cyclic Power Spectrum)");
     graPH->SetSubTitle(datafile[i]);
     if(calculate_average) graPH->SetXrange(0.0,(ncolumns-1)*turbinenominalfrequency);
     else graPH->SetXrange(0.0,(ncolumns-1)*turbinenominalfrequency/n_turns);
     graPH->SetXscaleTitle("cyclic frequency (Hz)");
     graPH->SetYscaleTitle("Cyclic Power Spectrum (V^2)");
     graPH->SetXNormalizationFactor(turbinenominalfrequency);
     graPH->SetNormalizedXscaleTitle("machine orders (Hz/Hz)");
     graPH->SaveToFile("bubble-CPS-"+datafile[i]);
     graPH->PlotSignal(cps,ncolumns,"");
     delete graPH;
     }

     {
     GL2D* graPH = new GL2D(3740,984);
     graPH->SetMainTitle("\\hat{P}_{X}(\\alpha) (Correlation with BPF)");
     graPH->SetSubTitle(datafile[i]);
     if(calculate_average) graPH->SetXrange(0.0,(ncolumns-1)*turbinenominalfrequency);
     else graPH->SetXrange(0.0,(ncolumns-1)*turbinenominalfrequency/n_turns);
     graPH->SetXscaleTitle("cyclic frequency (Hz)");
     graPH->SetYscaleTitle("correlation with BPF");
     graPH->SetXNormalizationFactor(turbinenominalfrequency);
     graPH->SetNormalizedXscaleTitle("machine orders (Hz/Hz)");
     graPH->SaveToFile("bubble-COR-"+datafile[i]);
     graPH->PlotSignal(corr,ncolumns,"");
     delete graPH;
     }

          //acha a soma de todas as potências da coluna da bubble cavitation:
          for(unsigned int column=nrunnerblades;column<ncolumns;column+=nrunnerblades) bubble_power+=cps[column];
          //subtrai a parte contaminada com a interferência do gerador
          for(unsigned int column=ngeneratorpoles;column<ncolumns;column+=ngeneratorpoles) bubble_power-
=cps[column];
          bubble_power*=2.0; //correção devido à componente de ordem de máquina zero
          printf("Total power modulated by Bubble Cavitation: %e V² (%f %
%)\r\n",bubble_power,100.0*bubble_power/stationary_power);

          //para a cloud cavitation, a coluna de referência é a do número de lâminas do rotor.
          for(unsigned int column=0;column<ncolumns;column++) { cps[column]=0.0; corr[column]=0.0; };
                    sum_x=0.0;
                    sum_x_squared=0.0;
                    for(unsigned int row=0;row<nrows;row++)
                          {
                                      if(calculate_average)
                                        {
                                        sum_x+=cabs(datamatrix[row][nguidevanes]);
                                        sum_x_squared+=pow(cabs(datamatrix[row][nguidevanes]),2.0);
```

```
                                    }
                                  else
                                    {
                                    sum_x+=cabs(datamatrix[row][nguidevanes*n_turns]);
                                    sum_x_squared+=pow(cabs(datamatrix[row][nguidevanes*n_turns]),2.0);
                                    }
                                  }

               for(unsigned int column=0;column<ncolumns;column++) //para todas as colunas:
                  {
                   //calcula o somatório de y, y ao quadrado e xy
                   sum_y=0.0;
                   sum_xy=0.0;
                   sum_y_squared=0.0;
                   for(unsigned int row=0;row<nrows;row++)
                      {
                     sum_y+=cabs(datamatrix[row][column]);
                     sum_y_squared+=pow(cabs(datamatrix[row][column]),2.0);
                     if(calculate_average) sum_xy+=cabs(datamatrix[row][column])*cabs(datamatrix[row][nguidevanes]);
                     else sum_xy+=cabs(datamatrix[row][column])*cabs(datamatrix[row][nguidevanes*n_turns]);
                      }
                   //finalmente aplica a fórmula para o coeficiente de correlação de Pearson
                  corr[column]=(nrows*sum_xy-sum_x*sum_y)/(sqrt((nrows*sum_x_squared-
sum_x*sum_x)*(nrows*sum_y_squared-sum_y*sum_y)));
                  //printf("column[%u]: sum_x=%f sum_y=%f sum_xy=%f sum_x_squared=%f sum_y_squared=
%f\r\n",column,sum_x,sum_y,sum_xy,sum_x_squared,sum_y_squared);
                  //aproveita para calcular a potência total em cada frequencia cíclica:
                  if((corr[column]>corr_threshold)&&(column!=0)) cps[column]=sum_y; else cps[column]=0.0;
                  }

               cps[0]=0.0;
                    {
                    GL2D* graPH = new GL2D(3740,984);
                    graPH->SetMainTitle("\\hat{P}_{X}(\\alpha) (Cyclic Power Spectrum)");
                    graPH->SetSubTitle(datafile[i]);
                    if(calculate_average) graPH->SetXrange(0.0,(ncolumns-1)*turbinenominalfrequency);
                    else graPH->SetXrange(0.0,(ncolumns-1)*turbinenominalfrequency/n_turns);
                    graPH->SetXscaleTitle("cyclic frequency (Hz)");
                    graPH->SetYscaleTitle("Cyclic Power Spectrum (V^2)");
                    graPH->SetXNormalizationFactor(turbinenominalfrequency);
                    graPH->SetNormalizedXscaleTitle("machine orders (Hz/Hz)");
                    graPH->SaveToFile("cloud-CPS-"+datafile[i]);
                    graPH->PlotSignal(cps,ncolumns,"");
                    delete graPH;
                    }

                    {
                    GL2D* graPH = new GL2D(3740,984);
                    graPH->SetMainTitle("\\hat{P}_{X}(\\alpha) (Correlation with VPF)");
                    graPH->SetSubTitle(datafile[i]);
                    if(calculate_average) graPH->SetXrange(0.0,(ncolumns-1)*turbinenominalfrequency);
                    else graPH->SetXrange(0.0,(ncolumns-1)*turbinenominalfrequency/n_turns);
                    graPH->SetXscaleTitle("cyclic frequency (Hz)");
                    graPH->SetYscaleTitle("correlation with VPF");
                    graPH->SetXNormalizationFactor(turbinenominalfrequency);
                    graPH->SetNormalizedXscaleTitle("machine orders (Hz/Hz)");
                    graPH->SaveToFile("cloud-COR-"+datafile[i]);
                    graPH->PlotSignal(corr,ncolumns,"");
                    delete graPH;
                    }

          //acha a soma de todas as potências da coluna da cloud cavitation:
          for(unsigned int column=nguidevanes;column<ncolumns;column+=nguidevanes) cloud_power+=cps[column];
          cloud_power*=2.0; //correção devido à componente de ordem de máquina zero
          printf("Total power modulated by Cloud Cavitation: %e V² (%f %
%)\r\n",cloud_power,100.0*cloud_power/stationary_power);

          free(corr);
          free(cps);
     }



   if(is_CMC) //se é CMC, os elementos da matriz têm significado de índice de modulação em potência
     {
          //double total_modulation = 0.0;
          //double bubble_modulation = 0.0;
          //double cloud_modulation = 0.0;
     }
   }

exit(EXIT_SUCCESS);
}
```

Arquivo **STFT-iSTFT.cc**:

```
#include "STFT-iSTFT.h"
```

```
#include <stdlib.h>
#include <math.h>
#include <fftw3.h>
#include <omp.h>

//construtor padrão
STFT_Computer::STFT_Computer()
{
this->samplingrate=0;
this->window_duration=0.0;
this->sample_duration = 0.0;
this->n_lines = 0;
this->n_columns = 0;
this->windowsize = 0;
this->stepsize = 0;
this->overlap_index = 0;
this->nominalperiod = 0.0;
this->time_limit = 0.0;
this->spectral_freq_limit = 0.0;
this->time_resolution = 0.0;
this->spectral_freq_resolution = 0.0;
this->window = NULL;
this->Kw=NULL;
this->original_samplingrate = 0;
this->S1 = 0.0;
this->S2 = 0.0;
this->NENBW = 0.0;
this->ENBW = 0.0;
}

double STFT_Computer::get_ENBW(void) const
{
return this->ENBW;
}

double* STFT_Computer::get_Kw(void) const
{
return this->Kw;
}

//destrutor
STFT_Computer::~STFT_Computer()
{
if(this->window) free(this->window);
if(this->Kw) free(this->Kw);
}

void STFT_Computer::set_nominal_period(double nomperiod) //configura o período nominal de rotação da máquina
{
this->nominalperiod=nomperiod;
}

void STFT_Computer::set_experience_name(const string name)
{
this->experience_name = string(name);
}

double STFT_Computer::get_time_resolution(void) const
{
return this->time_resolution;
}

double STFT_Computer::get_freq_resolution(void) const
{
return this->spectral_freq_resolution;
}

double STFT_Computer::get_time_limit(void) const
{
return this->time_limit;
}

double STFT_Computer::get_freq_limit(void) const
{
return this->spectral_freq_limit;
}

unsigned int STFT_Computer::get_n_columns(void) const //retorna o número de colunas que a STFT tem
{
return this->n_columns;
}

unsigned int STFT_Computer::get_n_lines(void) const //retorna o número de linhas que a STFT tem
{
return this->n_lines;
}

void STFT_Computer::set_overlap_index(unsigned int overlap) //configura o overlap index
{
this->overlap_index=overlap;
//a porcentagem de overlap é 100/(2^overlap) %
printf(" Overlap Factor %4.1f %%\r\n",100*(1.0-pow(0.5,this->overlap_index)));
this->stepsize = this->windowsize/pow(2,this->overlap_index);
}
```

```
unsigned int STFT_Computer::get_window_size(void) const //retorna o tamanho da janela
{
return this->windowsize;
}

unsigned int STFT_Computer::get_step_size(void) const //retorna o tamanho do step em amostras
{
return this->stepsize;
}


void STFT_Computer::setup_window(const string windowtype, unsigned int N,unsigned int overlap) //configura o tipo e
tamanho de janela que será usado
{
int window_number=0;
int window_size=0;
if(windowtype=="Rectangular") window_number=RECTANGULAR_WINDOW; //1
if(windowtype=="Barlett") window_number=BARLETT_WINDOW;         //2
if(windowtype=="Hanning") window_number=HANNING_WINDOW;         //3
if(windowtype=="Hamming") window_number=HAMMING_WINDOW;         //4
if(windowtype=="Blackman") window_number=BLACKMAN_WINDOW;       //5
if(windowtype=="Gaussian") window_number=GAUSSIAN_WINDOW;       //6
if(window_number==0) { printf("Invalid Window Type. Review your description file!\r\n"); exit(EXIT_FAILURE);}

window_size=round(log(N)/log(2.0));

this->overlap_index=overlap;

this->generate_window(window_number,window_size);

this->window_name=string(windowtype);
this->windowsize=pow(2.0,window_size);
printf("STFT: Created window type %s with %u samples. ",this->window_name.c_str(),this->windowsize);

//a porcentagem de overlap é 100/(2^overlap) %
printf(" Overlap Factor %4.1f %%\r\n",100*(1.0-pow(0.5,this->overlap_index)));
this->stepsize = this->windowsize/pow(2,this->overlap_index);

}

double* STFT_Computer::get_window(void) const//plota a window usada
{
return this->window;
}

//calcula STFT do sinal de entrada, retorna uma matriz de complexos
Matrix STFT_Computer::Compute_STFT(electric_signal input)
{
//primeiro checa se a janela já foi configurada:
if(this->windowsize==0) { printf("Error: STFT Window not setup\r\n"); exit(EXIT_FAILURE); };

double* sample = input.get_samples_pointer();

//e o número de linhas é n/2+1 pois a FFT é de um vetor real (simetria hermitiana)
this->n_lines = floor(this->windowsize/2)+1; //lembrar que uma linha é o valor DC(parte imaginária nula)
//pega o samplingarate do sinal de entrada
this->samplingrate = input.get_samplingrate();
//o limite em frequência é a frequência de Nyquist
this->spectral_freq_limit = (double)this->samplingrate/2.0;
//calcula a resolução temporal baseando-se no step da janela e na samplingrate
this->time_resolution = (double)this->stepsize/samplingrate;

//o número de colunas da STFT é igual ao número de janelas que cabem no comprimento do sinal
this->n_columns = 1+(input.get_num_samples()-this->windowsize)/this->stepsize;

//o limite em tempo é o número de colunas x a resolução temporal
this->time_limit = this->n_columns*this->time_resolution;

//calcula a frequencia fundamental baseando-se na definição da DFT. Esta é a resolução em frequencia
this->spectral_freq_resolution = this->spectral_freq_limit/(this->n_lines-1);
this->ENBW = this->NENBW * this->spectral_freq_resolution;

printf("STFT: Time resolution: %u intervals of %f s. Frequency resolution: %u bands of %f Hz.\r\n",this-
>n_columns,this->time_resolution,this->n_lines-1,this->spectral_freq_resolution);

//printf("Columns: %u  Lines %u\r\n",this->n_columns,this->n_lines);

//aloca a matriz resultante, a própria Short Time Fourier Transform:
Matrix returnmatrix(this->n_lines,this->n_columns);

unsigned int nlines=this->n_lines;
unsigned int ncolumns=this->n_columns;
unsigned int wsize=this->windowsize;
double* windowp = this->window;
unsigned int StepSize = this->stepsize;
unsigned int i;
double S1 = this->S1;
fftw_cleanup();

#pragma omp parallel default(none) private(i)
shared(S1,wsize,nlines,ncolumns,windowp,StepSize,returnmatrix,sample,stdout,input)
{
//aloca os vetores de entrada e de saída para as FFTs:
//aloca um vetor de entrada de doubles e um vetor de saída de complexos para a FFT:
double* in = fftw_alloc_real(wsize);
```

```
fftw_complex* out = fftw_alloc_complex(nlines);

fftw_plan plan;

//faz o plano de uma FFT:
#pragma omp critical (make_plan)
{
printf("STFT: Planning FFTs..."); fflush(stdout);
plan = fftw_plan_dft_r2c_1d(wsize,in,out,FFTW_ESTIMATE);
printf("Done.\r\n");
}

#pragma omp barrier

printf("STFT: CPU %u Computing Power Short Time Fourier Transform for signal
%s...\r\n",omp_get_thread_num(),input.get_name()); fflush(stdout);
//otimizar para usar o openMP
#pragma omp for
for(i=0;i<ncolumns;i++) // para cada coluna
   {
   //copia um pedaço do sinal de entrada para o buffer de entrada da FFTW, já aplicando a janela:
   for(register unsigned int j=0;j<wsize;j++) in[j]=windowp[j]*sample[j+(i*StepSize)];

   //executa o plano previamente criado:
   fftw_execute(plan);

   for(register unsigned int j=0;j<nlines;j++) out[j] = (2.0*pow(cabs(out[j]),2)/pow(S1,2)+I*carg(out[j]));
   out[0]/=2.0;
   //copia o buffer de saída da FFTW para a matriz de resultados:
   returnmatrix.SetColumn(out,i);
   }

//fim do cálculo
printf("Done.\r\n"); fflush(stdout);
//desaloca o plano
fftw_destroy_plan(plan);
//desaloca os vetores de entrada e saída da FFT
fftw_free(in);
fftw_free(out);

} //fim do pragma omp parallel

//aqui é necessário eliminar a parte de cima do espectrograma, se o sinal foi reamostrado
if(this->original_samplingrate!=0)
  {
  unsigned int newlines = floor(((double)this->original_samplingrate/2.0)/this->spectral_freq_resolution)+1.0;
  this->spectral_freq_limit = (newlines-1)*this->spectral_freq_resolution;
  Matrix aux(newlines,this->n_columns);
  //printf("Correcting to %u lines\r\n",newlines);
  for(register unsigned int i=0;i<newlines;i++)
           for(register unsigned int j=0;j<this->n_columns;j++)
              {
                    aux[i][j]=returnmatrix[i][j];
              }
  returnmatrix = aux;
  }

return returnmatrix;
}


//calcula STFT do sinal de entrada, retorna uma matriz de complexos
Matrix STFT_Computer::Compute_Mean_STFT(electric_signal input,unsigned int samples_per_turn)
{
//primeiro checa se a janela já foi configurada:
if(this->windowsize==0) { printf("Error: STFT Window not setup\r\n"); exit(EXIT_FAILURE); };
//pega o ponteiro do início dos dados do sinal de entrada para poder acessar mais rápido
double* sample = input.get_samples_pointer();
//e o número de linhas é n/2+1 pois a FFT é de um vetor real (simetria hermitiana)
this->n_lines = floor(this->windowsize/2)+1; //lembrar que uma linha é o valor DC(parte imaginária nula)
//pega o samplingarate do sinal de entrada
this->samplingrate = input.get_samplingrate();
//o limite em frequência é a frequência de Nyquist
this->spectral_freq_limit = (double)this->samplingrate/2.0;
//calcula a resolução temporal baseando-se no step da janela e na samplingrate
this->time_resolution = (double)this->stepsize/samplingrate;
//o número de colunas da STFT é igual ao número de steps que cabem por volta
unsigned long int n_turns = input.get_num_samples()/samples_per_turn;
this->n_columns = samples_per_turn/this->stepsize;
//o limite em tempo é o número de colunas x a resolução temporal
this->time_limit = this->n_columns*this->time_resolution;
//calcula a frequencia fundamental baseando-se na definição da DFT. Esta é a resolução em frequencia
this->spectral_freq_resolution = this->spectral_freq_limit/(this->n_lines-1);
this->ENBW = this->NENBW * this->spectral_freq_resolution;
//imprime alguns dados:
printf("STFT: Averaging %lu of %9f turns.\r\n",n_turns,((double)input.get_num_samples()/samples_per_turn));
printf("STFT: Time resolution: %u intervals of %f s. Frequency resolution: %u bands of %f Hz.\r\n",this-
>n_columns,this->time_resolution,this->n_lines-1,this->spectral_freq_resolution);

//aloca a matriz resultante, a própria Short Time Fourier Transform Averaged:
Matrix returnmatrix(this->n_lines,this->n_columns);
//transforma alguns dos parâmetros numéricos para variáveis locais a fim de deixar o OpenMP paralelizar
unsigned int nlines=this->n_lines;
unsigned int ncolumns=this->n_columns;
unsigned int wsize=this->windowsize;
```

```
double* windowp = this->window;
unsigned int StepSize = this->stepsize;
complex double** returndata = returnmatrix.GetData();
unsigned int i;
fftw_cleanup();

#pragma omp parallel default(none) private(i)
shared(n_turns,wsize,nlines,ncolumns,windowp,StepSize,returndata,sample,stdout,input)
{
//aloca os vetores de entrada e de saída para as FFTs:
//aloca um vetor de entrada de doubles e um vetor de saída de complexos para a FFT:
double* in = fftw_alloc_real(wsize);
fftw_complex* out = fftw_alloc_complex(nlines);
//cria um objeto que vai guardar os planos de uma FFT:
fftw_plan plan;

//faz o plano de uma FFT:
#pragma omp critical (make_plan)
{
printf("STFT: Planning FFTs for CPU %u...",omp_get_thread_num()); fflush(stdout);
plan = fftw_plan_dft_r2c_1d(wsize,in,out,FFTW_ESTIMATE);
printf("Done.\r\n"); fflush(stdout);
}


#pragma omp barrier

printf("STFT: CPU %u Computing Averaged Modulus of Short Time Fourier Transform for signal
%s...\r\n",omp_get_thread_num(),input.get_name()); fflush(stdout);

#pragma omp for
for(i=0;i<ncolumns;i++) // para cada coluna da matriz resultado
   {
   //para cada uma das voltas da turbina
   for(register unsigned int k=0; k<n_turns;k++)
      {
            //copia um pedaço do sinal de entrada da k-ésima volta para o buffer de entrada da FFTW, já aplicando a
janela:
            for(register unsigned int j=0;j<wsize;j++) in[j]=windowp[j]*sample[j+((i+k*ncolumns)*StepSize)];
            //executa o plano previamente criado:
            fftw_execute(plan);
            //copia o buffer de saída da FFTW para a matriz de resultados:
            //já multiplica por 2 pois a FFTW só computa as frequências positivas
            //A FFTW direta obriga a multiplicar o resultado por 1/windowsize
            //mas o janelamento obriga a dividir por windowsize/stepsize
            for(unsigned int j=0;j<nlines;j++) returndata[j][i]+=(2.0*cabs(out[j])/StepSize+I*carg(out[j]))/n_turns;
            // corrige a frequência 0:
            }
   returndata[0][i]/=2.0;
   }

//fim do cálculo

//desaloca o plano
fftw_destroy_plan(plan);
//desaloca os vetores de entrada e saída da FFT
fftw_free(in);
fftw_free(out);
} //fim do #pragma omp parallel default(none) private(i)

printf("Done.\r\n"); fflush(stdout);

//aqui é necessário eliminar a parte de cima do espectrograma, se o sinal foi reamostrado
if(this->original_samplingrate!=0)
  {
  unsigned int newlines = floor(((double)this->original_samplingrate/2.0)/this->spectral_freq_resolution)+1.0;
  this->spectral_freq_limit = (newlines-1)*this->spectral_freq_resolution;
  Matrix aux(newlines,this->n_columns);
  //printf("Correcting to %u lines\r\n",newlines);
  for(register unsigned int i=0;i<newlines;i++)
            for(register unsigned int j=0;j<this->n_columns;j++)
               {
                     aux[i][j]=returnmatrix[i][j];
               }
  returnmatrix = aux;
  }


return returnmatrix;
}


//calcula STFT do sinal de entrada, retorna uma matriz de complexos
Matrix STFT_Computer::Compute_RMS_STFT(electric_signal input,unsigned int samples_per_turn)
{
//primeiro checa se a janela já foi configurada:
if(this->windowsize==0) { printf("Error: STFT Window not setup\r\n"); exit(EXIT_FAILURE); };
//pega o ponteiro do início dos dados do sinal de entrada para poder acessar mais rápido
double* sample = input.get_samples_pointer();
//e o número de linhas é n/2+1 pois a FFT é de um vetor real (simetria hermitiana)
this->n_lines = floor(this->windowsize/2)+1; //lembrar que uma linha é o valor DC(parte imaginária nula)
//pega o samplingarate do sinal de entrada
this->samplingrate = input.get_samplingrate();
//o limite em frequência é a frequência de Nyquist
this->spectral_freq_limit = (double)this->samplingrate/2.0;
```

```
//calcula a resolução temporal baseando-se no step da janela e na samplingrate
this->time_resolution = (double)this->stepsize/samplingrate;
//o número de colunas da STFT é igual ao número de steps que cabem por volta
unsigned long int n_turns = input.get_num_samples()/samples_per_turn;
this->n_columns = samples_per_turn/this->stepsize;
//o limite em tempo é o número de colunas x a resolução temporal
this->time_limit = this->n_columns*this->time_resolution;
//calcula a frequencia fundamental baseando-se na definição da DFT. Esta é a resolução em frequencia
this->spectral_freq_resolution = this->spectral_freq_limit/(this->n_lines-1);
this->ENBW = this->NENBW * this->spectral_freq_resolution;
//imprime alguns dados:
printf("STFT: Averaging %lu of %9f turns.\r\n",n_turns,((double)input.get_num_samples()/samples_per_turn));
printf("STFT: Time resolution: %u intervals of %f s. Frequency resolution: %u bands of %f Hz.\r\n",this-
>n_columns,this->time_resolution,this->n_lines-1,this->spectral_freq_resolution);

//aloca a matriz resultante, a própria Short Time Fourier Transform Averaged:
Matrix returnmatrix(this->n_lines,this->n_columns);
//transforma alguns dos parâmetros numéricos para variáveis locais a fim de deixar o OpenMP paralelizar
unsigned int nlines=this->n_lines;
unsigned int ncolumns=this->n_columns;
unsigned int wsize=this->windowsize;
double* windowp = this->window;
unsigned int StepSize = this->stepsize;
complex double** returndata = returnmatrix.GetData();
unsigned int i;
double S1 = this->S1;
fftw_cleanup();

#pragma omp parallel default(none) private(i)
shared(S1,n_turns,wsize,nlines,ncolumns,windowp,StepSize,returndata,sample,stdout,input)
{
//aloca os vetores de entrada e de saída para as FFTs:
//aloca um vetor de entrada de doubles e um vetor de saída de complexos para a FFT:
double* in = fftw_alloc_real(wsize);
fftw_complex* out = fftw_alloc_complex(nlines);
//cria um objeto que vai guardar os planos de uma FFT:
fftw_plan plan;

//faz o plano de uma FFT:
#pragma omp critical (make_plan)
{
printf("STFT: Planning FFTs for CPU %u...",omp_get_thread_num()); fflush(stdout);
plan = fftw_plan_dft_r2c_1d(wsize,in,out,FFTW_ESTIMATE);
printf("Done.\r\n"); fflush(stdout);
}


#pragma omp barrier

printf("STFT: CPU %u Computing RMS of Short Time Fourier Transform for signal
%s...\r\n",omp_get_thread_num(),input.get_name()); fflush(stdout);

#pragma omp for
for(i=0;i<ncolumns;i++) // para cada coluna da matriz resultado
   {
   for(register unsigned int k=0; k<n_turns;k++)
      {
            //copia um pedaço do sinal de entrada da k-ésima volta para o buffer de entrada da FFTW, já aplicando a
janela:
            for(register unsigned int j=0;j<wsize;j++) in[j]=windowp[j]*sample[j+((i+k*ncolumns)*StepSize)];
            //executa o plano previamente criado:
            fftw_execute(plan);
            //copia o buffer de saída da FFTW para a matriz de resultados:
            //for(unsigned int j=0;j<nlines;j++) returndata[j][i]
+=2.0*(pow((cabs(out[j])/StepSize),2.0)+I*carg(out[j]))/n_turns;
            for(unsigned int j=0;j<nlines;j++) returndata[j][i]+=
(2.0*pow(cabs(out[j]),2)/pow(S1,2)+I*carg(out[j]))/n_turns;
      }
   returndata[0][i]/=2.0; //o nível DC deve ser dividido por 2 pois ele já vem dobrado da FFTW
   }

//fim do cálculo

//desaloca o plano
fftw_destroy_plan(plan);
//desaloca os vetores de entrada e saída da FFT
fftw_free(in);
fftw_free(out);
} //fim do #pragma omp parallel default(none) private(i)

printf("Done.\r\n"); fflush(stdout);

//aqui é necessário eliminar a parte de cima do espectrograma, se o sinal foi reamostrado
if(this->original_samplingrate!=0)
  {
  unsigned int newlines = floor(((double)this->original_samplingrate/2.0)/this->spectral_freq_resolution)+1.0;
  this->spectral_freq_limit = (newlines-1)*this->spectral_freq_resolution;
  Matrix aux(newlines,this->n_columns);
  //printf("Correcting to %u lines\r\n",newlines);
  for(register unsigned int i=0;i<newlines;i++)
            for(register unsigned int j=0;j<this->n_columns;j++)
               {
                     aux[i][j]=returnmatrix[i][j];
               }
  returnmatrix = aux;
```

```
  }


printf("Extracting Square root of each element of STFT Matrix (already in polar Form)...");
for (register unsigned int row = 0; row < returnmatrix.GetNumRows(); row++)
                                for (register unsigned int column = 0;column < returnmatrix.GetNumColumns(); column+
+)
                                    {
                                          returnmatrix[row][column] = sqrt(creal(returnmatrix[row][column]))
+I*cimag(returnmatrix[row][column]);
                                    }
printf("Done.\r\n");


return returnmatrix;
}

//calcula STFT do sinal de entrada, retorna uma matriz de complexos
Matrix STFT_Computer::Compute_PWR_STFT(electric_signal input,unsigned int samples_per_turn)
{
//primeiro checa se a janela já foi configurada:
if(this->windowsize==0) { printf("Error: STFT Window not setup\r\n"); exit(EXIT_FAILURE); };
//pega o ponteiro do início dos dados do sinal de entrada para poder acessar mais rápido
double* sample = input.get_samples_pointer();
//e o número de linhas é n/2+1 pois a FFT é de um vetor real (simetria hermitiana)
this->n_lines = floor(this->windowsize/2)+1; //lembrar que uma linha é o valor DC(parte imaginária nula)
//pega o samplingarate do sinal de entrada
this->samplingrate = input.get_samplingrate();
//o limite em frequência é a frequência de Nyquist
this->spectral_freq_limit = (double)this->samplingrate/2.0;
//calcula a resolução temporal baseando-se no step da janela e na samplingrate
this->time_resolution = (double)this->stepsize/samplingrate;
//o número de colunas da STFT é igual ao número de steps que cabem por volta
unsigned long int n_turns = input.get_num_samples()/samples_per_turn;
this->n_columns = samples_per_turn/this->stepsize;
//o limite em tempo é o número de colunas x a resolução temporal
this->time_limit = this->n_columns*this->time_resolution;
//calcula a frequencia fundamental baseando-se na definição da DFT. Esta é a resolução em frequencia
this->spectral_freq_resolution = this->spectral_freq_limit/(this->n_lines-1);
this->ENBW = this->NENBW * this->spectral_freq_resolution;
//imprime alguns dados:
printf("STFT: Averaging %lu of %9f turns.\r\n",n_turns,((double)input.get_num_samples()/samples_per_turn));
printf("STFT: Time resolution: %u intervals of %f s. Frequency resolution: %u bands of %f Hz.\r\n",this-
>n_columns,this->time_resolution,this->n_lines-1,this->spectral_freq_resolution);

//aloca a matriz resultante, a própria Short Time Fourier Transform Averaged:
Matrix returnmatrix(this->n_lines,this->n_columns);
//transforma alguns dos parâmetros numéricos para variáveis locais a fim de deixar o OpenMP paralelizar
unsigned int nlines=this->n_lines;
unsigned int ncolumns=this->n_columns;
unsigned int wsize=this->windowsize;
double* windowp = this->window;
unsigned int StepSize = this->stepsize;
complex double** returndata = returnmatrix.GetData();
unsigned int i;
double S1 = this->S1;
fftw_cleanup();

#pragma omp parallel default(none) private(i)
shared(S1,n_turns,wsize,nlines,ncolumns,windowp,StepSize,returndata,sample,stdout,input)
{
//aloca os vetores de entrada e de saída para as FFTs:
//aloca um vetor de entrada de doubles e um vetor de saída de complexos para a FFT:
double* in = fftw_alloc_real(wsize);
fftw_complex* out = fftw_alloc_complex(nlines);
//cria um objeto que vai guardar os planos de uma FFT:
fftw_plan plan;

//faz o plano de uma FFT:
#pragma omp critical (make_plan)
{
printf("STFT: Planning FFTs for CPU %u...",omp_get_thread_num()); fflush(stdout);
plan = fftw_plan_dft_r2c_1d(wsize,in,out,FFTW_ESTIMATE);
printf("Done.\r\n"); fflush(stdout);
}


#pragma omp barrier

printf("STFT: CPU %u Computing Average PWR of Short Time Fourier Transform for signal
%s...\r\n",omp_get_thread_num(),input.get_name()); fflush(stdout);

#pragma omp for
for(i=0;i<ncolumns;i++) // para cada coluna da matriz resultado
   {
   for(register unsigned int k=0; k<n_turns;k++)
      {
            //copia um pedaço do sinal de entrada da k-ésima volta para o buffer de entrada da FFTW, já aplicando a
janela:
            for(register unsigned int j=0;j<wsize;j++) in[j]=windowp[j]*sample[j+((i+k*ncolumns)*StepSize)];
            //executa o plano previamente criado:
            fftw_execute(plan);
            //copia o buffer de saída da FFTW para a matriz de resultados:
            for(unsigned int j=0;j<nlines;j++) returndata[j][i]+= (2.0*pow(cabs(out[j]),2)/
(pow(S1,2))/*+I*carg(out[j])*/)/n_turns;
```

```
      }
   returndata[0][i]/=2.0; //o nível DC deve ser dividido por 2 pois ele já vem dobrado da FFTW
   }

//fim do cálculo
//printf("++++++++++++++++++++++++++++++++++++++NENBW: %f\r\n",this->NENBW);

//desaloca o plano
fftw_destroy_plan(plan);
//desaloca os vetores de entrada e saída da FFT
fftw_free(in);
fftw_free(out);
} //fim do #pragma omp parallel default(none) private(i)

printf("Done.\r\n"); fflush(stdout);

//aqui é necessário eliminar a parte de cima do espectrograma, se o sinal foi reamostrado
if(this->original_samplingrate!=0)
  {
  unsigned int newlines = floor(((double)this->original_samplingrate/2.0)/this->spectral_freq_resolution)+1.0;
  this->spectral_freq_limit = (newlines-1)*this->spectral_freq_resolution;
  Matrix aux(newlines,this->n_columns);
  //printf("Correcting to %u lines\r\n",newlines);
  for(register unsigned int i=0;i<newlines;i++)
           for(register unsigned int j=0;j<this->n_columns;j++)
              {
                    aux[i][j]=returnmatrix[i][j];
              }
  returnmatrix = aux;
  }


return returnmatrix;
}


electric_signal STFT_Computer::Get_Overlapped_Windows(void)
{
this->samplingrate = 96000;
electric_signal returnsignal = electric_signal(10*this->windowsize,this->samplingrate,"Overlapped Windows");

unsigned int i=0;
unsigned int j=0;

while(i<=(9*windowsize))
     {
     for(j=0;j<this->windowsize;j++) returnsignal[j+i]+=this->window[j];
     i+=this->stepsize;
     }
return returnsignal;
}

electric_signal STFT_Computer::Compute_iSTFT(const Matrix input)
{
electric_signal returnsignal = electric_signal();
printf("Not implemented yet. Tried to compute iSTFT of %u x %u Matrix.\r\n",input.GetNumRows(),input.GetNumRows());
return returnsignal;
}

void STFT_Computer::generate_window(int window_type,const int N)
{
unsigned int i;
unsigned int n; //número de amostras na janela
const double PI=3.141592653589793;

n = pow(2,N); //N é sempre um numero par
this->window = (double*)malloc(n*sizeof(double)); //aloca 2^N doubles para guardar a janela
if(this->window==NULL) { printf("Couldn't allocate memory for window. Aborting.\r\n"); exit(EXIT_FAILURE); };

/*perform the window function requsted by the user*/
switch(window_type)
           {
           case 1:
           for(i=0;i<n;i++)
           this->window[i]=1.0;
           break;

           case 2:
           for(i=0;i<=(n-1)/2;i++)
           this->window[i]=2.0*i/(n-1);
           for(i=(n-1)/2+1;i<=n-1;i++)
           this->window[i]=2.0-2.0*i/(n-1);
           break;

           case 3:
           for(i=0;i<n;i++)
           this->window[i]=(1.0-cos(2.0*PI*i/(n-1)))/2.0;
           break;

           case 4:
           for(i=0;i<n;i++)
           this->window[i]=0.54-0.46*cos(2*PI*i/(n-1));
           break;

           case 5:
```

```
            for(i=0;i<n;i++)
            this->window[i]=0.42-0.5*cos(2*PI*i/(n-1))+0.08*cos(4*PI*i/(n-1));
            break;

            case 6:
            for(i=0;i<n;i++)
            this->window[i]=1.0/exp(18*((n-1)/2.0-i)*((n-1)/2.0-i)/(n*n));
            break;

            default:
            printf("Invalid Window Type. Review your description file!\r\n"); exit(EXIT_FAILURE);
            break;
              }
//normaliza a janela, para que a soma total com overlap dê uma janela unitária
for(i=0;i<n;i++) this->window[i]/=pow(2,this->overlap_index-1);

this->Kw = (double*)malloc(sizeof(double)*(n/2+1));

//calcula S1 e S2
for(register unsigned int i=0;i<n;i++) this->S1+=this->window[i];
for(register unsigned int i=0;i<n;i++) this->S2+=pow(this->window[i],2.0);
this->NENBW = n*this->S2/pow(this->S1,2.0);

fftw_cleanup();
double* in = fftw_alloc_real(n);
fftw_complex* out = fftw_alloc_complex(n/2+1);
fftw_plan plan = fftw_plan_dft_r2c_1d(n,in,out,FFTW_ESTIMATE);
for(register unsigned int i=0;i<n;i++) in[i]=pow(this->window[i],2.0);
fftw_execute(plan);
for(register unsigned int i=0;i<n/2+1;i++) this->Kw[i] = cabs(out[i])/this->S2;
fftw_destroy_plan(plan);
fftw_free(in);
fftw_free(out);
fftw_cleanup();

}

void STFT_Computer::set_original_samplingrate(unsigned int original)
{
this->original_samplingrate = original;
}
```

Arquivo **CMS-iCMS.cc**:

```
/*
 * CMS-iCMS.cc
 *
 *  Created on: Sep 15, 2014
 *      Author: rafael
 */
#include <fftw3.h>
#include <omp.h>
#include <math.h>
#include "CMS-iCMS.h"


Matrix CMS_Computer::Compute_CMS(Matrix Spectrogram_In)
{
unsigned int in_columns = Spectrogram_In.GetNumColumns();
unsigned int in_lines = Spectrogram_In.GetNumRows();
unsigned int out_columns = floor(in_columns/2)+1; //lembrar que uma linha é o valor DC(parte imaginária nula);
//calcula alguns parâmetros baseados nas informações passadas
this->n_lines = in_lines;
this->spectral_freq_resolution = this->spectral_freq_limit/(this->n_lines-1);
this->n_columns = out_columns;
this->cyclic_freq_limit = (double)in_columns/(2*this->time_limit);
this->cyclic_freq_resolution = this->cyclic_freq_limit/(this->n_columns-1);
//aloca a matriz de resultados
Matrix result(this->n_lines,this->n_columns);
unsigned int i;
string name = this->signal_name;

complex double** dataout = result.GetData();
complex double** datain = Spectrogram_In.GetData();

printf("CMS: Max Cyclic   frequency: %7.3f Hz (%u bands of %7.3f Hz)\r\n",this->cyclic_freq_limit,(this->n_columns-
1),this->cyclic_freq_resolution);
printf("CMS: Max Spectral frequency: %7.3f Hz (%u bands of %7.3f Hz)\r\n",this->spectral_freq_limit,(this->n_lines-
1),this->spectral_freq_resolution);

fftw_cleanup();

#pragma omp parallel default(none) private(i) shared(out_columns,in_columns,in_lines,datain,dataout,name,stdout)
{
//aloca os vetores de entrada e de saída para as FFTs:
//aloca um vetor de entrada de doubles e um vetor de saída de complexos para a FFT:
double* in = fftw_alloc_real(in_columns);
fftw_complex* out = fftw_alloc_complex(this->n_columns);
//cria um objeto que vai guardar os planos de uma FFT:
fftw_plan plan;

//faz o plano de uma FFT:
#pragma omp critical (make_plan)
```

```
{
printf("CMS: Planning FFTs for CPU %u...",omp_get_thread_num()); fflush(stdout);
plan = fftw_plan_dft_r2c_1d(in_columns,in,out,FFTW_ESTIMATE);
printf("Done.\r\n");
}

#pragma omp barrier

printf("CMS: CPU %u Computing Cyclic Modulation Spectrum for signal %s...\r\n",omp_get_thread_num(),name.c_str());
fflush(stdout);

#pragma omp for
for(i=0;i<in_lines;i++) // para cada linha
   {
   //copia a linha i (só a parte real) para o buffer de entrada da FFTW:
   for(unsigned int register j=0;j<in_columns;j++) in[j]=creal(datain[i][j]);

   //executa o plano previamente criado:
   fftw_execute(plan);

   //copia o resultado para a saída:
   //multiplica todos por 2 porque a FFTW só computa as frequências positivas
   for(unsigned int register j=0;j<out_columns;j++) dataout[i][j]=2.0*(((double complex*)out)[j])/in_columns;
   //corrige para o nível DC, senão sai dobrado.
   dataout[i][0]/=2.0;
   }

//fim do cálculo

//desaloca o plano
fftw_destroy_plan(plan);
//desaloca os vetores de entrada e saída da FFT
fftw_free(in);
fftw_free(out);


} // fim do pragma omp parallel default(none) private(i) shared(in_columns,in_lines,Spectrogram_In,result)

printf("Done.\r\n"); fflush(stdout);

return result;
}

Matrix CMS_Computer::Compute_iCMS(Matrix CMS_In)
{
Matrix result=CMS_In;


return result;
}

double CMS_Computer::get_cyclic_freq_resolution(void) const
{
return this->cyclic_freq_resolution;
}

double CMS_Computer::get_spectral_freq_resolution(void) const
{
return this->spectral_freq_resolution;
}

double CMS_Computer::get_cyclic_freq_limit(void) const
{
return this->cyclic_freq_limit;
}

double CMS_Computer::get_spectral_freq_limit(void) const
{
return this->spectral_freq_limit;
}

unsigned int CMS_Computer::get_n_columns(void) const
{
return this->n_columns;
}

unsigned int CMS_Computer::get_n_lines(void) const
{
return this->n_lines;
}

CMS_Computer::CMS_Computer()
{
this->cyclic_freq_limit = 0.0;
this->cyclic_freq_resolution = 0.0;
this->experience_name = string("");
this->n_columns = 0;
this->n_lines = 0;
this->spectral_freq_limit = 0.0;
this->spectral_freq_resolution =0.0;
this->time_limit = 0.0;
}

string CMS_Computer::get_experience_name(void) const
{
```

```
return string(this->experience_name);
}

string CMS_Computer::get_signal_name(void) const
{
return string(this->signal_name);
}

void CMS_Computer::set_experience_name(const string name)
{
this->experience_name = string(name);
}

void CMS_Computer::set_signal_name(const string name)
{
this->signal_name = string(name);
}

void CMS_Computer::set_time_limit(double lim)
{
this->time_limit = lim;
}

void CMS_Computer::set_spectral_freq_limit(double lim)
{
this->spectral_freq_limit = lim;
}

CMS_Computer::~CMS_Computer()
{

}
```

Arquivo **CMC-iCMC.cc**:

```
/*
 * CMC-iCMC.cc
 *
 *  Created on: Sep 15, 2014
 *      Author: rafael
 */

#include <stdlib.h>
#include <stdio.h>
#include <complex.h>
#include <omp.h>
#include "CMC-iCMC.h"


Matrix CMC_Computer::Compute_CMC(Matrix CMS_In)
{
unsigned int nlines = CMS_In.GetNumRows();
unsigned int ncolumns = CMS_In.GetNumColumns();
Matrix result(nlines,ncolumns);
unsigned int i;
this->spectral_freq_resolution = this->spectral_freq_limit/(nlines-1); // 1 linha é o nível DC
//double resolution = this->spectral_freq_resolution;
this->cyclic_freq_resolution = this->cyclic_freq_limit/(ncolumns-1);

printf("CMC: Computing CMC..."); fflush(stdout);
#pragma omp parallel for private(i) shared(CMS_In,nlines,ncolumns,result)
for(i=0;i<ncolumns;i++)
   {
   if(i==0) for(register unsigned int j=0;j<nlines;j++) result[j][i]=0.0;
   else for(register unsigned int j=0;j<nlines;j++)
                if(CMS_In[j][0]!=0.0) result[j][i] = CMS_In[j][i]/CMS_In[j][0];
   }
printf("Done.\r\n"); fflush(stdout);

return result;
}

complex double* CMC_Computer::get_PSD(Matrix CMS_In) const
{
complex double* PSD = (complex double*)malloc(sizeof(complex double)*CMS_In.GetNumRows());
printf("PSD: Extracting PSD for signal %s...",this->signal_name.c_str()); fflush(stdout);
CMS_In.GetColumn(PSD,0);
printf("\r\nPSD: Converting to Polar Form...");  fflush(stdout);
for(register unsigned int i=0;i<CMS_In.GetNumRows();i++) PSD[i]=cabs(PSD[i])+I*carg(PSD[i]);
printf("Done.\r\n");  fflush(stdout);
return PSD;
}


Matrix CMC_Computer::Compute_iCMC(Matrix CMC_In)
{
printf("Inverse CMC not implemented yet. Aborting\r\n"); exit(EXIT_FAILURE);
return CMC_In;
}
```

```
string CMC_Computer::get_experience_name(void) const {
return string(this->experience_name);
}


string CMC_Computer::get_signal_name(void) const {
return string(this->signal_name);
}


double CMC_Computer::get_spectral_freq_resolution(void) const {
return this->spectral_freq_resolution;
}


double CMC_Computer::get_spectral_freq_limit(void) const {
return this->spectral_freq_limit;
}


double CMC_Computer::get_cyclic_freq_resolution(void) const {
return this->cyclic_freq_resolution;
}


double CMC_Computer::get_cyclic_freq_limit(void) const {
return this->cyclic_freq_limit;
}


unsigned int CMC_Computer::get_n_columns(void) const {
return this->n_columns;
}


unsigned int CMC_Computer::get_n_lines(void) const {
return this->n_lines;
}


CMC_Computer::CMC_Computer() {
this->cyclic_freq_limit = 0.0;
this->cyclic_freq_resolution = 0.0;
this->spectral_freq_limit = 0.0;
this->spectral_freq_resolution = 0.0;
this->n_columns = 0;
this->n_lines = 0;
this->signal_name = string("");
this->experience_name = string("");
}

void CMC_Computer::set_experience_name(const string name) {
this->experience_name = string(name);
}

void CMC_Computer::set_signal_name(const string name) {
this->signal_name = string(name);
}

void CMC_Computer::set_spectral_freq_limit(double lim) {
this->spectral_freq_limit = lim;
}

void CMC_Computer::set_cyclic_freq_limit(double lim) {
this->cyclic_freq_limit = lim;
}

CMC_Computer::~CMC_Computer()
{
}
```

Arquivo **grpGL.cc**:

```
#include"grpGL.h"
#include<mgl2/mgl.h>



/
**********************************************************************************************************************
*****/
/
**********************************************************************************************************************
*****/
/
**********************************************************************************************************************
*****/

GL2D::GL2D(unsigned int x,unsigned int y)
```

```
{
this->MainTitle = string("");  //começa com os textos todos vazios
this->SubTitle = string("");
this->normalizedXscaleTitle = string("");
this->XscaleTitle = string("");
this->YscaleTitle = string("");
this->XscaleStart = 0.0;   //e com a escala zerada
this->XscaleEnd = 0.0;
this->XscaleNormalizationFactor = 0.0;
this->YscaleStart = 0.0;
this->YscaleEnd = 0.0;
this->draw_inverted = false; //e desenha com o fundo branco por default
this->Xsize = x;
this->Ysize = y;
}

void GL2D::SetXrange(double x1,double x2)
{
this->XscaleStart = x1;
this->XscaleEnd = x2;
}

void GL2D::SetMainTitle(string s)
{
this->MainTitle = string(s);
}

void GL2D::SetSubTitle(string s)
{
this->SubTitle = string(s);
}

void GL2D::set_inverted(bool inverted)
{
this->draw_inverted = inverted;
}

void GL2D::showGraph(void)
{

}

void GL2D::SetXscaleTitle(string s)
{
this->XscaleTitle = string(s);
}

void GL2D::SetYscaleTitle(string s)
{
this->YscaleTitle = string(s);
}

void GL2D::SetXNormalizationFactor(double x)
{
this->XscaleNormalizationFactor = x;
}

void GL2D::SetNormalizedXscaleTitle(string s)
{
this->normalizedXscaleTitle = string(s);
}

void GL2D::PlotSignal(double* data,unsigned long int num_samples,string c)
{
mglGraph graph;
mglData y;
register unsigned long int i;

graph.SetSize(this->Xsize,this->Ysize);
graph.SetFontSizePT(8); //configura o tamanho da fonte
graph.SetTickLen(-0.01,1);
graph.SetTicks('x',-10.0,-10,0.0);
graph.SetRange('x',this->XscaleStart,this->XscaleEnd);
graph.Axis("xAKDTVISO");


this->YscaleStart=data[0];
this->YscaleEnd=data[0];
y.Create(num_samples);
for(i=0;i<num_samples;i++)
   {
   y.a[i]=data[i]; //carrega o vetor com os dados a serem plotados
   if(this->YscaleStart>data[i]) this->YscaleStart=data[i];
   if(this->YscaleEnd<data[i]) this->YscaleEnd=data[i];
   }
graph.SetRange('y',this->YscaleStart,this->YscaleEnd);
graph.Axis("yAKDTVISO");
graph.Label('x',this->XscaleTitle.c_str(),1.0);
graph.Label('y',this->YscaleTitle.c_str(),0);

graph.Box("",false);
graph.Plot(y,c.c_str()); //definitivamente plota os dados

graph.SetOrigin(0.0,0.0);
```

```
if(this->XscaleNormalizationFactor!=0.0)
  {
  graph.SetRanges(this->XscaleStart/this->XscaleNormalizationFactor,this-
>XscaleEnd/XscaleNormalizationFactor,0.0,1.0);
  graph.SetOrigin(0.0,-0.20);
  graph.SetTickLen(0.02,1);
  graph.SetTicks('x',5);
  graph.Axis("xAKDTVISO","r");
  graph.SetOrigin(this->XscaleEnd/XscaleNormalizationFactor,0.0);
  graph.Label('x',string("#r{"+this->normalizedXscaleTitle+"}").c_str(),-1.0);
  }

graph.SetOrigin(0.0,0.0);


graph.Title(string(this->MainTitle+"\n@{"+this->SubTitle+"}").c_str());
graph.WriteTGA(string(this->Filename+".tga").c_str());
}

void GL2D::SaveToFile(string filename)
{
this->Filename = string(filename);
}

GL2D::~GL2D()
{

}

/
**********************************************************************************************************************
*****/
/
**********************************************************************************************************************
*****/
/
**********************************************************************************************************************
*****/

GL3D::GL3D(unsigned int x,unsigned int y)
{
this->MainTitle = string("");  //começa com os textos todos vazios
this->SubTitle = string("");
this->normalizedXscaleTitle = string("");
this->XscaleTitle = string("");
this->YscaleTitle = string("");
this->ZscaleTitle = string("");
this->Filename = string("");
this->XscaleStart = 0.0;   //e com a escala zerada
this->XscaleEnd = 0.0;
this->XscaleNormalizationFactor = 0.0;
this->YscaleStart = 0.0;
this->YscaleEnd = 0.0;
this->ZscaleStart = 0.0;
this->ZscaleEnd = 0.0;
this->draw_inverted = false; //e desenha com o fundo branco por default
this->Xsize=x;
this->Ysize=y;
}

void GL3D::SetXrange(double x1,double x2)
{
this->XscaleStart = x1;
this->XscaleEnd = x2;
}

void GL3D::SetYrange(double y1,double y2)
{
this->YscaleStart = y1;
this->YscaleEnd = y2;
}

void GL3D::SetMainTitle(string s)
{
this->MainTitle = string(s);
}

void GL3D::SetSubTitle(string s)
{
this->SubTitle = string(s);
}

void GL3D::set_inverted(bool inverted)
{
this->draw_inverted = inverted;
}

void GL3D::showGraph(void)
{

}

void GL3D::SetXscaleTitle(string s)
{
this->XscaleTitle=string(s);
```

```
}

void GL3D::SetYscaleTitle(string s)
{
this->YscaleTitle=string(s);
}

void GL3D::SetZscaleTitle(string s)
{
this->ZscaleTitle=string(s);
}

void GL3D::SetXNormalizationFactor(double x)
{
this->XscaleNormalizationFactor = x;
}

void GL3D::SetNormalizedXscaleTitle(string s)
{
this->normalizedXscaleTitle = string(s);
}

void GL3D::PlotSignal(double** data,unsigned long int num_columns,unsigned long int num_lines)
{
mglGraph graph;
graph.SetSize(this->Xsize,this->Ysize);
graph.Alpha(false);
graph.Light(false);
graph.SetFontSizePT(5);

mglData y;
register unsigned long int i,j;

y.Create(num_columns,num_lines);
this->ZscaleStart=data[0][0];
this->ZscaleEnd=data[0][0];
//olha o OPENMP
for(i=0;i<num_columns;i++) for(j=0;j<num_lines;j++)
                                {
                                      y.a[i+num_columns*j]=data[j][i]; //carrega o vetor com os dados a serem plotados
                                      if(this->ZscaleEnd<data[j][i]) this->ZscaleEnd=data[j][i];
                                      if(this->ZscaleStart>data[j][i]) this->ZscaleStart=data[j][i];
                                      }


graph.SetTicks('x',10.0,9,0.0);
graph.SetRanges(this->XscaleStart,this->XscaleEnd,this->YscaleStart,this->YscaleEnd,this->ZscaleStart,this-
>ZscaleEnd);



graph.Title(string(this->MainTitle+"\n@{"+this->SubTitle+"}").c_str());
graph.Light(true);
graph.Rotate(75,15); // é (X,Z,Y=0) !!!
graph.AddLight(1,mglPoint(100,100,100),'w',0.25);
graph.Light(1,true);
graph.Axis("xAKDTVISO");
graph.Axis("yz^AKDTVISO");
graph.Label('x',this->XscaleTitle.c_str(),0);
graph.Label('y',this->YscaleTitle.c_str(),0);
graph.Label('z',this->ZscaleTitle.c_str(),0);
graph.Box();
if(this->XscaleNormalizationFactor!=0.0)
  {
  graph.SetOrigin(0.0,0.0,this->ZscaleEnd);
  graph.SetTicks('x',1.0,0,0.0);
  graph.SetRanges(this->XscaleStart/this->XscaleNormalizationFactor,this->XscaleEnd/XscaleNormalizationFactor,this-
>YscaleStart,this->YscaleEnd,this->ZscaleStart,this->ZscaleEnd);
  graph.Axis("XAKDTVISO","r");
  graph.Label('x',string("#r{"+this->normalizedXscaleTitle+"}").c_str(),0);
  }
graph.Surf(y); //definitivamente plota os dados
graph.WriteTGA(string(this->Filename+".tga").c_str());
graph.ShowImage("display",true);
}

void GL3D::SaveToFile(string filename)
{
this->Filename = string(filename);
}

GL3D::~GL3D()
{

}

/
*****************************************************************************************************************
*****/
/
*****************************************************************************************************************
*****/
/
*****************************************************************************************************************
*****/
```

```
GLImg::GLImg(unsigned int x,unsigned int y)
{
this->MainTitle = string("");  //começa com os textos todos vazios
this->SubTitle = string("");
this->normalizedXscaleTitle = string("");
this->XscaleTitle = string("");
this->YscaleTitle = string("");
this->ZscaleTitle = string("");
this->Filename = string("");
this->XscaleStart = 0.0;   //e com a escala zerada
this->XscaleEnd = 0.0;
this->XscaleNormalizationFactor = 0.0;
this->YscaleStart = 0.0;
this->YscaleEnd = 0.0;
this->ZscaleStart = 0.0;
this->ZscaleEnd = 0.0;
this->draw_inverted = false; //e desenha com o fundo branco por default
this->Xsize=x;
this->Ysize=y;
this->XScaleTicks = 0.0;
this->XScaleInterTicks = 0;
this->YScaleTicks = 0.0;
this->YScaleInterTicks = 0;
}

void GLImg::SetXscaleticks(double ticks,signed int interticks)
{
this->XScaleTicks = ticks;
this->XScaleInterTicks = interticks;
}

void GLImg::SetYscaleticks(double ticks,signed int interticks)
{
this->YScaleTicks = ticks;
this->YScaleInterTicks = interticks;
}

void GLImg::SetXrange(double x1,double x2)
{
this->XscaleStart = x1;
this->XscaleEnd = x2;
}

void GLImg::SetYrange(double y1,double y2)
{
this->YscaleStart = y1;
this->YscaleEnd = y2;
}

void GLImg::SetMainTitle(string s)
{
this->MainTitle = string(s);
}

void GLImg::SetSubTitle(string s)
{
this->SubTitle = string(s);
}

void GLImg::set_inverted(bool inverted)
{
this->draw_inverted = inverted;
}

void GLImg::showGraph(void)
{

}

void GLImg::SetXscaleTitle(string s)
{
this->XscaleTitle=string(s);
}

void GLImg::SetYscaleTitle(string s)
{
this->YscaleTitle=string(s);
}

void GLImg::SetZscaleTitle(string s)
{
this->ZscaleTitle=string(s);
}

void GLImg::SetXNormalizationFactor(double x)
{
this->XscaleNormalizationFactor = x;
}

void GLImg::SetNormalizedXscaleTitle(string s)
{
this->normalizedXscaleTitle = string(s);
}
```

```
void GLImg::PlotSignal(double** data,unsigned long int num_columns,unsigned long int num_lines)
{
double zmedium=0.0;
//double y_resolution = (this->YscaleEnd-this->YscaleStart)/(num_lines-1);

mglGraph graph; //cria um objeto gráfico
mglData  z; //cria um objeto data
graph.SetSize(this->Xsize,this->Ysize); //configura o tamanho do gráfico
graph.SetFontSizePT(5); //configura o tamanho da fonte
z.Create(num_columns,num_lines); //aloca espaço para os dados
        //cria dois doubles para guardar o maior valor e o menor valor de todos na matriz
this->ZscaleStart = data[0][0]; //y_resolution;
this->ZscaleEnd = data[0][0]; //y_resolution;
//carrega a estrutura de dados do MathGL
for(register unsigned int column=0;column<num_columns;column++)
   for(register unsigned int row=0;row<num_lines;row++)
      {
           double tmp = data[row][column]; //y_resolution;
      z.a[column+num_columns*row] = tmp;
      if(this->ZscaleStart>tmp) this->ZscaleStart=tmp; //atualiza o máximo e o mínimo
      if(this->ZscaleEnd<tmp) this->ZscaleEnd=tmp;
      zmedium+=tmp;
      }
   zmedium/=num_columns;
   zmedium/=num_lines;

         if(this->XScaleTicks==0.0) graph.SetTicks('x',-10.0,-10,0.0);
               else graph.SetTicks('x',this->XScaleTicks,this->XScaleInterTicks,this->XscaleStart);

               if(this->YScaleTicks==0.0) graph.SetTicks('y',-10.0,-10,0.0);
               else
                  {
                  double order = pow(10.0,floor(log10(this->YScaleTicks)/3)*3);
                  this->YScaleTicks/=order;
                  this->YscaleStart/=order;
                  this->YscaleEnd/=order;
                  graph.SetTicks('y',this->YScaleTicks,this->YScaleInterTicks,this->YscaleStart);
                  }

               graph.SetRanges(this->XscaleStart,this->XscaleEnd,this->YscaleStart,this->YscaleEnd,this-
>ZscaleStart,this->ZscaleEnd);
               graph.Axis("xyAKDTVISO");
               graph.Colorbar(">",0.95,0.0);
               graph.Box("",false);
               graph.Label('x',this->XscaleTitle.c_str());
               if(this->XscaleNormalizationFactor!=0.0) graph.Label('x',string("#r{"+this-
>normalizedXscaleTitle+"}").c_str(),-1.0);
               graph.Label('y',this->YscaleTitle.c_str());

               char buffer[50];
               if(this->draw_inverted)
                 {
                 sprintf(buffer,"w{h,%5.3f}k",(zmedium-this->ZscaleStart)/(this->ZscaleEnd-this->ZscaleStart));
                 graph.Dens(z,buffer);
                 sprintf(buffer,">w{h,%5.3f}k",(zmedium-this->ZscaleStart)/(this->ZscaleEnd-this->ZscaleStart));
                 graph.Colorbar(buffer,0.95,0.0);
                 }
               else
                 {
                 sprintf(buffer,"Bbc{y,%3.1f}rR",(zmedium-this->ZscaleStart)/(this->ZscaleEnd-this->ZscaleStart));
                 graph.Dens(z,buffer);
                 sprintf(buffer,">Bbc{y,%3.1f}rR",(zmedium-this->ZscaleStart)/(this->ZscaleEnd-this->ZscaleStart));
                 graph.Colorbar(buffer,0.95,0.0);
                 }

               graph.SetOrigin(this->XscaleEnd,0.0);
               graph.Label('y',this->YscaleTitle.c_str(),0.0);
               graph.SetOrigin(0.0,0.0);

               if(this->XscaleNormalizationFactor!=0.0)
               {
               graph.SetRanges(this->XscaleStart/this->XscaleNormalizationFactor,this-
>XscaleEnd/XscaleNormalizationFactor,0.0,1.0);
               graph.SetOrigin(0.0,-0.15);
               graph.SetTicks('x',-10.0);
               graph.Axis("xAKDTVISO","r");
               graph.SetOrigin(this->XscaleEnd/XscaleNormalizationFactor,0.0);
               }

         graph.SetOrigin(0.0,0.0);
               graph.Title(string(this->MainTitle+"\r\n@{"+this->SubTitle+"}").c_str(),"",3);

               graph.WriteTGA(string(this->Filename+".tga").c_str());

}

void GLImg::PlotRealPartOfComplexSignal(double** data,unsigned long int num_columns,unsigned long int num_lines)
{
         double zmedium=0.0;

         //double y_resolution = (this->YscaleEnd-this->YscaleStart)/(num_lines-1);

         mglGraph graph; //cria um objeto gráfico
         mglData  z; //cria um objeto data
```

```
         graph.SetSize(this->Xsize,this->Ysize); //configura o tamanho do gráfico
         graph.SetFontSizePT(5); //configura o tamanho da fonte
         z.Create(num_columns,num_lines); //aloca espaço para os dados

         //cria dois doubles para guardar o maior valor e o menor valor de todos na matriz
         this->ZscaleStart = data[0][0]; //y_resolution;
         this->ZscaleEnd = data[0][0]; //y_resolution;
         unsigned int MaxX=0;
         unsigned int MaxY=0;

         //carrega a estrutura de dados do MathGL
         for(register unsigned int column=0;column<num_columns;column++)
            for(register unsigned int row=0;row<num_lines;row++)
               {
                     double tmp = data[row][column*2];// /y_resolution;
               z.a[column+num_columns*row] = tmp;
               if(this->ZscaleStart>tmp) this->ZscaleStart=tmp; //atualiza o máximo e o mínimo
               if(this->ZscaleEnd<tmp) { this->ZscaleEnd=tmp; MaxX = column; MaxY=row; }
               zmedium+=tmp;
               }
     zmedium/=num_columns;
     zmedium/=num_lines;

     printf("  Zmax = %f(%u,%u)  Zmin = %f Zmed = %f  ",this->ZscaleEnd,MaxX,MaxY,this->ZscaleStart,zmedium);

          //configura o eixo X
     if(this->XScaleTicks==0.0) graph.SetTicks('x',-10.0,-10,0.0);
                                else graph.SetTicks('x',this->XScaleTicks,this->XScaleInterTicks,this->XscaleStart);

          //configura o eixo Y
     if(this->YScaleTicks==0.0) graph.SetTicks('y',-10.0,-10,0.0);
                                else
                                      {
                                      double order = pow(10.0,floor(log10(this->YScaleTicks)/3)*3);
                                      this->YScaleTicks/=order;
                                      this->YscaleStart/=order;
                                      this->YscaleEnd/=order;
                                   graph.SetTicks('y',this->YScaleTicks,this->YScaleInterTicks,this->YscaleStart);
                                   }

           graph.SetRanges(this->XscaleStart,this->XscaleEnd,this->YscaleStart,this->YscaleEnd,this-
>ZscaleStart,this->ZscaleEnd);
           graph.Axis("xyAKDTVISO");
      graph.Box("",false);
      graph.Label('x',this->XscaleTitle.c_str());
      if(this->XscaleNormalizationFactor!=0.0) graph.Label('x',string("#r{"+this-
>normalizedXscaleTitle+"}").c_str(),-1.0);
      graph.Label('y',this->YscaleTitle.c_str());

      char buffer[50];
      if(this->draw_inverted)
               {
               sprintf(buffer,"w{h,%5.3f}k",(zmedium-this->ZscaleStart)/(this->ZscaleEnd-this->ZscaleStart));
               graph.Dens(z,buffer);
               sprintf(buffer,">w{h,%5.3f}k",(zmedium-this->ZscaleStart)/(this->ZscaleEnd-this->ZscaleStart));
               graph.Colorbar(buffer,0.95,0.0);
               }
      else
               {
          sprintf(buffer,"Bbc{y,%3.1f}rR",(zmedium-this->ZscaleStart)/(this->ZscaleEnd-this->ZscaleStart));
          graph.Dens(z,buffer);
               sprintf(buffer,">Bbc{y,%3.1f}rR",(zmedium-this->ZscaleStart)/(this->ZscaleEnd-this->ZscaleStart));
               graph.Colorbar(buffer,0.95,0.0);
               }


     graph.SetOrigin(this->XscaleEnd,0.0);
     graph.Label('y',this->ZscaleTitle.c_str(),0.0);
     graph.SetOrigin(0.0,0.0);

     if(this->XscaleNormalizationFactor!=0.0)
               {
               graph.SetRanges(this->XscaleStart/this->XscaleNormalizationFactor,this-
>XscaleEnd/XscaleNormalizationFactor,0.0,1.0);
               graph.SetOrigin(0.0,-0.15);
               graph.SetTicks('x',-10.0);
               graph.Axis("xAKDTVISO","r");
               graph.SetOrigin(this->XscaleEnd/XscaleNormalizationFactor,0.0);
               }

     graph.SetOrigin(0.0,0.0);
          graph.Title(string("\n"+this->MainTitle+"\r\n@{"+this->SubTitle+"}").c_str(),"",3);

          graph.WriteTGA(string(this->Filename+".tga").c_str());

}

void GLImg::PlotModulusOfComplexSignal(double** data,unsigned long int num_columns,unsigned long int num_lines)
{
         double zmedium=0.0;

         //double y_resolution = (this->YscaleEnd-this->YscaleStart)/(num_lines-1);

         mglGraph graph; //cria um objeto gráfico
         mglData  z; //cria um objeto data
```

```
         graph.SetSize(this->Xsize,this->Ysize); //configura o tamanho do gráfico
         graph.SetFontSizePT(8); //configura o tamanho da fonte
         z.Create(num_columns,num_lines); //aloca espaço para os dados
                 //cria dois doubles para guardar o maior valor e o menor valor de todos na matriz
         this->ZscaleStart = sqrt(pow(data[0][0],2.0)+pow(data[0][1],2.0)); //y_resolution;
         this->ZscaleEnd = this->ZscaleStart;
         unsigned int MaxX=0;
         unsigned int MaxY=0;

         //carrega a estrutura de dados do MathGL
         for(register unsigned int column=0;column<num_columns;column++)
            for(register unsigned int row=0;row<num_lines;row++)
               {
                     double tmp = sqrt(pow((data[row][column*2]),2.0)+pow((data[row][column*2+1]),2.0));
//y_resolution;
               z.a[column+num_columns*row] = tmp;
               if(this->ZscaleStart>tmp) this->ZscaleStart=tmp; //atualiza o máximo e o mínimo
               if(this->ZscaleEnd<tmp) { this->ZscaleEnd=tmp; MaxX=column; MaxY=row; }
               zmedium+=tmp;
               }
     zmedium/=num_columns;
     zmedium/=num_lines;

     printf("  Zmax = %f(%u,%u)  Zmin = %f Zmed = %f  ",this->ZscaleEnd,MaxX,MaxY,this->ZscaleStart,zmedium);

             graph.SetTickLen(-0.01,1);
             graph.SetTicks('c',-5);

                  if(this->XScaleTicks==0.0) graph.SetTicks('x',-10.0,-10,0.0);
                  else graph.SetTicks('x',this->XScaleTicks,this->XScaleInterTicks,this->XscaleStart);

                  if(this->YScaleTicks==0.0) graph.SetTicks('y',-10.0,-10,0.0);
                  else
                   {
                   double order = pow(10.0,floor(log10(this->YScaleTicks)/3)*3);
                   this->YScaleTicks/=order;
                   this->YscaleStart/=order;
                   this->YscaleEnd/=order;
                   graph.SetTicks('y',this->YScaleTicks,this->YScaleInterTicks,this->YscaleStart);
                   }

                         graph.SetRanges(this->XscaleStart,this->XscaleEnd,this->YscaleStart,this->YscaleEnd,this-
>ZscaleStart,this->ZscaleEnd);
                         graph.Axis("xyAKDTVISO");

                         graph.Box("",false);
                         graph.Label('x',this->XscaleTitle.c_str());
                         if(this->XscaleNormalizationFactor!=0.0) graph.Label('x',string("#r{"+this-
>normalizedXscaleTitle+"}").c_str(),-1.0);
                         graph.Label('y',this->YscaleTitle.c_str());

                         char buffer[50];
                         if(this->draw_inverted)
                         {
                         sprintf(buffer,"w{h,%5.3f}k",(zmedium-this->ZscaleStart)/(this->ZscaleEnd-this-
>ZscaleStart));
                         graph.Dens(z,buffer);
                         sprintf(buffer,">w{h,%5.3f}k",(zmedium-this->ZscaleStart)/(this->ZscaleEnd-this-
>ZscaleStart));
               graph.Colorbar(buffer,0.95,0.0);
                         }
                         else
                         {
                      sprintf(buffer,"Bbc{y,%3.1f}rR",(zmedium-this->ZscaleStart)/(this->ZscaleEnd-this-
>ZscaleStart));
                    graph.Dens(z,buffer);
                         sprintf(buffer,">Bbc{y,%3.1f}rR",(zmedium-this->ZscaleStart)/(this->ZscaleEnd-this-
>ZscaleStart));
               graph.Colorbar(buffer,0.95,0.0);
                         }


                         graph.SetOrigin(1.04*this->XscaleEnd,0.0);
                         graph.Label('y',this->ZscaleTitle.c_str(),0.0);
                         graph.SetOrigin(0.0,0.0);

                         if(this->XscaleNormalizationFactor!=0.0)
                         {
                         graph.SetRanges(this->XscaleStart/this->XscaleNormalizationFactor,this-
>XscaleEnd/XscaleNormalizationFactor,0.0,1.0);
                         graph.SetOrigin(0.0,-0.20);
                         graph.SetTickLen(0.02,1);
                         graph.SetTicks('x',5);
                         graph.Axis("xAKDTVISO","r");
                         graph.SetOrigin(this->XscaleEnd/XscaleNormalizationFactor,0.0);
                         }

                   graph.SetOrigin(0.0,0.0);
                         graph.Title(string(this->MainTitle+"\r\n"+this->SubTitle).c_str(),"",3);

                         graph.WriteTGA(string(this->Filename+".tga").c_str());


}
```

```
void GLImg::SaveToFile(string filename)
{
this->Filename = string(filename);
}

GLImg::~GLImg()
{

}

/
*****************************************************************************************************************
*****/
/
*****************************************************************************************************************
*****/
/
*****************************************************************************************************************
*****/

GLPolar::GLPolar(unsigned int x,unsigned int y) //construtor que já dá as dimensões do gráfico
{
this->MainTitle = string("");  //começa com os textos todos vazios
this->SubTitle = string("");
this->AngularScaleTitle = string("");
this->RadialScaleTitle = string("");
this->normalizedAngularScaleTitle = string("");
this->Filename = string("");
this->AngularScaleEnd = 0.0;
this->RadialScaleEnd = 0.0;
this->AngularScaleNormalizationFactor = 0.0;
this->draw_inverted = false; //e desenha com o fundo branco por default
this->Xsize = x;
this->Ysize = y;
}

void GLPolar::SetMainTitle(string s)
{
this->MainTitle = string(s);
}

void GLPolar::SetSubTitle(string s)
{
this->SubTitle = string(s);
}

void GLPolar::SetAngularScaleTitle(string s)
{
this->AngularScaleTitle = string(s);
}

void GLPolar::SetRadialScaleTitle(string s)
{
this->RadialScaleTitle = string(s);
}

void GLPolar::SetAngularScaleNormalizationFactor(double f)
{
this->AngularScaleNormalizationFactor = f;
}

void GLPolar::SetNormalizedAngularScaleTitle(string s)
{
this->normalizedAngularScaleTitle = string(s);
}

void GLPolar::PlotSignal(double* data,unsigned long int num_samples,string c)
{

mglGraph graph;
mglData y;
register unsigned long int i;

graph.SetSize(this->Xsize,this->Ysize);
//graph.SetCoor(mglPolar);
graph.SetFunc("y*cos(x)", "y*sin(x)");
graph.SetFontSizePT(5);
graph.Title(string(this->MainTitle+"\n@{"+this->SubTitle+"}").c_str());

this->RadialScaleEnd=data[0];
y.Create(num_samples);
for(i=0;i<num_samples;i++)
   {
   y.a[i]=data[i]; //carrega o vetor com os dados a serem plotados
   if(this->RadialScaleEnd<data[i]) this->RadialScaleEnd=data[i];
   }

graph.SetFontSizePT(8);
graph.SetRange('x',0.0,2*M_PI);
//graph.SetTicks('x',M_PI,3,0.0,"\\pi");
double val[]={0,M_PI/2,M_PI,3*M_PI/2};
graph.SetTicksVal('x', mglData(4,val), "\\pi\n3\\pi/2\n\\0\n\\pi/2");
graph.Axis("x");
graph.Grid("x!","K");
graph.SetRange('y',0.0,this->RadialScaleEnd);
```

```
graph.Axis("yAKDTVISO");
graph.Grid("y!","K");

graph.SetFontSizePT(5);
//graph.Label('x',this->AngularScaleTitle.c_str(),0);
graph.SetFontSizePT(5);
//graph.Label('y',this->RadialScaleTitle.c_str(),0);

if(this->AngularScaleNormalizationFactor!=0.0)
  {
  graph.SetFontSizePT(3);
  graph.SetTicks('x',1.0,0,0.0);
  graph.SetRanges(0.0,this->AngularScaleEnd/this->AngularScaleNormalizationFactor,0.0,this->RadialScaleEnd);
  graph.Axis("XAKDTVISO","r");
  graph.SetFontSizePT(5);
  graph.Label('x',string("#r{"+this->normalizedAngularScaleTitle+"}").c_str(),0);
  }

graph.Box();
graph.Plot(y,c.c_str()); //definitivamente plota os dados
graph.WriteTGA(string(this->Filename+".tga").c_str());
graph.ShowImage("display",true);

}

//void PlotSignal(signal* S,ColorRGB c)
void GLPolar::set_inverted(bool inverted)
{
this->draw_inverted = inverted;
}

void GLPolar::SaveToFile(string filename)
{
this->Filename = string(filename);
}

void GLPolar::showGraph(void)
{

}

GLPolar::~GLPolar()
{

}
```

Arquivo **matrices.cc**:

```
#include "matrices.h"
#include <stdlib.h>
#include <stdio.h>


/* First constructor: Uninitialized matrix. */
Matrix::Matrix()
{
//printf("Matrix Constructor!\r\n");
/* Private variables initialization. */
this->Ncolumns = 0;
this->Nrows = 0;
this->data = NULL;
}

Matrix::Matrix(unsigned int numberOfRows,unsigned int numberOfColumns)
{
//printf("Matrix Overloaded Constructor!\r\n");
this->Ncolumns = numberOfColumns;
this->Nrows = numberOfRows;
this->data = (double complex**)malloc(sizeof(double complex*)*this->Nrows);
for(register unsigned int i=0;i<this->Nrows;i++) this->data[i]=(double complex*)malloc(sizeof(double complex)*this-
>Ncolumns);
for(register unsigned int i=0;i<this->Nrows;i++) for(register unsigned int j=0;j<this->Ncolumns;j++) this->data[i]
[j]=0.0;
}

// Destrutor

Matrix::~Matrix()
{
//printf("Matrix Destructor!\r\n");
if(this->data)
  {
  for(register unsigned int i=0;i<this->Nrows;i++) free(this->data[i]);
  free(data);
  this->Ncolumns = 0;
  this->Nrows = 0;
  }
}

//Copy Constructor
```

```
Matrix::Matrix(const Matrix& source)
{
//printf("Matrix Copy Constructor!\r\n");
this->Ncolumns = source.Ncolumns;
this->Nrows = source.Nrows;
//if(this->data!=NULL) free(this->data);
this->data = (double complex**)malloc(sizeof(double complex*)*this->Nrows);
for(register unsigned int i=0;i<this->Nrows;i++) this->data[i]=(double complex*)malloc(sizeof(double complex)*this-
>Ncolumns);
for(register unsigned int i=0;i<this->Nrows;i++) for(register unsigned int j=0;j<this->Ncolumns;j++) data[i]
[j]=source.data[i][j];
}

//Copy assignment constructor
Matrix Matrix::operator=(Matrix source)
{
//printf("Matrix Assignment Constructor!\r\n");
this->Ncolumns = source.Ncolumns;
this->Nrows = source.Nrows;
if(this->data) free(this->data);
this->data = (double complex**)malloc(sizeof(double complex*)*this->Nrows);
for(register unsigned int i=0;i<this->Nrows;i++) this->data[i]=(double complex*)malloc(sizeof(double complex)*this-
>Ncolumns);
for(register unsigned int i=0;i<this->Nrows;i++) for(register unsigned int j=0;j<this->Ncolumns;j++) data[i]
[j]=source.data[i][j];
return *this;
}

/* Data insertion and removal methods. */
void Matrix::SetRow(const double* rowdata,unsigned int rownum)
{
if(rownum<this->Nrows)
  {
  for(register unsigned int j=0;j<this->Ncolumns;j++) data[rownum][j]=rowdata[j];
  }
else
   {
   printf("Matrix.SetRow: Subscript out of range\r\n");
   }
}

void Matrix::SetRow(const double complex* rowdata,unsigned int rownum)
{
if(rownum<this->Nrows)
  {
  for(register unsigned int j=0;j<this->Ncolumns;j++) data[rownum][j]=rowdata[j];
  }
else
   {
   printf("Matrix.SetRow: Subscript out of range\r\n");
   }
}

void Matrix::SetColumn(const double* columndata,unsigned int columnnum)
{
if(columnnum<this->Ncolumns)
  {
  for(register unsigned int i=0;i<this->Nrows;i++) data[i][columnnum]=columndata[i];
  }
else
   {
   printf("Matrix.SetColumn: Subscript out of range\r\n");
   }
}

void Matrix::SetColumn(const double complex* columndata,unsigned int columnnum)
{
if(columnnum<this->Ncolumns)
  {
  for(register unsigned int i=0;i<this->Nrows;i++) data[i][columnnum]=columndata[i];
  }
else
   {
   printf("Matrix.SetColumn: Subscript out of range\r\n");
   }
}

void Matrix::GetRealRow(double* rowdata,unsigned int rownum) const
{
if(rownum<this->Nrows)
  {
  for(register unsigned int j=0;j<this->Ncolumns;j++) rowdata[j]=creal(data[rownum][j]);
  }
else
   {
   printf("Matrix.GetRealRow: Subscript out of range\r\n");
   }
}

void Matrix::GetImagRow(double* rowdata,unsigned int rownum) const
{
if(rownum<this->Nrows)
  {
  for(register unsigned int j=0;j<this->Ncolumns;j++) rowdata[j]=cimag(data[rownum][j]);
  }
```

```
else
    {
    printf("Matrix.GetImagRow: Subscript out of range\r\n");
    }
}

void Matrix::GetRow(double complex* rowdata,unsigned int rownum) const
{
if(rownum<this->Nrows)
  {
  for(register unsigned int j=0;j<this->Ncolumns;j++) rowdata[j]=data[rownum][j];
  }
else
    {
    printf("Matrix.GetRow: Subscript out of range\r\n");
    }
}

void Matrix::GetRealColumn(double* columndata,unsigned int columnnum) const
{
if(columnnum<this->Ncolumns)
  {
  for(register unsigned int i=0;i<this->Ncolumns;i++) columndata[i]=creal(data[i][columnnum]);
  }
else
    {
    printf("Matrix.GetRealColumn: Subscript out of range\r\n");
    }
}

void Matrix::GetImagColumn(double* columndata,unsigned int columnnum) const
{
if(columnnum<this->Ncolumns)
  {
  for(register unsigned int i=0;i<this->Ncolumns;i++) columndata[i]=cimag(data[i][columnnum]);
  }
else
    {
    printf("Matrix.GetImagColumn: Subscript out of range\r\n");
    }
}

void Matrix::GetColumn(double complex* columndata,unsigned int columnnum) const
{
if(columnnum<this->Ncolumns)
  {
  for(register unsigned int i=0;i<this->Ncolumns;i++) columndata[i]=data[i][columnnum];
  }
else
    {
    printf("Matrix.GetColumn: Subscript out of range\r\n");
    }
}


/* Interface de dados */
double complex** Matrix::GetData() const
{
return this->data;
}

//double** Matrix::GetRealData() const { }

//double** Matrix::GetImagData() const { }

double complex Matrix::GetElement(unsigned int row, unsigned int column) const
{
if((row<this->Nrows)&&(column<this->Ncolumns)) return data[row][column];
else
    {
    printf("Matrix.GetElement: Subscript out of range\r\n");
    return 0.0;
    }
}

void Matrix::SetElement(unsigned int row, unsigned int column,double complex x)
{
if((row<this->Nrows)&&(column<this->Ncolumns)) data[row][column]=x;
else
    {
    printf("Matrix.SetElement: Subscript out of range\r\n");
    }
}

void Matrix::SetElement(unsigned int row, unsigned int column,double x)
{
if((row<this->Nrows)&&(column<this->Ncolumns)) data[row][column]=x;
else
    {
    printf("Matrix.SetElement: Subscript out of range\r\n");
    }
}

double complex* Matrix::operator[](unsigned int row)
{
```

```
return this->data[row];
}

unsigned int Matrix::GetNumRows(void) const
{
return this->Nrows;
}

unsigned int Matrix::GetNumColumns(void) const
{
return this->Ncolumns;
}

/* Interface de entrada e saída em disco e terminal */
void Matrix::SaveToFile(string filename) const
{
FILE* fout;
fout = fopen(filename.c_str(),"wb");
if(!fout) { printf("Matrix.SaveToFile failed!\r\n"); return; }

fwrite(&(this->Nrows),sizeof(unsigned int),1,fout);
fwrite(&(this->Ncolumns),sizeof(unsigned int),1,fout);

for(unsigned int i=0;i<this->Nrows;i++) fwrite(this->data[i],sizeof(double complex),this->Ncolumns,fout);

if(fout) fclose(fout);
}

void Matrix::LoadFromFile(string filename)
{
FILE* fin;
unsigned int rowsnum;
unsigned int columnsnum;

fin = fopen(filename.c_str(),"rb");
if(!fin) { printf("Matrix.LoadFromFile failed!\r\n"); return; }
rewind(fin);
fread(&rowsnum,sizeof(unsigned int),1,fin);
fread(&columnsnum,sizeof(unsigned int),1,fin);

if(data) {
         for(unsigned int i=0;i<this->Nrows;i++) free(data[i]);
              free(data);
         }

this->Nrows = rowsnum;
this->Ncolumns = columnsnum;

this->data = (double complex**)malloc(sizeof(double complex*)*this->Nrows);
for(register unsigned int i=0;i<this->Nrows;i++) this->data[i]=(double complex*)malloc(sizeof(double complex)*this-
>Ncolumns);

for(unsigned int i=0;i<this->Nrows;i++) fread(this->data[i],sizeof(double complex),this->Ncolumns,fin);

if(fin) fclose(fin);
}

void Matrix::PrintOut(void) const
{
printf("\r\n");
printf("    ");
for(register unsigned int j=0;j<this->Ncolumns;j++) printf(" [      %04u     ]",j);
for(register unsigned int i=0;i<this->Nrows;i++)
   {
   printf("%04u",i);
   for(register unsigned int j=0;j<this->Ncolumns;j++) printf(" [%+7.3f%+7.3fj]",creal(data[i][j]),cimag(data[i]
[j]));
   printf("\r\n");
   }
printf("\r\n");
}
```

Arquivo **signal.cc**:

```
#include "signal.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sndfile.h>
#include "wavfile.h"
#include <algorithm>
#include <math.h>
#include <time.h>

electric_signal::electric_signal()
{
//printf("signal.Default Constructor ");
this->samples = NULL;
this->num_samples = 0;
this->samplingrate = 0;
```

```
this->duration = 0.0;
this->sample_duration = 0.0;
this->minvalue = 0.0;
this->maxvalue = 0.0;
this->name = (char*)malloc(sizeof(char)); *this->name=0;
this->date=(char*)malloc(sizeof(char)); *this->date=0;
this->comments=(char*)malloc(sizeof(char)); *this->comments=0;
this->DCvalue = 0.0;
//printf("Created.\r\n");
}

//copy constructor
electric_signal::electric_signal(const electric_signal& source)
{
//printf("signal.Copy Constructor ");
//if(this->samples!=NULL) { free(samples); samples=NULL; }
this->num_samples = source.num_samples;
this->samplingrate = source.samplingrate;
this->duration = source.duration;
this->sample_duration = source.sample_duration;
this->minvalue = source.minvalue;
this->maxvalue = source.maxvalue;
this->DCvalue = source.DCvalue;
this->samples = (double*)malloc(sizeof(double)*this->num_samples);
for(unsigned int i=0;i<this->num_samples;i++) this->samples[i] = source.samples[i];
this->name = (char*)malloc((strlen(source.name)+1)*sizeof(char));
strcpy(this->name,source.name);
this->date = (char*)malloc((strlen(source.date)+1)*sizeof(char));
strcpy(this->date,source.date);
this->comments = (char*)malloc((strlen(source.comments)+1)*sizeof(char));
strcpy(this->comments,source.comments);
//printf("%s\r\n",this->name);
}

//assignment constructor
electric_signal& electric_signal::operator=(const electric_signal& source)
{
//printf("\r\nsignal.Assignment Constructor: "); fflush(stdout);
this->name = (char*)malloc((strlen(source.name)+1)*sizeof(char));
strcpy(this->name,source.name);
this->date = (char*)malloc((strlen(source.date)+1)*sizeof(char));
strcpy(this->date,source.date);
this->num_samples = source.num_samples;
this->samplingrate = source.samplingrate;
this->duration = source.duration;
this->sample_duration = source.sample_duration;
this->minvalue = source.minvalue;
this->maxvalue = source.maxvalue;
this->DCvalue = 0.0;
this->samples = (double*)malloc(sizeof(double)*this->num_samples);
for(register unsigned int i=0;i<this->num_samples;i++) this->samples[i] = source.samples[i];
//printf("Name pointer: %p ",this->name);
//printf("Assign Constructed %s (copy)\r\n",this->name); fflush(stdout);
return *this;
}


//destructor
electric_signal::~electric_signal()
{
//printf("signal.Destructor: "); fflush(stdout);
if(this->samples!=NULL) { free(this->samples); this->samples=NULL; }
if(this->date) { free(this->date); this->date=NULL; }
if(this->comments) { free(this->comments); this->comments=NULL; }
//printf("%s Destroyed\r\n",this->name); fflush(stdout);
if(this->name) { free(this->name); this->name=NULL; }
}

double* electric_signal::get_samples_pointer(void)
{
return this->samples;
}

unsigned long int electric_signal::get_num_samples(void) const
{
return this->num_samples;
}

unsigned int electric_signal::get_samplingrate(void) const
{
return this->samplingrate;
}

double electric_signal::get_duration(void)
{
return this->duration;
}

double electric_signal::get_sample_duration(void)
{
return this->sample_duration;
}

//cria um objeto da classe signal definindo quantas samples, a samplingrate e seu nome. Inicialmente todas as
amostras são zeradas.
```

```
/*****************************************************************************************************/
electric_signal::electric_signal(unsigned long int nsamples,unsigned int nsamplingrate,const char* signalname)
{
//printf("\r\nsignal.Overloaded Constructor: %s %lu %p ",signalname,strlen(signalname),signalname); fflush(stdout);
this->DCvalue = 0.0;
this->maxvalue = 0.0;
this->minvalue = 0.0;
this->num_samples = nsamples;
this->samplingrate = nsamplingrate;
this->samples = (double*)malloc(sizeof(double)*nsamples);
this->sample_duration = 1.0/this->samplingrate;
this->duration = 1.0*this->num_samples/this->samplingrate;
for(unsigned int i=0;i<this->num_samples;i++) this->samples[i]=0.0; //zera inicialmente o sinal
//printf("\r\nsignal.Overloaded Constructor: %s %lu %p ",signalname,strlen(signalname),signalname);

this->date=(char*)malloc(sizeof(char)); *this->date=0;
this->comments=(char*)malloc(sizeof(char)); *this->comments=0;
this->name=(char*)malloc((strlen(signalname)+1)*sizeof(char));
strcpy(this->name,signalname);

//printf("Name pointer: %p ",this->name);
//printf("Zero Constructor %s\r\n",this->name); fflush(stdout);
}


electric_signal::electric_signal(unsigned long int nsamples,double* source,unsigned int nsamplingrate,const char*
signalname)
{
this->num_samples = nsamples;
this->samplingrate = nsamplingrate;
this->samples = (double*)malloc(sizeof(double)*nsamples);
this->sample_duration = 1.0/this->samplingrate;
this->duration = 1.0*this->num_samples/this->samplingrate;
for(unsigned int i=0;i<this->num_samples;i++) this->samples[i]=source[i]; //copia inicialmente o sinal
this->name=(char*)malloc((strlen(signalname)+1)*sizeof(char));
strcpy(this->name,signalname);
this->date=(char*)malloc(sizeof(char)); *this->date=0;
this->comments=(char*)malloc(sizeof(char)); *this->comments=0;
this->DCvalue = this->get_DC();
this->maxvalue = this->get_max();
this->minvalue = this->get_min();
//printf("signal: Buffer Constructor %s\r\n",this->name.c_str());
}

//construtor que cria um objeto signal a partir de um canal de um arquivo WAV e batiza com um nome:
/*****************************************************************************************************/
electric_signal::electric_signal(const char* filename, unsigned int n_channel)
{
unsigned int num_channels;
WavFile wavfile(filename);

num_channels = wavfile.get_num_channels();

if(n_channel>=num_channels)
  {
  printf("Invalid channel number: %u. Max %u. Aborting.\r\n",n_channel,num_channels-1);
  exit(EXIT_FAILURE);
  }

this->DCvalue = 0.0;
this->maxvalue = 0.0;
this->minvalue = 0.0;

this->num_samples = wavfile.get_num_samples();
this->samplingrate = wavfile.get_samplingrate();
this->duration = wavfile.get_duration();
this->sample_duration = wavfile.get_sample_duration();

//aloca o espaço para as amostras do sinal:
this->samples = (double*)malloc(sizeof(double)*this->num_samples);

//preenche o nome do sinal com o nome armazenado no metadata daquele canal
this->name=(char*)malloc((strlen(wavfile.get_channel_name(n_channel))+1)*sizeof(char));
strcpy(this->name,wavfile.get_channel_name(n_channel));

//faz o mesmo para a data
this->date=(char*)malloc((strlen(wavfile.get_metadata_date())+1)*sizeof(char));
strcpy(this->date,wavfile.get_metadata_date());

//faz o mesmo para os comentários
this->comments=(char*)malloc((strlen(wavfile.get_metadata_comments())+1)*sizeof(char));
strcpy(this->comments,wavfile.get_metadata_comments());

//aloca um buffer de recepção
//double* buf = wavfile.load_samples();

//aloca um buffer de recepção para somente um frame:
double * buf = (double*)malloc(sizeof(double)*num_channels);

//copia somente as amostras do canal de interesse:
//for(unsigned int i=0;i<this->num_samples;i++) { this->samples[i]=buf[i*num_channels+n_channel]; }

SF_INFO sfinfo;

SNDFILE* file = sf_open(filename,SFM_READ,&sfinfo);
```

```
sf_seek(file,0,SEEK_SET);

//lê somente as amostras de interesse:
for(unsigned int i=0;i<this->num_samples;i++)
   {
   //lê um frame de cada vez
   if(sf_readf_double(file,buf,1)!=1)
     {
          printf("Error reading sample %u, channel %s from file %s.",i,this->name,filename);
     };
   //armazena no buffer final
   this->samples[i]=buf[n_channel];
   }

sf_close(file);

//aplica o ganho:
this->apply_gain(wavfile.get_channel_range(n_channel));

//desaloca o buffer de recepção do arquivo
free(buf); buf = NULL;

printf("Read file %s channel %u: %lu samples.\r\n",filename,n_channel,this->get_num_samples());
printf("Max: %+f   Min: %+f  DC: %+f Range: ±%4.1f V (%s)\r\n",this->get_max(),this->get_min(),this-
>get_DC(),wavfile.get_channel_range(n_channel),this->name);

//printf("signal: WavFile channel Constructor %s\r\n",this->name.c_str());
}
/*****************************************************************************************************/
void electric_signal::SaveSignalSetToDisk(char* filename,electric_signal* signalpointer,unsigned int numsignals)
{
WavFile outputfile; //construtor para ser usado para gravação de arquivo

outputfile.set_basename(filename);
outputfile.set_type(WavFile::WAV);
//configura o numero de canais
outputfile.set_num_channels(numsignals);
//configura a taxa de amostragem
outputfile.set_samplingrate(signalpointer[0].get_samplingrate());

double* outbuffer = (double*)malloc(numsignals*signalpointer[0].get_num_samples()*sizeof(double));
//faz o interleave:
double* p = outbuffer;
for(unsigned int i=0;i<signalpointer[0].get_num_samples();i++)
   {
   for(unsigned int j=0;j<numsignals;j++) { *p=signalpointer[j][i]; p++; };
   }

//configura a data:
char buffer [800];
/*time_t rawtime;
struct tm * timeinfo;
time(&rawtime);
timeinfo=localtime(&rawtime);
strftime(buffer,80,"%d/%m/%Y %H:%M:%S",timeinfo);*/
strcpy(buffer,signalpointer[0].get_date());
outputfile.set_metadata_date(buffer);

//configura os comentários:
strcpy(buffer,"Range");
for(unsigned int i=0;i<numsignals;i++) strcat(buffer,";1,00");
outputfile.set_metadata_comments(buffer);

outputfile.set_metadata_software("");
//configura os títulos dos sinais nos canais:
strcpy(buffer,"Channel Name");
for(unsigned int i=0;i<numsignals;i++) { strcat(buffer,";"); strcat(buffer,signalpointer[i].get_name()); }
outputfile.set_metadata_title(buffer);

//abre os arquivos e escreve seus conteúdos:
outputfile.save_samples(outbuffer,signalpointer[0].get_num_samples());
free(outbuffer);
}

/*****************************************************************************************************/

/*****************************************************************************************************/
void electric_signal::SaveSignalSetToDisk(char* filename,electric_signal** signalpointer,unsigned int numsignals)
{
WavFile outputfile; //construtor para ser usado para gravação de arquivo

outputfile.set_basename(filename);
outputfile.set_type(WavFile::WAV);
//configura o numero de canais
outputfile.set_num_channels(numsignals);
//configura a taxa de amostragem
outputfile.set_samplingrate((signalpointer[0])->get_samplingrate());

double* outbuffer = (double*)malloc(numsignals*(*signalpointer[0]).get_num_samples()*sizeof(double));
//faz o interleave:
double* p = outbuffer;
for(unsigned int i=0;i<(*signalpointer[0]).get_num_samples();i++)
   {
   for(unsigned int j=0;j<numsignals;j++) { *p=(*signalpointer[j])[i]; p++; };
   }
```

```
//configura a data:
char buffer [800];
/*time_t rawtime;
struct tm * timeinfo;
time(&rawtime);
timeinfo=localtime(&rawtime);
strftime(buffer,800,"%d/%m/%Y %H:%M:%S",timeinfo); */
strcpy(buffer,(*signalpointer[0]).get_date());
outputfile.set_metadata_date(buffer);

//configura os comentários:
strcpy(buffer,"Range");
for(unsigned int i=0;i<numsignals;i++) strcat(buffer,";1,00");
outputfile.set_metadata_comments(buffer);

outputfile.set_metadata_software("");
//configura os títulos dos sinais nos canais:
strcpy(buffer,"Channel Name");
for(unsigned int i=0;i<numsignals;i++)
   {
         //printf("%s \r\n",(*signalpointer[i]).get_name()); fflush(stdout);
   strcat(buffer,";");
   strcat(buffer,(*signalpointer[i]).get_name());
   }
outputfile.set_metadata_title(buffer);

//abre os arquivos e escreve seus conteúdos:
outputfile.save_samples(outbuffer,(*signalpointer[0]).get_num_samples());
free(outbuffer);
}

/*****************************************************************************************************/
void electric_signal::LoadSignalSetFromDisk(char* filename,electric_signal* signalpointer,unsigned int numsignals)
{
electric_signal* channel;
if(signalpointer==NULL)
  {
  printf("Invalid reception buffer: Function LoadSignalSetFromDisk. Aborting\r\n");
  exit(EXIT_FAILURE);
  }

channel = signalpointer;
//para cada canal:
for(unsigned int i=0;i<numsignals;i++)
   {
   channel[i] = electric_signal(filename,i);
   }
}
/*****************************************************************************************************/
double electric_signal::get_max(void)
{
this->maxvalue = this->samples[0];
for(unsigned int i=0;i<this->num_samples;i++) this->maxvalue = std::max(this->maxvalue,this->samples[i]);
return this->maxvalue;
}
/*****************************************************************************************************/
double electric_signal::get_min(void)
{
this->minvalue = this->samples[0];
for(unsigned int i=0;i<this->num_samples;i++) this->minvalue = std::min(this->minvalue,this->samples[i]);
return this->minvalue;
}
/*****************************************************************************************************/
double& electric_signal::operator[](unsigned int i)
{
return this->samples[i];
}
/*****************************************************************************************************/
double electric_signal::get_DC(void)
{
long double acum;
acum=0.0;
for(unsigned int i=0;i<this->num_samples;i++) acum+=this->samples[i];
this->DCvalue = acum/this->num_samples;
return this->DCvalue;
}
/*****************************************************************************************************/
void electric_signal::removeDC(void)
{
double DC;
DC = this->get_DC();
for(unsigned int i=0;i<this->num_samples;i++) this->samples[i]-=DC;
this->get_DC();
}
/*****************************************************************************************************/
void electric_signal::add_DC(double value)
{
for(unsigned int i=0;i<this->num_samples;i++) this->samples[i]+=value;
this->get_DC();
this->get_max();
this->get_min();
}
/*****************************************************************************************************/
void electric_signal::apply_gain(double gain)
```

```
{
for(unsigned int i=0;i<this->num_samples;i++) this->samples[i]*=gain;
this->get_DC();
this->get_max();
this->get_min();
}
/*****************************************************************************************************/
double electric_signal::maximize(void) //maximiza o sinal e retorna o inverso do ganho aplicado necessário para
maximização
{
double Yrange;
double gain;
double meanvalue;

meanvalue = (this->get_max()+this->get_min())/2.0;
this->add_DC(-meanvalue); //centraliza o sinal em torno do zero.

Yrange = (this->get_max()-this->get_min())/2.0;
gain = 1.0/Yrange;
for(unsigned int i=0;i<this->num_samples;i++) this->samples[i]*=gain;
return Yrange;
}
/*****************************************************************************************************/
void electric_signal::SmithTrigger(double Hhigh,double Hlow)
{
double out;
out= -0.75;
for(unsigned int i=0;i<this->num_samples;i++)
   {
   if((out<0)&&(this->samples[i]>Hhigh)) out = 0.75;
   if((out>0)&&(this->samples[i]<Hlow)) out = -0.75;
   this->samples[i] = out;
   }
}


/*****************************************************************************************************/
void electric_signal::set_name(const char* newname)
{

this->name = (char*)malloc((strlen(newname)+1)*sizeof(char));
strcpy(this->name,newname);
}
/*****************************************************************************************************/
const char* electric_signal::get_name(void) const
{
return this->name;
}
/*****************************************************************************************************/
void electric_signal::set_date(const char* newdate)
{
this->date = (char*)malloc((strlen(newdate)+1)*sizeof(char));
strcpy(this->date,newdate);
}
/*****************************************************************************************************/
const char* electric_signal::get_date(void) const
{
return this->date;
}
/*****************************************************************************************************/
void electric_signal::set_comments(const char* newcomments)
{
this->comments = (char*)malloc((strlen(newcomments)+1)*sizeof(char));
strcpy(this->comments,newcomments);
}
/*****************************************************************************************************/
const char* electric_signal::get_comments(void) const
{
return this->comments;
}
/*****************************************************************************************************/
void electric_signal::monoestable(bool change,unsigned int timing)
{

unsigned int i=0;
unsigned int j;

if(change) //configurado para detectar transições positivas
  {
  while(i<(this->num_samples-1))
       {
       if(this->samples[i]<this->samples[i+1])
         {
           j=timing;
         while(j-->0) this->samples[i++]=+0.75;
          this->samples[i]=-0.75;
         }
       else this->samples[i]=-0.75;
       i++;
       }
  }
else //configurado para detectar transições negativas
  {
  while(i<(this->num_samples-1))
       {
```

```
       if(this->samples[i]>this->samples[i+1])
          {
          j=timing;
         while(j-->0) this->samples[i++]=+0.75;
          this->samples[i]=-0.75;
         }
       else this->samples[i]=-0.75;
       i++;
       }
  }
}
/*************************************************************************************************/
void electric_signal::DivideFrequency(unsigned int divfactor)
{
unsigned long int i=0;
unsigned int j=0;
double last_sample;
bool beginning = true;

last_sample=this->samples[0];

while(i<(this->num_samples-1))
     {
     if((this->samples[i]<0.0)&&(this->samples[i+1]>=0.0)) //se achou uma transição positiva
       {
       j++;
       last_sample = this->samples[i];
       if (beginning) { for(unsigned long int k=0;k<i;k++) this->samples[k]=last_sample; beginning = false; }
       }
     if(j==divfactor) j=0;
     if(j!=1) this->samples[i]=last_sample;
     i++;
     }
}
/*************************************************************************************************/
double electric_signal::MeasureDigitalFrequency(void)
{
double freq;
long double time_elapsed;
unsigned int pulses=0;
unsigned long int firstpulse;
unsigned long int lastpulse;

firstpulse = 0;
lastpulse = this->num_samples;

unsigned long int i=0;
while(i<(this->num_samples-1))
     {
     if(this->samples[i]<this->samples[i+1]) //se encontrou uma transição positiva
       {
       pulses++;
       if(pulses==1) firstpulse=i;
       lastpulse = i;
       }
     i++;
     }

time_elapsed = (lastpulse-firstpulse)*this->sample_duration;

//printf("Cycles detected: %u\r\n",pulses-1);
//printf("Time Elapsed: %Lf seconds\r\n",time_elapsed);

freq = 1.0*(pulses-1)/time_elapsed;
return freq;
}
/*************************************************************************************************/
double electric_signal::MeasureSinusoidalFrequency(void)
{
double freq;
long double time_elapsed,first_transition,last_transition;
unsigned int pulses=0;
unsigned long int firstpulse;
unsigned long int lastpulse;

firstpulse = 0;
lastpulse = this->num_samples;

unsigned long int i=0;
while(i<(this->num_samples-1))
     {
     if((this->samples[i]<0.0)&&(this->samples[i+1]>=0.0)) //se encontrou uma transição positiva
       {
       pulses++;
       if(pulses==1) firstpulse=i;
       lastpulse = i;
       }
     i++;
     }

first_transition = this->sample_duration*(firstpulse+0.5)+this->sample_duration*(-1.0*this->samples[firstpulse]/
(this->samples[firstpulse+1]-this->samples[firstpulse]));
last_transition = this->sample_duration*(lastpulse+0.5)+this->sample_duration*(-1.0*this->samples[lastpulse]/(this-
>samples[lastpulse+1]-this->samples[lastpulse]));
```

```
time_elapsed = last_transition-first_transition;

//printf("Cycles detected: %u\r\n",pulses-1);
//printf("Time Elapsed: %Lf seconds\r\n",time_elapsed);

freq = 1.0*(pulses-1)/time_elapsed;
return freq;
}
/*****************************************************************************************************/
double electric_signal::MeasureAveragePower(void) const
{
long double acumulator=0.0;
for(unsigned long int i=0;i<this->num_samples;i++) acumulator+=pow(this->samples[i],2.0);
acumulator/=this->num_samples;
return acumulator;
}
/*****************************************************************************************************/
unsigned int electric_signal::CountCycles(void)
{
unsigned int pulses=0;
unsigned long int i=0;
while(i<(this->num_samples-1))
     {
     if((this->samples[i]<0.0)&&(this->samples[i+1]>=0.0)) //se encontrou uma transição positiva
       {
       pulses++;
       }
     i++;
     }
return (pulses-1);
}
/*****************************************************************************************************/
unsigned long int* electric_signal::CyclesStartPoints(void)
{
unsigned int cycles;
unsigned long int* retvector;
unsigned long int i;
unsigned int j;

cycles = this->CountCycles();
retvector = (unsigned long int*)malloc(sizeof(unsigned long int)*(cycles+1));
i=0;
j=0;
while(i<(this->num_samples-1))
     {
     if((this->samples[i]<0.0)&&(this->samples[i+1]>=0.0)) //se encontrou uma transição positiva
       {
       retvector[j++]=i+1;
       }
     i++;
     }
return retvector;
}

void electric_signal::SaveToDisk(string filename)
{
if(!this->samples) return;
WavFile outputfile; //construtor para ser usado para gravação de arquivo

outputfile.set_basename(filename.c_str());
outputfile.set_type(WavFile::WAV);
//configura o numero de canais
outputfile.set_num_channels(1);
//configura a taxa de amostragem
outputfile.set_samplingrate(this->get_samplingrate());

//configura os comentários:
outputfile.set_metadata_comments("Range;1,00");
//configura a data:
char buffer [80];
time_t rawtime;
struct tm * timeinfo;
time(&rawtime);
timeinfo=localtime(&rawtime);
strftime(buffer,80,"%d/%m/%Y %H:%M:%S",timeinfo);
outputfile.set_metadata_date(buffer);

outputfile.set_metadata_software("");
//configura os títulos dos sinais nos canais:
outputfile.set_metadata_title(string(string("Channel Name;")+string(this->name)).c_str());

//abre os arquivos e escreve seus conteúdos:
outputfile.save_samples(this->samples,this->get_num_samples());
}


/*****************************************************************************************************/
/
*****************************************************************************************************
***/
void IIRFilter::init(void)
{
this->order =0;
this->numerator_order=0;
this->denominator_order=0;
```

```
this->shifted_inputs=NULL;
this->shifted_outputs=NULL;
this->denominator=NULL;
this->numerator=NULL;
this->input=0.0;
this->output=0.0;
this->constant=0.0;
}

unsigned int IIRFilter::get_order(void)
{
return this->order;
}

void IIRFilter::reset_state(void)
{
if(this->shifted_inputs!=NULL) for(unsigned int i=0;i<this->order+1;i++) this->shifted_inputs[i]=0.0;
if(this->shifted_outputs!=NULL) for(unsigned int i=0;i<this->order+1;i++) this->shifted_outputs[i]=0.0;
}

void IIRFilter::set_coeficients(double* n,unsigned int norder,double* d,unsigned int dorder,double cte)
{
if(n==NULL)
  {
  printf("Null pointer passed to IIRFilter numerator. Aborting\r\n");
  exit(-1);
  }

if(d==NULL)
  {
  printf("Null pointer passed to IIRFilter denominator. Aborting\r\n");
  exit(-1);
  }

this->numerator_order = norder;
this->denominator_order = dorder;

this->numerator = (double*)malloc(sizeof(double)*(this->numerator_order+1));
this->denominator = (double*)malloc(sizeof(double)*(this->denominator_order+1));
for(unsigned int i=0;i<=this->numerator_order;i++) this->numerator[i]=n[i];
for(unsigned int i=0;i<=this->denominator_order;i++) this->denominator[i]=d[i];
this->constant = cte;

this->order = this->denominator_order;
//if(this.order<this.numerator_order-1) this.order = this.numerator_order-1;

this->shifted_inputs = (double*)malloc(sizeof(double)*(this->order+1));
this->shifted_outputs = (double*)malloc(sizeof(double)*(this->order+1));
this->reset_state();
}

void IIRFilter::feedforward(void)
{
//Esta função calcula a saída do filtro, levando em conta a entrada (input)

//primeiro desloca as n últimas entradas:
for(unsigned int i=this->order;i>0;i--) this->shifted_inputs[i]=this->shifted_inputs[i-1];
this->shifted_inputs[0]=this->input*this->constant;
//desloca as n ultimas saídas:
for(unsigned int i=this->order;i>1;i--) this->shifted_outputs[i]=this->shifted_outputs[i-1];
this->shifted_outputs[1]=this->output;
//calcula a saída de acordo com as condições atualizadas
this->output = 0.0;
for(unsigned int i=0;i<=this->numerator_order;i++) this->output+=this->numerator[i]*this->shifted_inputs[i];
for(unsigned int i=1;i<=this->denominator_order;i++) this->output+=this->denominator[i]*this->shifted_outputs[i];
//repare que o coeficiente a[0] (denominador) é ignorado, ou seja, é 1
}

void IIRFilter::apply(electric_signal* inputsignal)
{
//recebe um sinal e aplica a filtragem,
//o sinal de saída terá o mesmo tamanho do sinal de entrada
double* out = (double*)malloc(sizeof(double)*inputsignal->get_num_samples());

//aplica a filtragem
for(unsigned int i=0;i<inputsignal->get_num_samples();i++)
   {
   this->input=(*inputsignal)[i];
   this->feedforward();
   out[i] = this->output;
   }
//copia o resultado da filtragem para o sinal de entrada, sobrescrevendo-o
for(unsigned int i=0;i<inputsignal->get_num_samples();i++)
   {
   (inputsignal->get_samples_pointer())[i] = out[i];
   }
free(out);
}

void IIRFilter::printout(void)
{
printf("      ");
for (unsigned int i=0;i<=this->order;i++)
    {
          printf("%+7.3f",this->numerator[i]);
```

```
         if(i!=0) printf("*z^-%u ",i);
         fflush(stdout);
    }
printf("\r\nH(z)=----------");
for(unsigned int i=0;i<this->order;i++) printf("----------------");
printf("\r\n      ");
for (unsigned int i=0;i<=this->order;i++)
    {
         printf("%+7.3f",this->denominator[i]);
         if(i!=0) printf("*z^-%u ",i);
         fflush(stdout);
         }
printf("\r\n");
}

IIRFilter::IIRFilter()
{
this->init();
}

IIRFilter::~IIRFilter()
{
if(this->shifted_inputs) free(this->shifted_inputs);
if(this->shifted_outputs) free(this->shifted_outputs);
if(this->numerator) free(this->numerator);
if(this->denominator) free(this->denominator);
}

/
*************************************************************************************************************
***/
void FIRFilter::init(void)
{
this->order =0;
this->numerator_order=0;
this->shifted_inputs=NULL;
this->numerator=NULL;
this->input=0.0;
this->output=0.0;
this->constant=0.0;
}

unsigned int FIRFilter::get_order(void)
{
return this->order;
}

void FIRFilter::reset_state(void)
{
if(this->shifted_inputs!=NULL) for(unsigned int i=0;i<this->order+1;i++) this->shifted_inputs[i]=0.0;
}

void FIRFilter::set_coeficients(double* n,unsigned int norder,double cte)
{
if(n==NULL)
  {
  printf("Null pointer passed to IIRFilter numerator. Aborting\r\n");
  exit(-1);
  }

this->numerator_order = norder;

this->numerator = (double*)malloc(sizeof(double)*(this->numerator_order+1));
for(unsigned int i=0;i<=this->numerator_order;i++) this->numerator[i]=n[i];
this->constant = cte;

this->order = this->numerator_order;
//if(this.order<this.numerator_order-1) this.order = this.numerator_order-1;

this->shifted_inputs = (double*)malloc(sizeof(double)*(this->order+1));
this->reset_state();
}

void FIRFilter::feedforward(void)
{
//Esta função calcula a saída do filtro, levando em conta a entrada (input)

//primeiro desloca as n últimas entradas:
for(unsigned int i=this->order;i>0;i--) this->shifted_inputs[i]=this->shifted_inputs[i-1];
this->shifted_inputs[0]=this->input*this->constant;
//calcula a saída de acordo com as condições atualizadas
this->output = 0.0;
for(unsigned int i=0;i<=this->numerator_order;i++) this->output+=this->numerator[i]*this->shifted_inputs[i];
}

void FIRFilter::apply(electric_signal* inputsignal)
{
//recebe um sinal e aplica a filtragem,
//o sinal de saída terá o mesmo tamanho do sinal de entrada
double* out = (double*)malloc(sizeof(double)*inputsignal->get_num_samples());

//aplica a filtragem
for(unsigned int i=0;i<inputsignal->get_num_samples();i++)
   {
   this->input=(*inputsignal)[i];
```

```
   this->feedforward();
   out[i] = this->output;
   }
//copia o resultado da filtragem para o sinal de entrada, sobrescrevendo-o
for(unsigned int i=0;i<inputsignal->get_num_samples();i++)
   {
   (inputsignal->get_samples_pointer())[i] = out[i];
   }
free(out);
}

void FIRFilter::printout(void)
{
printf("H(z)= ");
for (unsigned int i=0;i<=this->order;i++)
              {
                    printf("%+7.3f",this->numerator[i]);
                    if(i!=0) printf("*z^-%u ",i);
                    fflush(stdout);
              }
printf("\r\n");
}

FIRFilter::FIRFilter()
{
this->init();
}

FIRFilter::~FIRFilter()
{
if(this->shifted_inputs) free (this->shifted_inputs);
if(this->numerator) free(this->numerator);
}
/
*****************************************************************************************************************
***/
```

Arquivo **wavfile.cc**:

```
#include "wavfile.h"
#include <stdlib.h>
#include <string.h>
#include <sndfile.h>

void WavFile::init(void)
{
this->filename = string("");
this->basename = string("");
this->extention = string("");
this->metadata_title = string("");
this->metadata_copyright = string("");
this->metadata_software = string("");
this->metadata_artist = string("");
this->metadata_comments = string("");
this->metadata_date = string("");
this->metadata_album = string("");
this->metadata_license = string("");
this->metadata_tracknumber = string("");
this->metadata_genre = string("");
this->num_channels = 0;
this->samplingrate = 0;
this->time_duration = 0.0;
this->bits_per_sample = 0;
this->filesize = 0;
this->file_type = INVALID;
this->num_samples = 0;
this->bits_per_second = 0;
this->channel_name = NULL;
}

WavFile::WavFile(void)
{
this->init();
}

WavFile::WavFile(const char* file_name)
{
this->init();
this->filename = string(file_name);
this->basename = string(file_name);
this->basename.erase(this->basename.rfind('.'),string::npos);
this->extention = this->filename.substr(this->filename.rfind('.')+1,string::npos);

if(this->extention.compare("bin")==0)
  {
  this->file_type = BIN;
```

```
  //adicionar código para procurar no arquivo _info.txt associado
  printf("Not implemented .bin files yet\r\n"); exit(EXIT_FAILURE);
  }

SNDFILE *sf;
SF_INFO info;

info.format = 0;
sf = sf_open(this->filename.c_str(),SFM_READ,&info);
if (sf == NULL) { printf("Failed to open the file %s.\r\n",this->filename.c_str()); exit(EXIT_FAILURE);  }

this->num_channels = info.channels;
this->samplingrate = info.samplerate;
this->num_samples = info.frames;
this->time_duration = 1.0*this->num_samples/this->samplingrate;

this->channel_name = new string[this->num_channels];

//printf("file type: %X\r\n",info.format&SF_FORMAT_TYPEMASK);

switch(info.format&SF_FORMAT_TYPEMASK)
  {
  case SF_FORMAT_WAV:
    this->file_type = WAV;
  break;
  case SF_FORMAT_W64:
    this->file_type = W64;
  break;
  case SF_FORMAT_RF64:
    this->file_type = RF64;
  break;
  case SF_FORMAT_WAVEX:
    this->file_type = WAV;
  break;
  }

switch(info.format&SF_FORMAT_SUBMASK)
  {
  case SF_FORMAT_PCM_S8:
    this->bits_per_sample = 8;
  break;
  case SF_FORMAT_PCM_U8:
    this->bits_per_sample = 8;
  break;
  case SF_FORMAT_PCM_16:
    this->bits_per_sample = 16;
  break;
  case SF_FORMAT_PCM_24:
    this->bits_per_sample = 24;
  break;
  case SF_FORMAT_PCM_32:
    this->bits_per_sample = 32;
  break;
  case SF_FORMAT_FLOAT:
    this->bits_per_sample = 32;
  break;
  case SF_FORMAT_DOUBLE:
    this->bits_per_sample = 64;
  break;
  }

this->bits_per_second = this->bits_per_sample*this->samplingrate*this->num_channels;
this->filesize = (this->bits_per_sample/8)*this->num_channels*this->num_samples;

switch(info.format&SF_FORMAT_ENDMASK)
  {
  case SF_ENDIAN_FILE:
  break;
  case SF_ENDIAN_CPU:
  break;
  case SF_ENDIAN_LITTLE:
  break;
  case SF_ENDIAN_BIG:
  break;
  }

if(sf_get_string(sf,SF_STR_TITLE)!=NULL) this->metadata_title=string(sf_get_string(sf,SF_STR_TITLE));
if(sf_get_string(sf,SF_STR_COPYRIGHT)!=NULL) this->metadata_copyright=string(sf_get_string(sf,SF_STR_COPYRIGHT));
if(sf_get_string(sf,SF_STR_SOFTWARE)!=NULL) this->metadata_software=string(sf_get_string(sf,SF_STR_SOFTWARE));
if(sf_get_string(sf,SF_STR_ARTIST)!=NULL) this->metadata_artist=string(sf_get_string(sf,SF_STR_ARTIST));
if(sf_get_string(sf,SF_STR_COMMENT)!=NULL) this->metadata_comments=string(sf_get_string(sf,SF_STR_COMMENT));
if(sf_get_string(sf,SF_STR_DATE)!=NULL) this->metadata_date=string(sf_get_string(sf,SF_STR_DATE));
if(sf_get_string(sf,SF_STR_ALBUM)!=NULL) this->metadata_album=string(sf_get_string(sf,SF_STR_ALBUM));
if(sf_get_string(sf,SF_STR_LICENSE)!=NULL) this->metadata_license=string(sf_get_string(sf,SF_STR_LICENSE));
if(sf_get_string(sf,SF_STR_TRACKNUMBER)!=NULL) this-
>metadata_tracknumber=string(sf_get_string(sf,SF_STR_TRACKNUMBER));
if(sf_get_string(sf,SF_STR_GENRE)!=NULL) this->metadata_genre=string(sf_get_string(sf,SF_STR_GENRE));


sf_close(sf);
}

const char* WavFile::get_metadata_title(void) const
{
return this->metadata_title.c_str();
```

```
}

const char* WavFile::get_metadata_copyright(void) const
{
return this->metadata_copyright.c_str();
}

const char* WavFile::get_metadata_software(void) const
{
return this->metadata_software.c_str();
}

const char* WavFile::get_metadata_artist(void) const
{
return this->metadata_artist.c_str();
}

const char* WavFile::get_metadata_comments(void) const
{
return this->metadata_comments.c_str();
}

const char* WavFile::get_metadata_date(void) const
{
return this->metadata_date.c_str();
}

const char* WavFile::get_metadata_album(void) const
{
return this->metadata_album.c_str();
}

const char* WavFile::get_metadata_license(void) const
{
return this->metadata_license.c_str();
}

const char* WavFile::get_metadata_tracknumber(void) const
{
return this->metadata_tracknumber.c_str();
}

const char* WavFile::get_metadata_genre(void) const
{
return this->metadata_genre.c_str();
}

void WavFile::set_metadata_title(const char* source)
{
this->metadata_title = string(source);
}

void WavFile::set_metadata_copyright(const char* source)
{
this->metadata_copyright = string(source);
}

void WavFile::set_metadata_software(const char* source)
{
this->metadata_software = string(source);
}

void WavFile::set_metadata_artist(const char* source)
{
this->metadata_artist = string(source);
}

void WavFile::set_metadata_comments(const char* source)
{
this->metadata_comments = string(source);
}

void WavFile::set_metadata_date(const char* source)
{
this->metadata_date = string(source);
}

void WavFile::set_metadata_album(const char* source)
{
this->metadata_album = string(source);
}

void WavFile::set_metadata_license(const char* source)
{
this->metadata_license = string(source);
}

void WavFile::set_metadata_tracknumber(const char* source)
{
this->metadata_tracknumber = string(source);
}

void WavFile::set_metadata_genre(const char* source)
{
this->metadata_genre = string(source);
```

```
}


const char* WavFile::get_basename(void) const
{
return this->basename.c_str();
}

const char* WavFile::get_filename(void) const
{
return this->filename.c_str();
}

void WavFile::set_basename(const char* name)
{
if(this->basename.compare("")!=0) { printf("Error: Redefining WavFile object.\r\n"); exit(EXIT_FAILURE); }
this->basename = string(name);
}

void WavFile::set_type(filetype type)
{
if(type==INVALID) { printf("Error: Invalid WavFile object.\r\n"); exit(EXIT_FAILURE); }
if(this->filename.compare("")!=0) { printf("Error: Redefining WavFile object.\r\n"); exit(EXIT_FAILURE); } //não vai
tentar mudar o nome de um arquivo WAV que já existe...
if(this->basename.compare("")==0) { printf("Error: Invalid WavFile filename.\r\n"); exit(EXIT_FAILURE); } //não vale
mudar o tipo de arquivo sem ter um nome base.
this->file_type = type;
this->filename = this->basename;
switch (this->file_type)
     {
  case INVALID:
  break;
  case WAV:
  this->extention=string("wav");
  break;
  case W64:
  this->extention=string("w64");
  break;
  case RF64:
  this->extention=string("rf64");
  break;
  case BIN:
  this->extention=string("bin");
  break;
     }
this->filename.append(".");
this->filename.append(this->extention);
}


unsigned int WavFile::get_num_channels(void) const
{
return this->num_channels;
}

void WavFile::set_num_channels(unsigned int num)
{
if(this->num_channels!=0) { printf("Error: Invalid file.\r\n"); exit(EXIT_FAILURE); }  //não vai tentar mudar o nome
de um arquivo WAV que já existe...
this->num_channels = num;
}

unsigned int WavFile::get_samplingrate(void) const
{
return this->samplingrate;
}

void WavFile::set_samplingrate(unsigned int rate)
{
if(this->samplingrate!=0) { printf("Error: Invalid file.\r\n"); exit(EXIT_FAILURE); }  //não vai tentar mudar o nome
de um arquivo WAV que já existe...
this->samplingrate = rate;
}

unsigned long int WavFile::get_num_samples(void) const
{
return this->num_samples;
}

unsigned int WavFile::get_bits_per_sample(void) const
{
return this->bits_per_sample;
}

unsigned int WavFile::get_bits_per_second(void) const
{
return this->bits_per_second;
}

unsigned long int WavFile::get_filesize(void) const
{
return this->filesize;
}

double WavFile::get_duration(void) const
```

```
{
return this->time_duration;
}

double WavFile::get_sample_duration(void) const
{
return 1.0/this->samplingrate;
}

double* WavFile::load_samples(void) const
{
if(this->file_type==INVALID) { printf("Error: Invalid file.\r\n"); exit(EXIT_FAILURE); }

SNDFILE *sf;
double* buf;
SF_INFO info;

info.format = 0;
sf = sf_open(this->filename.c_str(),SFM_READ,&info);

if (sf == NULL) { printf("Failed to open the file %s.\r\n",this->filename.c_str()); exit(EXIT_FAILURE);  }
sf_command(sf,SFC_SET_NORM_FLOAT,NULL,SF_TRUE);
buf = (double*) malloc(this->num_samples*this->num_channels*sizeof(double));
if((unsigned long int)sf_read_double(sf,buf,this->num_samples*this->num_channels)<(this->num_samples*this-
>num_channels))
  {
  printf("Failed to open the file: %s. File is too short!\n",this->filename.c_str()); exit(EXIT_FAILURE);
  }
sf_close(sf);
return buf;
}


void WavFile::save_samples(double* samplespointer,unsigned long int n_samples) const
{
if(this->file_type==INVALID) { printf("Error: Trying to write an invalid file ?\r\n"); exit(EXIT_FAILURE); }
if(this->basename.compare("")==0) { printf("Error: Trying to write an unnamed file ?\r\n"); exit(EXIT_FAILURE); }
if(this->num_channels==0) { printf("Error: Trying to write an empty file ?\r\n"); exit(EXIT_FAILURE); }
if(this->samplingrate==0) { printf("Error: Did you forget the samplerate?\r\n"); exit(EXIT_FAILURE); }

SNDFILE* sf;
SF_INFO info;
//configura a taxa de amostragem
info.samplerate=this->samplingrate;
//configura o numero de canais
info.channels=this->num_channels;
//configura o formato como WAV microsoft little endian
switch(this->file_type)
  {
  case INVALID:
  break;
  case WAV:
  info.format=(SF_FORMAT_WAV|SF_FORMAT_DOUBLE);
  break;
  case W64:
  info.format=(SF_FORMAT_W64|SF_FORMAT_DOUBLE);
  break;
  case RF64:
  info.format=(SF_FORMAT_RF64|SF_FORMAT_DOUBLE);
  break;
  case BIN:
  info.format=(SF_FORMAT_RAW|SF_FORMAT_DOUBLE);
  break;
  }
sf = sf_open(this->filename.c_str(),SFM_WRITE,&info);

sf_set_string(sf,SF_STR_TITLE,this->metadata_title.c_str());
sf_set_string(sf,SF_STR_COPYRIGHT,this->metadata_copyright.c_str());
sf_set_string(sf,SF_STR_SOFTWARE,this->metadata_software.c_str());
sf_set_string(sf,SF_STR_ARTIST,this->metadata_artist.c_str());
sf_set_string(sf,SF_STR_COMMENT,this->metadata_comments.c_str());
sf_set_string(sf,SF_STR_DATE,this->metadata_date.c_str());
sf_set_string(sf,SF_STR_ALBUM,this->metadata_album.c_str());
sf_set_string(sf,SF_STR_LICENSE,this->metadata_license.c_str());
sf_set_string(sf,SF_STR_TRACKNUMBER,this->metadata_tracknumber.c_str());
sf_set_string(sf,SF_STR_GENRE,this->metadata_genre.c_str());

printf("Written %u frames to file %s.\r\n",(unsigned int)sf_writef_double(sf,samplespointer,n_samples),this-
>filename.c_str());
sf_close(sf);
}


const char* WavFile::get_channel_name(unsigned int channel_num) const
{
//Channel Name;Channel 1;Channel 2;Channel 3;Channel 4
if(this->metadata_title.length()==0)
  {
  char retvalue[20];
  sprintf(retvalue,"No Name %02u",channel_num);
  return string(retvalue).c_str();
  }

unsigned int pos=0; //começa do início da string:
unsigned int end=0;
```

```
for(unsigned int i=0;i<=channel_num;i++)
   {
   pos = this->metadata_title.find(';',pos);  //acha o (i+1)-ésimo ponto e vírgula
   pos++;
   }
end = this->metadata_title.find(';',pos);
this->channel_name[channel_num] = string(this->metadata_title.substr(pos,end-pos));
return this->channel_name[channel_num].c_str();
}

double WavFile::get_channel_range(unsigned int channel_num) const
{
//Range;2,5;2,5;1,25;10
if(this->metadata_comments.length()==0) return 1.0;
unsigned int pos=0; //começa do início da string:
unsigned int end=0;
for(unsigned int i=0;i<=channel_num;i++)
   {
   pos = this->metadata_comments.find(';',pos);  //acha o (i+1)-ésimo ponto e vírgula
   pos++;
   }
end = this->metadata_comments.find(';',pos);

string tmpvalue = string(this->metadata_comments.substr(pos,end-pos));

for(unsigned int i=0;i<tmpvalue.length();i++)
   if(tmpvalue.at(i)==',') tmpvalue[i]='.';

return atof(tmpvalue.c_str());
}

WavFile::~WavFile()
{
//printf("Calling Destructor\r\n");
this->filename = string("");
this->basename = string("");
this->extention = string("");
this->metadata_title = string("");
this->metadata_copyright = string("");
this->metadata_software = string("");
this->metadata_artist = string("");
this->metadata_comments = string("");
this->metadata_date = string("");
this->metadata_album = string("");
this->metadata_license = string("");
this->metadata_tracknumber = string("");
this->metadata_genre = string("");
if(this->channel_name) { delete[] this->channel_name; this->channel_name = NULL;};
}
```

Arquivo **window.c**:

```
#include "window.h"
#include <math.h>
#include <stdlib.h>
#include <complex.h>

double* generate_window(int window,const int N)
{
int i;
double* w = NULL; //os valores das amostras da janela
int n; //número de amostras na janela
const double PI=3.141592653589793;

n = pow(2,N); //N é sempre um numero par
w = (double*)malloc(n*sizeof(double)); //aloca 2^N doubles para guardar a janela

if(w==NULL) return NULL;

/*perform the window function requsted by the user*/
    switch(window)
    {
        case 1:
            for(i=0;i<=n-1;i++)
                w[i]=1.0;
            break;
        case 2:
            for(i=0;i<=(n-1)/2;i++)
                w[i]=2.0*i/(n-1);
            for(i=(n-1)/2+1;i<=n-1;i++)
                w[i]=2.0-2.0*i/(n-1);
            break;
        case 3:
            for(i=0;i<=n-1;i++)
                w[i]=(1.0-cos(2.0*PI*i/(n-1)))/2.0;
            break;
        case 4:
            for(i=0;i<=n-1;i++)
                w[i]=0.54-0.46*cos(2*PI*i/(n-1));
            break;
```

```
        case 5:
            for(i=0;i<=n-1;i++)
                w[i]=0.42-0.5*cos(2*PI*i/(n-1))+0.08*cos(4*PI*i/(n-1));
            break;
        case 6:
            for(i=0;i<=n-1;i++)
                w[i]=1.0/exp(18*((n-1)/2.0-i)*((n-1)/2.0-i)/(n*n));
            break;
        default:
            return NULL;
            break;
    }
return w;
}
```

Arquivo **STFT-iSTFT.h**:

```
#ifndef STFT_H
#define STFT_H

#include "signal.h"
#include "matrices.h"


class STFT_Computer //implementa um analizador STFT e iSTFT
{
private: static const unsigned int RECTANGULAR_WINDOW=1;
private: static const unsigned int BARLETT_WINDOW=2;
private: static const unsigned int HANNING_WINDOW=3;
private: static const unsigned int HAMMING_WINDOW=4;
private: static const unsigned int BLACKMAN_WINDOW=5;
private: static const unsigned int GAUSSIAN_WINDOW=6;


private: double spectral_freq_limit; //o limite superior de frequência
private: double time_limit; //o limite superior de tempo
private: double time_resolution; //a duração de cada janela em segundos é a resolução temporal
private: double spectral_freq_resolution; //a resolução em frequência
private: unsigned int n_lines;   //as dimensões do array bidimensional
private: unsigned int n_columns;

private: string window_name; //uma string com o nome da window
private: unsigned int windowsize; //o comprimento em numero de amostras da janela
private: unsigned int samplingrate;
private: unsigned int original_samplingrate;
private: double window_duration;
private: double sample_duration;
private: double* window; //um vetor de doubles onde a janela (gaussiana, hanning, hamming...) é armazenada.
private: double* Kw;
private: unsigned int stepsize; //o passo de avanço da janela em numeros de amostras
private: unsigned int overlap_index; //o índice de overlap (0 = 0%, 1 = 50%, 2 = 75%...) (1 -
1/2^overlap_index)*100%

private: double nominalperiod; //armazena o período nominal de rotação da máquina
private: string experience_name; //armazena uma string com a identificação da experiência.

private: double NENBW; //normalized noise equivalent bandwidth
private: double ENBW; //equivalent noise bandwidth
private: double S1; // soma de todas as amostras da janela
private: double S2; // soma de todas as amostras da janela ao quadrado

private: void generate_window(int window_type,const int N);

public: void set_nominal_period(double nomperiod); //configura o período nominal de rotação da máquina
public: void set_original_samplingrate(unsigned int original); //configura a taxa de amostragem original do sinal
public: void set_experience_name(const string name);
public: double get_time_resolution(void) const;
public: double get_freq_resolution(void) const;
public: double get_time_limit(void) const;
public: double get_freq_limit(void) const;
public: unsigned int get_n_columns(void) const; //retorna o número de colunas que a STFT tem
public: unsigned int get_n_lines(void) const;   //retorna o número de linhas que a STFT tem
public: void set_overlap_index(unsigned int overlap); //configura o overlap index
public: unsigned int get_window_size(void) const; //retorna o tamanho da janela em amostras
public: unsigned int get_step_size(void) const; //retorna o tamanho do step em amostras
public: void setup_window(const string windowtype, unsigned int N,unsigned int overlap); //configura o tipo e
tamanho de janela que será usado
public: double* get_window(void) const; //retorna a window usada
public: double* get_Kw(void) const; //retorna o fator de correção
public: double get_ENBW(void) const; //retorna o effective noise bandwidth
public: Matrix Compute_STFT(electric_signal input); //calcula STFT do sinal de entrada, retorna uma matriz de
complexos
public: Matrix Compute_RMS_STFT(electric_signal input,unsigned int samples_per_turn); //calcula a STFT de um sinal
repetitivo e faz a média entre várias iterações.
public: Matrix Compute_Mean_STFT(electric_signal input,unsigned int samples_per_turn); //calcula a STFT de um sinal
repetitivo e faz a média entre várias iterações.
public: Matrix Compute_PWR_STFT(electric_signal input,unsigned int samples_per_turn); //calcula a STFT de um sinal
repetitivo e faz a média entre várias iterações.

public: electric_signal Compute_iSTFT(const Matrix input); //calcula a iSTFT da matriz complexa de entrada, retorna
```

```
um sinal temporal
public: electric_signal Get_Overlapped_Windows(void); //

public: STFT_Computer(); //construtor padrão
public: ~STFT_Computer(); //destrutor

};

#endif
```

Arquivo **CMS-iCMS.h**:

```
/*
 * CMS-iCMS.cc
 *
 *  Created on: Sep 15, 2014
 *      Author: rafael
 */

#ifndef CMS_ICMS_CC_
#define CMS_ICMS_CC_

#include <string.h>
#include "matrices.h"
using namespace std;

//implementa um analizador CMS e iCMS
class CMS_Computer
{

private: double spectral_freq_limit;
private: double cyclic_freq_limit;
private: double time_limit;
private: double spectral_freq_resolution; //a resolução em frequência
private: double cyclic_freq_resolution; //a resolução em frequência
private: unsigned int n_lines;   //as dimensões do array bidimensional
private: unsigned int n_columns;
private: string experience_name; //armazena uma string com a identificação da experiência.
private: string signal_name;

public: Matrix Compute_CMS(Matrix Spectrogram_In);
public: Matrix Compute_iCMS(Matrix CMS_In);
public: void set_experience_name(const string name);
public: string get_experience_name(void) const;
public: void set_signal_name(const string name);
public: string get_signal_name(void) const;
public: void set_time_limit(double lim);
public: void set_spectral_freq_limit(double lim);
public: double get_cyclic_freq_resolution(void) const;
public: double get_spectral_freq_resolution(void) const;
public: double get_cyclic_freq_limit(void) const;
public: double get_spectral_freq_limit(void) const;
public: unsigned int get_n_columns(void) const; //retorna o número de colunas que a STFT tem
public: unsigned int get_n_lines(void) const;   //retorna o número de linhas que a STFT tem
public: CMS_Computer();
public: ~CMS_Computer();

};

#endif /* CMS_ICMS_CC_ */
```

Arquivo **CMC-iCMC.h**:

```
/*
 * CMC-iCMC.h
 *
 *  Created on: Sep 15, 2014
 *      Author: rafael
 */

#ifndef CMC_ICMC_H_
#define CMC_ICMC_H_

#include <string.h>
#include <complex.h>
#include "matrices.h"
using namespace std;

class CMC_Computer
{
```

```
private: unsigned int n_lines;   //as dimensões do array bidimensional
private: unsigned int n_columns;
private: string experience_name; //armazena uma string com a identificação da experiência.
private: string signal_name;
private: double spectral_freq_limit;
private: double cyclic_freq_limit;
private: double spectral_freq_resolution; //a resolução em frequência
private: double cyclic_freq_resolution; //a resolução em frequência

public: Matrix Compute_CMC(Matrix CMS_In);
public: complex double* get_PSD(Matrix CMS_In) const;
public: Matrix Compute_iCMC(Matrix CMC_In);


public: void set_experience_name(const string name);
public: string get_experience_name(void) const;
public: void set_signal_name(const string name);
public: string get_signal_name(void) const;

public: double get_spectral_freq_resolution(void) const;
public: double get_spectral_freq_limit(void) const;
public: void set_spectral_freq_limit(double lim);

public: double get_cyclic_freq_resolution(void) const;
public: double get_cyclic_freq_limit(void) const;
public: void set_cyclic_freq_limit(double lim);

public: unsigned int get_n_columns(void) const; //retorna o número de colunas que a STFT tem
public: unsigned int get_n_lines(void) const;   //retorna o número de linhas que a STFT tem
public: CMC_Computer();
public: ~CMC_Computer();

};


#endif /* CMC_ICMC_H_ */
```

Arquivo **grpGL.h**:

```
#ifndef GRPGL_H
#define GRPGL_H

#include <string>
using namespace std;

class GLImg
{
private: string MainTitle;
private: string SubTitle;
private: string normalizedXscaleTitle;
private: string XscaleTitle;
private: string YscaleTitle;
private: string ZscaleTitle;
private: string Filename;
private: double XscaleStart;
private: double XscaleEnd;
private: double XScaleTicks;
private: signed int XScaleInterTicks;
private: double YScaleTicks;
private: signed int YScaleInterTicks;
private: double XscaleNormalizationFactor;
private: double YscaleStart;
private: double YscaleEnd;
private: double ZscaleStart;
private: double ZscaleEnd;
private: unsigned int Xsize;
private: unsigned int Ysize;
private: bool   draw_inverted;

public: GLImg(unsigned int x,unsigned int y); //construtor que já dá as dimensões do gráfico
public: void SetXrange(double x1,double x2); //configura o range do eixo X
public: void SetYrange(double y1,double y2); //configura o range do eixo Y
public: void SetXscaleticks(double ticks,signed int interticks);
public: void SetYscaleticks(double ticks,signed int interticks);
public: void SetMainTitle(string s);
public: void SetSubTitle(string s);
public: void set_inverted(bool inverted); //configura se a imagem será normal (false) ou o negativo (true)
public: void showGraph(void);
public: void SetXscaleTitle(string s);
public: void SetYscaleTitle(string s);
public: void SetZscaleTitle(string s);
public: void SetXNormalizationFactor(double x);
public: void SetNormalizedXscaleTitle(string s);
public: void PlotSignal(double** signal,unsigned long int num_columns,unsigned long int num_lines);
public: void PlotRealPartOfComplexSignal(double** signal,unsigned long int num_columns,unsigned long int num_lines);
public: void PlotModulusOfComplexSignal(double** signal,unsigned long int num_columns,unsigned long int num_lines);
public: void SaveToFile(string filename);
public: ~GLImg();
};
```

```
class GLPolar
{
private: string AngularScaleTitle;
private: string RadialScaleTitle;
private: string MainTitle;    //Título geral do gráfico
private: string SubTitle;     //Sub título, com info adicional
private: string normalizedAngularScaleTitle; //Legenda do eixo X normalizado
private: string Filename;
private: double AngularScaleEnd;     //Valor de fundo de escala do eixo Y
private: double AngularScaleNormalizationFactor; //Fator de normalização da escala em X
private: double RadialScaleEnd;     //valor de fundo de escala em Y
private: unsigned int Xsize;
private: unsigned int Ysize;
private: bool   draw_inverted; //se true, desenha com o fundo preto

public: GLPolar(unsigned int x,unsigned int y); //construtor que já dá as dimensões do gráfico
public: void SetMainTitle(string s);
public: void SetSubTitle(string s);
public: void SetAngularScaleTitle(string s);
public: void SetRadialScaleTitle(string s);
public: void SetAngularScaleNormalizationFactor(double f);
public: void SetNormalizedAngularScaleTitle(string s);
public: void PlotSignal(double* signal,unsigned long int num_samples,string c);
//public: void PlotSignal(signal* S,ColorRGB c);
public: void set_inverted(bool inverted);
public: void SaveToFile(string filename);
public: void showGraph(void);
public: ~GLPolar();
};


class GL2D
{
private: string MainTitle;    //Título geral do gráfico
private: string SubTitle;     //Sub título, com info adicional
private: string normalizedXscaleTitle; //Legenda do eixo X normalizado
private: string XscaleTitle;   //Legenda do eixo X sem normalização
private: string YscaleTitle;   //Leganda do eixo Y
private: string Filename;
private: double XscaleStart;   //Valor de início de escala do eixo X
private: double XscaleEnd;     //Valor de fundo de escala do eixo Y
private: double XscaleNormalizationFactor; //Fator de normalização da escala em X
private: double YscaleStart;   //Valor de início de escala em Y
private: double YscaleEnd;     //valor de fundo de escala em Y
private: unsigned int Xsize;
private: unsigned int Ysize;
private: bool   draw_inverted; //se true, desenha com o fundo preto

public: GL2D(unsigned int x,unsigned int y); //construtor sobrecarregado
public: void SetXrange(double x1,double x2); //configura o range do eixo X
public: void SetMainTitle(string s);
public: void SetSubTitle(string s);
public: void set_inverted(bool inverted); //configura se a imagem será normal (false) ou o negativo (true)
public: void showGraph(void);
public: void SetXscaleTitle(string s);
public: void SetYscaleTitle(string s);
public: void SetXNormalizationFactor(double x);
public: void SetNormalizedXscaleTitle(string s);
public: void PlotSignal(double* signal,unsigned long int num_samples,string c);
public: void SaveToFile(string filename);
public: ~GL2D(); //destrutor
};

class GL3D
{
private: string MainTitle;
private: string SubTitle;
private: string normalizedXscaleTitle;
private: string XscaleTitle;
private: string YscaleTitle;
private: string ZscaleTitle;
private: string Filename;
private: double XscaleStart;
private: double XscaleEnd;
private: double XscaleNormalizationFactor;
private: double YscaleStart;
private: double YscaleEnd;
private: double ZscaleStart;
private: double ZscaleEnd;
private: unsigned int Xsize;
private: unsigned int Ysize;
private: bool   draw_inverted;


public: GL3D(unsigned int x,unsigned int y); //construtor que já dá as dimensões do gráfico
public: void SetXrange(double x1,double x2); //configura o range do eixo X
public: void SetYrange(double y1,double y2); //configura o range do eixo Y
public: void SetMainTitle(string s);
public: void SetSubTitle(string s);
public: void set_inverted(bool inverted); //configura se a imagem será normal (false) ou o negativo (true)
public: void showGraph(void);
public: void SetXscaleTitle(string s);
public: void SetYscaleTitle(string s);
```

```
public: void SetZscaleTitle(string s);
public: void SetXNormalizationFactor(double x);
public: void SetNormalizedXscaleTitle(string s);
public: void PlotSignal(double** signal,unsigned long int num_columns,unsigned long int num_lines);
public: void SaveToFile(string filename);
public: ~GL3D();
};

#endif
```

Arquivo **matrices.h**:

```
#ifndef MATRICES_H
#define MATRICES_H

#include <complex.h>
#include <string>
using namespace std;

class Matrix
{

/* private data */
private: unsigned int Ncolumns;
private: unsigned int Nrows;
private: double complex** data;

/* Constructor(s)/destructor. */
public: Matrix(); /* Uninitialized matrix. */
public: Matrix(unsigned int numberOfColumns,unsigned int numberOfRows);  /*Initialize the matrix.*/
public: ~Matrix();

/* Data insertion and removal methods. */
public: void SetRow(const double* rowdata,unsigned int rownum);
public: void SetRow(const double complex* rowdata,unsigned int rownum);
public: void SetColumn(const double* columndata,unsigned int columnnum);
public: void SetColumn(const double complex* columndata,unsigned int columnnum);
public: void GetRealRow(double* rowdata,unsigned int rownum) const;
public: void GetImagRow(double* rowdata,unsigned int rownum) const;
public: void GetRow(double complex* rowdata,unsigned int rownum) const;
public: void GetRealColumn(double* columndata,unsigned int columnnum) const;
public: void GetImagColumn(double* columndata,unsigned int columnnum) const;
public: void GetColumn(double complex* columndata,unsigned int columnnum) const;

/* Interface de dados */
public: double complex** GetData() const;
//public: double** GetRealData() const;
//public: double** GetImagData() const;
public: double complex GetElement(unsigned int row, unsigned int column) const;
public: void SetElement(unsigned int row, unsigned int column,double complex x);
public: void SetElement(unsigned int row, unsigned int column,double x);
public: unsigned int GetNumRows(void) const;
public: unsigned int GetNumColumns(void) const;
public: double complex* operator[](unsigned int row);

/* Interface de entrada e saída em disco e terminal */
public: void SaveToFile(string filename) const;
public: void LoadFromFile(string filename);
public: void PrintOut(void) const;


/* Arithmatic operators. */
public: Matrix operator=(Matrix source);
public: Matrix(const Matrix& source);
/*public: Matrix operator= ( double );
public: Matrix operator+ ( Matrix );
public: Matrix operator+ ( double );
public: Matrix operator- ( Matrix );
public: Matrix operator- ( double );
public: Matrix operator* ( Matrix );
public: Matrix operator* ( double );
public: Matrix operator/ ( Matrix );
public: Matrix operator/ ( double );*/

};


#endif
```

Arquivo **signal.h**:

```
#ifndef ELECTRIC_SIGNAL_H
#define ELECTRIC_SIGNAL_H
```

```
#include <string>
using namespace std;

class electric_signal
{
private: char* name;                    //o nome do sinal
private: double* samples;                //um ponteiro para o início das amostras
private: unsigned long int num_samples;  //número de amostas
private: unsigned int samplingrate;      //a taxa de amostragem
private: double duration;                //a duração temporal do sinal
private: double sample_duration;         //a duração de cada amostra
private: double minvalue;                //o valor mínimo encontrado
private: double maxvalue;                //o valor máximo encontrado
private: char* date;                    //a data e hora que o sinal foi adquirido
private: char* comments;                //comentários sobre o sinal
private: double DCvalue;                 //o seu valor DC (valor médio)

public: static void SaveSignalSetToDisk(char* filename,electric_signal* signalpointer,unsigned int numsignals);
public: static void SaveSignalSetToDisk(char* filename,electric_signal** signalpointer,unsigned int numsignals);

public: static void LoadSignalSetFromDisk(char* filename,electric_signal* signalpointer,unsigned int numsignals);

public: electric_signal(); //construtor padrão
public: electric_signal(const electric_signal& source); //copy constructor
public: electric_signal& operator=(const electric_signal& source); //assignment constructor
public: double& operator[](unsigned int i); //acessa uma amostra individual do sinal
public: ~electric_signal(); //destrutor;
public: electric_signal(unsigned long int nsamples,unsigned int nsamplingrate,const char* signalname); //construtor
sobrecarregado (cria um sinal vazio)
public: electric_signal(unsigned long int nsamples,double* source,unsigned int nsamplingrate,const char*
signalname); //construtor sobrecarregado (cria um sinal preenchido com uma cópia de source)
public: electric_signal(const char* filename, unsigned int n_channel); //construtor sobrecarregado (cria um sinal a
partir de um arquivo WAV e numero do canal)
public: double* get_samples_pointer(void);
public: void set_name(const char* newname);
public: const char* get_name(void) const;
public: void set_date(const char* newdate);
public: const char* get_date(void) const;
public: void set_comments(const char* newcomments);
public: const char* get_comments(void) const;
public: unsigned long int get_num_samples(void) const;
public: unsigned int get_samplingrate(void) const;
public: double get_duration(void);
public: double get_sample_duration(void);
public: double get_max(void);
public: double get_min(void);
public: double get_DC(void);
public: void removeDC(void);
public: void add_DC(double value);
public: void apply_gain(double gain);
public: double maximize(void);
public: void SmithTrigger(double Hhigh,double Hlow);
public: void monoestable(bool change,unsigned int timing);
public: void DivideFrequency(unsigned int divfactor);
public: double MeasureDigitalFrequency(void);
public: double MeasureSinusoidalFrequency(void);
public: double MeasureAveragePower(void) const;
public: unsigned int CountCycles(void);
public: unsigned long int* CyclesStartPoints(void);
public: void SaveToDisk(string filename);
};

/*********************************************************************************************/

//ATENÇÃO: estas classes FIR e IIR não estão prontas!
//Faltam construtores, o destrutor, e os copy constructors e checar tudo, testar tudo.

class IIRFilter
{
private: unsigned int order;
private: unsigned int numerator_order;
private: unsigned int denominator_order;
private: double* shifted_inputs;
private: double* shifted_outputs;
private: double* denominator;
private: double* numerator;
private: double input;
private: double output;
private: double constant;

private: void init(void);
public: unsigned int get_order(void);
public: void reset_state(void);
public: void set_coeficients(double* n,unsigned int norder,double* d,unsigned int dorder,double cte);
private: void feedforward(void);
public: void apply(electric_signal* inputsignal);
public: void printout(void);
IIRFilter();
~IIRFilter();
};

/*********************************************************************************************/

class FIRFilter
{
```

```
private: unsigned int order;
private: unsigned int numerator_order;
private: double* shifted_inputs;
private: double* numerator;
private: double input;
private: double output;
private: double constant;

private: void init(void);
public: unsigned int get_order(void);
public: void reset_state(void);
public: void set_coeficients(double* n,unsigned int norder,double cte);
private: void feedforward(void);
public: void apply(electric_signal* inputsignal);
public: void printout(void);
FIRFilter();
~FIRFilter();
};



#endif
```

Arquivo **wavfile.h**:

```
#ifndef WAVFILE_H
#define WAVFILE_H

#include <string>
using namespace std;

class WavFile
{

public: enum filetype {INVALID,WAV,W64,RF64,BIN};

private:
string filename; //o nome do arquivo no sistema de arquivos
string basename; //o nome do arquivo mas sem a extensão .wav ou .w64 ou .rf64
string extention; //a extensão do arquivo sem o ponto
filetype file_type; //o tipo do arquivo
unsigned int num_channels; //número de canais do arquivo wav
unsigned int samplingrate; //a taxa de amostragem em samples/segundo.
unsigned long int num_samples; //o número de amostras do arquivo, por canal.
double time_duration; //duração do arquivo em segundos.
unsigned int bits_per_sample; //o número de bits por amostra que o arquivo .Wav usa para armazenar dados
unsigned int bits_per_second; //a taxa de bits por segundo que o arquivo inteiro necessita para ser transmitido
unsigned long int filesize; //o tamanho em bytes que o arquivo ocupa no filesystem

string metadata_title;
string metadata_copyright;
string metadata_software;
string metadata_artist;
string metadata_comments;
string metadata_date;
string metadata_album;
string metadata_license;
string metadata_tracknumber;
string metadata_genre;

string* channel_name;


private:
void init(void); //inicializador default;

public:
WavFile(); //construtor padrão
WavFile(const char* file_name); //construtor sobrecarregado

const char* get_basename(void) const; //retorna o nome sem o .wav
void set_basename(const char* name); //configura o nome base para um Wav novo. Seu nome completo será com a adição
de .rf64
const char* get_filename(void) const; //retorna o nome completo do arquivo
void set_type(filetype type); //configura o tipo de arquivo.
unsigned int get_num_channels(void) const; //retorna o número de canais que este arquivo tem
void set_num_channels(unsigned int num); //configura o número de canais para um arquivo novo
unsigned int get_samplingrate(void) const; //retorna a taxa de amostragem
void set_samplingrate(unsigned int rate); //configura a taxa de amostragem para um arquivo novo
unsigned long int get_num_samples(void) const;
unsigned int get_bits_per_sample(void) const;
unsigned int get_bits_per_second(void) const;
unsigned long int get_filesize(void) const;
double get_duration(void) const;
double get_sample_duration(void) const;
double* load_samples(void) const;  //retorna o ponteiro de um array de doubles onde estão todas as amostas do
arquivo, sendo que estão os canais intercalados
                                   //o ponteiro retornado deve ser obrigatoriamente desalocado posteriormente pelo
usuário
```

```
void save_samples(double* samplespointer,unsigned long int n_samples) const; //grava em disco um arquivo com o
formato já definido anteriormente os dados do buffer apontado

const char* get_metadata_title(void) const; //
const char* get_metadata_copyright(void) const;
const char* get_metadata_software(void) const;
const char* get_metadata_artist(void) const;
const char* get_metadata_comments(void) const; //retorna uma string com todos os comentários do arquivo .WAV
const char* get_metadata_date(void) const; //retorna uma string com a data do arquivo .WAV
const char* get_metadata_album(void) const;
const char* get_metadata_license(void) const;
const char* get_metadata_tracknumber(void) const; //
const char* get_metadata_genre(void) const; //

void set_metadata_title(const char* title); //
void set_metadata_copyright(const char* title);
void set_metadata_software(const char* title);
void set_metadata_artist(const char* title);
void set_metadata_comments(const char* title);
void set_metadata_date(const char* title);
void set_metadata_album(const char* title);
void set_metadata_license(const char* title);
void set_metadata_tracknumber(const char* title);
void set_metadata_genre(const char* title);

const char* get_channel_name(unsigned int channel_num) const;
double get_channel_range(unsigned int channel_num) const;

~WavFile(); //destrutor
};

#endif
```

Arquivo **window.h**:

```
#ifndef WINDOW_H
#define WINDOW_H

#define RECTANGULAR_WINDOW 1
#define BARLETT_WINDOW 2
#define HANNING_WINDOW 3
#define HAMMING_WINDOW 4
#define BLACKMAN_WINDOW 5
#define GAUSSIAN_WINDOW 6

double* generate_window(int window,const int N);

#endif
```

APÊNDICE F – ESQUEMAS ELÉTRICOS E *LAYOUTS*

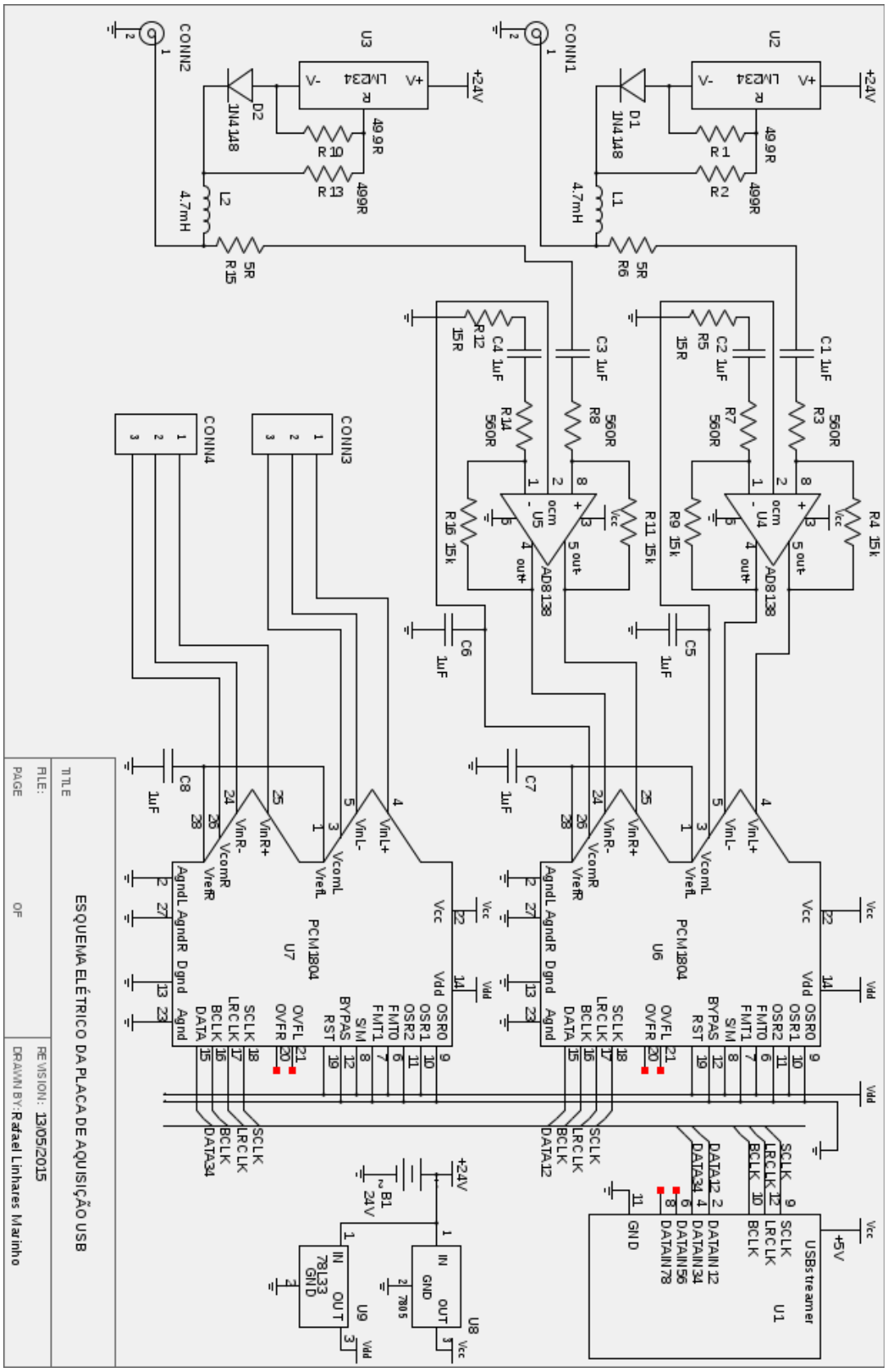


Figura 76 - Esquema elétrico da placa de aquisição de sinais com interface USB.

Fonte: Elaboração própria com *freeware* gschem 1.8.2 Linux.

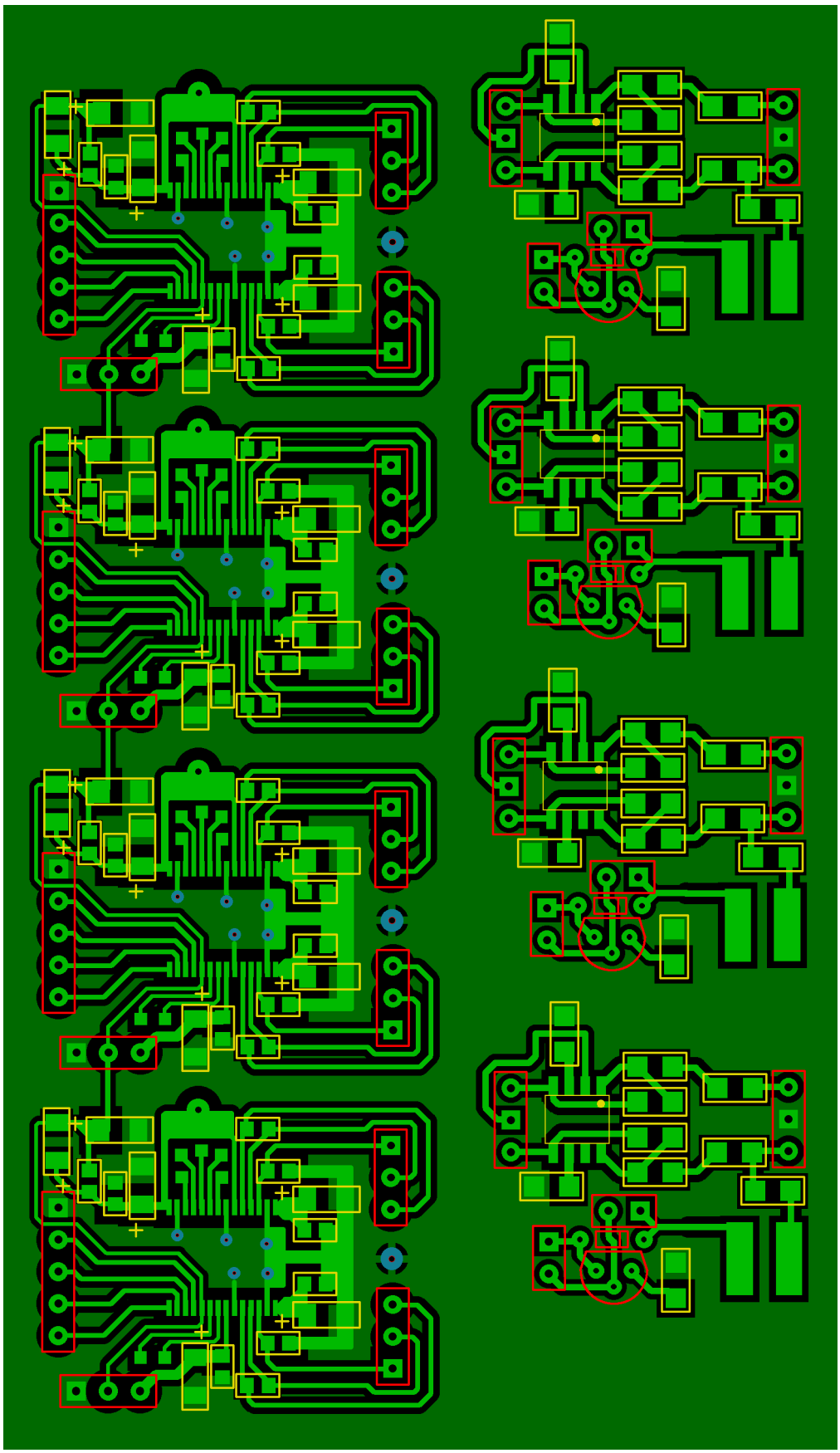


Figura 77 - *Lay-out* da placa de circuito impresso com componentes SMD.

Fonte: Elaboração própria.

ANEXO A – *DATASHEETS* MAIS IMPORTANTES

9700A Accelerometer (EK-43799I)

Attribute	Metric	Imperial
Performance		
Sensitivity (±10%)[1]	1.02 mV/(m/s²)	10 mV/g
Measurement range	±4905 m/s²	±500 g
Frequency range (±10%)[2]	3.4…18000 Hz	204…1080000 cpm
Frequency range (±3 dB)	1.6…30000 Hz	96…1800000 cpm
Resonant frequency, typical	85 kHz	5100 kcpm
Broadband resolution (1…10000 Hz), typical	1176 µm/s²	1.2 mg rms
Non-linearity[3]	±1%	±1%
Transverse sensitivity	≤ 5%	≤ 5%
Environmental		
Overload limit (shock)	±98100 m/s² pk	±10000 g pk
Temperature range	-54…121 °C	-65…+50 °F
Electrical		
Settling time (within 1% of bias)	≤ 3 .0 s	≤ 3.0 s
Discharge time constant	≥ 0.1 s	≥ 0.1 s
Excitation voltage	18…28V DC	18…28V DC
Constant current excitation	2…20 mA	2…20 mA
Output impedance	< 100 Ω	< 100 Ω
Output bias voltage	8…12V DC	8…12V DC
Spectral noise (10 Hz), typical	980 (µm/s²)/√Hz	100 µg/√Hz
Spectral noise (100 Hz), typical	294 (µm/s²)/√Hz	30 µg/√Hz
Spectral noise (1 kHz), typical	98.1 (µm/s²)/√Hz	10 µg/√Hz
Physical		
Size (hex x height)	9.53 x 16.8 mm	3/8 x 0.66 in.
Weight, approx.	2.8 g	0.1 oz
Mounting thread[4]	5-40 male	5-40 male
Mounting torque	203…226 N•cm	18…20 lb•ft
Sensing element	Ceramic	Ceramic
Sensing geometry	Shear	Shear
Housing material	Titanium	Titanium
Sealing	Welded hermetic	Welded hermetic
Electrical connector	5-44 Coaxial jack	5-44 Coaxial jack
Electrical connection position	Top	Top

(1) Conversion factor 1g = 9.81 m/s².

(2) Frequency response with adhesive base.

(3) Mounted resonance (nominal) without magnet.

(4) Zero-based, least-squares, straight line method.

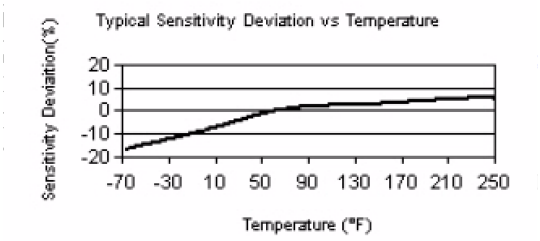
Requires accelerometer cable coaxial to BNC (cat. no. EK-35579).

All specifications are at room temperature unless otherwise specified.

See Declaration of Conformance at http://www.rockwellautomation.com/products/certification

Supplied accessories include the following:

- ICS-3 3 pt calibration 10 Hz, 100 Hz, 1 kHz



National Semiconductor

March 2005

LM134/LM234/LM334
3-Terminal Adjustable Current Sources

General Description

The LM134/LM234/LM334 are 3-terminal adjustable current sources featuring 10,000:1 range in operating current, excellent current regulation and a wide dynamic voltage range of 1V to 40V. Current is established with one external resistor and no other parts are required. Initial current accuracy is ±3%. The LM134/LM234/LM334 are true floating current sources with no separate power supply connections. In addition, reverse applied voltages of up to 20V will draw only a few dozen microamperes of current, allowing the devices to act as both a rectifier and current source in AC applications.

The sense voltage used to establish operating current in the LM134 is 64mV at 25˚C and is directly proportional to absolute temperature (˚K). The simplest one external resistor connection, then, generates a current with ≈+0.33%/˚C temperature dependence. Zero drift operation can be obtained by adding one extra resistor and a diode.

Applications for the current sources include bias networks, surge protection, low power reference, ramp generation, LED driver, and temperature sensing. The LM234-3 and LM234-6 are specified as true temperature sensors with guaranteed initial accuracy of ±3˚C and ±6˚C, respectively. These devices are ideal in remote sense applications because series resistance in long wire runs does not affect accuracy. In addition, only 2 wires are required.

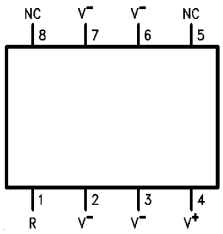
The LM134 is guaranteed over a temperature range of −55˚C to +125˚C, the LM234 from −25˚C to +100˚C and the LM334 from 0˚C to +70˚C. These devices are available in TO-46 hermetic, TO-92 and SO-8 plastic packages.

Features

- Operates from 1V to 40V
- 0.02%/V current regulation
- Programmable from 1µA to 10mA
- True 2-terminal operation
- Available as fully specified temperature sensor
- ±3% initial accuracy

Connection Diagrams

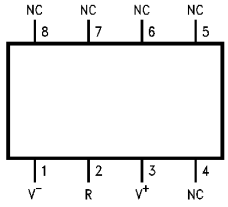
SO-8
Surface Mount Package



00569724

Order Number LM334M or LM334MX
See NS Package Number M08A

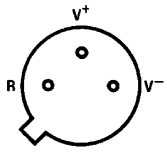
SO-8 Alternative Pinout
Surface Mount Package



00569725

Order Number LM334SM or LM334SMX
See NS Package Number M08A

TO-46
Metal Can Package

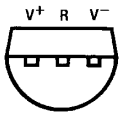


00569712

V⁻ Pin is electrically connected to case.

Bottom View
Order Number LM134H,
LM234H or LM334H
See NS Package
Number H03H

TO-92 Plastic Package



00569710

Bottom View
Order Number LM334Z, LM234Z-3 or LM234Z-6
See NS Package Number Z03A

Low Distortion Differential ADC Driver

AD8138

FEATURES

Easy to use, single-ended-to-differential conversion
Adjustable output common-mode voltage
Externally adjustable gain
Low harmonic distortion
−94 dBc SFDR @ 5 MHz
−85 dBc SFDR @ 20 MHz
−3 dB bandwidth of 320 MHz, G = +1
Fast settling to 0.01% of 16 ns
Slew rate 1150 V/µs
Fast overdrive recovery of 4 ns
Low input voltage noise of 5 nV/√Hz
1 mV typical offset voltage
Wide supply range +3 V to ±5 V
Low power 90 mW on 5 V
0.1 dB gain flatness to 40 MHz
Available in 8-Lead SOIC and MSOP packages

APPLICATIONS

ADC drivers
Single-ended-to-differential converters
IF and baseband gain blocks
Differential buffers
Line drivers

PIN CONFIGURATION

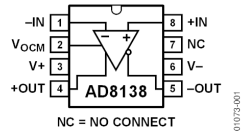


Figure 1.

TYPICAL APPLICATION CIRCUIT

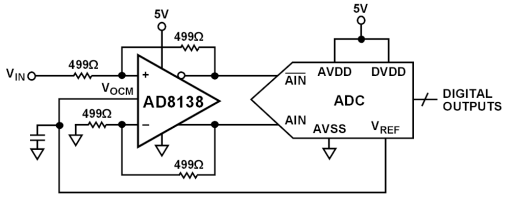


Figure 2.

GENERAL DESCRIPTION

The AD8138 is a major advancement over op amps for differential signal processing. The AD8138 can be used as a single-ended-to-differential amplifier or as a differential-to-differential amplifier. The AD8138 is as easy to use as an op amp and greatly simplifies differential signal amplification and driving. Manufactured on ADI's proprietary XFCB bipolar process, the AD8138 has a −3 dB bandwidth of 320 MHz and delivers a differential signal with the lowest harmonic distortion available in a differential amplifier. The AD8138 has a unique internal feedback feature that provides balanced output gain and phase matching, suppressing even order harmonics. The internal feed-back circuit also minimizes any gain error that would be associated with the mismatches in the external gain setting resistors.

The AD8138's differential output helps balance the input to differential ADCs, maximizing the performance of the ADC.

The AD8138 eliminates the need for a transformer with high performance ADCs, preserving the low frequency and dc information. The common-mode level of the differential output is adjustable by a voltage on the V_{OCM} pin, easily level-shifting the input signals for driving single-supply ADCs. Fast overload recovery preserves sampling accuracy.

The AD8138 distortion performance makes it an ideal ADC driver for communication systems, with distortion performance good enough to drive state-of-the-art 10-bit to 16-bit converters at high frequencies. The AD8138's high bandwidth and IP3 also make it appropriate for use as a gain block in IF and baseband signal chains. The AD8138 offset and dynamic performance makes it well suited for a wide variety of signal processing and data acquisition applications.

The AD8138 is available in both SOIC and MSOP packages for operation over −40°C to +85°C temperatures.

Rev. F

One Technology Way, P.O. Box 9106, Norwood, MA 02062-9106, U.S.A.
Tel: 781.329.4700 www.analog.com
Fax: 781.461.3113

FULL DIFFERENTIAL ANALOG INPUT 24-BIT, 192-kHz STEREO A/D CONVERTER

Check for Samples: PCM1804-Q1

FEATURES

- Qualified for Automotive Applications
- AEC-Q100 Test Guidance With the Following Results:
  - Device Temperature Grade 3: –40°C to 85°C Ambient Operating Temperature Range
  - Device HBM ESD Classification Level H2
  - Device CDM ESD Classification Level C3B
- 24-Bit Delta-Sigma Stereo A/D Converter
- High Performance:
  - Dynamic Range: 112 dB (Typical)
  - SNR: 111 dB (Typical)
  - THD+N: –102 dB (Typical)
- High-Performance Linear Phase Antialias Digital Filter:
  - Pass-Band Ripple: ±0.005 dB
  - Stop-Band Attenuation: –100 dB
- Fully Differential Analog Input: ±2.5 V
- Audio Interface: Master- or Slave-Mode Selectable
- Data Formats: Left-Justified, I^2S, Standard 24-Bit, and DSD
- Function:
  - Peak Detection
  - High-Pass Filter (HPF): –3 dB at 1 Hz, f_S = 48 kHz
- Sampling Rate up to 192 kHz
- System Clock: 128 f_S, 256 f_S, 384 f_S, 512 f_S, or 768 f_S
- Dual Power Supplies:
  - 5 V for Analog
  - 3.3 V for Digital
- Power Dissipation: 225 mW
- Small 28-Pin SSOP
- DSD Output: 1 Bit, 64 f_S

APPLICATIONS

- AV Amplifier
- MD Player
- Digital VTR
- Digital Mixer
- Digital Recorder

DESCRIPTION

The PCM1804-Q1 device is a high-performance, single-chip stereo A/D converter with fully differential analog voltage input which uses a precision delta-sigma modulator and includes a linear-phase antialias digital filter and high-pass filter (HPF) that removes DC offset from the input signal. The PCM1804-Q1 device is suitable for a wide variety of mid- to high-grade consumer and professional applications, where excellent performance and 5-V analog supply and 3.3-V digital power-supply operation are required. The PCM1804-Q1 device can achieve both PCM audio and DSD format due to the precision delta-sigma modulator. The PCM1804-Q1 device is fabricated using an advanced CMOS process and is available in a small 28-pin SSOP package.

minidsp

USBStreamer

Features
- 10 x in + 10 x out multi-channel USB audio asynchronous interface
- Native support up to 24bits/192kHz
- Easy integration for OEM/DIY

Technical
- USB Audio Class 2.0 interface
- 2ch in + out over Toslink (optical)
- 8ch in + out over I2S
- Asynchronous USB synchronization
- 24bit Resolution I2S lines
- 44.1/48/88.2/96/176.4/192KHz
- Optical Receiver/Transmitter
- Buffered I2S interface on header

OS compatibility
- WinXP/Vista/7 Thesycon driver
- Mac OS X driverless

Power
- USB Bus powered
- External DC supply header

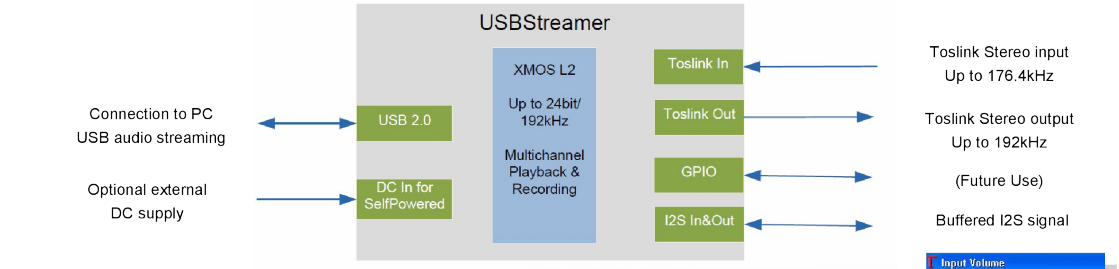
Applications
- Custom USB Audio interface
- USB-Toslink-I2S interface
- High resolution Playback Recording
- Multi-channel surround preamplifier

USBStreamer is a miniature multi-channel USB to Toslink & I2S interface with native support of audio files with format up to 24bit/192kHz. Packaged on a tiny PCB of only 2.4" by 1.4" (40x62mm), this interface is the perfect fit for OEM integration or as an element of a customized high performance A/V product.

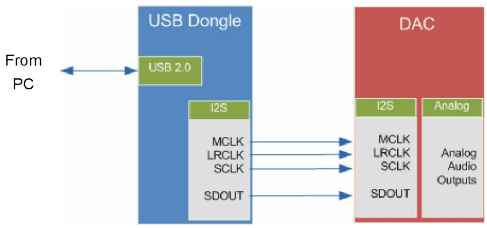
The USBStreamer utilizes high speed USB2.0 along with the Audio Class 2.0 implementation to simultaneously provide up to 10channels of playback along with 10channel of recording. Out of the 10channels, 2ch are provided on the toslink interface with the remaining provided on the I2S expansion connector. The dongle is up & running in a matter of seconds with compatible drivers for both Mac/Windows platforms. An easy and flexible system integration that follows the footsteps of the miniDSP product line. From a high end stereo USB interface to a multi-channel USB interface, the USBStreamer is an innovative yet cost effective USB 2.0 audio interface for a wide range of custom A/V products.

miniDSP, innovative Digital Audio Products

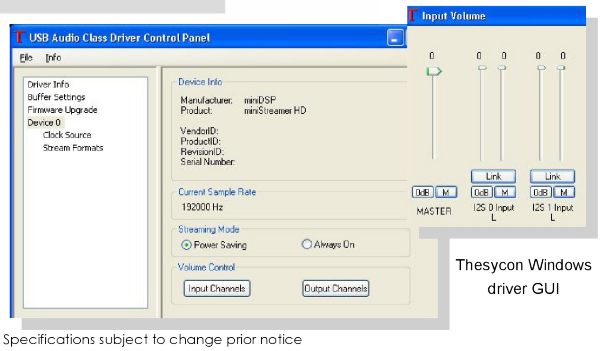
miniSTREAMER Diagram



Sample Application circuit



USBStreamer in I2S master, Audio playback



Thesycon Windows driver GUI