UNIVERSIDADE FEDERAL DO PARÁ

INSTITUTO DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

TESE DE DOUTORADO

Desenvolvimento de Framework para Análise e Simulação Dinâmica de Sistemas Elétricos de Potência

José Adolfo da Silva Sena

TD: 01/2013

UFPA / ITEC / PPGEE

Campus Universitário do Guamá

Belém-Pará-Brasil

2013

UNIVERSIDADE FEDERAL DO PARÁ

INSTITUTO DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

José Adolfo da Silva Sena

Desenvolvimento de Framework para Análise e Simulação Dinâmica de Sistemas Elétricos de Potência

UFPA / ITEC / PPGEE

Campus Universitário do Guamá

Belém-Pará-Brasil

2013

UNIVERSIDADE FEDERAL DO PARÁ

INSTITUTO DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

José Adolfo da Silva Sena

Desenvolvimento de Framework para Análise e Simulação Dinâmica de Sistemas Elétricos de Potência

Tese de Doutorado submetida à Banca Examinadora do Programa de Pós-Graduação em Engenharia Elétrica (PPGEE) da Universidade Federal do Pará (UFPA) para obtenção do Grau de Doutor em Engenharia Elétrica.

Orientador: Prof. Dr. Walter Barra Jr.

Área de Concentração: Sistemas de Energia Elétrica

UFPA / ITEC / PPGEE

Campus Universitário do Guamá

Belém-Pará-Brasil

2013

AGRADECIMENTOS

Em primeiro lugar agradeço ao Pai Celestial que me concedeu todas as bênçãos necessárias para que eu atingisse mais essa vitória. Agradeço à minha mãe por ter dado sua vida para educar seus filhos, ela sabia, como a grande maioria das mães humildes deste país, que a educação liberta e nos tira da miséria. Agradeço à minha esposa pelo seu esforço em me fazer levantar e retornar ao caminho para chegar ao meu destino. Agradeço ao Professor Walter Barra Jr. que foi muito mais que um orientador, ele é um grande amigo. Agradeço, sobretudo, ao povo brasileiro mais humilde que pagou meus estudos com seu suor, suas lágrimas e seu sangue na esperança por dias melhores.

Abstract

This thesis presents a flexible Object Oriented (OO) methodology for the design and implementation of simulation software system used to perform dynamic studies of large electrical power systems. The proposed OO methodology aims at make easier the development, upgrade and maintenance of complex software systems. The user's requirements are mapped to set base classes which are atomic classes to perform the modeling of the dynamic devices such as electrical generators. In order to evaluate the methodology, a simulation software system was developed using the OO modeling. In order to evaluate the Framework performance, two study cases were carried out. The first one apply the Framework on the modeling and simulation of generation units at Tucuruí Power Plant. The simulation results were comparated to data measurements from field tests and shown the good performance of the Framework on reproducing electromechanical phenomena of this large Power Plant. In the second study, by its turn, the Framework was applied to the modeling of a Photovoltaic (PV) generation system, along with its Maximum Power Tracking (*MPPT*). The *MPPT* control was implemented using digital techniques. The simulation results show the good performance of the Framework on modeling the current as well as *MPPT* control of PV generations systems

Resumo

Esta tese apresenta uma metodologia flexível orientada a objetos (OO) para a aplicação no projeto e implementação de sistemas de software utilizados na realização de estudos dinâmicos de sistemas elétricos de grande porte. A metodologia OO proposta objetiva tornar mais simples o desenvolvimento, a atualização e a manutenção de complexos sistemas de software para estudos de transitórios eletromecânicos em sistemas elétricos de potência. Os requisitos de usuário são mapeados para um conjunto de classes básicas, as quais são usadas para efetuar a modelagem de dispositivos dinâmicos tais como geradores elétricos. Para avaliação da metodologia foram realizados dois estudos de casos. No primeiro estudo caso o Framework foi aplicado na simulação das unidades geradoras da Usina Hidrelétrica de Tucuruí. Os resultados da simulação foram comparados com medições obtidos em ensaios no campo e mostrou a boa performance do Framework na reprodução dos fenômenos eletromecânicos desta usina de grande porte. No segundo estudo de caso, por outro lado, o Framework foi aplicado na modelagem de um sistema de geração fotovoltaico (PV) com seu sistema de Rastreamento da Potência Máxima (*MPPT*). O controle *MPPT* foi implementado usando técnicas digitais. Os resultados das simulações demonstram a performance do Framework na modelagem do sistema de controle de corrente, assim como no controle *MPPT*, dos sistemas de geração PV.

Lista de Figuras

FIGURA 2-1 - PACOTES QUE COMPÕE O FRAMEWORK 21

FIGURA 2-2 - CLASSES DO PACOTE "MATRIZES E VETORES" 24

FIGURA 2-3 - RELAÇÃO ENTRE CLASSE REFERENCIA (NOMEADA) E CLASSE TEMPORÁRIA (NÃO NOMEADA) 26

FIGURA 2-4 - NA CÓPIA PRESUMIDA SÃO COPIADOS OS PONTEIROS, NÃO OS CONTEÚDOS DE ÁREAS DE MEMÓRIA. EM (A) ANTES DA FUNÇÃO RETORNAR. EM (B) APÓS O RETORNO DO OPERADOR DE ATRIBUIÇÃO. EM (C) A MEMÓRIA É REPASSADA AO OBJETO DECLARADO. EM (D) O OBJETO TEMPORÁRIO É DESTRUÍDO 27

FIGURA 2-5 - DECLARAÇÃO SIMPLIFICADA DAS CLASSES PARA VETORES NÃO ESPARSOS 29

FIGURA 2-6 - CÓDIGOS FONTES DOS CONSTRUTORES DE CÓPIA DE CONTEÚDO E PRESUMIDA 30

FIGURA 2-7 - SOBRECARGA DO OPERADOR DE ATRIBUIÇÃO 31

FIGURA 2-8 - CÓDIGOS FONTES DAS FUNÇÕES DE SOBRECARGA DO OPERADOR "+" 32

FIGURA 2-9 - DECLARAÇÃO DA CLASSE TMPTVECTOR 33

FIGURA 2-10 - CÓDIGO FONTE DOS CONSTRUTORES DE CÓPIA 34

FIGURA 2-11 - EXEMPLO DE USO 34

FIGURA 2-12 - EXEMPLO DE PROGRAMA QUE UTILIZA MATRIZES 35

FIGURA 2-13 - REPRESENTAÇÃO EM UML DOS GABARITOS DE CLASSES DE VETORES ESPARSOS 36

FIGURA 2-14 - OBTENÇÃO DE UM SUBVETOR 37

FIGURA 2-15 - MÉTODOS USADOS PARA INICIALIZAR O VETOR ESPARSO 39

FIGURA 2-16 - INICIALIZAÇÃO RÁPIDA DO CONTEÚDO DE UM VETOR ESPARSO 39

FIGURA 2-17 - DECLARAÇÃO DOS OPERADORES DE SUBSCRIÇÃO DE ELEMENTOS 41

FIGURA 2-18 - INICIALIZAÇÃO DE UMA MATRIZ ESPARSA 42

FIGURA 3-1 - DIAGRAMA DE CLASSES QUE PRESENTÃO A REDE ELÉTRICA 45

FIGURA 3-2 - EXEMPLO DE CADASTRAMENTO DE UMA ÁREA E DE UM MODELO DE CARGA 54

FIGURA 3-3 - EXEMPLO DE REDE ELÉTRICA 55

FIGURA 3-4 - CADASTRAMENTO DAS BARRAS DO SISTEMA APRESENTADO NA FIGURA 3-3 55

FIGURA 3-5 - CADASTRO DE UMA LINHA DE TRANSMISSÃO DO SISTEMA NA FIGURA 3-3 56

FIGURA 3-6 - CÁLCULO DO FLUXO DE CARGA 56

FIGURA 4-1 - EXEMPLO DE DIAGRAMA EM BLOCOS 58

FIGURA 4-2 - DIAGRAMA DE CLASSES QUE REPRESENTAM SISTEMAS DINÂMICOS 59

FIGURA 4-3 - DECLARAÇÃO DA CLASSE TFIRSTORDER 64

FIGURA 4-4 - IMPLEMENTAÇÃO DO MÉTODO QUE AVALIA A EXPRESSÃO (4.5) 65
FIGURA 4-5 - IMPLEMENTAÇÃO DO MÉTODO QUE AVALIA A EXPRESSÃO (4.6) 65
FIGURA 4-6 - IMPLEMENTAÇÃO DO MÉTODO DE INICIALIZAÇÃO DAS VARIÁVEIS DO BLOCO 66
FIGURA 4-7 - DECLARAÇÃO DA CLASSE BASE PARA OS MÉTODOS DE INTEGRAÇÃO NUMÉRICA 69
FIGURA 4-8 - IMPLEMENTAÇÃO DO INTEGRADOR NUMÉRICO PARA O MÉTODO DE RUNGE-KUTTA 73
FIGURA 4-9 - DEFINIÇÃO DA CLASSE TTRAPEZOIDAL 77
FIGURA 4-10 - DIAGRAMA EM BLOCOS DE UM SISTEMA DINÂMICO 78
FIGURA 4-11 - EXEMPLO DE PROGRAMA DE SIMULAÇÃO 79
FIGURA 5-1 - MODELO BÁSICO 81
FIGURA 5-2 - MODELO DE UM SISTEMA MULTIMÁQUINAS 82
FIGURA 5-3 - REPRESENTAÇÃO DE UM GERADOR ATRAVÉS DE SEU NÓ INTERNO 82
FIGURA 5-4 - MODELO COMPUTACIONAL PARA A REDE ELÉTRICA 87
FIGURA 5-5 - MÉTODO QUE CALCULA AS SAÍDAS DO BLOCO QUE REPRESENTA A REDE ELÉTRICA 88
FIGURA 5-6 - REPRESENTAÇÃO COMPUTACIONAL DE UMA MÁQUINA SÍNCRONA 92
FIGURA 5-7 - MODELO CLÁSSICO DA MÁQUINA EM REGIME PERMANENTE 92
FIGURA 5-8 - CLASSE FUNDAMENTAL PARA GERADORES 94
FIGURA 5-9 - DECLARAÇÃO DA CLASSE TSYNCGEN 95
FIGURA 5-10 - CÁLCULO DAS DERIVADAS DAS VARIÁVEIS DE ESTADO DA MÁQUINA SÍNCRONA 97
FIGURA 5-11 - MÉTODO PARA O CÁLCULO DAS SAÍDAS 98
FIGURA 5-12 - CÁLCULO DA CONTRIBUIÇÃO DA MÁQUINA SÍNCRONA PARA A FORMAÇÃO DA MATRIZ JACOBIANA 101
FIGURA 5-13 - CÁLCULO DAS CONDIÇÕES INICIAIS DA MÁQUINA 103
FIGURA 6-1 - MÁQUINAS DA PRIMEIRA ETAPA DA CONSTRUÇÃO DA UHE DE TUCURUÍ 104
FIGURA 6-2 - MÁQUINAS DA SEGUNDA ETAPA DA CONSTRUÇÃO DA UHE DE TUCURUÍ 104
FIGURA 6-3 - DIAGRAMA UNIFILAR DO SISTEMA MODELADO 105
FIGURA 6-4 - ENSAIOS SENDO REALIZADOS NO REGULADOR DE TENSÃO DA UGH 8 106
FIGURA 6-5 - DIAGRAMA DE BLOCOS DO RAT DA PRIMEIRA ETAPA 107
FIGURA 6-6 - MALHA PRINCIPAL DO RAT DA SEGUNDA ETAPA 107
FIGURA 6-7 – COMPENSAÇÃO DE POTÊNCIA REATIVA DA SEGUNDA ETAPA 107
FIGURA 6-8 - LIMITADOR DE CORRENTE DE ARMADURA DA SEGUNDA ETAPA 108
FIGURA 6-9 - LIMITADOR DE SUBEXCITAÇÃO DA SEGUNDA ETAPA 108

FIGURA 6-10 - LIMITADOR DE SOBREXCITAÇÃO DA SEGUNDA ETAPA 108

FIGURA 6-11 - LIMITADOR DE CORRENTE DE EXCITAÇÃO DA SEGUNDA ETAPA 108

FIGURA 6-12 - LIMITADOR DE ENLACE DE FLUXO (VOLT/HZ) DA SEGUNDA ETAPA 109

FIGURA 6-13 - REPRESENTAÇÃO DO REGULADOR DE TENSÃO NO FRAMEWORK 111

FIGURA 6-14 - DECLARAÇÃO DA CLASSE QUE REPRESENTA O REGULADOR DE TENSÃO 112

FIGURA 6-15 - CONSTRUTOR DA CLASSE TAVRTUCURUI 112

FIGURA 6-16 - MÉTODO PARA O CÁLCULO DAS DERIVADAS DAS VARIÁVEIS DE ESTADO 113

FIGURA 6-17 - IMPLEMENTAÇÃO DO MÉTODO JACOBIAN 114

FIGURA 6-18 - IMPLEMENTAÇÃO DO MÉTODO OUTPUTS 115

FIGURA 6-19 - INICIALIZAÇÃO DAS VARIÁVEIS DE ESTADO DE TAVRTUCURUI 116

FIGURA 6-20 - GAVETA ONDE SE ENCONTRA O ESP 117

FIGURA 6-22 - DIAGRAMA DE BLOCOS DO ESP DA SEGUNDA ETAPA 118

FIGURA 6-21 - DIAGRAMA DE BLOCOS DO ESP DA PRIMEIRA ETAPA 117

FI GURA 6-23 - TURBINA FRANCIS, SIMILAR ÀS INSTALADAS NA UHE DE TUCURUÍ (WIKIPÉDIA: A ENCICLOPÉDIA LIVRE) 120

FIGURA 6-24 - MODELO DAS TURBINAS EMPREGADO 120

FIGURA 6-25 - ENSAIOS SENDO REALIZADOS NO RV DA UGH 8 122

FIGURA 6-27 - DIAGRAMA DE BLOCOS DO RV DA SEGUNDA ETAPA 123

FIGURA 6-26 - DIAGRAMA DE BLOCOS DO RV DA PRIMEIRA ETAPA 122

FIGURA 6-28 - CURTO-CIRCUITO TRIFÁSICO APLICADO NA MÁQUINA 13 127

FIGURA 6-29 - DEGRAU DE +10% APLICADO NA REFERÊNCIA DO RAT DA MÁQUINA 13 127

FIGURA 6-30 - DEGRAU DE +10% APLICADO REFERÊNCIA DO RV DA MÁQUINA 13 128

FIGURA 6-31 - DEGRAU DE +10% APLICADO NA REFERÊNCIA DO RV DE UMA MÁQUINA DA PRIMEIRA ETAPA, OBTIDO PELO ONS, EM AZUL DADOS MEDIDOS E EM VERMELHO A SIMULAÇÃO 128

FIGURA 6-32 - DEGRAU DE +10% APLICADO NA REFERÊNCIA DO RV DE UMA MÁQUINA DA PRIMEIRA ETAPA, UTILIZANDO FRAMEWORK 129

FIGURA 6-33 - DEGRAU DE -9,09% APLICADO NA REFERÊNCIA DO RV DE UMA MÁQUINA DA PRIMEIRA ETAPA, OBTIDO PELO ONS (OPERADOR NACIONAL DO SISTEMA, 2003A) 129

FIGURA 6-34 - DEGRAU DE -9,09% APLICADO NA REFERÊNCIA DO RV DE UMA MÁQUINA DA PRIMEIRA ETAPA, UTILIZANDO O FRAMEWORK 130

FIGURA 6-35 - DEGRAU DE +2,9% APLICADO NA REFERÊNCIA DO RV DA UGH 8, MEDIDO EM CAMPO E COMPARADO COM SIMULAÇÕES UTILIZANDO O FRAMEWORK 130

FIGURA 6-36 - DEGRAU DE -25% APLICADO NA REFERÊNCIA DO RV DE UMA MÁQUINA DA SEGUNDA ETAPA, OBTIDO PELO ONS 131

FIGURA 6-37 - DEGRAU DE -25% APLICADO NA REFERÊNCIA DO RV DE UMA MÁQUINA DA SEGUNDA ETAPA, UTILIZANDO O FRAMEWORK 131

FIGURA 7-1 DIAGRAMA UNIFILAR DO SISTEMA UTILIZADO 134

FIGURA 7-2 - DECLARAÇÃO DE UMA REDE ELÉTRICA 135

FIGURA 7-3 - CADASTRANDO BARRAS NO SISTEMA 137

FIGURA 7-4 - CADASTRO DE RAMOS DA REDE ELÉTRICA 139

FIGURA 7-5 - CÁLCULO DO FLUXO DE CARGA 139

FIGURA 7-6 - CRIAÇÃO DAS INSTÂNCIAS DOS GERADORES SÍNCRONOS 140

FIGURA 7-7 - DIAGRAMA EM BLOCOS DO REGULADOR DE TENSÃO 141

FIGURA 7-8 - DECLARAÇÃO DA CLASSE QUE REPRESENTA O REGULADOR DE TENSÃO 142

FIGURA 7-9 - IMPLEMENTAÇÃO DO MÉTODO DERIVATIVES PARA O REGULADOR DE TENSÃO 143

FIGURA 7-10 - CÁLCULO DAS CONDIÇÕES INICIAIS DO REGULADOR DE TENSÃO 144

FIGURA 7-11 - CÁLCULO DA SAÍDA DO REGULADOR DE TENSÃO 144

FIGURA 7-12 - CRIAÇÃO DAS INSTÂNCIAS DOS REGULADORES DE TENSÃO 145

FIGURA 7-13 - MODELO DE REGULADOR DE VELOCIDADE PROPOSTO EM (ARRILAGA & ARNOLD, 1990) 145

FIGURA 7-14 - DECLARAÇÃO DA CLASSE QUE REPRESENTA O REGULADOR DE VELOCIDADE 146

FIGURA 7-15 - CONSTRUTOR DA CLASSE QUE REPRESENTA O REGULADOR DE VELOCIDADE 147

FIGURA 7-16 - MÉTODO PARA O CÁLCULO DA DERIVADA DAS VARIÁVEIS DE ESTADO 148

FIGURA 7-17 - MÉTODO DE CÁLCULO DAS CONDIÇÕES INICIAIS DO REGULADOR DE VELOCIDADE 149

FIGURA 7-18 - IMPLEMENTAÇÃO DO MÉTODO OUTPUTS DO REGULADOR DE VELOCIDADE 150

FIGURA 7-19 - CRIAÇÃO DAS INSTÂNCIAS DOS REGULADORES DE VELOCIDADE 151

FIGURA 7-20 - DIAGRAMA EM BLOCOS DA TURBINA 151

FIGURA 7-21 - CRIAÇÃO DAS INSTÂNCIAS DAS TURBINAS 151

FIGURA 7-22 - CRIAÇÃO DAS INSTÂNCIAS DAS CLASSES TSYSTEM E TRUNGEKUTTA4 152

FIGURA 7-23 - CADASTRO DOS BLOCOS NO SISTEMA DINÂMICO 152

FIGURA 7-24 - DIAGRAMA EM BLOCOS DE UMA UNIDADE DE GERAÇÃO 152

FIGURA 7-25 - CONEXÃO DAS ENTRADAS DA REDE NAS SAÍDAS DA MÁQUINA SÍNCRONA 153

FIGURA 7-26 - CONEXÃO DAS ENTRADAS DE G1 NAS SAÍDAS DA REDE ELÉTRICA 153

FIGURA 7-27 - FRAGMENTO DE CÓDIGO QUE COMPLETA AS CONEXÕES DA UNIDADE DE GERAÇÃO G1 153

FIGURA 7-28 - CIRCUITO EQUIVALENTE DAS CÉLULAS PV 154
FIGURA 7-29 - DECLARAÇÃO DA CLASSE QUE REPRESENTA UM MÓDULO PV 156
FIGURA 7-30 - REPRESENTAÇÃO COMPUTACIONAL DO MÓDULO PV 156
FIGURA 7-31 - IMPLEMENTAÇÃO DO MÉTODO OUTPUTS DA CLASSE QUE REPRESENTA OS MÓDULOS PV 157
FIGURA 7-32 - MÉTODO INITIALIZE DA CLASSE TPVARRAY 159
FIGURA 7-33 - CIRCUITO EQUIVALENTE DO MODELO DA MÉDIA PARA O INVERSOR CC/CA 160
FIGURA 7-34 - DECLARAÇÃO DA CLASSE TPVINVERTER 163
FIGURA 7-35 - REPRESENTAÇÃO PARA DIAGRAMA DE BLOCOS DA CLASSE TPVINVERTER 164
FIGURA 7-36 - MÉTODO OUTPUTS DA CLASSE TPVINVERTER 165
FIGURA 7-37 - CÁLCULO DA DERIVADA DA VARIÁVEL DE ESTADO 166
FIGURA 7-38 - CALCULO DAS CONDIÇÕES INICIAIS DO INVERSOR 169
FIGURA 7-39 - DIAGRAMA EM BLOCOS DO SISTEMA DE CONTROLE DE CORRENTE 170
FIGURA 7-40 - COMPENSADORES DE CORRENTE DO EIXO *D* E *Q* 171
FIGURA 7-41 - EQUIVALENTE SISO DO COMPENSADOR DO EIXO *Q* 171
FIGURA 7-42 - EQUIVALENTE SISO DO COMPENSADOR DO EIXO *D* 172
FIGURA 7-43 - DIAGRAMA EM BLOCOS DO COMPENSADOR DO EIXO *Q* 172
FIGURA 7-44 - DIAGRAMA EM BLOCOS DO COMPENSADOR DO EIXO *D* 173
FIGURA 7-45 - CONTROLE PI EXPANDIDO DO EIXO *Q* 173
FIGURA 7-46 - DECLARAÇÃO DA CLASSE QUE REPRESENTA O COMPENSADOR DO EIXO *Q* 175
FIGURA 7-47 - IMPLEMENTAÇÃO DO MÉTODO *DERIVATIVES* DA CLASSE *TPVCOMPQ* 176
FIGURA 7-48 - SISTEMA DE CONTROLE DE CARGA 177
FIGURA 7-49 - DECLARAÇÃO DA CLASSE QUE IMPLEMENTA O CONVERSOR BOOST 178
FIGURA 7-50 - REPRESENTAÇÃO EM DIAGRAMA DE BLOCOS DO CONVERSOR BOOST 178
FIGURA 7-51 - DIAGRAMA EM BLOCOS DO CONTROLADOR PI 178
FIGURA 7-52 - DECLARAÇÃO DO CONTROLADOR PI DO CONVERSOR BOOST 179
FIGURA 7-53 - CURVA DA POTÊNCIA VERSUS TENSÃO DE UM SISTEMA FOTOVOLTAICO 180
FIGURA 7-54 - FLUXOGRAMA DO MÉTODO DE CONDUTÂNCIA INCREMENTAL 181
FIGURA 7-55 - DIAGRAMA EM BLOCOS DO MPPT 182
FIGURA 7-56 - DECLARAÇÃO DA CLASSE QUE REPRESENTA O MPPT 183
FIGURA 7-57 - IMPLEMENTAÇÃO DO MÉTODO *OUTPUTS* DO MPPT 184

FIGURA 7-58 - SUBSTITUIÇÃO DO MÉTODO *INITIALIZE* PARA A CLASSE DO MPPT 184

FIGURA 7-59 - DIAGRAMA EM BLOCOS DE UMA UNIDADE DE GERAÇÃO PV 185

FIGURA 7-60 - CRIAÇÃO DAS INSTÂNCIAS QUE REPRESENTAM AS FONTES 185

FIGURA 7-61 - CONEXÕES DO INVERSOR COM A REDE ELÉTRICA 186

FIGURA 7-62 - SIMULAÇÃO DE UM DEGRAU NA REFERÊNCIA DE CORRENTE NO EIXO *Q* 186

FIGURA 7-63 - SIMULAÇÃO DE UM DEGRAU NA REFERÊNCIA DE CORRENTE NO EIXO *D* 187

FIGURA 7-64 - CURVAS DA SIMULAÇÃO EM QUE O *MPPT* É ATIVADO 188

FIGURA 7-65 - COMPORTAMENTO DO ÍNDICE DE MODULAÇÃO DO CONVERSOR BOOST 189

Lista de Tabelas

TABELA 2-1 - CLASSES PARA MATRIZES E VETORES 23

TABELA 3-1 - ATRIBUTOS DA CLASSE TBUS 47

TABELA 3-2 - ARGUMENTOS DOS MÉTODOS DE CRIAÇÃO DE INSTÂNCIAS DE BARRAS 48

TABELA 3-3 - ARGUMENTOS DO MÉTODO DE CRIAÇÃO DE INSTÂNCIAS DA CLASSE *TLOAD* 49

TABELA 3-4 - ATRIBUTOS DA CLASSE *TBRANCH* 50

TABELA 3-5 - ARGUMENTOS DO MÉTODO DE CRIAÇÃO DE INSTÂNCIAS DA CLASSE *TBRANCH* 51

TABELA 3-6 - ATRIBUTOS DA CLASSE *TAREA* 52

TABELA 3-7 - MÉTODOS DA CLASSE TNETWORK DE CADASTRO DE DISPOSITIVOS NA REDE 53

TABELA 3-8 - MÉTODOS PARA REFERENCIAR OS DISPOSITIVOS CADASTRADOS 54

TABELA 4-1 - ATRIBUTOS DA CLASSE TBLOCK. 61

TABELA 4-2 - MÉTODOS DA CLASSE TBLOCK 63

TABELA 4-3 - ALGUNS BLOCOS LINEARES DEFINIDOS EM *LINEAR.H* 66

TABELA 4-4 - ATRIBUTOS DA CLASSE *TSYSTEM* 67

TABELA 4-5 - MÉTODOS DA CLASSE *TSYSTEM* 68

TABELA 4-6- ATRIBUTOS DA CLASSE TINTEGRATOR 70

TABELA 4-7 - MÉTODOS DA CLASSE TINTEGRATOR 71

TABELA 5-1 - GRANDEZAS RELACIONADAS COM AS EQUAÇÕES DE BALANÇO MECÂNICO DA MÁQUINA 89

TABELA 5-2 - GRANDEZAS RELACIONADAS ÀS EQUAÇÕES ELÉTRICAS DA MÁQUINA 91

TABELA 5-3 - ATRIBUTOS DA CLASSE QUE REPRESENTA AS MÁQUINAS SÍNCRONAS 95

TABELA 5-4 - VARIÁVEIS DE ESTADO DO MODELO DO GERADOR SÍNCRONO 96

TABELA 6-1 - RESULTADO DO FLUXO DE CARGA 126

TABELA 7-1 - DADOS DE BARRAS APÓS O FLUXO DE CARGA 135

TABELA 7-2 - ARGUMENTOS USADOS NO CADASTRO DE BARRAS 136

TABELA 7-3 - ARGUMENTOS DO MÉTODO DE CADASTRO DE RAMOS 138

TABELA 7-4 - ATRIBUTOS DA CLASSE CRIADA 162

TABELA 7-5 - LISTA DE PARÂMETROS A SER PASSADO PARA A CLASSE NO CONSTRUTOR 176

Lista de Abreviaturas e Siglas

Eletronorte	Centrais Elétricas do Norte do Brasil S/A
ESP	Estabilizador de Sistema de Potência
FEE	Faculdade de Engenharia Elétrica
ONS	Operador Nacional do Sistema Elétrico
OO	Orientação a Objetos ou Orientado(a) a Objetos
P&D	Pesquisa e Desenvolvimento
p.u.	Por Unidade
PC	Personal Computer
PPGEE	Programa de Pós-Graduação em Engenharia Elétrica
PSP	Personal Software Process
RAT	Regulador Automático de Tensão
RV	Regulador de Velocidade
SEP	Sistema Elétrico de Potência
SIN	Sistema Interligado Nacional
UFPA	Universidade Federal do Pará
UGH	Unidade Geradora Hidrelétrica
UHE	Usina Hidrelétrica
MPPT	Rastreador de Máxima Potência
PV	Fotovoltaico

Sumário

1 Introdução *16*

1.1 Objetivo da tese de doutoramento **16**

1.2 Justificativa **17**

1.3 Estado da arte **18**

1.4 Publicações **19**

1.5 Organização da tese **20**

2 Matrizes e vetores não – esparsos e esparsos *21*

2.1 Introdução **21**

2.2 Modelagem orientada a objetos **23**

2.3 Vetores e Matrizes Não – Esparsos **24**

2.3.1 Introdução 24

2.3.2 Reaproveitamento de instâncias temporárias 24

2.3.3 Regra dos objetos temporários 25

2.3.3.1 Reutilização de temporários 25

2.3.3.2 Atribuição presumida 25

2.3.4 Cópia presumida 26

2.3.5 Implementação da técnica de reaproveitamento de objetos temporários na classe de vetores não esparsos no Framework 27

2.3.5.1 Gabaritos de classes para vetores não esparsos 27

2.3.5.2 Exemplo de utilização do gabarito de classe TVector 34

2.3.5.3 Comentários 34

2.3.6 Matrizes não esparsas 34

2.4 Esparsidade **35**

2.4.1 Vetores esparsos 36

2.4.1.1 Armazenamento dos elementos 36

2.4.1.2 Subvetores esparsos 36

2.4.1.3 Operações de subscrição 37

2.4.1.4 Inicialização de vetores esparsos 38

2.4.2 Matrizes esparsas 39

2.4.2.1 Armazenamento dos elementos 40

2.4.2.2 Operações de subscrição de elementos 41

2.4.2.3 Inicialização de matrizes esparsas 41

2.5 Conclusão 43

3 Modelagem e implementação orientada a objetos para execução de fluxo de carga 44

3.1 Modelagem orientada a objetos da rede de transmissão 44

3.1.1 Definição das classes e suas relações 44

3.1.2 A classe que representa as barras 45

3.1.3 Classe que representa modelos de cargas 48

3.1.4 Classe que representa ramos 49

3.1.5 Classe que representa as áreas 51

3.1.6 Classe que representa a rede de transmissão 52

3.1.6.1 Definição da classe 52

3.1.6.2 Cadastrando dispositivos na rede 53

3.2 Conclusão 56

4 Modelagem e implementação orientada a objetos para representação de sistemas dinâmicos 57

4.1 Introdução 57

4.2 Modelagem orientada a objetos do problema 57

4.2.1 Introdução 57

4.2.2 Representação em espaço de estados 57

4.2.3 Representação em diagramas em blocos 58

4.2.4 Definição das classes 58

4.2.4.1 Definindo as abstrações e as implementações 58

4.2.4.2 A classe TBlock 60

4.2.4.3 Classes derivadas de TBlock 63

4.2.4.4 A classe TSystem 66

4.2.4.5 A integração Numérica: classe TIntegrator 68

4.2.4.6 Integração numérica: classe TRungeKutta4 71

4.2.4.7 Integração numérica: classe TTrapezoidal 73

4.3 Exemplo de uso da classes para dispositivos dinâmicos 78

4.4 Conclusão 80

5 Implementação orientada a objetos para a representação da dinâmica de sistemas de potência 81

5.1 Introdução 81

5.2 Modelo clássico de sistemas multimáquinas 81

5.2.1 Introdução 81

5.2.2 Representação de sistemas multimáquinas 81

5.2.3 Implementação das equações da rede na classe TNetwork 87

5.3 Implementação dos modelos das máquinas síncronas no Framework 88

5.3.1 Eixos de referência 88

5.3.2 Equações mecânicas 88

5.3.3 Equações elétricas 89

5.3.4 Cálculo das condições iniciais 92

5.3.5 Implementação no Framework da classe base para geradores 93

5.3.5.1 Método de cálculo das derivadas 95

5.3.5.2 Método de cálculo das saídas 97

5.3.5.3 Método de cálculo da matriz jacobiana para o método trapezoidal 99

5.3.5.4 Método de inicialização das variáveis da máquina 101

5.4 Conclusão 103

6 Estudo de caso 1: aplicação e validação do Framework na UHE Tucuruí 104

6.1 Modelagem da Hidrelétrica Tucuruí 104

6.1.1 Geradores 106

6.1.2 Reguladores de tensão 106

6.1.2.1 Modelagem em Diagrama de Blocos 106

6.1.2.2 Modelagem em Espaços de Estados 109

6.1.2.3 Implementação da classe que representa o regulador de tensão 110

6.1.3 Estabilizadores de Sistemas de Potência 117

6.1.3.1 Modelagem em Diagrama de Blocos 117

6.1.3.2 Modelagem em Espaços de Estados 118

6.1.4 Turbinas 120

6.1.4.1 Modelagem em Diagrama de Blocos 120

6.1.4.2 Modelagem em Espaços de Estados 121

6.1.5 Reguladores de Velocidade 121

6.1.5.1 Modelagem em Diagrama de Blocos 122

6.1.5.2 Modelagem em Espaços de Estados 123

6.2 Comparações entre as simulações e medições em campo 124

6.2.1 Simulações implementadas 124

6.2.2 Comparações com medições obtidas em campo 128

6.3 Conclusão 131

7 Estudo de caso 2: aplicação do Framework em estudo da interconexão de parques fotovoltaicos em sistemas multimáquinas 133

7.1 Introdução 133

7.2 Descrição do sistema de potência simulador 133

7.3 Implementando a rede elétrica 135

7.4 Modelagem das máquinas síncronas e seus controladores 140

7.4.1 Modelagem da máquina síncrona 140

7.4.2 Modelagem dos reguladores de tensão 141

7.4.3 Modelagem dos reguladores de velocidade 145

7.4.4 Implementação do modelo das turbinas 151

7.4.5 Conexão das máquinas síncronas à rede elétrica e aos seus controladores 152

7.5 Modelagem do Sistema Fotovoltaico Conectado a Rede Elétrica 153

7.5.1 Modelagem dos módulos PV 154

7.5.2 Inversor CC/CA 159

7.5.3 Controle de corrente 169

7.5.4 Conversor CC/CC Boost e o controle de carga do link CC 176

7.5.5 Rastreador de máxima potência 179

7.5.6 Conexões entre os dispositivos que compõe o gerador fotovoltaico 185

7.6 Simulações 186

7.6.1 Desacoplamento dos compensadores de corrente 186

7.6.2 Simulação do efeito do rastreamento de máxima potência 188

7.7 Conclusão 189

8 Conclusão 190

Referências Bibliográficas 192

1 Introdução

1.1 Objetivo da tese de doutoramento

O objetivo desta tese é contribuir com novas ferramentas destinadas a facilitar a execução de tarefas usuais em estudos sobre a dinâmica de sistemas elétricos de potência. Para isso, apresenta-se o desenvolvimento e testes de um novo Framework para esta finalidade. O Framework foi desenvolvido em linguagem C++ e fornece um conjunto de classes orientadas a objetos que tornam o desenvolvimento de programas de simulação uma tarefa mais simples e intuitiva. Neste trabalho são apresentados os conjuntos de classes orientadas a objetos, suas regras de implementação e convenções de desenvolvimento. Tais conjuntos encapsulam, de uma forma transparente, as rotinas de álgebra linear, matrizes não – esparsas e esparsas, integração numérica e modelagens dinâmicas, normalmente necessárias em estudos de sistemas de potência, de forma que estes conjuntos definem um novo Framework orientado a objetos para o desenvolvimento de simuladores.

A contribuição principal desta tese é propor uma abordagem alternativa ao que foi já apresentado até o momento na literatura técnica, conforme levantamento do estado da arte apresentada na seção 1.3 deste capítulo. É proposto um Framework com uma abordagem análoga aos diagramas em blocos, onde cada dispositivo dinâmico é representado por um bloco com entradas e saídas, que devem ser conectados entre si para formar dispositivos mais complexos. A rede elétrica, por exemplo, é representada por um grande bloco em torno do qual os blocos que representam geradores, turbinas, reguladores, etc., são, então, conectados. São definidos padrões de codificação de novos dispositivos dinâmicos visando minimizar a quantidade de código que se necessita escrever e evitando proliferação de classes derivadas. Além disso, a metodologia proposta torna muitos aspectos do desenvolvimento transparentes como, por exemplo, as rotinas para tratamento de matrizes e a integração numérica.

Esta tese se concentra e contribui nos seguintes aspectos da aplicação da orientação a objetos em sistemas elétricos de potência:

1. Trata especificamente da modelagem orientada a objetos da dinâmica dos sistemas elétricos de potência, propondo um novo Framework.
2. Desenvolve e implementa um novo conjunto de classes para tratamento de vetores e matrizes esparsos e não – esparsos que, através de técnicas especiais de programação, amenizam a perda de desempenho provocado pelos overheads, inerentes à implementação da orientação a objetos na linguagem de programação.
3. Desenvolve e implementa suporte para utilização das rotinas computacionais de álgebra linear mais eficientes, disponíveis na atualidade, e que foram desenvolvidas na forma de bibliotecas padronizadas. Estas bibliotecas são a BLAS descrita em (Lawson, Hanson, Kincaid, & Krogh, 1979) e LAPACK (Anderson E. , et al., 1990). Estas duas bibliotecas possuem várias implementações, geralmente

fornecidas por fabricantes de microprocessadores.

A tese contribui, ainda, com suporte para implementação de métodos de integração numérica implícitos e explícitos. Utilizando-se o Framework proposto, é possível implementar e estudar diversos métodos de integração numérica aplicados em sistemas elétricos de potência.

1.2 Justificativa

O desenvolvimento de softwares de simulação de sistemas dinâmicos tem sido feito utilizando-se a metodologia procedural (Neyer, Wu, & Imhof, 1990), onde as funcionalidades são os elementos em torno do qual a programação é desenvolvida. Os programas desenvolvidos utilizando-se esta metodologia podem levar a códigos fontes complexos e difíceis de manter e atualizar. Apesar dessas limitações, as rotinas desenvolvidas utilizando-se a metodologia procedural foram e continuam sendo aprimoradas como resultado de décadas de pesquisa e desenvolvimento.

Atualmente a metodologia de desenvolvimento mais empregada, em aplicações em geral, é a orientada a objetos (Deitel & Deitel, 2001). Nesta metodologia, as entidades que compõe o problema a ser resolvido são mapeadas na linguagem de programação como estruturas que contém tanto as características da entidade (atributos) quanto às funcionalidades oferecidas pela entidade (métodos), esta estrutura é denominada *classe*.

Podem-se destacar duas vantagens, muito importantes, obtidas no uso da orientação a objetos:

- Torna a programação mais intuitiva e mais próxima da forma como os problemas são tratados na vida real.
- Permite um gerenciamento muito mais efetivo do desenvolvimento de softwares complexos (Neyer, Wu, & Imhof, 1990).

Por outro lado, pode-se destacar uma desvantagem fundamental:

- A considerável quantidade de código "*overhead*", produzido pelos compiladores, com o objetivo de suportar as características das linguagens orientadas a objetos, tornam os códigos de máquina produzidos maiores e mais lentos. Mesmo assim, a utilização do desenvolvimento orientado a objetos (OOD) em problemas de simulação de sistemas dinâmicos é justificável pelo número de vantagens excederem amplamente o número de desvantagens (Dingle & Hildebrandt, 1998).

Outra questão importante é: "por que desenvolver um Framework ao invés de simplesmente utilizar um aplicativo disponível comercialmente?". Existem pacotes comerciais para análise e simulação de sistemas de potência, como por exemplo, os aplicativos Anatem (Oliveira, Rangel, Thomé, Baitelli, & Guimarães, 1994), EuroStag (Stubbe, Bihain, Deuse, & Baader, 1989), DigSilent PowerFactory (digSILENT, 2012), PSCAD (Manitoba - HVDC RESEARCH CENTRE, 2012), dentre outros que possuem um excelente desempenho computacional e, muitos deles, possuem uma interface do usuário amigável. No entanto sua arquitetura é fechada, tornando difícil ou até mesmo impossível adaptá-los para necessidades muito específicas.

Existem pacotes, como por exemplo, o Anatem que possibilitam que o usuário defina modelos de controladores usando uma representação de diagrama em blocos ou usando linguagens textuais de modelagem. No entanto, tal aplicativo fornece suporte somente para modelagem de sistemas dinâmicos de tempo contínuo. Desta forma, carecem de ferramentas para modelar e simular sistemas dinâmicos de tempo discreto, ou sistemas dinâmicos variantes no tempo, tais como controladores adaptativos.

O aplicativo DigSilent PowerFactory permite a implementação de novos modelos de dispositivos através da linguagem denominada *DigSILENT Simulation Language* ou *DSL* (Leelaruji & Vanfretti, 2012), onde é possível definir as equações diferenciais, a inicialização das variáveis de estado, não linearidades e implementação de modelos de dispositivos digitais. Apesar destas características, a DSL é implementada como parte do DigSILENT limitando o usuário às características do aplicativo.

No meio científico, muitos pesquisadores, quando desejam avaliar novas técnicas de controle automático e/ou implementação numérica de modelos de dispositivos, desenvolvem programas próprios para simular o comportamento dos sistemas em estudo. Desta forma, é necessário que esteja acessível ao pesquisador ferramentas para o desenvolvimento dos seus próprios aplicativos (daí a popularidade de softwares como o Matlab). Para atender a esta necessidade, esta tese de doutorado apresenta um Framework concebido para ser flexível e de fácil atualização e que provê os componentes de software básicos para o desenvolvimento de aplicativos para a análise dinâmica de sistemas de potência.

1.3 Estado da arte

Em função das vantagens da orientação a objetos (mesmo considerando-se as possíveis desvantagens) e a necessidade dos pesquisadores em desenvolver seus próprios programas, vários trabalhos foram publicados sobre a aplicação da orientação a objetos em sistemas elétricos de potência. Um dos trabalhos pioneiros nesta área está na referência (Neyer, Wu, & Imhof, 1990), onde os autores apresentam modelagem orientada a objetos para equipamentos da rede elétrica. Outros artigos publicados que seguem esta linha são (Zhou, 1996), (Zhu & Lubkeman, 1997) e (McMorran, Ault, Morgan, Elders, & McDonald, 2006). Nestes artigos são aplicados vários conceitos relacionados ao projeto orientado a objetos no desenvolvimento de programas para estudos de sistemas elétricos de potência. Embora estes artigos apresentem resultados bastante relevantes para a aplicação da modelagem orientada a objetos em sistemas elétricos de potência, estes artigos não tratam da representação da dinâmica dos sistemas.

Uma ferramenta interessante é o "*ObjectStab*" que é apresentado na referência (Larsson, 2004). Neste artigo é apresentada uma ferramenta de pesquisa e ensino em sistemas elétricos de potência, com características de reutilização e adaptação muito flexíveis utilizando a linguagem de modelagem *Modelica* (Fritzson & Bunus, 2002). A linguagem *Modelica* é interpretada por programas de simulação de sistemas físicos, como por exemplo, o *Dymola* descrito na referência (Cellier & Elmqvist, 1993).

Um trabalho que realiza a modelagem de sistemas elétricos de potência de uma forma mais ampla, incluindo aspectos relacionados com a dinâmica do sistema, é descrito nas referências (Manzoni, Silva, &

Decker, 1999) e (Agostini, Decker, & Silva, 2007). Entretanto, a modelagem descrita no artigo é pouco flexível, na representação da parte dinâmica dos sistemas de potência, porque apresenta dispositivos complexos como unidades básicas de modelagem, presentes em sistemas elétricos de potência, que possuem funcionalidades e atributos já conhecidos (como por exemplo, as classes *C_Turbine*, *C_Vel_Reg* e *C_Sync_Machine* apresentadas na referência (Agostini, Decker, & Silva, 2007)). Caso houver a necessidade de representação de um novo modelo de dispositivo, como por exemplo, algum tipo de controlador inteligente, poderia levar a necessidade de reescrita de várias rotinas computacionais ou poderia provocar a proliferação de classes derivadas.

1.4 Publicações

Durante as pesquisas e desenvolvimentos realizados no doutoramento, foram publicados vários trabalhos. Dentre os quais se podem destacar:

1. Artigos em periódicos:
    a. Referência (Sena, Fonseca, Di Paolo, & Barra, 2011): artigo publicado em periódico A1 para Engenharia IV. Neste artigo é apresentada a aplicação do Framework na simulação das oscilações eletromecânicas na hidrelétrica de Tucuruí cujos resultados foram validados por ensaios em campo.
    b. Referência (Di Paolo, Cordeiro, Sena, Fonseca, & Barra, 2010): artigo publicado em periódico B3 para Engenharia IV. Neste artigo é apresentado um Framework de aplicação derivado do Framework apresentado nesta tese no contexto de padrões de projeto.
2. Artigos em conferências
    a. Na referência (Sena, Barra, Barreiros, Fonseca, Campos, & Costa, 2005), são apresentados os primeiros resultados obtidos neste doutoramento: a modelagem do problema e uma implementação do Framework onde os resultados são validados com aplicativo de mercado.
    b. Na referência (Sena, Campos, Fonseca, Barra, Barreiros, & Costa, 2006), é apresentada a estrutura atual do Framework e alguns resultados de simulação.
    c. Na referência (Di Paolo, Fonseca, Sena, Nogueira, Damasceno, & Barra, 2011) é apresentado um Framework de aplicação derivado do Framework apresentado nesta tese de doutoramento.
3. O Framework apresentado nesta tese de doutoramento foi empregado no estudo de oscilações eletromecânicas intra – planta da usina termelétrica de Santana (AP), pertencente a Eletrobrás – Eletronorte. Os resultados obtidos com o uso do Framework e o desenvolvimento de um estabilizador de sistema de potência, na malha de velocidade, para os grupos diesel de grande porte existentes na usina, foram usados na elaboração de um capítulo de livro da referência (Nogueira, Sena, Fonseca, & Barra, 2012) (a ser publicado em breve) sobre grandes motores a diesel.

Além das publicações referenciadas acima, os resultados obtidos na pesquisa realizada durante este doutoramento, foram empregados na elaboração da dissertação de mestrado (Di Paolo Í. F., 2009)

1.5 Organização da tese

O texto desta tese está organizado nos seguintes capítulos:

1. **Introdução**. Capítulo que apresenta a motivação e contribuições da tese;
2. **Matrizes e Vetores Não - Esparsos e Esparsos**. Apresenta a implementação das classes orientadas a objetos que encapsulam as rotinas de tratamento de vetores e matrizes esparsas e não – esparsas, propostas na tese;
3. **Fluxo de Carga**. Capítulo onde é apresentada a implementação das classes propostas para a descrição de redes elétricas e a execução de fluxo de carga;
4. **Representação da dinâmica de sistemas físicos**. Capítulo onde as classes propostas para sistemas dinâmicos em geral são apresentadas. Neste capítulo, as implementações destas classes são discutidas e detalhadas;
5. **Representação da dinâmica de sistemas de potência**. Neste capítulo, são apresentadas as classes que implementam os geradores síncronos e a rede elétrica, além das diretrizes básicas para que o usuário do Framework crie as classes especializadas que implementam outros dispositivos, tais como controladores de tensão, de velocidade, estabilizadores de sistemas de potência, etc.;
6. **Aplicação do Framework em um simulador para a UHE Tucuruí**. Neste capítulo são apresentados e discutidos os resultados da aplicação do Framework proposto na implementação de um simulador para estudos da dinâmica eletromecânica da usina hidrelétrica de Tucuruí;
7. **Aplicação do Framework em estudo da interconexão de parques fotovoltaicos em sistemas multimáquinas**. Neste capítulo é apresentado um estudo de caso com o objetivo de demonstrar a aplicabilidade do Framework na simulação do comportamento de sistemas de geração não convencionais. Neste estudo de caso também é demonstrada a aplicação do Framework na simulação de sistemas mistos, onde dispositivos de tempo contínuo (analógicos) e de tempo discreto (digitais) estão presentes no mesmo sistema.
8. **Conclusão**. Neste capítulo são resumidos os resultados obtidos nesta tese e quais as perspectivas para futuros trabalhos.

2 Matrizes e vetores não – esparsos e esparsos

2.1 Introdução

O Framework desenvolvido nesta tese está organizado em pacotes como ilustrado na Figura 2-1. Cada pacote é constituído por um conjunto de rotinas computacionais e classes orientadas a objetos organizadas de tal forma que os pacotes nas posições inferiores na figura fornecem um conjunto de serviços aos pacotes superiores.

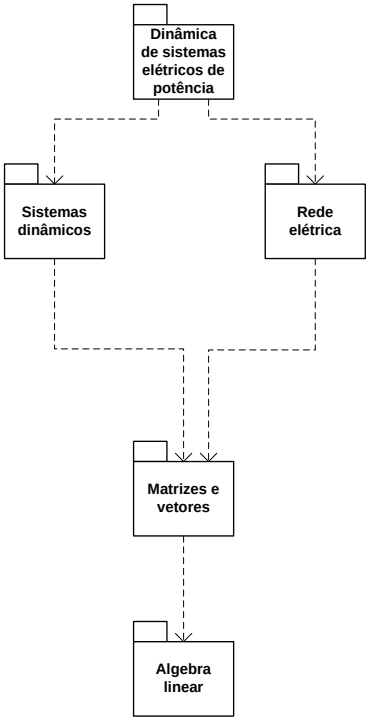

Figura 2-1 - Pacotes que compõe o Framework

Na Figura 2-1 o pacote nomeado de "Álgebra linear" contém as rotinas computacionais de alto desempenho e não orientadas a objetos para tratamento de matrizes e vetores. O pacote denominado "Álgebra linear" é constituído por duas bibliotecas: BLAS e LAPACK. A sigla BLAS significa Basic Linear Algebra Subprograms que constitui um conjunto de sub-rotinas para execução de tarefas de álgebra linear. A BLAS é constituída por três conjuntos de rotinas (Blackford, et al., 1996-2000):

1. BLAS level 1. Conjunto de rotinas referentes a operações entre vetores.
2. BLAS level 2. Conjunto de rotinas referentes a operações entre vetores e matrizes.
3. BLAS level 3. Conjunto de rotinas referentes a operações entre matrizes.

Um conjunto de rotinas derivadas da BLAS é a SP-BLAS (Duff, Heroux, & Pozo, 2002) que implementam operações envolvendo vetores e matrizes esparsos. Outra variante da biblioteca é a PB-BLAS (Choi, Dongarra, & Walker, 1994), em que as operações são paralelizadas para execução em máquinas com vários processadores.

A sigla LAPACK significa *Linear Algebra Package* (Anderson E. , et al., 1990) e constitui um conjunto de sub-rotinas para solução de problemas comumente encontrados em álgebra linear como, por exemplo, solução de sistemas de equações, busca de soluções através de mínimos quadrados para sistemas de equações sobredeterminados e solução de problemas de autovalores. São fornecidos também métodos de fatoração de matrizes para solução de sistemas de equações (LU, Cholesky, QR, SVD, Schur, Schur generalizado), mais informações podem ser obtidas na referência (Anderson E. , et al., 1999).

Estas bibliotecas são, na verdade, especificações e existem no mercado diversas implementações. Uma das implementações é a ATLAS (*Automatically Tuned Linear Algebra Software*) que implementa a BLAS de uma forma otimizada, detectando as configurações da máquina onde está instalada e ajustando seus parâmetros de desempenho automaticamente (Whaley & Dongarra, 1998).

Os fabricantes de processadores mantêm implementações específicas, com suporte para suas famílias de processadores, para as bibliotecas BLAS e LAPACK. Por exemplo, a Intel fornece a MKL (*Math Kernel Library*), otimizada para suas linhas de processadores. A cada lançamento de um novo processador no mercado, esta biblioteca é atualizada para dar suporte ao mesmo (Intel Corporation, 2012). Como a implementação destas bibliotecas é fornecida pelo fabricante do respectivo microprocessador alvo, pode-se considerar as implementações destas bibliotecas como "drivers de dispositivos".

No Framework desenvolvido nesta tese, o pacote denominado "Matrizes e Vetores" (ver Figura 2-1) contêm diversos conjuntos, classes e funções que implementam as operações fundamentais necessária à criação e tratamento de vetores e matrizes não esparsos e esparsos. No pacote "Sistemas dinâmicos" estão implementadas as classes utilizadas para simular (no domínio do tempo) sistemas dinâmicos. Neste pacote estão classes que representam dispositivos físicos onde a dinâmica deve ser levada em consideração. Neste pacote também são encontradas as classes que implementam os métodos de integração numérica. No pacote denominado de "Rede Elétrica e Fluxo de Carga" estão as funções que implementam as rotinas de fluxo de carga em sistemas elétricos de potência. No pacote "Dinâmica de Sistemas Elétricos de Potência" estão as classes específicas para sistemas elétricos de potência, onde são implementadas as dinâmicas de geradores, turbinas, reguladores, etc.

Neste capítulo, será apresentado o pacote desenvolvido para tratamento de matrizes e vetores. Serão apresentadas as técnicas empregadas com o objetivo de melhorar o desempenho face aos *overheads* presentes na implementação baseada em linguagens orientadas a objetos.

2.2 Modelagem orientada a objetos

As bibliotecas citadas anteriormente são constituídas por um conjunto de funções (não orientada a objetos) cujas listas de argumentos são muito extensas. Cada função implementa uma operação de álgebra linear relativamente complexa, como pode ser verificado nas referências (Blackford, et al., 1996-2000) e (Anderson E. , et al., 1999).

Por exemplo, sendo x e y dois vetores não esparsos reais de precisão dupla e a um número escalar de precisão dupla, a *BLAS* fornece a operação:

$$\bar{y} = a\bar{x} + \bar{y} \tag{2.1}$$

A equação (2.1) deve ser utilizada para programar as operações de soma de dois vetores e produto de um escalar com um vetor. A função em linguagem C para esta operação possui a seguinte sintaxe:

cblas_daxpy(N, a, x, 1, y, 1);

Onde N é o número de elementos dos vetores. Este padrão de utilização se estende a todas as funções disponíveis nas bibliotecas. Portanto, para efetuar uma simples operação de soma elemento a elemento de dois vetores, por exemplo, várias operações devem ser feitas.

Uma das características desejáveis no Framework consiste em tornar a sua utilização intuitiva e fácil, portanto, para vetores e matrizes, que podem ser esparsos e não – esparsos, são criadas classes que representam estas entidades.

Entidade	**Classe**
Vetor não esparso	*TVector*
Vetor esparso	*TSparseVector*
Matriz não esparsa	*TMatrix*
Matriz esparsa	*TSparseMatrix*

Tabela 2-1 - Classes para matrizes e vetores

Os elementos de vetores e de matrizes esparsas e não – esparsas podem ser reais de precisão simples, reais de precisão dupla, complexos de precisão simples e complexos de precisão dupla. Para representar estas possibilidades, poder-se-ia criar uma classe para cada um destes casos de tipos de elementos de vetores e matrizes, no entanto, esta solução resultaria em um número muito grande de classes para representar vetores e matrizes o que poderia dificultar o uso do Framework desenvolvido. Para evitar esta proliferação de classes, foram criados gabaritos de classes para vetores e matrizes esparsos e não esparsos (classes parametrizadas ou genéricas), cujo único parâmetro *T* é o tipo de dado de seus elementos, como ilustrado na Figura 2-2.

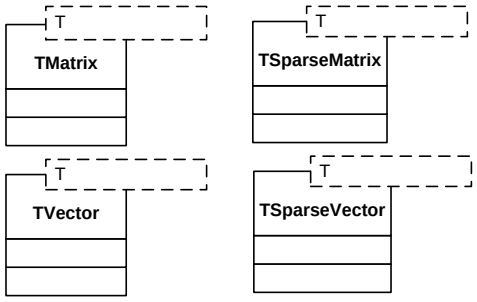

Figura 2-2 - Classes do pacote "Matrizes e Vetores"

Cada classe do pacote “Matrizes e Vetores” consistirão em uma interface unificada para um subsistema complexo constituído pelas bibliotecas *BLAS* e *LAPACK*. Esta abordagem é uma aplicação do padrão de projeto (design pattern) denominado *Facade* (fachada) onde cada uma das classes *TVector*, *TSparseVector*, *TMatrix* e *TSparseMatrix* são classes de *fachada* (*facade classes*) para as rotinas de tratamento de vetores e matrizes esparsos e não – esparsos contidas nas bibliotecas de álgebra linear (*BLAS*, *LAPACK* e bibliotecas correlatas) (Gamma, Helm, Johnson, & Vlissides, 1995).

2.3 Vetores e Matrizes Não – Esparsos

2.3.1 Introdução

O Framework desenvolvido nesta tese possui um conjunto de classes que representam tanto vetores e matrizes não esparsos quanto esparsos. Além destas classes, foram sobrecarregados vários operadores aritméticos e funções comumente utilizadas em problemas científicos. Os operadores e as funções, em grande parte, são implementadas no Framework através de chamadas às funções existentes nas bibliotecas padrões BLAS e LAPACK.

Com a sobrecarga de várias funções e de operadores aritméticos é possível, por exemplo, escrever em linguagem C++ expressões matriciais do tipo A = B*inv(C)+D. Poder escrever expressões envolvendo vetores e matrizes de forma literal representa um grande ganho de produtividade, tornando o código desenvolvido mais facilmente gerenciável. No entanto, para implementar este recurso várias operações de suporte devem ser executadas, o que diminui a velocidade com que o resultado é disponibilizado.

Nas seções subsequentes será apresentada a técnica de “reaproveitamento de instancias temporária”, a técnica utilizada nesta tese para amenizar a perda de desempenho devido aos *overheads* inerentes à programação orientada a objetos.

2.3.2 Reaproveitamento de instâncias temporárias

O desenvolvimento de software orientado a objetos possui inúmeras vantagens, tais como sobrecarga de operadores, reaproveitamento intensivo de código, abstrações mais próximas da realidade material e permite gestão do desenvolvimento de forma mais eficiente. No entanto, quando utilizada em áreas da computação científica, que exigem elevada precisão numérica e rapidez de execução, a metodologia orientada a objetos

possui características que podem diminuir consideravelmente o desempenho da aplicação alvo. O baixo desempenho computacional geralmente associado a orientação a objetos possui como causa os *overheads* que envolvem a alocação e liberação de espaços de memória necessários na criação e na destruição de objetos, assim como na inicialização de objetos a um estado consistente e as cópias de conteúdos de vetores e matrizes. Além disso, funções podem retornar objetos, causando o surgimento de objetos temporários não nomeados, criados no escopo da chamada de funções. Ambas as situações impactam de uma forma significativa no desempenho de programas orientados a objetos, quando comparados com programas baseados em técnicas procedurais. Isso, de certa forma, explica em parte a existência de uma certa rejeição inicial ao uso da programação orientada a objetos em aplicações numericamente intensiva, por parcela da comunidade científica (Dingle & Hildebrandt, 1998).

2.3.3 Regra dos objetos temporários

Uma função, escrita em uma linguagem orientada a objetos, cria um objeto de retorno através da criação de uma instância de um objeto do tipo declarado como valor de retorno. Em expressões, o objeto retornado não tem nome e ele pode ser acessado apenas como um operando. Em seguida este objeto é destruído quando a avaliação da expressão onde ele foi criado estiver concluída.

Por exemplo, em A = C+B, a expressão C+B retorna um objeto temporário que é usado em seguida pelo operador de atribuição (=). Este operador, tipicamente, faz uma cópia do conteúdo do objeto temporário em A e em seguida o objeto temporário é destruído.

2.3.3.1 Reutilização de temporários

Criar e em seguida destruir instâncias temporárias envolve alocação, desalocação e cópia de conteúdo para outra área de memória para reter o resultado, além do gasto de tempo de processamento na inicialização deste objeto que possui um tempo de vida muito curto. Devido a isto é justificável que se tenha o menor número possível de objetos temporários.

Em linguagens orientadas a objetos, é virtualmente impossível alterar a forma de gerenciamento de objetos temporários. Portanto, é necessário criar técnicas que evitem a alocação, desalocação e cópia de áreas de memória para que a quantidade de "overheads" na criação de objetos temporários seja reduzida. Estas áreas de memória, ao invés de serem descartadas, podem ser então passadas de um objeto temporário a outro implementando, desta forma, a reutilização de objetos temporários.

2.3.3.2 Atribuição presumida

No caso de linguagens orientadas a objeto, um método conhecido como técnica da atribuição presumida (Dingle & Hildebrandt, 1998) elimina cópias desnecessárias de objetos temporários quando em uma operação de atribuição. Nesta técnica, o objeto alvo (a esquerda do sinal de atribuição), assume a memória usada pelo objeto temporário. No entanto, para que o compilador saiba a diferença entre o objeto alvo e o objeto temporário, eles devem pertencer a classes diferentes, mas relacionadas através de herança. Na Figura 2-3, a classe do objeto

alvo é denominada como “Classe referência” e a classe do objeto temporário é denominado “Classe temporária”. Na mesma figura pode-se verificar que as duas classes estão relacionadas pela relação de herança, onde a classe temporária é derivada da classe referência.

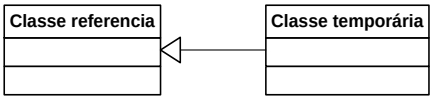

Figura 2-3 - Relação entre classe referencia (nomeada) e classe temporária (não nomeada)

2.3.4 Cópia presumida

A técnica de atribuição presumida pode ser generalizada para criar uma técnica chamada cópia presumida, onde qualquer função, não apenas as sobrecargas de operadores podem potencialmente reutilizar a memória alocada por objetos temporários. Em operações encadeadas (expressões), um objeto temporário pode tornar-se um operando de outra função, para depois o resultado ser utilizado em um operador de atribuição. Como forma de maximizar os ganhos de eficiência sugeridos pela técnica de atribuição presumida, é necessária a disponibilização de funções com reuso de operandos temporários.

Usualmente, o operador de atribuição faz uma cópia do conteúdo do operando para um objeto alvo e retorna uma referência a este objeto atualizado (o alvo). A cópia de conteúdo gera um objeto distinto do original: uma alteração no objeto copia não altera o objeto original. Uma alteração na referência altera também o original ao qual ele se refere.

Quando o operando de uma operação de atribuição é um objeto referenciado (declarado ou nomeado), como em A = B, uma cópia de conteúdo é necessária. Entretanto, se o operando é um temporário como em A = temp1 (onde temp1 = B+C), pode-se utilizar a cópia presumida e reaproveitar a memória usada em temp1.

Na cópia presumida, a memória usada pelo objeto operando é transferida ao objeto alvo (não há cópia de conteúdo), como apresentado na Figura 2-4. Cópia de ponteiros (transferência de memória), objetos que ocupam espaço muito pequeno na memória, gasta muito menos tempo que a cópia de um bloco de dados na memória (cópia de conteúdo).

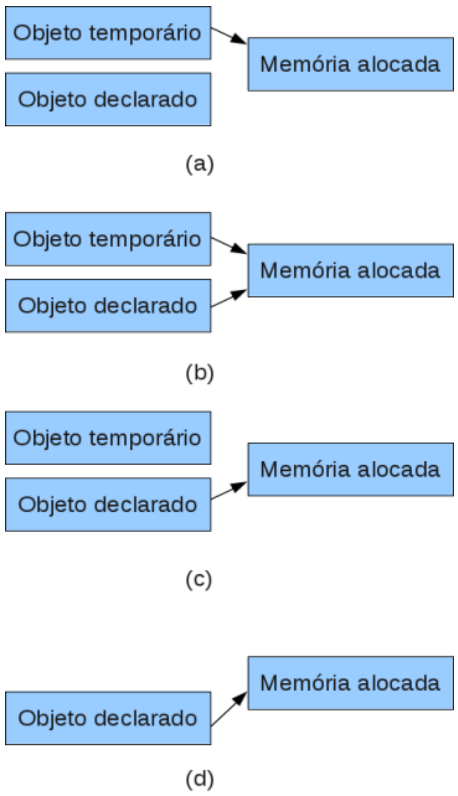

Figura 2-4 - Na cópia presumida são copiados os ponteiros, não os conteúdos de áreas de memória. Em (a) antes da função retornar. Em (b) após o retorno do operador de atribuição. Em (c) a memória é repassada ao objeto declarado. Em (d) o objeto temporário é destruído

A área de memória usada pelo objeto temporário é transferida ao objeto declarado seguindo três passos:

1. Liberar qualquer memória que o objeto declarado (denominado também de objeto nomeado) esteja utilizando;
2. Copiar o conteúdo apenas dos ponteiros do objeto temporário para o objeto declarado;
3. Ajustar o valor dos ponteiros do objeto temporário para o valor *null*.

2.3.5 Implementação da técnica de reaproveitamento de objetos temporários na classe de vetores não esparsos no Framework

Como pode ser identificado nas subseções anteriores, existem duas entidades diferentes envolvidas no processo de reaproveitamento de instâncias temporárias: a classe do objeto declarado (classe referência) e a classe do objeto temporário (classe temporária). Desta forma, no Framework desenvolvido nesta tese existe uma classe de vetores não esparsos para objetos declarados e uma classe de vetores não esparsos para objetos temporários.

2.3.5.1 Gabaritos de classes para vetores não esparsos

Vetores e matrizes não esparsos são aqueles em que a grande maioria dos seus elementos são não nulos e todos os seus elementos, incluindo os nulos, são armazenados. Nas operações envolvendo tais matrizes e vetores

todos os elementos da matriz ou vetor são utilizados nos cálculos.

Quando o número de elementos for pequeno, as operações envolvendo vetores não esparsos (denominados também de vetores cheios) são bastante eficientes. Para números elevados de elementos, a eficiência cai drasticamente e novas representações de vetores devem ser desenvolvidas.

Com o objetivo de implementar o reuso de objetos temporários, foram desenvolvidas nesta tese dois gabaritos de classes: *TVector* e *tmpTVector*. A declaração simplificada da classe *TVector* é apresentada na Figura 2-5.

```
1 template <typename T>
2 class TVector
3 {
4    friend tmpTVector<float> real(TVector<float> &v);
.....................................................
5    friend tmpTVector<float> real( const tmpTVector<float> &v);
.....................................................
6    friend
7    tmpTVector<float> operator + (TVector<float> &lhs,
8                                  TVector<float> &rhs);
.....................................................
9    friend
10   tmpTVector<float> operator + (TVector<float> &lhs,
11                                 const tmpTVector<float> &rhs);
.....................................................
12   friend
13   tmpTVector<float> operator + (const tmpTVector<float> &lhs,
14                                 TVector<float> &rhs);
.....................................................
15   friend
16   tmpTVector<float> operator + (const tmpTVector<float> &lhs,
17                                 const tmpTVector<float> &rhs);
.....................................................
18   TVector();
19   TVector(int len);
20   TVector(TVector &v);
21   TVector(const tmpTVector<T> &v);
```

```
22   ~TVector();
23   void operator = (TVector<float> &v);
.....................................................
24   void operator = (const tmpTVector<float> &v);
.....................................................
};
```

Figura 2-5 - Declaração simplificada das classes para vetores não esparsos

Ao obervar a Figura 2-5, nas linhas 4 e 5 são apresentadas duas funções não membro que retornam um vetor temporário representado pela classe *tmpTVector<float>*. Outra característica desta função é possuir duas versões com lista de parâmetros diferentes, um com a classe *TVector* e outra com a classe *tmpTVector* (linha 5). Todas as funções com um único parâmetro seguem este padrão.

Para as funções com dois parâmetros como, por exemplo, da sobrecarga do operador "+", apresentadas nas linhas 9 a 17 da Figura 2-5, também retornam um objeto da classe *tmpTVector* observa-se também que existem quatro versões da função de sobrecarga cobrindo as combinações possíveis de tipos para os parâmetros.

Nas linhas 20 e 21 da Figura 2-5, pode-se observar que existem duas versões para o construtor de cópia. O construtor de cópia da linha 20 é o construtor utilizado nas operações envolvendo a cópia do conteúdo das áreas de memória do vetor passado como argumento (dependendo do tamanho do vetor passado no argumento pode ser uma operação demorada). O construtor na linha 21 é utilizado nas operações envolvendo a cópia presumida, onde o vetor passado no argumento transfere suas áreas de memória copiando apenas os seus ponteiros (portanto, é uma operação extremamente rápida). Na Figura 2-6 são apresentados os códigos fontes destes dois construtores.

```
1  template <typename T>
2  TVector<T>::TVector(TVector<T> &v)
3  {
4   if (values) delete values;
5   values = new T [v.length];
6   memcpy(this->values,v.values,v.length*sizeof(T));
7   length = v.length;
8   base = v.base;
9   erase = true;
10 }

11 template <typename T>
```

```
12 TVector<T>::TVector(const tmpTVector<T> &v)
13 {
14  if (values) delete values;
15  length = v.length;
16  values = v.values;
17  base = v.base;
18  erase = true;
19 }
```

Figura 2-6 - Códigos fontes dos construtores de cópia de conteúdo e presumida

Ao observar a Figura 2-6, pode-se verificar que na cópia de conteúdo, localizado nas linhas de 1 a 11, possui os seguintes passos:

1. Caso houver memória alocada para o objeto alvo (this) então desalocá-la e alocar uma nova (linhas 4 e 5).
2. Em seguida na linha 6 é feita a cópia do conteúdo das áreas de memória do objeto passado como argumento para o objeto alvo. Esta operação pode ser extremamente demorada.

No construtor de cópia presumida as etapas são:

1. Caso haja memória alocada para o objeto alvo então desalocar, linha 14;
2. Copiar apenas o ponteiro da área de memória usada pelo objeto passado como argumento, linha 16;

Os construtores de cópia das demais classes, matrizes densas, vetores esparsos e matrizes esparsas, devem seguir o mesmo modelo dos construtores apresentados nesta seção para a classe de vetores não esparsos *TVector*.

Os operadores de atribuição seguem a mesma lógica dos construtores de cópia. Na Figura 2-5 na linha 23 está à declaração do operador de atribuição em que será feita uma cópia de conteúdo e na linha 24 é feita uma atribuição com cópia presumida. Na Figura 2-7 são apresentados os códigos fontes destas funções.

```
1  template <>
2  void TVector<float>::operator = (TVector<float> &v)
3  {
4   if (length!=v.length)
5   {
6    if (values) delete values;
7    values = new float[v.length];
8    length = v.length;
9   }
```

```
10  base = v.base;
11 #if defende(_MKL) || defende(_ATLAS_)
12  cblas_scopy(length,v.values,1,values,1);
13 #else
14  memcpy(values,v.values,length*sizeof(float));
15 #endif
16 }

17 template <>
18 void TVector<float>::operator = (const tmpTVector<float> &v)
19 {
20  if (values) delete values;
21  values = v.values;
22  length = v.length;
23  base = v.base;
24 }
```

Figura 2-7 - Sobrecarga do operador de atribuição

Na Figura 2-7 pode-se verificar que o primeiro passo da rotina de atribuição por cópia de conteúdo, nas linhas de 1 até 16. Caso o objeto alvo possui o mesmo número de elementos do objeto passado no argumento, a memória já utilizada é mantida, caso seja diferente a memória será realocada. Em seguida é feita a cópia dos conteúdos das áreas de memória dos objetos passados para o objeto alvo nas linhas de 11 a 15. Pode-se verificar que há diretivas do pré-processador para definir quais os trechos de código serão compilados. Caso o Framework estiver configurado para usar a MKL ou a ATLAS, o código da linha 12 será compilado e executado, onde uma função BLAS é chamada. Caso contrário o código a ser compilado e executado é o da linha 14. O operador de atribuição com cópia presumida faz a cópia apenas dos ponteiros. Assim como no caso dos construtores de cópia, as funções de sobrecarga do operador de atribuição das demais classes de matrizes e vetores do Framework desenvolvido nesta tese seguem este padrão.

Com relação a funções do tipo não membro, também existe uma forma padrão para a sua implementação. Como exemplos, serão analisadas a funções de sobrecarga do operador "+", cuja declaração das suas versões encontram-se nas linhas de 6 até 17 da Figura 2-5. Na Figura 2-8 são apresentados os códigos fontes destas funções.

```
1  tmpTVector<float> operator + (TVector<float> &lhs,
2                                TVector<float> &rhs)
```

```
3  {
4   tmpTVector<float> temp(lhs.length);
5   memcpy(temp.values,lhs.values,lhs.length*sizeof(float));
6   temp+=rhs;
7   return temp;
8  }

9  tmpTVector<float> operator + (TVector<float> &lhs,
10                                const tmpTVector<float> &rhs)
11 {
12  tmpTVector<float> temp(rhs);
13  temp+=lhs;
14  return temp;
14 }

15 tmpTVector<float> operator + (const tmpTVector<float> &lhs,
16                                TVector<float> &rhs)
17 {
18  tmpTVector<float> temp(lhs);
19  temp+=rhs;
20  return temp;
21 }

22 tmpTVector<float> operator + (const tmpTVector<float> &lhs,
23                                const tmpTVector<float> &rhs)
24 {
25  tmpTVector<float> temp(lhs);
26  temp+=rhs;
27  return temp;
28 }
```

Figura 2-8 - Códigos fontes das funções de sobrecarga do operador "+"

Na Figura 2-8 deve-se observar que, em cada função, foram declarados objetos *temp*, da classe *tmpTVector*, que serão processados e retornados pelas funções. Ao observar à criação do objeto *temp* nas linhas 12, 18 e 25 é usado o construtor de cópia presumida. Na linha 4 foi utilizado o uma cópia de conteúdo. Todas as funções

com dois parâmetros devem seguir esta formatação.

O gabarito de classe que representa os vetores temporários é denominada de *tmpTVector*. Sua declaração é apresentada na Figura 2-9.

```
1  template<typename T>
2  class tmpTVector: public TVector<T>
3  {
4   public:
5     tmpTVector();
6     tmpTVector(int len);
7     tmpTVector(const tmpTVector &v);
8     ~tmpTVector();
9     void operator = (TVector<T> &v);
10    void operator = (const tmpTVector &v);
11 };
```

Figura 2-9 - Declaração da classe tmpTVector

Para completar a descrição da aplicação da técnica de reaproveitamento de objetos temporários é necessário apresentar os construtores das linhas 6 e 7 da Figura 2-9. Os códigos fontes dos construtores são apresentados na Figura 2-10.

```
1  template<typename T>
2  tmpTVector<T>::tmpTVector(int len)
3  :TVector<T>(len)
4  {
5    this->erase = false;
6  }


7  template<typename T>
8  tmpTVector<T>::tmpTVector(const tmpTVector<T> &v)
9  :TVector<T>(v)
10 {
11   this->erase = false;
```

```
12 }
```

Figura 2-10 - Código fonte dos construtores de cópia

Ao examinar os códigos fontes na Figura 2-10, pode-se observar que os objetos do parâmetro de cada construtor são repassados aos construtores da classe ancestral (*TVector*), e a um membro chamado "*erase*" é atribuído o valor *false* para impedir que o destrutor libere a memória usada. As funções de sobrecarga do operador de atribuição segue o padrão apresentado anteriormente para a classe *TVector*.

2.3.5.2 Exemplo de utilização do gabarito de classe TVector

Para utilizar o gabarito *TVector* em um programa, deve-se incluir o arquivo de cabeçalho *bcssd.h* e declarar os objetos indicando qual a especialização desejada. O código na Figura 2-11 apresenta um programa exemplo.

```
1  #include <iostream>
2  #include <bcssd/core/include/bcssd.h>
3  using namespace std;
4  int main() {
5    bcssd::TVector<double> A(2), B(2), C(2);
6    A[0]=1.0; A[1]=2.0;
7    B = 2.0*A;
8    C = 2.0*A+B;
9    cout << "C = \n" << C << endl;
10   return 0;
11 }
```

Figura 2-11 - Exemplo de uso

Na linha 5 da Figura 2-11 é feita a declaração de três objetos, observa-se o uso de "<*double*>" para indicar que a especialização para números reais de precisão dupla será usada. Na linha 6 é feito atribuição de valores aos elementos do vetor e nas linhas 7 e 8 são feitas algumas operações.

2.3.5.3 Comentários

Esta seção apresentou a técnica de reaproveitamento de valores temporários utilizando a classe de vetores não esparsos para ilustrar esta implementação. Esta técnica foi aplicada também nas classes *TMatrix* para matrizes densas, *TSparseVector* para vetores esparsos e *TSparseMatrix* para matrizes esparsas.

2.3.6 Matrizes não esparsas

Matrizes não esparsas são aquelas em que todos os seus elementos são armazenados. A forma de implementação segue o padrão apresentado na seção 2.3.5. As matrizes não esparsas foram implementadas

através dos gabaritos de classes *TMatrix* e *tmpTMatrix*. Na Figura 2-12 é apresentado um programa exemplo que ilustra a utilização de algumas funcionalidades da biblioteca desenvolvida para tratamento de vetores e matrizes.

```
1  #include <bcssd/core/include/bcssd.h>
2  int main () {
3     bcssd::TMatrix<double> A(2,2), B(2,2), C(2,2);
4     unsigned int i,j;
5     for (i = 0; i < A.getNumRow(); ++i)
6     {
7        for (j = 0; j < A.getNumCol(); ++j)
8        {
9           A(i,j) = i+j;
10          B(i,j) = 0.5*i-j;
11       }
12    }
13    C = 0.5*A + B;
14    std::cout<<"Resultado: \n"<<C<<std::endl;
15 }
```

Figura 2-12 - Exemplo de programa que utiliza matrizes

Como pode ser observado na Figura 2-12, os objetos são declarados na linha 3, onde são definidos os números de linhas e colunas das matrizes. Nas linhas 9 e 10 são atribuídos valores aos elementos de duas matrizes. Na linha 13 é feita uma operação entre estas matrizes.

2.4 Esparsidade

Em estudos de sistemas elétricos de potência, é comum a necessidade de se trabalhar com matrizes e vetores com dimensões muito grandes. Isso demanda elevada capacidade de memória e de tempo de CPU para o processamento dos dados. No entanto, tais matrizes e vetores são geralmente esparsos, ou seja, apresentam um elevado percentual de elementos nulos, recebendo a denominação de matrizes e vetores esparsos.

Define-se como grau de esparsidade a relação entre o número de elementos não nulos, *nnz*, e o número máximo de elementos que o vetor ou a matriz poderiam ter, *total*: $e = \frac{nnz}{total}$.

Uma estratégia eficiente para reduzir a quantidade de memória necessária é armazenar apenas os valores não nulos. Havendo muito menos elementos armazenados, a quantidade de memória e o tempo de processamento despendido nas operações pode ser reduzido consideravelmente.

2.4.1 Vetores esparsos

2.4.1.1 Armazenamento dos elementos

Os vetores esparsos são aqueles em que o número de elementos não nulos é muito inferior ao número de elementos nulos. Como o número de elementos que farão parte do processamento é muito menor, então obtêm-se um ganho de velocidade muito grande. No entanto, este ganho de desempenho é diminuído em função da necessidade de *overheads* relacionados ao armazenamento e recuperação dos dados, portanto, para vetores pequenos a utilização das técnicas de vetores esparsos é contra - indicado.

A princípio, o que se precisa armazenar em um vetor esparso são os elementos não nulos, armazenados em um vetor convencional denominado "*sa*" e seus respectivos índices dos elementos em um vetor convencional denominado "*ija*". Considerando como exemplo o vetor na expressão (2.2).

$$v = \begin{bmatrix} 0 \\ -1{,}25 \\ 0 \\ 0 \\ 0 \\ 2{,}5 \end{bmatrix} \tag{2.2}$$

é armazenado como:

$$sa = \begin{bmatrix} -1{,}25 & 2{,}5 \end{bmatrix} \text{ e } ija = \begin{bmatrix} 2 & 6 \end{bmatrix} \tag{2.3}$$

Neste caso o menor índice é "1".

O número de elementos de "*sa"* e "*ija"* é igual ao número de elementos não nulos no vetor, que será representado pelo símbolo "*numnzelem"*.

Como se pode intuir, as operações de subscrito podem ser bastante lentas, portanto, sempre que possível devem ser evitadas. Para agilizar as operações de busca de elementos, necessariamente *ija* deve estar ordenado de forma crescente. Os gabaritos de classes que representam os vetores esparsos são ilustrados través da Figura 2-13.

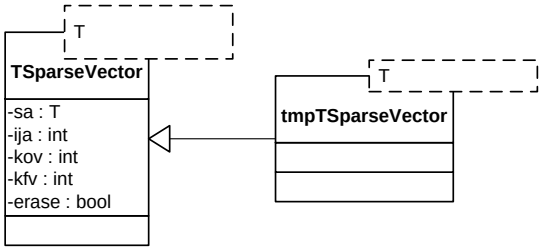

Figura 2-13 - Representação em UML dos gabaritos de classes de vetores esparsos

2.4.1.2 Subvetores esparsos

O conceito de *subvetores esparsos* foi criado no Framework desenvolvido nesta tese devido à necessidade de evitar cópias desnecessárias de áreas de memória. Os subvetores são usados quando há a necessidade de obtenção de um vetor esparso constituído por uma faixa de elementos de outro vetor esparso maior. Outra aplicação é na necessidade de obtensão de uma linha ou coluna de uma matriz esparsa. Considerando a Figura 2-14, onde é ilustrada a necessidade de obter um subvetor B a partir de um vetor maior com mais elementos A.

Através da Figura 2-14 pode-se verificar que o sub-vetor *B* possui a mesma dimensão do vetor original *A,* mas com os elementos nas posições 3 e 13 excluídos.

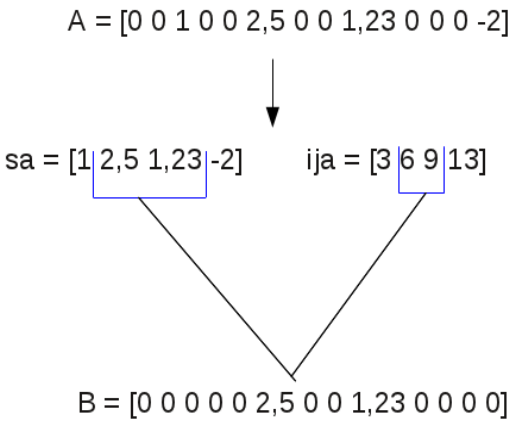

Figura 2-14 - Obtenção de um subvetor

Desta forma, para que seja implementado o conceito de subvetor esparso, é necessário armazenar a posição inicial em *sa* e *ija* que será representada pelo símbolo *kov* e a posição final que será representado pelo símbolo *kfv*. Para este exemplo deve-se fazer:

B.sa = A.sa;

B.ija = A.ija;

B.kov = 2;

B.kfv = 3;

2.4.1.3 Operações de subscrição

As operações de subscrição são aquelas em que se têm acesso a um determinado elemento para ler ou escrever. No gabarito de classes *TSparseVector* existem duas sobrecargas, uma para recuperação de um elemento e outra para escrever um elemento.

Operador para recuperação de um elemento

Este operador retorna o valor de um elemento para o índice informado como argumento. Por exemplo, em "*A(1)*". A declaração deste operador de subscrição é:

```
T operator()(int indx) const;
```

Onde "*indx*" é o índice no vetor.

Nesta rotina é feita uma busca em *ija* para verificar em que posição encontra-se o elemento e em seguida retorná-lo de *sa*. Caso não seja encontrado o índice, o valor zero é retornado.

Operador para escrita de um elemento

Este operador possui a seguinte declaração:

```
void operator()(int indx, T value);
```

Onde "*indx*" é o índice no vetor e "*value*" é o valor a ser armazenado.

Nesta rotina, a primeira atividade é encontrar em *ija* se consta o índice igual a *indx*. Caso afirmativo o valor passado para a rotina vai substituir o valor correspondente em *sa*. Caso não seja encontrado, é feito uma busca em *ija* pelo índice cujo valor seja o inferior mais próximo de *indx*. Em seguida *ija* e *sa* são redimensionadas e os seus elementos movidos para acomodar o novo elemento. Sendo, portanto, uma operação que pode ser lenta.

2.4.1.4 Inicialização de vetores esparsos

A inicialização de vetores esparsos começa pela criação de uma instância do mesmo. O construtor a ser utilizado com esta finalidade é declarado como:

```
TSparseVector(int len, int ne)
```

Onde *len* é o número de posições do vetor esparso e *ne* é o número máximo de elementos não nulos.

Exemplo:

```
bcssd::TSparseVector<double> A(3500, 100);
```

Onde *A* é um vetor de 3500 posições sendo que apenas 100 elementos podem ser não nulos. Para uma inicialização rápida dos elementos deste vetor, devem ser utilizados os métodos *Position* e *Element*. A declaração de ambos está na Figura 2-15.

```
1  int &Position(int p)
2  {
3    return ija[p];
4  }

5  T &Element (int p)
6  {
7    return sa[p];
8  }
```

Figura 2-15 - Métodos usados para inicializar o vetor esparso

Onde *p* é a posição em *sa* e *ija*. Para ilustrar a inicialização de um vetor esparso, pode-se empregar como o exemplo o vetor na expressão (2.2). Na Figura 2-16 encontra-se o fragmento de código em que a inicialização do conteúdo é feita da forma rápida.

```
1  #include <bcssd/core/include/bcssd.h>
2  void main ()
3  {
4     bcssd::TSparseVector<double> A(6,2);
5     A.Position(1) = 2;
6     A.Position(2) = 6;
7     A.Element(1) = -1.25;
8     A.Element(2) = 2.5;
9     A.End() = 2;
10    std::cout<<"A = \n"<<A<<std::endl;
11 }
```

Figura 2-16 - Inicialização rápida do conteúdo de um vetor esparso

Na Figura 2-16, na linha 4 é feita a criação da instância *A* do vetor. Nas linhas 5 e 6 são definidos os índices dos elementos não nulos, deve-se notar que a indexação começa no um. Nas linhas 7 e 8 os valores dos elementos são definidos. Na linha 9 é informado ao objeto que a posição (não o índice) do último elemento não nulo é dois (*sa*[2] e *ija*[2]). Os métodos *Element* e *Position* escrevem e leem diretamente de *sa* e *ija*, sem haver deslocamentos de grandes blocos de memória, portanto, é o meio mais eficiente de inicializar o conteúdo de um vetor esparso.

2.4.2 Matrizes esparsas

2.4.2.1 Armazenamento dos elementos

Analogamente aos vetores esparsos, as matrizes esparsas são aquelas onde o número de elementos não nulos é muito inferior ao número total de elementos da matriz. Ao armazenar apenas os elementos não nulos, obtêm-se ganhos significativos em termos de economia de memória e de tempo de processamento.

Um exemplo de matriz esparsa é mostrado através da expressão (2.4):

$$A = \begin{bmatrix} 1 & 0 & 0 & 2{,}5 & 0 & 0 \\ 0 & -1{,}25 & 0 & 0 & 0 & -0{,}5 \\ 0 & 0 & -2{,}5 & 0 & 1{,}1 & 0 \\ 0 & 0 & 0 & -0{,}34 & 0 & 0 \\ 1{,}1 & 0 & 0 & 0 & 1{,}2 & 0 \\ 0 & -1{,}5 & 0 & 0 & 0 & -0{,}25 \end{bmatrix} \tag{2.4}$$

Nos gabaritos de classes de matrizes esparsas, denominados *TSparseMatrix* e *tmpTSparseMatrix*, os valores não nulos são armazenados em "*sa*". Para o caso da matriz utilizada como exemplo o vetor *sa* conterá:

1	2	3	4	5	6	7	8	9	10	11
sa = [1	2,5	-1,25	-0,5	-2,5	1,1	-0,34	1,1	1,2	-1,5	-0,25]

Em *ija* serão armazenados os índices de cada coluna ao qual o elemento correspondente em *sa* pertence. Portanto, para exemplo utilizado *ija* conterá:

$$ija = [1;\ 4;\ 2;\ 6;\ 3;\ 5;\ 4;\ 1;\ 5;\ 2;\ 6] \tag{2.5}$$

Além de armazenar o índice das colunas onde os elementos estão localizados, é necessário armazenar alguma referência às linhas. Esta referência é armazenada em *index* onde cada posição corresponde a uma linha da matriz esparsa e um elemento a mais que armazena o número de elementos não nulos da matriz mais um. Em cada posição de *index*, com exceção da última posição, é armazenada a posição em *sa* do primeiro elemento não nulo da linha correspondente. Portanto, para este exemplo:

Linha→	[1	2	3	4	5	6]	
index =	[1	3	5	7	8	10	12]

Para saber o número de elementos não nulos de uma determinada linha *n* basta realizar uma subtração *index[n+1]-index[n]*.

2.4.2.2 Operações de subscrição de elementos

Para determinar o número da linha e da coluna de um determinado elemento também é simples, por exemplo, o elemento cujo valor é 2,5, pode-se verificar em *sa* que ele corresponde à posição dois então ao examinar *ija* verifica-se que a coluna correspondente é quatro. Ao examinar *index* verifica-se que este elemento só pode estar na linha um, visto que o primeiro elemento não nulo da linha dois é três e da linha um é um. Como se pode verificar, a busca individual por elementos em uma matriz esparsa é uma busca sequencial e, portanto, para várias aplicações pode ser lento.

São disponibilizados dois métodos: um para recuperação de elementos e outro para introduzir ou modificar elementos, cujas declarações estão na Figura 2-17.

```
T operator ()(int row,  int col) const;       //Subscript operator for get
void operator ()(int row,  int col, T value);//Subscript operator for set
```

Figura 2-17 - Declaração dos operadores de subscrição de elementos

Onde *row*, *col* e *value* são respectivamente linhas, colunas e valores.

2.4.2.3 Inicialização de matrizes esparsas

Da mesma forma que em vetores esparsos, a inicialização utilizando os operadores de subscrição não é recomendada.

Para inicializar uma matriz esparsa, o primeiro passo consiste em obter uma instância de matriz esparsa através do construtor:

TSparseMatrix (**int** nr, **int** nc, **int** nnz)

Onde *nr*, *nc* e *nnz* são: o número de linhas, número de colunas e o número máximo de elementos não nulos. Por exemplo:

bcssd::TSparseMatrix<double> A(6, 6, 11);

Na Figura 2-18 é apresentado um exemplo em que uma matriz esparsa é inicializada da forma recomendada.

```
1  #include <bcssd/core/include/bcssd.h>
2  int main ()
3  {
4    bcssd::TSparseMatrix<double> A(6,6,11);
5    A.Element(1) = 1;
```

```
6     A.Element(2) = 2.5;
7     A.Element(3) = -1.25;
8     A.Element(4) = -0.5;
9     A.Element(5) = -2.5;
10    A.Element(6) = 1.1;
11    A.Element(7) = -0.34;
12    A.Element(8) = 1.1;
13    A.Element(9) = 1.2;
14    A.Element(10) = -1.5;
15    A.Element(11) = -0.25;
16    A.Column(1) = 1;
17    A.Column(2) = 4;
18    A.Column(3) = 2;
19    A.Column(4) = 6;
20    A.Column(5) = 3;
21    A.Column(6) = 5;
22    A.Column(7) = 4;
23    A.Column(8) = 1;
24    A.Column(9) = 5;
25    A.Column(10) = 2;
26    A.Column(11) = 6;
27    A.BeginRow(1) = 1;
28    A.BeginRow(2) = 3;
29    A.BeginRow(3) = 5;
30    A.BeginRow(4) = 7;
31    A.BeginRow(5) = 8;
32    A.BeginRow(6) = 10;
33    A.BeginRow(7) = 12;
34    std::cout<<"A = \n"<<A<<std::endl;
35    return 0;
36 }
```

Figura 2-18 - Inicialização de uma matriz esparsa

Na linha 4 da Figura 2-18 é declarada uma matriz esparsa de precisão dupla (double) 6x6 com onze elementos não nulos. Nas linhas de 5 até 15 são armazenados os elementos não nulos, ou seja, *sa* é preenchida.

Das linhas 16 até 26 são informadas as colunas de cada elemento não nulo, ou seja, *ija* é preenchido. Finalmente nas linhas de 27 até 33 são armazenados os índices dos primeiros elementos não nulos de cada linha, ou seja, *index* é preenchido. Na linha 34 é exibida a matriz esparsa para verificar se tudo ocorreu corretamente.

2.5 Conclusão

Neste capítulo foram apresentados os gabaritos de classes para matrizes e vetores não esparsos e esparsos. Estes gabaritos de classes são fundamentais no Framework desenvolvido nesta tese, pois eles fornecem a base computacional necessário para a implementação de todas as outras classes e rotinas numéricas. Foram apresentadas algumas técnicas utilizadas que tentam minimizar a perda de desempenho inerente ao uso de linguagens orientadas a objetos.

A quantidade de rotinas e técnicas utilizadas na implementação das funcionalidades referentes a matrizes e vetores esparsos e não esparsos é muito grande, não havendo espaço para abordá-las completamente. Portanto, recomenda-se fortemente que seja consultada a literatura referenciada para que mais informações sejam obtidas, por exemplo, (Anderson E. , et al., 1999), (Blackford, et al., 1996-2000), (Blackfordy, et al., 1996), (Choi, Dongarra, & Walker, 1994) e (Crow, 2003).

3 Modelagem e implementação orientada a objetos para execução de fluxo de carga

3.1 Modelagem orientada a objetos da rede de transmissão

3.1.1 Definição das classes e suas relações

A rede de transmissão possui várias entidades que devem ser modeladas como classes orientadas a objetos com relações muito bem definidas. Para os objetivos do Framework desenvolvido nesta tese de doutorado, foram definidas as seguintes entidades:

1. **Barras**. Entidade que representa as barras do sistema. Podem ser de três tipos básicos: referência, PV e PQ, no entanto a modelagem matemática destes tipos de barras é idêntica e não se justifica a criação de uma classe para cada tipo de barra, desta forma, então, define-se a classe *TBus* que define as barras existentes em uma rede elétrica. Uma barra pode estar associada a uma ou nenhuma barra controlada remotamente e a um ou vários ramos chegando ou partindo da barra.

2. **Modelos de carga**. Entidade que representa os modelos de carga variantes com a tensão. Um modelo de carga pode estar associado a uma ou mais barras para modelar as cargas presentes na mesma. A classe que representa esta entidade é denominada *TLoad*.

3. **Ramos**. Entidade que representa os ramos de circuito da rede, ou seja, dispositivos que conectam duas barras. Os ramos podem ser linhas de transmissão, transformadores, compensadores série, etc. Apesar da variedade de dispositivos, nesta tese, todos eles são representados pelo mesmo modelo matemático o que não justifica a criação de classes específicas para cada um destes tipos de ramos. A criação de várias classes específicas para cada tipo de ramo foi evitado nesta tese, então, definiu-se uma única classe denominada *TBranch* para representar os ramos da rede de elétrica. Um ramo sempre está associado a uma barra de origem (*from bus*) e a uma barra de destino (*to bus*). Um ramo pode estar associado a uma ou nenhuma barra controlada remotamente por ajuste automático de tap (*control bus*).

4. **Área.** Esta entidade representa a área de concessão de uma companhia e possui uma ou várias barras de fronteira, uma ou várias barras internas e uma barra de folga.

5. **Rede de transmissão**. Esta entidade representa a rede de transmissão completa. Ela possui uma ou muitas barras, uma ou muitos ramos, modelos de cargas, áreas, etc. A rede é representada no Framework pela classe *TNetwork.* A classe *TNetwork* é a responsável por todos os cálculos relacionados ao fluxo de carga e análise de redes. Dentre seus atributos estão: as tolerâncias ao erro de potências ativas e reativas e o número máximo de iterações do método de cálculo de fluxo de carga. Dentre suas funcionalidades estão o cálculo da matriz admitância de rede, cálculo da matriz jacobiana, implementação do método de Newton-Raphson, etc.

O diagrama de classes onde se pode visualizar as relações entre estas diversas classes é apresentado através da Figura 3-1.

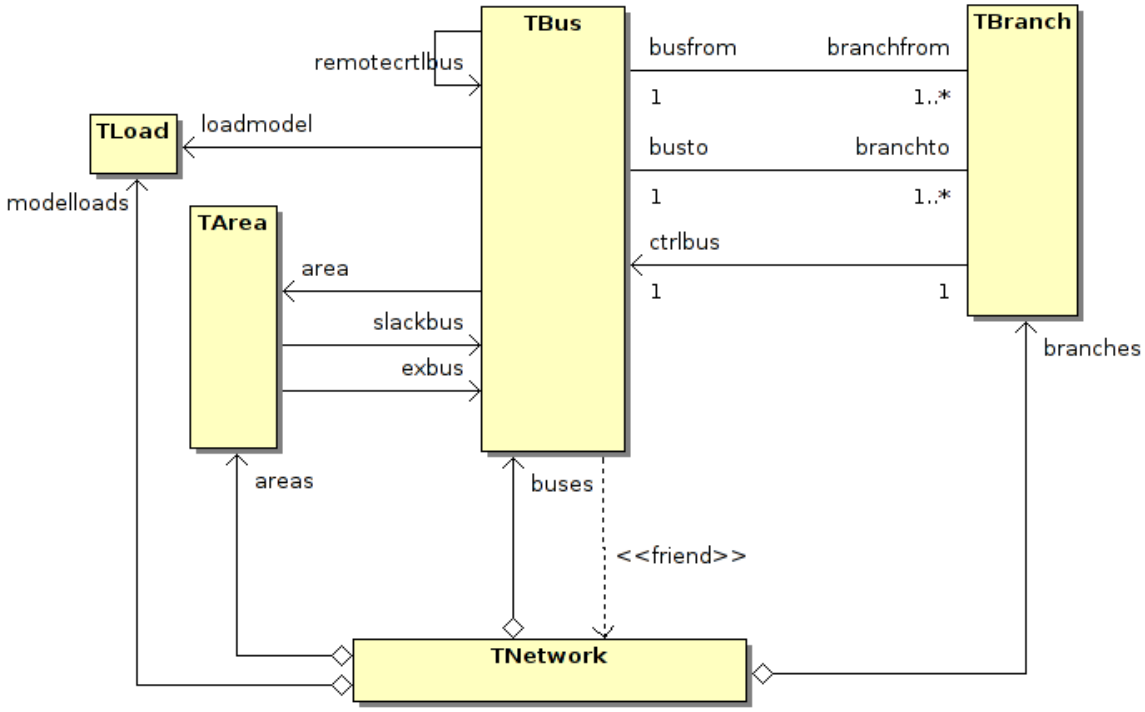

Figura 3-1 - Diagrama de classes que presentão a rede elétrica

Na Figura 3-1 pode-se verificar que as barras possuem uma relação unidirecional que parte dela para ela mesma que representa as barras controladas remotamente por injeção de reativos (*remotecrtlbus*). As barras também estão relacionadas a vários ramos que partem dela (*branchfrom*) e vários ramos que chegam nela (*branchto*). As barras estão relacionadas a apenas uma área (*area*). Outra relação é que uma barra está associada a apenas um único modelo de carga (*loadmodel*).

Pode-se verificar, na mesma figura, que um ramo está associado a apenas uma barra de origem *(busfrom*) e a uma única barra de destino (*busto*). Ela está associada a uma ou nenhuma barra controlada remotamente (*ctrlbus*).

Uma área está associada a uma ou várias barras de fronteira (*exbus*) e a apenas uma barra de folga (*slackbus*).

Todos as classes estão relacionadas à classe *TNetwork* através de relações de agregação, ou seja, todos os dispositivos devem ser cadastrados em uma instância da classe *TNetwork*.

3.1.2 A classe que representa as barras

No decorrer da explanação teórica a respeito das barras e suas relações com outras entidades pode-se definir que a classe que a representa, *TBus*, possui os seguintes atributos:

Atributo	Tipo (C++)	Descrição
name	String	Nome de identificação da barra
ID	unsigned int	Número que inicia em um que estabelece a ordem da barra dentro da rede. Normalmente, a barra de referência recebe o número um, em seguida as barras PV são numeradas e por último as barras PQ.
type	TBusType	Valor enumerado que identifica o tipo da barra. Para barra de referência recebe o valor *REFBUS*, para barras PV recebe *PVBUS* e para barras PQ recebe *PQBUS*.
Vk	double	Módulo da tensão da barra em pu da barra.
phasek	double	Fase da tensão da barra em graus.
Pk	double	Potência ativa líquida em MW.
Qk	double	Potência reativa líquida em MVAr.
Pgk	double	Potência ativa gerada na barra em MW.
Qgk	double	Potência reativa gerada na barra em MVAr.
Plk	double	Potência ativa de carga na barra em MW.
Qlk	double	Potência reativa de carga na barra em MVAr.
bshk	double	Susceptância shunt de barra em MVAr.
Vmax	double	Limite superior do módulo da tensão da barra em pu na base da barra.
Vmin	double	Limite inferior do módulo da tensão da barra em pu na base da barra.
Qmax	double	Limite superior da potência reativa em MVAr.

Qmin	double	Limite inferior da potência reativa em MVAr.
Vesp	double	Tensão especificada da barra controlada em pu da barra controlada.
Sbase	double	Potência base da barra em MVA.
Vbase	double	Tensão base da barra em kV.

Tabela 3-1 - Atributos da classe TBus

Para criar uma instância de uma barra, utiliza-se um dos métodos estáticos:

```
TBus *CreateRefBus(name,bshk,Vk,Phk,Pgk,Qgk,Plk,Qlk,ml,ctrl,esp,area,f,Sbase,Vbase);

TBus *CreatePVBus(name,bshk,Vk,Phk,Pgk,Qgk,Plk,Qlk,ml,ctrl,esp,area,f,Sbase,Vbase);

TBus *CreatePQBus(name,bshk,Vk,Phk,Pgk,Qgk,Plk,Qlk,ml,ctrl,esp,area,f,Sbase,Vbase);
```

Onde os argumentos são:

Argumento	Tipo (C++)	Descrição
name	string	Nome de identificação da barra.
bshk	double	Susceptância shunt da barra em MVAr.
Vk	double	Módulo da tensão em pu da barra.
Phk	double	Fase da tensão em graus.
Pgk	double	Potência ativa gerada em MW.
Qgk	double	Potência reativa gerada em MVAr.

Plk	double	Potência ativa de carga em MW.
Qlk	double	Potência reativa de carga em MVAr.
ml	TLoad *	Ponteiro para o modelo de carga.
ctrl	TBus *	Ponteiro para a barra controlada. Caso não houver barra a ser controlada, fazer este argumento igual a NULL.
esp	double	Tensão especificada para a barra controlada. Caso ctrl for NULL este argumento não é significativo.
area	TArea *	Ponteiro para a área ao qual esta barra pertence.
f	TAreaFunc	Enumeração que identifica qual a função da barra dentro da área. Pode ser *INBUS* caso seja uma barra interna ordinária, *SLACKBUS* caso seja uma barra de folga e *EXBUS* se for uma barra de intercâmbio.
Sbase	double	Potência base em MVA da barra.
Vbase	double	Tensão base em kV da barra.

Tabela 3-2 - Argumentos dos métodos de criação de instâncias de barras

3.1.3 Classe que representa modelos de cargas

Um modelo de carga pode ser empregado em uma ou muitas barras na rede de transmissão. No Framework ela representada pela classe *TLoad*.

Os atributos desta classe correspondem ao modelo geral das cargas variantes com a tensão cuja modelagem encontra-se na referência (Monticelli, 1983) e mais um nome que identifica o modelo de carga. Para criar um modelo de carga utiliza-se o método estático:

```
TLoad *CreateLoadModel(n,_ap,_bp,_cp,_aq,_bq,_cq);
```

Onde os argumentos são:

Argumento	Tipo (C++)	Descrição
n	string	Nome de identificação do modelo de carga.

_ap	double	a_p
_bp	double	b_p
_cp	double	c_p
_aq	double	a_q
_bq	double	b_q
_cq	double	c_q

Tabela 3-3 - Argumentos do método de criação de instâncias da classe *TLoad*

3.1.4 Classe que representa ramos

Os ramos são os dispositivos que conectam duas barras na rede. A classe que representa os ramos é a classe *TBranch*. A classe que representam os ramos de circuito possui os seguintes atributos:

Atributo	Tipo (C++)	Descrição
branchname	string	Nome do ramo.
akm	double	Módulo do fator de transformação.
phikm	double	Fase do fator de transformação.
rkm	double	Resistência série do ramo em pu da linha.
xkm	double	Reatância série do ramo em pu da linha.
bshkm	double	Susceptância shunt do modelo da linha em pu da linha.
Pkm	double	Fluxo de potência ativa no sentido k-m em MW.
Qkm	double	Fluxo de potência reativa no sentido k-m em MW.
Pmk	double	Fluxo de potência ativa no sentido m-k em MW.

Qmk	double	Fluxo de potência reativa no sentido m-k em MW.
Pl	double	Perda de potência ativa em MW.
Ql	double	Potência reativa armazenada em MVAr.
Sbase	double	Potência base em MVA do ramo.
Vbase	double	Tensão base em kV do ramo.
Zbase	double	Impedância base do ramo.
Pmax	double	Limite superior de potência ativa do ramo.
Qmax	double	Limite superior de potência reativa do ramo.

Tabela 3-4 - Atributos da classe *TBranch*

Para criar instâncias da classe *TBranch* utiliza-se o método estático:

```
TBranch * CreateBranch(name,from,to,akm,phikm,rkm,xkm,bshkm,ctrl,esp,bcrtl,
                       Sbase,Vbase);
```

Os argumentos do método de criação de um ramo são:

Argumento	Tipo (C++)	Descrição
name	string	Nome do ramo.
from	TBus *	Ponteiro para a barra origem.
to	TBus *	Ponteiro para a barra destino.
akm	double	Módulo do fato de transformação em pu.
phikm	double	Fase do fator de transformação em graus.
rkm	double	Resistência série do ramo em pu do ramo.

xkm	double	Reatância série do ramo em pu do ramo.
bshkm	double	Susceptância shunt do ramo em pu do ramo.
ctrl	TControlType	Enumeração que indica o tipo de controle a ser realizado pelo transformador. Caso seja *BRANCHVOLT* o controle será do módulo da tensão por ajuste do tap em transformado em fase. Caso seja *BRANCHFLOW* o controle será do fluxo de potência onde o transformador está inserido através do ajuste da fase do tap do transformador defasador. Caso *BRANCHNONE* nenhum controle será realizado.
esp	double	Caso ctrl = *BRANCHVOLT*, esp é o valor da tensão especificada para a barra remota. Caso seja *BRANCHFLOW* esp é o valor do fluxo de potência ativa especificada.
bctrl	TBus *	Barra controlada remotamente se ctrl = *BRANCHVOLT* caso contrário, este argumento pode receber o valor NULL.
Sbase	double	Potência base em MVA do ramo.
Vbase	double	Tensão base em kV do ramo.

Tabela 3-5 - Argumentos do método de criação de instâncias da classe *TBranch*

3.1.5 Classe que representa as áreas

Uma área corresponde a uma região do sistema interligado atribuída a uma concessionária. No Framework as áreas são representadas pela classe *TArea*. Uma área, representada pela classe *TArea*, possui os seguintes atributos:

Atributo	Tipo (C++)	Descrição
name	string	Nome da área.
exP	double	Potência ativa líquida de intercâmbio.
exQ	double	Potência reativa líquida de intercâmbio.

Pesp	double	Potência ativa especificada para o controle de intercâmbio.
Pmax	double	Limite superior de potência ativa.
Pmin	double	Limite inferior de potência reativa.

Tabela 3-6 - Atributos da classe *TArea*

Para criar uma instância de uma área é utilizado o método:

```
TArea *CreateArea(n, esp, max, min);
```

Onde *n* é o nome da área, *esp* é o valor da potência especificada do controle de intercâmbio de potência ativa, *max* e *min* são os limites de potência ativa quando o controle de intercâmbio está desativado.

3.1.6 Classe que representa a rede de transmissão

3.1.6.1 Definição da classe

A rede de transmissão é representada no Framework através da classe *TNetwork*. Nesta classe serão cadastrados todos os dispositivos que fazem parte do sistema através de métodos específicos.

Os principais atributos desta classe são:

1. Potência base comum em MVA representada por *Sbase*;
2. Tensão base comum em kV representada por *Vbase*;
3. Erros de potência ativa e reativa a serem alcançados pelo método de cálculo do fluxo de carga, representados por *errP* e *errQ* respectivamente.
4. Número máximo de iterações do método de cálculo do fluxo de carga que, quando alcançado, caracteriza uma não convergência do método de solução do fluxo de carga, representado por *maxIter*.

Para criar uma instância da rede deve-se declarar como objeto local através da seguinte sintaxe:

```
TNetwork rede(Sbase, Vbase, errP, errQ, maxIter);
```

Onde *rede* é o objeto criado, *Sbase* é potência base em MVA, *Vbase* é a tensão base em kV, *errP* é o erro de potência ativa, *errQ* é o erro de potência reativa e *maxIter* é o número máximo de iterações do método de cálculo do fluxo de carga.

3.1.6.2 Cadastrando dispositivos na rede

Para cadastrar os dispositivos na rede (no objeto da classe *TNetwork*) é necessário utilizar um dos métodos de cadastro apresentados na Tabela 3-7.

Método	Descrição
AddArea (TArea *)	Cadastra uma área na rede. O argumento é um ponteiro para a área.
AddLoadModel (TLoad *)	Cadastra um modelo de carga. O argumento é um ponteiro para o modelo de carga.
AddBus (TBus *)	Cadastra uma barra. O argumento é um ponteiro para uma barra.
AddBranch (TBranch *)	Cadastra um ramo. O argumento é um ponteiro para um ramo.

Tabela 3-7 - Métodos da classe TNetwork de cadastro de dispositivos na rede

Cada dispositivo cadastrado recebe um identificador sequencial que deverá ser usado quando for necessária uma referência ao dispositivo cadastrado. Ao realizar o cadastro das barras é necessário que as barras sejam cadastradas seguindo a sequencia:

1. Primeiro a barra de referência;
2. Em seguida as barras PV;
3. Por último as barras PQ.

Para fazer referência a um dos dispositivos cadastrados, pode-se utilizar um dos métodos apresentados na Tabela 3-8.

Método	Descrição
TArea *GetArea(unsigned int indx)	Retorna o ponteiro da área cadastrada que recebeu o identificador *indx*.
TLoad *GetLoadModel(unsigned int indx)	Retorna o ponteiro do modelo de carga cadastrado que recebeu o identificador *indx*.

TBus *GetBus(unsigned int indx)	Retorna o ponteiro da barra cadastrada que recebeu o identificador *indx*.
TBranch *GetBranch(unsigned int indx)	Retorna o ponteiro para o ramo cadastrado que recebeu o identificador *indx*.

Tabela 3-8 - Métodos para referenciar os dispositivos cadastrados

O fragmento de código na Figura 3-2 ilustra o cadastramento de uma área e de um modelo de carga, forma que é recomendado para este Framework.

```
1 #include <bcssd/core/include/bcssd.h>
2 using namespace bcssd;
3 int main ()
4 {
5     TNetwork rede(100.0, 230, 1e-3, 1e-3, 5);
6     rede.AddArea(TArea::CreateArea("Eletronorte Pará", 4000.0,
                                     9000.0, 2000.0));
7     rede.AddLoadModel(TLoad::CreateLoadModel("Carga pesada", 0.6,
                                               0.9, 0.6, 0.1,8
                                               0.05, 0.09));
.....................................................
9 }
```

Figura 3-2 - Exemplo de cadastramento de uma área e de um modelo de carga

Na linha 1 da Figura 3-2 é incluído o arquivo de cabeçalho mestre que carrega todas as definições necessárias ao Framework. Na linha 5 é criada a instância nomeada como *rede* com potência base de 100MVA, tensão base de 230kV, erros de 1e-3 e o número máximo de iterações é cinco. Na linha 6 é criada e cadastrada uma instância da área nomeada "*Eletronorte Pará*". Nas linhas 7 e 8 é criado e cadastrado um modelo de carga.

Quando o objeto *rede* for destruído (quando sair do contexto, por exemplo) todos os dispositivos cadastrados serão destruídos e a memória usada é devolvida ao sistema operacional.

Para exemplificar o cadastro de barras e ramos para montagem da rede elétrica foi usado o exemplo de nove barras e três máquinas encontrado na referência (Anderson & Fouad, 2002):

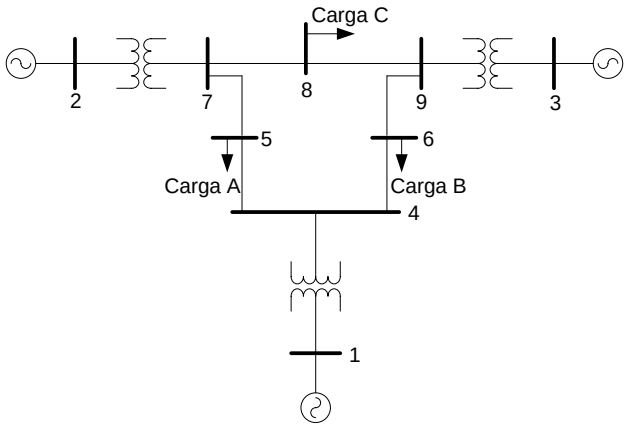

Figura 3-3 - Exemplo de rede elétrica

A primeira barra a ser cadastrada é a barra de referência que receberá o identificador um como indicado na Figura 3-3, em seguida as barras PV e por último as barras PQ, como apresentado no fragmento de código na Figura 3-4.

```
#include <bcssd/core/include/bcssd.h>
using namespace bcssd;
int main ()
{
      TNetwork rede(100.0, 230, 1e-3, 1e-3, 5);
      rede.AddBus(TBus::CreateRefBus(
                  "Barra 1",0.0,1.04,0.0,0.0,0.0,0.0,0.0,0.0,
                  NULL,NULL,0.0,NULL,INBUS,100.0,230.0));
      rede.AddBus(TBus::CreatePVBus(
                  "Barra 2",0.0,1.025,0.0,163.0,0.0,0.0,0.0,
                  NULL,NULL,0.0,NULL,INBUS,100.0,230.0));
      rede.AddBus(TBus::CreatePVBus(
                  "Barra 3",0.0,1.025,0.0,85.0,0.0,0.0,0.0,0.0,
                  NULL,NULL,0.0,NULL,INBUS,100.0,230.0));
      rede.AddBus(TBus::CreatePQBus(
                  "Barra 4",0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,
                  NULL,NULL,0.0,NULL,INBUS,100.0,230.0));
.....................................................................
}
```

Figura 3-4 - Cadastramento das barras do sistema apresentado na Figura 3-3

Para criar e cadastrar as instâncias dos ramos é necessário fazer referência às barras utilizando o método *GetBus* da classe *TNetwork* para definir as barras *from* e *to*. No fragmento de código na Figura 3-5 é ilustrado

como se deve proceder ao cadastrar o ramo que conecta as barras 4 e 5 do diagrama na Figura 3-3.

```
rede.AddBranch(TBranch::CreateBranch(
                "Linha 1",rede.GetBus(4),rede.GetBus(5),
                1.0,0.0,0.01,0.085,0.088,
                BRANCHNONE,0.0,NULL,100.0,230));
```

Figura 3-5 - Cadastro de uma linha de transmissão do sistema na Figura 3-3

Pode-se notar na Figura 3-5 que foi feito referencia às barras 4 e 5 utilizando *rede.GetBus(4)* e *rede.GetBus(5)*.

Após o cadastro de todos os equipamentos, podem-se utilizar os métodos para cálculo de fluxo de carga. O método *Arrange()* deve ser chamado para que as estruturas de dados internas sejam inicializadas e a matriz admitância de rede seja calculada. Ao chamar o método *LoadFlow()* o fluxo de carga é calculado e então o resultado pode ser obtido nos objetos das barras e ramos. O fragmento de código na Figura 3-6 ilustra este procedimento.

```
1 rede.Arrange();
2 if (rede.LoadFlow())
3     cout<<*rede.GetBus(1)<<endl;
```

Figura 3-6 - Cálculo do fluxo de carga

Na Figura 3-6, na linha 1 são inicializadas as estruturas de dados necessárias ao cálculo do fluxo de carga e a matriz admitância de rede é determinada. Na linha 2 o método *LoadFlow* retorna verdadeiro se o fluxo de carga convergiu ou falso se não houver convergência. Caso o fluxo de carga tenha convergido (*LoadFlow* retornou verdadeiro), a linha 3 é executada e o estado da "*barra 1*" é exibida (potências geradas, consumidas e tensão). Em todas as classes que modelam a rede elétrica está sobrecarregado o operador de inserção em uma *stream*, como mostrado através da linha 3.

3.2 Conclusão

Neste capítulo foi apresentada a modelagem orientada a objetos implementada no Framework para representar a rede elétrica dos sistemas elétricos de potência. O resultado do fluxo de carga será usado pelas classes que representam os dispositivos dinâmicos para o estabelecimento das condições de contorno das suas equações diferenciais. A classe *TNetwork* também é responsável pelo cálculo da matriz admitância usada na solução das equações da rede elétrica na simulação dinâmica do sistema de potência

Ainda referente à classe *TNetwork*, ela é derivada da classe *TBlock* e, portanto, como será visto no capítulo 5, ela é modelada como um dispositivo dinâmico.

4 Modelagem e implementação orientada a objetos para representação de sistemas dinâmicos

4.1 Introdução

No capítulo 3 foram apresentadas as classes que compõe o Framework para representar a parte estática dos sistemas elétricos de potência. Neste capítulo serão apresentadas as classes que implementam o comportamento dinâmico de sistemas físicos em geral. Este Framework foi desenvolvido de forma que:

- Permita ao usuário do Framework criar facilmente seus próprios dispositivos dinâmicos;
- Implementar seus próprios métodos de integração numérica;
- Possa ampliar o Framework de forma simples e eficiente.

4.2 Modelagem orientada a objetos do problema

4.2.1 Introdução

O requisito fundamental do Framework é permitir que o usuário possa criar seus próprios dispositivos dinâmicos e implementar seus próprios métodos de integração numérica tornando possível testar técnicas de controle não convencional em sistemas elétricos de potência de grande porte e avaliar quais os impactos, nos resultados obtidos, dos métodos de integração numérica nos resultados da simulação. Portanto, o Framework deve prover suporte a requisitos de software que podem variar com as necessidades do usuário desenvolvedor no campo de estudo da dinâmica de sistemas elétricos de potência. Para incluir no processo de desenvolvimento do software a possibilidade dos requisitos variarem, deve-se desacoplar a abstração da implementação de maneira que os dois possam variar de forma independente (Sena, Barra, Barreiros, Fonseca, Campos, & Costa, 2005).

4.2.2 Representação em espaço de estados

Todos os sistemas físicos possuem algum tipo de dinâmica. Pode-se conceituar que a dinâmica representa a inércia dos sistemas físicos ao mudarem seu estado face uma excitação externa. A dinâmica de um sistema físico pode ser representada através de dois sistemas de equações: um sistema de equações diferenciais e um sistema de equações algébricas que podem ser representadas pelas expressões:

$$\frac{d\overline{X}}{dt} = \overline{f}\left(\overline{X},\overline{U}\right) \tag{4.1}$$

$$\overline{Y} = \overline{g}\left(\overline{X},\overline{U}\right) \tag{4.2}$$

As expressões (4.1) e (4.2) são expressões matriciais. Onde $\overline{X}$ é o vetor de estados (vetor não esparso),

$\overline{Y}$ é o vetor das saídas e $\overline{U}$ é o vetor de entradas ou excitações.

As funções $\overline{f}$ e $\overline{g}$ podem ser lineares ou não lineares. Caso sejam lineares elas podem ser escritas na forma de espaço de estados na forma:

$$\frac{d\overline{X}}{dt} = A\overline{X} + B\overline{U} \tag{4.3}$$

$$\overline{Y} = C\overline{X} + D\overline{U} \tag{4.4}$$

No entanto em sistemas elétricos de potência muitos dispositivos são não lineares e as expressões (4.3) e (4.4) podem ser empregadas apenas para representar sistemas lineares e invariantes no tempo.

4.2.3 Representação em diagramas em blocos

Em Engenharia os sistemas dinâmicos são mais bem compreendidos se forem divididos em unidades menores que são conhecidos como blocos. Os sistemas dinâmicos são representados pelos diagramas em blocos. Um diagrama em bloco é constituído de blocos e conexões entre estes blocos. Cada bloco possui um conjunto de entradas e um conjunto de saídas. Nos diagramas em blocos sempre as entradas de um bloco estão conectadas as saídas de outros blocos.

Na Figura 4-1 é apresentado um modelo em diagrama de blocos de um sistema dinâmico hipotético.

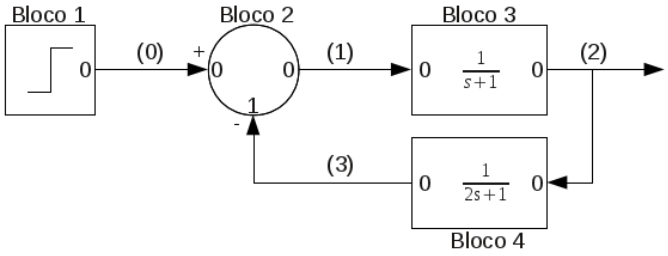

Figura 4-1 - Exemplo de diagrama em blocos

O bloco 1 representa um degrau unitário e não possui entrada apenas uma saída. O bloco 2 é um subtrador e possui duas entradas numeradas por 0 e 1 e uma saída numera como 0. Os blocos 3 e 4 são blocos de primeira ordem e possuem uma entrada numerada por 0 e uma saída numerada também por 0. Os números entre parênteses são as variáveis que no Framework são denominadas de variáveis do sistema.

4.2.4 Definição das classes

4.2.4.1 Definindo as abstrações e as implementações

O problema da simulação dinâmica de sistemas pode ser dividido no modelo do sistema propriamente dito (que será chamado de sistema), os blocos e os métodos de integração numérica.

O sistema possui várias características: possui um ou vários blocos e fornece informações necessárias ao método de integração numérica. O sistema a ser simulado pode ser representado por uma classe denominada

TSystem.

Os blocos definem as unidades estruturais do sistema. Possuem conexões entre si, um conjunto de equações diferenciais e algébricas que definem seu comportamento e um conjunto de parâmetros. Um bloco pode ser representado por uma classe que será chamada de *TBlock*.

O método de integração numérica solicita do sistema (classe *TSystem*) informações, como por exemplo, as derivadas das variáveis de estado ou a matriz jacobiana, para que a integração numérica possa ser feita.

Os blocos são as abstrações, eles representam entidades do mundo real que podem implementar computacionalmente, por exemplo, um controlador PID, uma máquina síncrona, um modelo estatístico de vento, etc, mas que no Framework possui a função de abstrair, fornecendo informações sobre suas variáveis que serão utilizadas na tarefa principal na simulação: a integração numérica. O sistema e o método de integração numérica constituem a implementação. Um sistema possui vários blocos em uma relação de agregação e, por sua vez, repassa as informações fornecidas pelos blocos, de forma organizada, ao método de integração numérica.

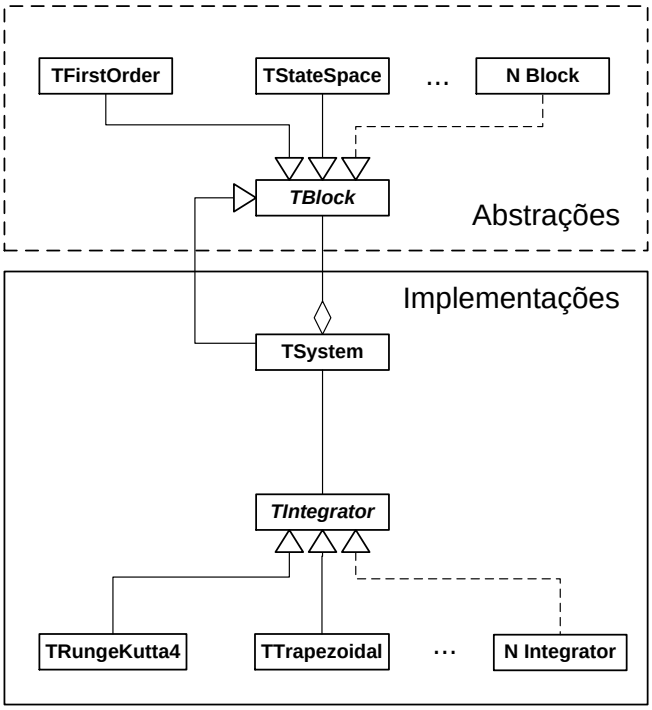

Figura 4-2 - Diagrama de classes que representam sistemas dinâmicos

Na Figura 4-2 pode-se verificar que a classe *TBlock* e suas derivadas estão relacionadas à classe *TSystem* por uma relação de agregação. Neste diagrama pode-se verificar que alterações nas classes *TBlock* e suas classes derivadas não vão interferir no funcionamento das classes *TSystem*, *TIntegrator* e classes derivadas de *TIntegrator*.

A modelagem expressa através da Figura 4-2 é uma aplicação do padrão de projeto denominado *Bridge Pattern* (Padrão ponte) (Gamma, Helm, Johnson, & Vlissides, 1995), (Sena, Barra, Barreiros, Fonseca, Campos, & Costa, 2005).

4.2.4.2 A classe TBlock

Um dispositivo dinâmico é descrito por dois sistemas de equações da forma:

$$\frac{d\overline{X}}{dt} = \overline{f}\left(\overline{X},\overline{U}\right) \tag{4.5}$$

$$\overline{Y} = \overline{g}\left(\overline{X},\overline{U}\right) \tag{4.6}$$

Onde o sistema (4.5) é um sistema de equações diferenciais e (4.6) um sistema de equações algébricas. O vetor $\overline{X}$ contem as variáveis de estado, o vetor $\overline{Y}$ contem as variáveis de saída, o $\overline{U}$ são as variáveis de entrada. As funções $\overline{f}(\cdot)$ e $\overline{g}(\cdot)$ são vetores de funções.

A solução numérica utilizando um método de integração explícito do sistema de equações formado pelas equações (4.5) e (4.6) pode ser executada em três passos:

1. Calcular os valores iniciais das variáveis de estado (*Initialize(.)*).
2. Usar a equação (4.5) para calcular as derivadas das variáveis de estado em função de $\overline{X}$ e $\overline{U}$ (*Derivatives(.)*).
3. Calcular o valor das saídas através da equação (4.6). Pode-se utilizar este passo para aplicar limites e não linearidades nas saídas do bloco (*Outputs(.)*).
4. Na solução numérica do sistema de equações, que descreve o comportamento do sistema dinâmico através das equações (4.3) e (4.4), utilizando métodos implícitos é necessário que se determine a parcela da matriz jacobiana correspondente ao bloco em questão (*Jacobian(.)*).

Outras funcionalidades necessárias a uma classe que represente um bloco em um diagrama em blocos são as associações de cada entrada e cada saída do bloco às variáveis do sistema dinâmico e uma funcionalidade que permita realizar a conexão entre blocos.

Cada bloco deve armazenar as associações entre suas entradas, saídas e variáveis de estado com as variáveis do sistema dinâmico. O armazenamento destas informações é realizado através de três áreas de memória cujos endereços estão nos atributos *in*, *out* e *state*.

Na Tabela 4-1 são apresentados os atributos da classe *TBlock*. Na Tabela 4-2 são apresentados os métodos desta classe.

Atributos	Tipo (C++)	Descrição
in	int *	Vetor de inteiros onde cada posição corresponde a uma entrada do bloco. São armazenados os identificadores das variáveis do sistema que estão associadas às entradas do bloco.

out	int *	Vetor de inteiros onde cada posição corresponde a uma saída do bloco. São armazenados os identificadores das variáveis do sistema que estão associadas às saídas do bloco.
state	int *	Vetor de inteiros onde cada posição corresponde a uma variável de estado do bloco. São armazenados os identificadores das variáveis de estado do sistema que estão associadas às variáveis de estado do bloco.
numinputs	int	Número de entradas do bloco.
numoutputs	int	Número de saídas do bloco.
numstates	int	Número de variáveis de estado do bloco.

Tabela 4-1 - Atributos da classe TBlock.

Método	Descrição
TBlock ();	Construtor default. Nenhuma estrutura de dados é inicializada.
TBlock (numIN,numStates,numOUT);	Construtor onde são inicializadas as estruturas de dados do bloco. O argumento *numIN* é o número de entradas do bloco, *numStates* é o número de variáveis de estado e *numOUT* é o número de saídas.
MemInit (numIN,numStates,numOUT);	Inicializa as estruturas de dados do bloco. O argumento *numIN* é o número de entradas do bloco, *numStates* é o número de variáveis de estado e *numOUT* é o número de saídas.
InputConnect (inputID,outputID,block);	Faz a conexão da entrada *inputID* do bloco com a saída *outputID* de outro bloco apontado por *block* .
SetInput(inID,varID)	Associa a entrada *inID* do bloco a variável do sistema *varID*.
SetOutput(outID,varID)	Associa a saída *outID* do bloco a variável do sistema *varID*.

SetState(stateID,varID)	Associa a variável de estado *stateID* do bloco a variável de estado do sistema *varID*.
GetInput (inputID);	Retorna o identificador da variável do sistema associado à entrada *inputID* do bloco.
GetOutput (outputID);	Retorna o identificador da variável do sistema associado à saída *outputID* do bloco.
GetState (stateID);	Retorna o identificador da variável de estado do sistema associado à variável de estado *stateID* do bloco.
Derivatives (Ysys,Xstate,Fstate,T);	Método virtual que realiza o cálculo das derivadas das varáveis de estado do bloco conforme expressão (4.5) e as coloca nas posições corretas no vetor de derivadas das variáveis de estado do sistema referenciado por *Fstate*. *Ysys* é uma referencia ao vetor das variáveis do sistema e *Xstate* é uma referência às variáveis de estado do sistema. Na classe *TBlock* este método não executa ação alguma, ele deve ser obrigatoriamente sobrecarregado nas classes derivadas. O argumento *T* contém o instante de tempo atual no processo de simulação.
Outputs (Ysys,Xstate,T);	Método virtual onde a expressão (4.6) é avaliada. A partir das variáveis de estado contidas em *Xstate* e os resultados são colocados nas posições adequadas do vetor *Ysys*. O argumento *T* é o instante atual na simulação. Na classe *TBlock* este método não executa nenhuma ação, ele deve ser substituído nas classes derivadas de *TBlock*.
Jacobian (Ysys,Xstate,T);	Função que retorna uma matriz com a contribuição do bloco na formação da matriz jacobiana do sistema dinâmico. Este método deve ser sobrecarregado nas classes derivadas de *TBlock.*
Initialize (Ysys,Xstate);	Método responsável por preencher nos vetores Ysys e Xstate os valores iniciais das saídas, entradas ou variáveis de estado

	em suas respectivas posições.
GetNumInputs ();	Retorna o número de entradas do bloco.
GetNumOutputs ();	Retorna o número de saídas do bloco.
GetNumStates ();	Retorna o número de variáveis de estado do bloco.

Tabela 4-2 - Métodos da classe TBlock

A classe *TBlock* é uma classe abstrata, ou seja, uma classe que não pode ser usada na criação de instâncias. Seus métodos *Initialize*, *Derivatives*, *Outputs* e *Jacobian* são vazios, ou seja, não realizam qualquer operação. Estes quatro métodos deverão ser sobrecarregados nas classes derivadas de *TBlock*.

4.2.4.3 Classes derivadas de TBlock

A classe *TBlock* é uma classe abstrata, ou seja, não pode ser usada na criação de objetos. Esta classe fornece um gabarito para criação de blocos que representam dispositivos dinâmicos.

Para criar um bloco que implemente a dinâmica de algum dispositivo é necessário que seja criada uma classe derivada de *TBlock* em que os métodos *Initialize*, *Drivative, Outputs* e *Jacobian* sejam substituídos. A substituição do método *Jacobian* será apresentado em mais detalhes após a apresentação da implementação do método de integração numérica trapezoidal.

Supondo-se que se queira desenvolver um bloco que implemente uma dinâmica de primeira ordem expressa pela função de transferência:

$$G(s) = \frac{Y(s)}{U(s)} = \frac{k}{Ts+1} \tag{4.7}$$

A primeira providencia a ser tomada é converter a função de transferência na expressão (4.7) por uma equação diferencial na forma expressa por (4.5) e (4.6). Ao fazer as manipulações necessárias em (4.7) obtêm-se as equações:

$$\frac{dx(t)}{dt} = \frac{1}{T}\left[ku(t) - x(t)\right] \tag{4.8}$$

$$y(t) = x(t) \tag{4.9}$$

Onde *x(t)* é a variável de estado, *u(t)* é a entrada e *y(t)* a saída do bloco.

A classe receberá a denominação de *TFirstOrder* e a sua declaração é apresentada na Figura 4-3

.

```
1 class TFirstOrder: public TBlock
2 {
3  private:
4    double K, T;
5  public:
6    TFirstOrder(){MemInit(1,1,1);};

7    virtual void
8    Derivatives (TVector<double> & Ysys, TVector<double> & Xstate,
9                 TVector<double> & Fstate, double Time);

10   virtual void
11   Initialize (TVector<double> & Ysys, TVector<double> & Xstate);

12   virtual tmpTMatrix<double>
13   Jacobian (TVector<double> & Ysys, TVector<double> & Xstate, double T);

14   virtual void
15   Outputs (TVector<double> & Ysys, TVector<double> & Xstate, double Time);

16   void SetParam (double k, double t){K = k; T = t;};
17 };
```

Figura 4-3 - Declaração da classe TFirstOrder

Na Figura 4-3 na linha 1 é declarado que a classe *TFirstOrder* é derivada da classe *TBlock*. São declarados dois atributos privativos para as constantes presentes na expressão (4.7) *K* e *T*. Os atributos *K* e *T* são definidos chamando o método *SetParam* que é declarado na linha 16. No construtor da classe declarado na linha 6, é chamado o método *MemInit* para inicializar as estruturas de dados da classe, onde são informados o número de entradas, número de estados e número de saídas.

O método responsável pela avaliação da expressão (4.8) é o *Derivatives* e a sua implementação é apresentada na Figura 4-4.

```
1 void TFirstOrder::Derivatives(TVector<double> &Ysys, TVector<double> &Xstate,
2                               TVector<double> &Fstate, double Time)
3 {
```

```
4   double pX,X,U;
5   U = Ysys[GetInput(0)];
6   X = Xstate[GetState(0)];
7   pX = (1.0/T)*(K*U-X);
8   Fstate[GetState(0)] = pX;
9 }
```

Figura 4-4 - Implementação do método que avalia a expressão (4.5)

Na Figura 4-4 pode-se verificar que na linha 5 é obtido a partir do vetor de variáveis do sistema *Ysys* o valor da entrada 0 do bloco. Na linha 6 é obtido o valor da variável de estado do bloco a partir do vetor de variáveis de estado do sistema *Xstate*. Na linha 7 é feito o cálculo da derivada da variável de estado do bloco de acordo com a equação (4.8) e na linha 8 o valor da derivada da variável de estado do bloco é passado para o vetor das derivadas das variáveis de estado do sistema.

O cálculo da variável de saída do bloco de primeira ordem *TFirstOrder* é feito através da sobrecarga do método *Outputs* da classe *TBlock*. Na Figura 4-5 é apresentado o código fonte do método sobrecarregado em *TFirstOrder*.

```
1 void TFirstOrder::Outputs (TVector<double> &Ysys, TVector<double> &Xstate,
2                            double Time)
3 {
4     double Y,X;
5     X = Xstate[GetState(0)];
6     Y = X;
7     Ysys[GetOutput(0)] = Y;
8 }
```

Figura 4-5 - Implementação do método que avalia a expressão (4.6)

Na Figura 4-5 pode-se observar que na linha 5 é obtido o valor da variável de estado do bloco no vetor de variáveis de estado do sistema *Xstate*. Na linha 6 é calculada a saída do bloco conforme a equação (4.9) e na linha 7 o valor da saída do bloco é atribuído à posição correspondente no vetor de variáveis do sistema.

O outro método que precisa ser sobrecarregado é o método de inicialização das variáveis do bloco, o método *Initialize* cujo código fonte é apresentado na Figura 4-6.

```
1 void TFirstOrder::Initialize (TVector<double> &Ysys, TVector<double> &Xstate)
2 {
3     Ysys[GetOutput(0)] = 0;
4     Xstate[GetState(0)] = 0;
```

```
5 }
```

Figura 4-6 - Implementação do método de inicialização das variáveis do bloco

Na Figura 4-6 pode-se verificar que nas linhas 3 e 4 as variáveis de saída e de estado são inicializadas em zero.

Ao seguir o padrão apresentado no exemplo da classe *TFirstOrder* pode-se implementar qualquer dispositivo dinâmico, seja ele contínuo ou discreto.

No arquivo de cabeçalho *linear.h*, por exemplo, estão definidas várias classes que implementam alguns blocos lineares básicos apresentados na Tabela 4-3.

Classe	Descrição
TStep	Classe que representa uma fonte de degrau unitário.
TSub2in	Classe que implementa um subtrador com as entradas + e -.
TFirstOrder	Classe que implementa um bloco de primeira ordem.
TStateSpace	Classe que implementa um bloco na forma de espaço de estados.
TTransFunc	Classe que implementa um bloco na forma de função de transferência.
TSum	Classe que implementa um bloco somador de várias entradas.

Tabela 4-3 - Alguns blocos lineares definidos em *linear.h*

Os blocos referentes a sistemas elétricos de potência serão apresentados no capítulo 5.

4.2.4.4 A classe TSystem

Um sistema dinâmico é representado, no Framework, pela classe *TSystem*. A classe *TSystem* relaciona-se com a classe *TBlock* através de agregação, ou seja, a classe *TSystem* possui um ou vários objetos da classe *TBlock* mas as duas classes são funcionalmente independente entre si. Outra relação entre as classes *TSystem* e *TBlock* é a herança, onde a classe *TSystem* é uma classe derivada de *TBlock*, o que implica dizer que um sistema dinâmico pode ser interpretado também como um bloco em um sistema maior.

A classe *TSystem* possui um vetor que armazena as referências aos blocos, representado pelo atributo *blocks* e possui várias funcionalidades básicas. A primeira funcionalidade básica é acrescentar um bloco ao sistema e é implementada através do método *PutBlock*. Outra funcionalidade importante é a que retorna um ponteiro para algum bloco e que é implementada através do método *GetBlock*. Outra funcionalidade consiste em avaliar a equação (4.1) para o sistema inteiro, que é uma sobrecarga do método *Derivatives* da classe

ancestral *TBlock*. Outra funcionalidade é a avaliação da equação (4.2) que é implementada através da sobrecarga do método *Outputs*. Na Tabela 4-4 são apresentados os atributos da classe *TSystem* e na Tabela 4-5 são apresentados os métodos implementados e sobrecarregados nesta classe.

Atributo	Tipo (C++)	Descrição
Blocks	TBlock *	Ponteiro para a área de memória onde as referências aos blocos são armazenados.
variablescount	Int	Contador de variáveis do sistema. A medida que os blocos são cadastrados, este atributo têm seu valor atualizado somando ao valor antigo o número de variáveis de saída do bloco novo.
statescount	Int	Contador de variáveis de estado do sistema. A medida que os blocos são cadastrados, este atributo têm seu valor atualizado somando ao valor antigo o número de variáveis de estado do bloco novo.
blockcount	Int	Contador de blocos. Sempre que um novo bloco é cadastrado, este atributo têm seu valor incrementado.

Tabela 4-4 - Atributos da classe *TSystem*

Método	Descrição
TSystem (n_maxblocks);	Construtor da classe. O seu único argumento é o número máximo de blocos que podem ser manipulados pelo sistema dinâmico. Este método cria a área de memória onde as referências dos blocos serão armazenadas.
PutBlock (pblock);	Faz o cadastro do bloco apontado por *pblock*. Cada bloco recebe um identificador sequencial que inicia em zero. Neste método são feitas as associações da saída de cada bloco a uma variável do sistema dinâmico e a associação de cada variável de estado de cada bloco a uma variável e estado do sistema dinâmico. O número de blocos cadastrados deve ser menor ou igual ao informado no construtor ou pode haver uma falha de segmentação.

GetBlock (blockID);	Retorna um ponteiro para o bloco com o identificador *blockID*. O identificado de bloco na verdade é uma posição na área de memória dos blocos, portando se *blockID* for maior que o número de blocos cadastrados menos um o valor retornado pode ser qualquer endereço e pode haver uma falha de segmentação caso o valor retorna for usado nestas condições.
GetNumBlocks ();	Retorna o número de blocos cadastrados.
Derivatives (Ysys,Xstate,Fstate,T);	Sobrecarga do método correspondente na classe ancestral *TBlock*. Esta classe realiza o cálculo das derivadas das variáveis de estado do sistema de acordo com a equação (4.1). Este método funciona chamando todos os métodos *Derivatives* dos blocos cadastrados.
Outputs (Ysys,Xstate,T);	Sobrecarga do método correspondente na classe ancestral *TBlock*. Esta classe realiza o cálculo das variáveis de saída do sistema de acordo com a equação (4.2). Este método funciona chamando todos os métodos *Outputs* dos blocos cadastrados.
Jacobian (Ysys,Xstat,T);	Este método retorna a matriz jacobiana usada no método de integração numérica trapezoidal. Ela sobrecarrega o método correspondente na classe ancestral *TBlock*. Este método funciona chamando os métodos *Jacobian* de todos os blocos e compõe cada submatriz de cada bloco na matriz jacobiana do sistema dinâmico inteiro.
Initialize (Ysys,Xstate);	Sobrecarrega o método correspondente na classe ancestral *TBlock*. Este método funciona chamando o método *Initialize* de cada bloco cadastrado.

Tabela 4-5 - Métodos da classe *TSystem*

4.2.4.5 A integração Numérica: classe TIntegrator

A integração numérica é implementada utilizando uma classe abstrata que representa um método de integração genérico e as classes derivadas dela implementam os métodos de integração específicos, como pode ser visto através do diagrama de classes na Figura 4-2. Esta arquitetura permite que vários métodos de integração possam ser desenvolvidos e testados para um mesmo sistema dinâmico a ser simulado.

A classe abstrata que será usada como base para a implementação das classes de integração numérica é a classe *TIntegrator* cuja declaração está na Figura 4-7.

```
1 class TIntegrator {
2 public:
3    TIntegrator ();
4    virtual ~TIntegrator ();
5    void    RegisterSystem (TSystem * sys);
6    void    StepParam (double max, double min, double init);
7    double  GetVariableValue (int IDvar);
8    double  GetStateValue (int IDvar);
9    TSystem *GetSystem ();
10   void    ErrorParam (double precision, int niter);

11   virtual int  Integration (double & Time){};

12 protected:
13   TSystem         *system;
14   TVector<double> X;
15   TVector<double> F;
16   TVector<double> Y;
17   double          h;
18   double          hmin;
19   double          hmax;
20   double          prec;
21   double          T;
22   int             numvar;
23   int             numstates;
24   int             numiter;
25 };
```

Figura 4-7 - Declaração da classe base para os métodos de integração numérica

Na Figura 4-7 observa-se que nas linhas 14-16 e na linha 21 estão declarados os atributos que são passados como argumentos nos métodos *Initialize*, *Derivatives*, *Outputs* e *Jacobian* de cada bloco e da classe *TSystem*. Na Tabela 4-6 são apresentados os atributos da classe com explicações detalhadas sobre cada um e na Tabela 4-7 é feito o mesmo para os métodos.

Atributo	Tipo (C++)	Descrição

System	TSystem *	Ponteiro para o sistema dinâmico a ser simulado.
X	TVector	Vetor com os valores das variáveis de estado do sistema dinâmico. É este objeto que é passado no argumento *Xstate* nos métodos *Initialize*, *Derivatives*, *Outputs* e *Jacobian* dos blocos e da classe *TSystem*.
Y	TVector	Vetor com os valores das variáveis do sistema dinâmico. É este objeto que é passado no argumento *Ysys* nos métodos *Initialize*, *Derivatives*, *Outputs* e *Jacobian* dos blocos e da classe *TSystem*.
F	TVector	Vetor com os valores das derivadas das variáveis de estado do sistema dinâmico. É este objeto que é passado no argumento *Fstate* no método *Derivatives* dos blocos e da classe *TSystem*.
h	Double	Comprimento atual do passo de integração numérica em segundos. Em métodos com passo adaptativo este passo á recalculado a cada passo de integração. Em métodos com passo fixo ele permanece constante.
hmax	Double	Valor máximo do passo de integração numérica.
hmin	Double	Valor mínimo do passo de integração numérica. Em métodos com o passo de integração adaptativo um passo *h* menor que *hmin* indica que pode haver um problema de estabilidade numérica.
prec	Double	Erro tolerável para cada passo de integração.
T	Double	Instante de tempo atual. Este objeto é passado no argumento de vários métodos das classes *TBlock* e das classes *TSystem*.
numvar	Int	Número de variáveis do sistema dinâmico.
numstates	Int	Número de variáveis de estado do sistema dinâmico.
numiter	Int	Em métodos de integração que precisam de iterações para cada passo de integração este atributo define o número máximo de iterações que podem ser executadas para cada passo de integração. Caso este número for excedido caracteriza uma falha de convergência.

Tabela 4-6- Atributos da classe TIntegrator

Método	Descrição
RegisterSystem (sys);	Faz o cadastro do sistema dinâmico no integrador numérico.
StepParam (max,min,init);	Define os parâmetros do passo de integração. O argumento *max* é valor máximo do passo de integração, *min* o valor mínimo do passo e *init* é valor inicial do passo de integração.
GetVariableValue (IDvar);	Retorna o valor da variável do sistema identificada por *IDvar*.
GetStateValue (IDvar);	Retorna o valor da variável de estado do sistema identificada por *IDvar*.
GetSystem ();	Retorna um ponteiro para o sistema dinâmico cadastrado.
ErrorParam (precision,niter);	Define parâmetros de erro do método de integração. O argumento *precision* define qual a precisão desejada para cada passo de integração e *niter* é o número máximo de iterações por passo de integração.
Integration (Time);	Método virtual que realiza os cálculos da integração numérica. Na classe *TIntegrator* este método é vazio, ele deverá ser sobrecarregado nas classes derivadas de *TIntegrator* para implementar os métodos de integração numérica.

Tabela 4-7 - Métodos da classe TIntegrator

4.2.4.6 Integração numérica: classe TRungeKutta4

Um integrador numérico explícito implementado no Framework é o Runge-Kutta de quarta ordem. Considerando a equação diferencial na expressão (4.10):

$$\frac{dx(t)}{dt} = f\left[x(t),u(t)\right] \tag{4.10}$$

Onde *x(t)* é uma variável de estado e *u(t)* um sinal de excitação e ambos são funções do tempo. Um método de integração numérica calcula um ponto atual no instante $t_n = t_{n-1} + h$ da variável de estado que será representado por $x(t_n)$utilizando um valor anterior desta mesma variável representado por $x(t_{n-1})$ O método de Runge-Kutta de quarta ordem define quatro coeficientes *k1*, *k2*, *k3* e *k4* que são dadas pelas expressões:

$$k_1 = f\left[x(t_{n-1}), u(t_{n-1})\right] \tag{4.11}$$

$$k_2 = f\left[x(t_{n-1}) + 0{,}5hk_1, u(t_{n-1} + 0{,}5h)\right] \tag{4.12}$$

$$k_3 = f\left[x(t_{n-1}) + 0{,}5hk_2, u(t_{n-1} + 0{,}5h)\right] \tag{4.13}$$

$$k_4 = f\left[x(t_{n-1}) + hk_3, u(t_{n-1} + h)\right] \tag{4.14}$$

A partir das coeficientes é possível determinar $x(t_n)$através da expressão:

$$x(t_n) = x(t_{n-1}) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{4.15}$$

As equações (4.11) a (4.15) foram desenvolvidas para apenas uma equação diferencial. Para o caso de um sistema de equações diferenciais, como é o caso deste trabalho, deve-se generalizar estas equações para o caso matricial. Os coeficientes do método de Runge-Kutta passam a ser vetores que serão denominados $\overline{K_1}$, $\overline{K_2}$, $\overline{K_3}$ e $\overline{K_4}$ respectivamente e as equações (4.11) a (4.15) serão aplicadas à expressão (4.1), desta forma:

$$\overline{K_1} = \overline{f}\left[\overline{X}(t_{n-1}), \overline{U}(t_{n-1})\right] \tag{4.16}$$

$$\overline{K_2} = \overline{f}\left[\overline{X}(t_{n-1}) + 0{,}5h\overline{K_1}, \overline{U}(t_{n-1} + 0{,}5h)\right] \tag{4.17}$$

$$\overline{K_3} = \overline{f}\left[\overline{X}(t_{n-1}) + 0{,}5h\overline{K_2}, \overline{U}(t_{n-1} + 0{,}5h)\right] \tag{4.18}$$

$$\overline{K_4} = \overline{f}\left[\overline{X}(t_{n-1}) + h\overline{K_3}, \overline{U}(t_{n-1} + h)\right] \tag{4.19}$$

$$\overline{X}(t_n) = \overline{X}(t_{n-1}) + \frac{1}{6}\left(\overline{K_1} + 2\overline{K_2} + 2\overline{K_3} + K_4\right) \tag{4.20}$$

Ao conjunto das equações (4.16) até (4.20) pode-se agregar a equação (4.2) para calcular as saídas. A definição da classe que implementa o método de integração de Runge-Kutta de quarta ordem é apresentada na Figura 4-8.

```
1  class TRungeKutta4: public TIntegrator
2  {
3    public:
4     TRungeKutta4();
```

```
5     virtual int  Integration (double &Time);
6  };

7  int  TRungeKutta4::Integration (double &Time)
8  {
9    int numvar = system->GetNumStates();
10   T = Time;
11   TVector<double> K1(numvar),K2(numvar),
12                   K3(numvar),K4(numvar),x(numvar);
13   system->Derivatives(Y,X,K1,T);      //K1=f[X(tn-1),U(tn-1)]
14   x = X+0.5*h*K1;
15   system->Derivatives(Y,x,K2,T+0.5*h);//K2=f[X(tn-1)+0.5*h*K1,U(tn-1+0.5*h)]
16   x = X+0.5*h*K2;
17   system->Derivatives(Y,x,K3,T+0.5*h);//K3=f[X(tn-1)+0.5*h*K2,U(tn-1+0.5*h)]
18   x = X+h*K3;
19   system->Derivatives(Y,x,K4,T+h);     //K4=f[X(tn-1)+h*K3,U(tn-1+h)]
20   X = X + (h/6.0)*(K1+2.0*K2+2.0*K3+K4);
21   system->Outputs(Y,X,T); // Avalia as equações algébricas
22   T+=h;
23   Time = T;
24   return 0;
25 }
```

Figura 4-8 - Implementação do integrador numérico para o método de Runge-Kutta

4.2.4.7 Integração numérica: classe TTrapezoidal

No método trapezoidal a equação diferencial da expressão (4.10) é transformada em uma equação algébrica e então resolvida para $x(t_n)$. Quando a equação diferencial é linear, a equação algébrica resultante é linear e sua solução é trivial, no entanto, se a equação algébrica é não linear então a solução é possível apenas por métodos iterativos, como o método de Newton.

A equação que define o método trapezoidal é dada por:

$$x(t_{n+1}) = x(t_n) + \frac{h}{2}\left[f\left(x(t_n),u(t_n)\right) + f\left(x(t_{n+1}),u(t_{n+1})\right)\right] \tag{4.21}$$

Portanto, a equação que se deseja encontrar a raiz $x(t_{n+1})$ é dada por:

$$x(t_{n+1}) - \frac{h}{2} f\left(x(t_{n+1}), u(t_{n+1})\right) - x(t_n) - \frac{h}{2} f\left(x(t_n), u(t_n)\right) = 0 \tag{4.22}$$

Definindo que:

$$\phi[x(t_{n+1})] = x(t_{n+1}) - x(t_n) - \frac{h}{2}\left[f\left(x(t_{n+1}), u(t_{n+1})\right) + f\left(x(t_n), u(t_n)\right)\right] \tag{4.23}$$

Então para determinar a solução $x(t_{n+1})$da equação (4.22) aplica-se o método de Newton na equação (4.23), então sabendo que *k* é a iteração anterior e que *k+1* é a próxima então:

$$x^{k+1}(t_{n+1}) = x^k(t_{n+1}) - \left\{\frac{\partial \phi\left[x^k(t_{n+1})\right]}{\partial x(t_{n+1})}\right\}^{-1} \phi\left[x^k(t_{n+1})\right] \tag{4.24}$$

A derivada parcial no denominador da equação (4.24) aplicada na equação (4.23) é:

$$\frac{\partial \phi[x(t_{n+1})]}{\partial x(t_{n+1})} = 1 - \frac{h}{2} \frac{\partial}{\partial x(t_{n+1})}\left[f\left(x(t_{n+1}), u(t_{n+1})\right)\right] \tag{4.25}$$

Estendo as equações (4.23), (4.24) e (4.25) para o caso matricial (sistema de equações), então:

$$\overline{\Phi}\left[\overline{X}(t_{n+1})\right] = \overline{X}(t_{n+1}) - \overline{X}(t_n) - \frac{h}{2}\left[\overline{f}\left(\overline{X}(t_{n+1}), \overline{U}(t_{n+1})\right) + \overline{f}\left(\overline{X}(t_n), \overline{U}(t_n)\right)\right] \tag{4.26}$$

$$\overline{X^{k+1}}(t_{n+1}) = \overline{X^k}(t_{n+1}) - \left[\frac{\partial \overline{\Phi}\left(\overline{X^k}(t_{n+1})\right)}{\partial \overline{X}(t_{n+1})}\right]^{-1} \overline{\Phi}\left[\overline{X^k}(t_{n+1})\right] \tag{4.27}$$

$$\frac{\partial \overline{\Phi}\left[\overline{X}(t_{n+1})\right]}{\partial \overline{X}(t_{n+1})} = I - \frac{h}{2} \frac{\partial}{\partial \overline{X}(t_{n+1})}\left[\overline{f}\left(\overline{X}(t_{n+1}), \overline{U}(t_{n+1})\right)\right] \tag{4.28}$$

Definindo como a Jacobiana do sistema de equações diferencias *J* como:

$$J = \frac{\partial}{\partial \overline{X}(t_{n+1})}\left[\overline{f}\left(\overline{X}(t_{n+1}), \overline{U}(t_{n+1})\right)\right] \tag{4.29}$$

Substituindo a equação (4.29) na equação (4.28) obtêm-se:

$$\frac{\partial \overline{\Phi}\left[\overline{X}(t_{n+1})\right]}{\partial \overline{X}(t_{n+1})} = I - \frac{h}{2} J \tag{4.30}$$

Substituindo a (4.30) na (4.27) obtêm-se:

$$\overline{X^{k+1}}(t_{n+1}) = \overline{X^k}(t_{n+1}) - \left[I - \frac{h}{2} J\right]^{-1} \overline{\Phi}\left(\overline{X^k}(t_{n+1})\right) \tag{4.31}$$

A matriz *J* possui dimensão elevada para problemas de simulação práticos, então a equação (4.31) possui

um custo computacional excessivo devido a necessidade da inversão matricial, no entanto, ela pode ser reescrita na forma:

$$\overline{X^{k+1}}(t_{n+1}) - \overline{X^{k}}(t_{n+1}) = -\left[I - \frac{h}{2} J \right]^{-1} \overline{\Phi}\left(\overline{X^{k}}(t_{n+1})\right) \tag{4.32}$$

Definindo:

$$\Delta\overline{X}(t_{n+1}) = \overline{X^{k+1}}(t_{n+1}) - \overline{X^{k}}(t_{n+1}) \tag{4.33}$$

Pré – multiplicando ambos os membros por $-\left[I - \frac{h}{2} J \right]$ obtém-se:

$$-\left[I - \frac{h}{2} J \right] \Delta\overline{X}(t_{n+1}) = \overline{\Phi}\left(\overline{X^{k}}(t_{n+1})\right) \tag{4.34}$$

A equação (4.34) está na forma $Ax = b$ onde se deseja obter x dispondo de A e b. Então resolvendo o sistema de equações lineares (4.34) para obter $\Delta\overline{X}(t_{n+1})$ e utilizando a equação (4.33) para obter $\overline{X^{k+1}}(t_{n+1})$, o vetor com os novos valores das variáveis de estado do sistema dinâmico.

Como o algoritmo é iterativo precisa-se encontrar uma estimativa inicial conveniente para $\overline{X}(t_{n+1})$. Esta estimativa pode ser obtida empregando algum método explícito.

Então o algoritmo para a integração numérica utilizando o método trapezoidal é constituído pelos passos:

1. Calcular uma estimativa inicial de $\overline{X}(t_{n+1})$ utilizando um método explícito;
2. Calcular $\overline{f}\left[\overline{X}(t_{n+1}), \overline{U}(t_{n+1})\right]$ com a equação (4.1) (método *Derivates*);
3. Calcular a matriz jacobiana (método *Jacobian*);
4. Calcular $\overline{\Phi}\left[\overline{X}(t_{n+1})\right]$ com a equação (4.26);
5. Calcular $\Delta\overline{X}(t_{n+1})$ resolvendo o sistema de equações lineares dada pela expressão (4.34);
6. Utilizar o valor de $\Delta\overline{X}(t_{n+1})$ para corrigir o valor de $\overline{X}(t_{n+1})$;
7. Calcular o erro através da norma de $\Delta\overline{X}(t_{n+1})$;
8. Caso o erro calculado no passo 7 for maior que o especificado e o número de iterações for menor que o máximo permitido então repetir os passos anteriores a partir do passo 2.
9. Calcular as variáveis de saída pela equação (4.2) (método *Outputs*);

A definição da classe que implementa o método trapezoidal é apresentada na Figura 4-9.

```
1  class TTrapezoidal: public TIntegrator
2  {
3    public:
4      TTrapezoidal();
5      virtual int  Integration (double &Time);
6  };

7  int  TTrapezoidal::Integration (double &Time)
8  {
9    int i,numvar = system->GetNumStates(),niter = 0;
10   TMatrix<double>    I(numvar,numvar,CblasColMajor);
11   TVector<double>    Xn(numvar), Xn_1(numvar),
12                      DV(numvar), phi(numvar),
13                      y(numvar),Fn_1(numvar);
14   TMatrix<double>    J(numvar,numvar);
15   double             erro;

/*   Método explícito usado para inicializar o
     processo iterativo do método trapezoidal*/

  //Passo 1
16   Xn = X;
17   T = Time;

  //Obtendo estimativa inicial
18   TVector<double> K1(numvar),K2(numvar),
19                   K3(numvar),K4(numvar);
20   system->Derivatives(Y,Xn,K1,T);     //k1 = f(X,T)
21   y = Xn+0.5*h*K1;
22   system->Derivatives(Y,y,K2,T+0.5*h); //k2 = f(X+0.5*h*k1,T+0.5*h)
23   y = Xn+0.5*h*K2;
24   system->Derivatives(Y,y,K3,T+0.5*h); //k3 = f(X+0.5*h*k2,T+0.5*h)
25   y = Xn+h*K3;
```

```
26    system->Derivatives(Y,y,K4,T+h);     //k4 = f(X+h*k3,T+h)
27    Xn = Xn + (h/6.0)*(K1+2.0*K2+2.0*K3+K4);
28    system->Outputs(Y,Xn,T); /* Avalia as equações algébricas */

  // Formação da matriz identidade
29    for (i=0;i<numvar;i++)
30    {
31      I(i,i)=1.0;
32    }

33    T += h;
34    do
35    {
36      Xn_1 = Xn;
37      system->Derivatives(Y,Xn_1,Fn_1,T);   //Passo 2
38      J = system->Jacobian(Y,Xn_1,T);       //Passo 3
39      phi = Xn_1-X-0.5*h*(Fn_1+F);          //Passo 4
40      DV = solve(I-0.5*h*J,phi);            //Passo 5
42      Xn = Xn_1 + DV;                       //Passo 6
43      erro = nrm2(DV);                      //Passo 7
44      niter++;
45    } while ((erro>prec)&&(niter<numiter)); //Passo 8
46    X = Xn;
47    system->Outputs(Y,X,T);                 //Passo 9
48    system->Derivatives(Y,X,F,T);
49    Time = T;
50    if (niter>=numiter)
51    {
52       return 1;
53    }
54    else return 0;
55 }
```

Figura 4-9 - Definição da classe TTrapezoidal

A analisar a Figura 4-9 pode verificar que a estimativa inicial das variáveis de estado é obtida nas linhas

de 18 a 28 (método de Runge-Kutta). As linhas de 34 à 49 implementam o método trapezoidal.

4.3 Exemplo de uso da classes para dispositivos dinâmicos

Para encerrar a explanação sobre as classes que implementam a simulação dinâmica é apresentado um exemplo de uso de Framework. O exemplo consiste em simular um sistema dinâmico representado pelo diagrama em blocos da Figura 4-10.

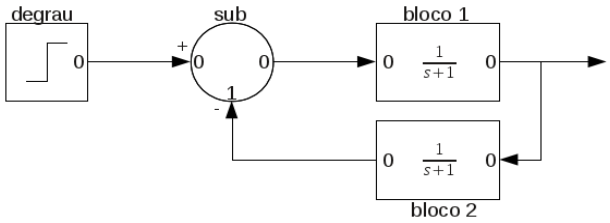

Figura 4-10 - Diagrama em blocos de um sistema dinâmico

Na Figura 4-11 está o código fonte do programa de exemplo. Este programa faz uma simulação simples e armazena em um arquivo de texto os pontos calculados de uma das variáveis do sistema dinâmico.

```
1 #include <iostream>
2 #include <fstream>
3 #include <bcssd/core/include/bcssd.h>

4 using namespace bcssd;
5 using namespace std;
6 int main (void)
7 {
8   TStep         *degrau;
9   TSub2in       *sub;
10  TFirstOrder   *bloco1, *bloco2;
11  TSystem       *sis;
12  TTrapezoidal *integracao;

13  sis =         new TSystem(4);
14  degrau =      new TStep;
15  sub =         new TSub2in;
16  bloco1 =      new TFirstOrder;
17  bloco2 =      new TFirstOrder;
18  integracao = new TTrapezoidal;
```

```
19  degrau->SetParam(1.0);
20  bloco1->SetParam(1.0,1.0);
21  bloco2->SetParam(1.0,1.0);

22  sis->PutBlock(degrau);
23  sis->PutBlock(bloco1);
24  sis->PutBlock(bloco2);
25  sis->PutBlock(sub);

26  sub->InputConnect(0,0,degrau);/* Entrada 0 de sub é conectada a saída 0 de
                                     degrau */
27  sub->InputConnect(1,0,bloco2);/* Entrada 1 de sub é conectada a saída 0 de
                                     bloco2 */
28  bloco2->InputConnect(0,0,bloco1);/* Entrada 0 de bloco2 é conectada a saída
                                        0 de bloco1 */
29  bloco1->InputConnect(0,0,sub);/* Entrada 0 de bloco1 é conectada a saída 0
                                     de sub */

30  integracao->RegisterSystem(sis);
31  integracao->StepParam(0.01,0.01,0.01);
32  integracao->ErrorParam(1e-6,4);

33  int i;
34  double tempo=0;
35  ofstream arq("result.dat",ios::out);
36  for (i=0;i<10000;i++)
37  {
38   arq<<tempo<<" "<<integracao->GetVariableValue(bloco1->GetOutput(0))<<endl;
39   integracao->Integration(tempo);
40  }
41  return 0;
42}
```

Figura 4-11 - Exemplo de programa de simulação

Na Figura 4-11 pode-se observar que das linhas 8 a 18 são criadas as instâncias dos objetos que vão representar os blocos da Figura 4-10 e o sistema dinâmico (*sis*). Das linhas 19 a 21 são passados os parâmetros dos objetos. Nas linhas de 22 a 25 são cadastrados os blocos dinâmicos no sistema dinâmico (*sis*), durante este cadastramento o Framework faz a associação das saídas dos blocos às variáveis do sistema dinâmico. Nas linhas de 26 a 29 são realizadas as conexões entre os blocos, deve-se observar que as conexões são feitas das entradas para as saídas e estas conexões devem ser feitas somente quando todos os blocos estão cadastrados no sistema dinâmico. Na linha 30 é feito o cadastro do sistema dinâmico no integrador numérico que neste caso é o trapezoidal. O laço definido nas linhas 36 a 40 constituem o laço de simulação onde se pode verificar que a integração numérica ocorre na chamada realizada na linha 39.

Neste exemplo pode-se verificar que a simulação de um sistema dinâmico é simples e direta. Todos os detalhes referentes a integração numérica, métodos de solução de equações não lineares e lineares e outras técnicas estão escondidas do usuário do Framework. O programa apresentado é apenas didático, em programas práticos devem-se obter as informações de configuração do sistema em uma base de dados e, a partir destes dados, criar as instâncias e realizar as conexões de forma automática.

4.4 Conclusão

Neste capítulo foram apresentadas as ferramentas básicas para simulação de sistemas dinâmicos. Foram apresentadas as classes que definem um sistema dinâmico, os blocos dinâmicos e os integradores numéricos de equações diferenciais. Com estas classes é possível simular qualquer sistema dinâmico que possa ser representado através de diagramas em blocos.

No próximo capítulo serão apresentadas as classes que definem os dispositivos existentes em sistemas elétricos de potência, como os geradores, reguladores de velocidade, reguladores de tensão e será apresentado também como fazer a conexão destes dispositivos com a rede elétrica cuja modelagem foi apresentada no capítulo sobre fluxo de carga.

5 Implementação orientada a objetos para a representação da dinâmica de sistemas de potência

5.1 Introdução

No capítulo anterior foi apresentado o ferramental básico para a realização da simulação de sistemas físicos. Neste capítulo será apresentada a aplicação destas ferramentas para a simulação da dinâmica de sistemas elétricos de potência para estudos de controle e estabilidade eletromecânica transitória de sistemas elétricos de potência. Serão apresentados os blocos que implementam os modelos de geradores síncronos, rede elétrica, reguladores, etc.

5.2 Modelo clássico de sistemas multimáquinas

5.2.1 Introdução

O Framework apresentado neste trabalho foi desenvolvido com o intuito de fornecer ferramentas para o desenvolvimento de programas para estudos de estabilidade transitória e para estudos de técnicas de controle avançadas para sistemas elétricos de potência. Para realizar o desenvolvimento dos modelos matemáticos são feitas várias considerações e simplificações que são apresentadas na referência (Anderson & Fouad, 2002) e cujos resultados são apresentados nas seções seguintes.

5.2.2 Representação de sistemas multimáquinas

Na rede elétrica existem três tipos básicos de barras: swing, PV e PQ. As barras swing e PV são barras onde há a injeção de corrente na rede elétrica e, portanto, existe a presença de geradores, ao contrário das barras PQ onde a injeção de corrente é nula.

Os geradores podem ser representados pelo seu modelo clássico de uma tensão atrás de uma impedância como apresentado na Figura 5-1.

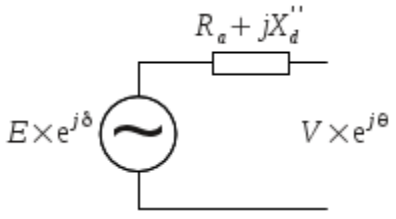

Figura 5-1 - Modelo básico

Ao acoplar um número *n* de geradores na rede elétrica, o modelo torna-se o apresentado na Figura 5-2 a seguir:

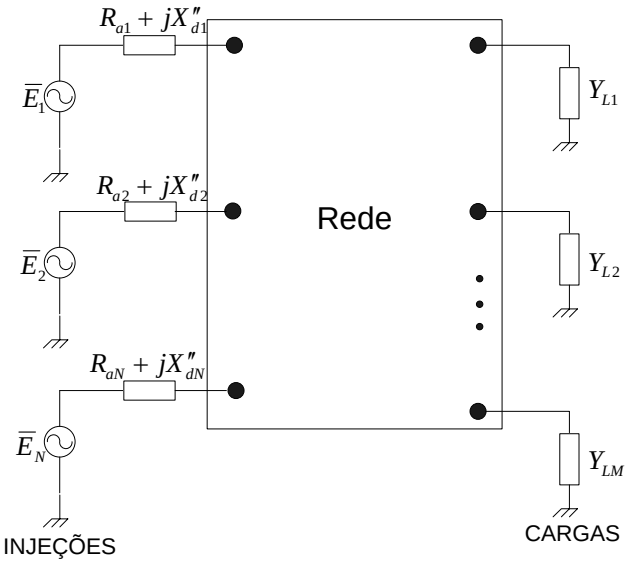

Figura 5-2 - Modelo de um sistema multimáquinas

Onde à esquerda estão os geradores representados pelo seu modelo clássico e a direita as cargas representadas pelos seus modelos de admitâncias constantes.

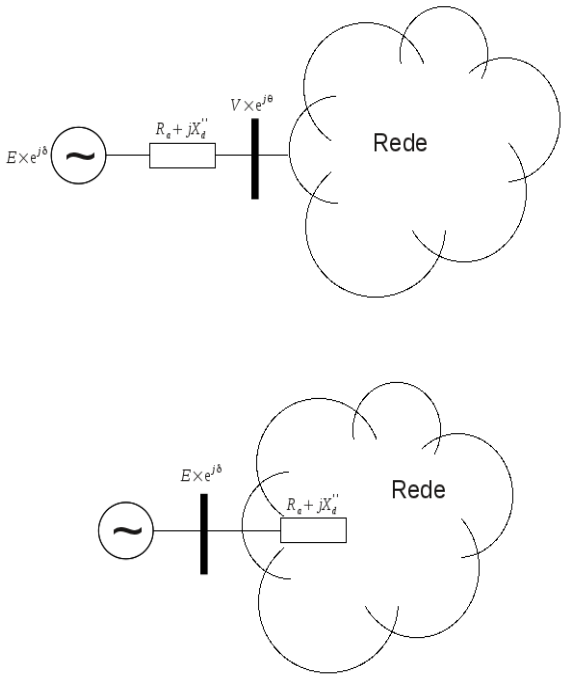

Figura 5-3 - Representação de um gerador através de seu nó interno

Na parte superior da Figura 5-3 é mostrada a representação clássica de uma máquina síncrona em um sistema elétrico: uma tensão interna atrás de uma impedância série. Para representar as máquinas síncronas no sistema elétrico, deve-se ter em mente que sua dinâmica é representada por um sistema de equações diferenciais e algébricas, enquanto que a rede elétrica é representada por um sistema de equações algébricas (para estudos de transitórios eletromecânicos).

Para simplificar as soluções das equações da rede e das máquinas, várias equações algébricas do modelo das máquinas podem ser eliminadas se a impedância série for considerada como parte integrante da rede elétrica, como ilustrado na parte inferior da Figura 5-3. Esta formulação é apresentada em (Sauer & Pai, 1998), ver também a referência (Sena, Barra, Barreiros, Fonseca, Campos, & Costa, 2005).

A tensão interna é uma variável calculada através de equações diferenciais que necessitam dos valores da corrente injetada na rede para serem calculadas. Estas correntes são obtidas através das equações da rede elétrica e da sua interface com as máquinas.

A matriz admitância a ser utilizada na simulação dinâmica é obtida a partir da matriz admitância da rede através de algumas operações:

1. As impedâncias série das máquinas serão incluídas na rede elétrica, como indicado na Figura 5-3, isto implica que a ordem da matriz admitância da rede elétrica será aumentada de NPV nós. Além de aumentar a ordem da matriz admitância, para cada barra de geração (swing ou PV) k ($1 \le k \le NPV+1$) então cada elemento *kk* deverá ser adicionado de: $\left(R_{ak} + jX''_{dk}\right)^{-1}$.

2. Para todas as barras, as cargas locais devem ser transformadas em impedâncias constantes, utilizando as potências ativas e reativas para calculá-las, e assim alterar a matriz admitância de rede somando estes valores de admitâncias em todos os elementos da sua diagonal principal. Considerando *k* uma barra qualquer com uma carga local em p.u. de $P_{Lk} + jQ_{Lk}$ então a admitância equivalente da carga é dada por:

$$\overline{Y}_{Lk} = \frac{P_{Lk}}{V_k^2} - j\frac{Q_{Lk}}{V_k^2}, \text{ para } 1 \le k \le NB \tag{5.1}$$

Pode-se definir agora uma matriz que deverá ser somada a $\overline{Y}_{bus}$ para obter uma matriz admitância $\overline{Y}_{rr}$ onde os efeitos das cargas e da impedância série dos geradores sejam representados:

$$\Delta\overline{Y}_{kk} = \left(R_{ak} + jX''_{dk}\right)^{-1} + \frac{P_{Lk}}{V_k^2} - j\frac{Q_{Lk}}{V_k^2}, \text{ para } 1 \le k \le NPV+1 \tag{5.2}$$

$$\Delta\overline{Y}_{kk} = \frac{P_{Lk}}{V_k^2} - j\frac{Q_{Lk}}{V_k^2}, \text{ para } NPV+1 < k \le NB \tag{5.3}$$

Pode-se agora definir que:

$$\overline{Y}_{rr} = \overline{Y}_{bus} + \Delta\overline{Y} \tag{5.4}$$

3. Serão acrescentadas NPV+1 elementos (que correspondem às novas barras acrescentadas) na parte superior da matriz admitância de rede, onde os elementos da diagonal desta submatriz serão ocupados pelas admitâncias série das máquinas:

$$\bar{Y}_{nn}=\begin{bmatrix} \left(R_{a1}+jX''_{d1}\right)^{-1} & 0 & \cdots & 0 \\ 0 & \left(R_{a2}+jX''_{d2}\right)^{-1} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \left(R_{an}+jX''_{dn}\right)^{-1} \end{bmatrix}_{n\times n} \tag{5.5}$$

Estas novas barras (fictícias) estarão conectadas apenas nas barras swing e PV originais, então as outras submatrizes que serão acrescentadas são:

$$\bar{Y}_{nr}=\begin{bmatrix} -\left(R_{a1}+jX''_{d1}\right)^{-1} & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & -\left(R_{a2}+jX''_{d2}\right)^{-1} & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & 0 & 0 & 0 \\ 0 & 0 & \cdots & -\left(R_{an}+jX''_{dn}\right)^{-1} & 0 & 0 & 0 \end{bmatrix}_{n\times r} \tag{5.6}$$

e

$$\bar{Y}_{rn}=\bar{Y}_{rn}^{t} \tag{5.7}$$

Onde na equação (5.7), o operador *t* é a transposta sem o conjugado dos elementos.

Denominando:

$$\bar{Y}_{rr}=\bar{Y}_{bus}+\Delta Y \tag{5.8}$$

$$n=NPV+1 \tag{5.9}$$

$$r=NB \tag{5.10}$$

Então se pode definir a matriz admitância ampliada onde as admitâncias série dos geradores passam a fazer parte da rede e os nós internos dos geradores são conectados diretamente a rede elétrica:

$$\bar{Y}_{net}=\begin{bmatrix} \bar{Y}_{nn} & \bar{Y}_{nr} \\ \bar{Y}_{rn} & \bar{Y}_{rr} \end{bmatrix} \tag{5.11}$$

Desta forma, a equação de rede para estudos de controle e estabilidade transitória de sistemas elétricos de potência torna-se:

$$\begin{bmatrix} \bar{I}_n \\ 0 \end{bmatrix}=\begin{bmatrix} \bar{Y}_{nn} & \bar{Y}_{nr} \\ \bar{Y}_{rn} & \bar{Y}_{rr} \end{bmatrix}\begin{bmatrix} \bar{E}_n \\ \bar{V}_r \end{bmatrix} \tag{5.12}$$

Onde $\bar{I}_n$ são as correntes injetadas pelos geradores, $\bar{E}_n$ são as tensões internas dos geradores e $\bar{V}_r$ são as tensões das demais barras. As correntes injetadas são as incógnitas que precisam ser determinadas. As tensões internas dos geradores são calculadas através das equações diferenciais dos geradores e as demais tensões também são incógnitas e não são usadas na solução das equações diferenciais dos geradores.

Para calcular as correntes injetadas, deve-se eliminar $\overline{V}_r$ da equação (5.12) expandindo-a da seguinte forma:

$$\overline{I}_n = \overline{Y}_{nn}\overline{E}_n + \overline{Y}_{nr}\overline{V}_r \tag{5.13}$$

$$0 = \overline{Y}_{rn}\overline{E}_n + \overline{Y}_{rr}\overline{V}_r \tag{5.14}$$

Na equação (5.14) pode-se agora explicitar $\overline{V}_r$:

$$\overline{Y}_{rr}\overline{V}_r = -\overline{Y}_{rn}\overline{E}_n \tag{5.15}$$

$$\overline{V}_r = -\left(\overline{Y}_{rr}\right)^{-1}\overline{Y}_{rn}\overline{E}_n \tag{5.16}$$

Pode-se agora substituir (5.16) em (5.13) e obter:

$$\overline{I}_n = \left(\overline{Y}_{nn} - \overline{Y}_{nr}\overline{Y}_{rr}{}^{-1}\overline{Y}_{rn}\right)\overline{E}_n \tag{5.17}$$

Desta forma, pode-se definir a matriz admitância reduzida, em que todos os nós onde não há injeção de corrente foram eliminados:

$$\overline{Y}_{red} = \overline{Y}_{nn} - \overline{Y}_{nr}\overline{Y}_{rr}{}^{-1}\overline{Y}_{rn} \tag{5.18}$$

A equação (5.18) é a fórmula de Kron para redução de redes às barras de injeção de corrente. Para pequenos sistemas de potência esta fórmula é bastante conveniente, mas para sistemas de grande porte que possuem dezenas de milhares de barras e uma matriz admitância com elevada esparsidade, esta fórmula não pode ser aplicada devido à inversão de $\overline{Y}_{rr}$ que resulta em uma matriz não esparsa.

A equação (5.15) está na forma $Ax = b$ então deve-se determinar as tensões através da solução do sistema de equações lineares e então substituir o resultado na equação (5.13) e, então, obter as correntes injetadas.

Quando as tensões internas dos geradores são calculadas, através das equações diferenciais do modelo, elas estão na referência *dq0* de cada máquina, então é necessário que haja a conversão destas referências para a referência da rede elétrica. Na referência bibliográfica (Anderson & Fouad, 2002) é mostrado que:

$$\overline{E}_n = \overline{T}\overline{E}_{ndq} \tag{5.19}$$

Onde $\overline{E}_{ndq}$ são as tensões internas de cada máquina nas suas respectivas referências *dq0*. A matriz $\overline{T}$ é a matriz de conversão usada para que as grandezas na referência da rede elétrica sejam calculadas a partir das grandezas nas referências *dq0*. Esta matriz é dada por:

$$\overline{T} = \begin{bmatrix} e^{j\delta_1} & 0 & \cdots & 0 \\ 0 & e^{j\delta_2} & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & e^{j\delta_n} \end{bmatrix}_{n\times n} \tag{5.20}$$

Onde cada δ_i corresponde à posição angular do rotor da máquina *i* em relação ao ângulo de referência. Estes ângulos são calculados através das equações diferenciais que modelam as máquinas.

A mesma transformação pode ser aplicada nas correntes injetadas em que:

$$\bar{I}_n = \overline{T}\overline{I}_{ndq} \tag{5.21}$$

Na solução das equações diferenciais do modelo das máquinas são necessárias as correntes na referência *dq0*, desta forma, deve-se resolver o sistema de equações lineares (5.15), utilizando as tensões nodais na referência das máquinas, esta equação é reescrita como:

$$\overline{Y}_{rr}\overline{V}_r = -\overline{Y}_{rn}\overline{TE}_{ndq} \tag{5.22}$$

Para em seguida utilizar as tensões calculadas nas demais barras V_r na equação (5.13) que será reescrita como:

$$\overline{T}\overline{I}_{ndq} = \overline{Y}_{nn}\overline{TE}_{ndq} + \overline{Y}_{nr}\overline{V}_r \tag{5.23}$$

$$\bar{I}_{ndq} = \overline{T}^{-1}\overline{Y}_{nn}\overline{TE}_{ndq} + \overline{T}^{-1}\overline{Y}_{nr}\overline{V}_r \tag{5.24}$$

Como a matriz *T* é ortogonal então $\overline{T}^{-1} = \overline{T}^{*}$, desta forma a equação (5.24) pode ser reescrita na forma:

$$\bar{I}_{ndq} = \overline{T}^{*}\overline{Y}_{nn}\overline{TE}_{ndq} + \overline{T}^{*}\overline{Y}_{nr}\overline{V}_r \tag{5.25}$$

Portanto, a solução da equação da rede elétrica será obtida em dois passos:

1. Solução do sistema de equações lineares expressas através da equação (5.22) para a determinação de $\overline{V}_r$, onde $\overline{E}_{ndq}$ e $\overline{T}$ são conhecidos e calculados através das equações das máquinas;

2. Dispondo de $\overline{V}_r$ emprega-se a equação (5.25) para calcular as correntes injetadas pelas máquinas $\bar{I}_{ndq}$.

A partir da formulação apresentada pode-se inferir um modelo computacional no Framework para a rede elétrica representada pela classe *TNetwork*:

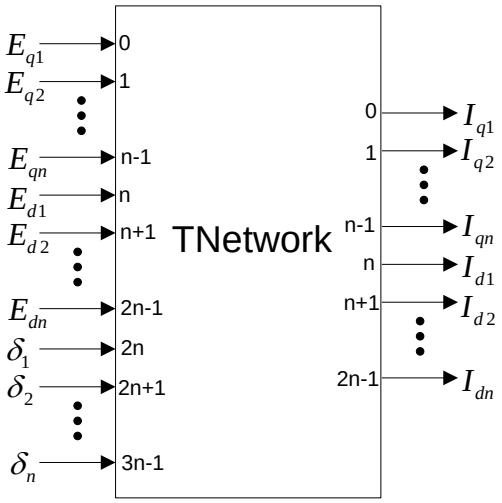

Figura 5-4 - Modelo computacional para a rede elétrica

Sendo *n* o número de máquinas então o número de entradas da classe *TNetwork* é 3*n*, de saídas é 2*n* e nenhuma variável de estado. O número de máquinas é igual ao número de barras PV mais uma barra de referência, ou seja, $n = PV+1$.

5.2.3 Implementação das equações da rede na classe TNetwork

A classe da rede elétrica *TNetwork* possui apenas equações algébricas então o único método a ser substituído é o *Outputs* cujo código fonte é apresentado na Figura 5-5.

```
1  void TNetwork::Outputs(TVector<double> &Ysys,TVector<double> &Xstate,
2                         double t)
3  {
4     int i, m = NPV+1;
5     complex<double> j(0.0,1.0), c_1(1.0,0.0);
6     TMatrix< complex<double> > T(m,m),Endq(m,1),Vr,Indq,b;

7     for (i=0;i<m;i++)
8     {
9           T(i,i) = exp(j*Ysys[GetInput(i+2*m)]);
10          Endq(i,0) = Ysys[GetInput(i)]+j*Ysys[GetInput(i+m)];
11    }

12    b = -c_1*Yrn*T*Endq;
13    Vr = b / Yrr;
14    Indq = conj(T)*Ynn*T*Endq + conj(T)*Ynr*Vr;
```

```
15    for (i=0;i<m;i++)
16    {
17          Ysys[GetOutput(i)] = Indq(i,0).real();
18          Ysys[GetOutput(i+m)] = Indq(i,0).imag();
19    }
20  }
```

Figura 5-5 - Método que calcula as saídas do bloco que representa a rede elétrica

Nas linhas de 7 a 11 da Figura 5-5 são obtidos os ângulos dos rotores das máquinas para a formação da matriz de conversão de referência $\overline{T}$ e as tensões internas na referência *dq* das máquinas formando o vetor $\overline{E}_{ndq}$. Nas linhas 12 e 13 são calculadas as tensões nas barras $\overline{V}_r$ utilizando a equação (5.22). As correntes injetadas pelas máquinas são calculadas na linha 14 utilizando a equação (5.25). Nas linhas de 15 a 19 as correntes calculadas são transferidas para as saídas do bloco que representa a rede elétrica.

5.3 Implementação dos modelos das máquinas síncronas no Framework

5.3.1 Eixos de referência

A escolha dos eixos de referência no qual o sistema de equações que modelam o gerador síncrono é formulado é de grande importância na análise.

Para máquinas síncronas, a referência mais conveniente é aquela onde os eixos de referência estão girando com o rotor. Esta escolha tem como vantagem a obtenção de equações diferenciais com coeficientes constantes (invariantes no tempo). Foram escolhidos como eixos para a referência da máquina o eixo na direção do enrolamento amortecedor de eixo direto (que corresponderá ao eixo imaginário) *d* e o outro eixo estará na direção do enrolamento amortecedor de eixo em quadratura *q* (que corresponderá ao eixo real).

Durante o funcionamento dinâmico, cada máquina síncrona está girando independentemente uma da outra e a transformação entre as referencias das máquinas e a rede é difícil. Uma solução é a escolha de uma referência independente e quando é necessário realizar os cálculos de correntes injetadas converte-se da referência de cada máquina para a esta referência e vice-versa. A escolha mais conveniente para ser a referência da rede é aquela que gira na velocidade síncrona e seus dois eixos são obtidos das condições de regime permanente obtidos no fluxo de carga na barra de referência.

5.3.2 Equações mecânicas

As equações mecânicas das máquinas síncronas são obtidas considerando as seguintes hipóteses:

1. A velocidade do rotor não varia muito em torno da velocidade síncrona (1.0pu);

2. Perdas nos enrolamentos e por fricção são ignorados;
3. A potência mecânica no eixo da máquina é constante exceto quando o regulador de velocidade atua.

Ao considerar estas simplificações podem-se utilizar as seguintes equações mecânicas para um gerador síncrono:

$$\frac{d\omega}{dt} = \frac{1}{2\mathrm{H}_g}\left[P_m - P_e - D_a(\omega - 1.0)\right] \tag{5.26}$$

$$\frac{d\delta}{dt} = 2\pi f_0(\omega - 1.0) \tag{5.27}$$

As grandezas envolvidas nestas equações são:

Grandezas	Descrição
ω	Velocidade angular do rotor em pu
P_m	Potência mecânica aplicada ao eixo da máquina em pu
P_e	Potência elétrica gerada pela máquina em pu
H_g	Momento angular
D_a	Constante de amortecimento

Tabela 5-1 - Grandezas relacionadas com as equações de balanço mecânico da máquina

5.3.3 Equações elétricas

Modelos precisos para máquinas síncronas obtidas como funções das variações do fluxo magnético são apresentados em (Anderson & Fouad, 2002) e (Kundur, 1994). Utilizando a transformada de Park e fazendo várias hipóteses sobre o comportamento das máquinas obtêm-se os modelos apresentados em (Arrilaga & Arnold, 1990). As hipóteses simplificadoras são:

1. A velocidade do rotor sempre é próxima de 1.0pu de forma que em certas situações a velocidade pode ser considerada constante;
2. Todas as indutâncias são independentes da corrente, ou seja, os efeitos da saturação do ferro não serão considerados;
3. As indutâncias dos enrolamentos da máquina podem ser representadas por uma componente constante mais uma componente senoidal que acompanha a rotação do rotor da máquina;

4. Enrolamentos distribuídos podem ser representados por enrolamentos concentrados;
5. A máquina pode ser representada por uma fonte de tensão atrás de uma impedância;
6. Não são consideradas as perdas por histerese do ferro;
7. Reatâncias de fuga existem apenas no estator.

No Framework foi escolhido implementar o modelo denominado de modelo quatro em (Arrilaga & Arnold, 1990). Neste modelo são levados em consideração os efeitos sub - transitórios nos eixos *d* e *q*. São utilizadas as equações (5.26) e (5.27) mais as equações:

$$\frac{dE_q^{'}}{dt} = \frac{1}{T_{d0}^{'}}\left[E_f + \left(X_d - X_d^{'}\right)I_d - E_q^{'}\right] \tag{5.28}$$

$$\frac{dE_q^{''}}{dt} = \frac{1}{T_{d0}^{''}}\left[E_q^{'} + \left(X_d^{'} - X_d^{''}\right)I_d - E_q^{''}\right] \tag{5.29}$$

$$\frac{dE_d^{''}}{dt} = \frac{1}{T_{q0}^{''}}\left[-\left(X_q - X_q^{''}\right)I_q - E_d^{''}\right] \tag{5.30}$$

$$E_q^{''} - V_q = R_a I_q - X_d^{''} I_d \tag{5.31}$$

$$E_d^{''} - V_d = R_a I_d + X_q^{''} I_q \tag{5.32}$$

$$P_e = V_q I_q + V_d I_d + R_a\left(I_q^2 + I_d^2\right) \tag{5.33}$$

$$P_t = V_d I_d + V_q I_q \tag{5.34}$$

$$Q_t = V_d I_q - V_q I_d \tag{5.35}$$

Na simulação as equações (5.31) e (5.32) são utilizadas apenas no cálculo das condições iniciais.

As grandezas envolvidas nestas equações são:

Grandeza	Descrição
$E_q^{'}$	Tensão transitória no eixo em quadratura em pu
$E_q^{''}$	Tensão sub - transitória no eixo em quadratura em pu

$E_d^{''}$	Tensão sub - transitória no eixo direto em pu
V_q	Tensão terminal no eixo em quadratura em pu
V_d	Tensão terminal no eixo direto em pu
I_q	Injeção de corrente no eixo em quadratura
I_d	Injeção de corrente no eixo direto
P_e	Potência elétrica interna em pu (antes das perdas)
P_t	Potência ativa elétrica nos terminais da máquina em pu (após as perdas)
Q_t	Potência reativa elétrica nos terminais da máquina em pu (após as perdas)
X_d , X_q , $X_d^{'}$, $X_d^{''}$, $X_q^{''}$	Reatâncias de regime permanente, transitórias e subtransitórias nos eixos direto e em quadratura em pu
R_a	Resistência de armadura em pu
$T_{d0}^{'}$, $T_{d0}^{''}$, $T_{q0}^{''}$	Constantes de tempo transitórias e subtransitórias de circuito aberto nos eixos direto e em quadratura
E_f	Tensão de campo em pu

Tabela 5-2 - Grandezas relacionadas às equações elétricas da máquina

As tensões $E_q^{''}$ e $E_d^{''}$ são as componentes da tensão interna da máquina, portanto para uma máquina *i* qualquer:

$$\overline{E}_{dqi} = E''_{qi} + jE''_{di} \tag{5.33}$$

Então se podem definir para *n* máquinas:

$$\overline{E}_{ndq} = \begin{bmatrix} E''_{q1} + jE''_{d1} \\ E''_{q2} + jE''_{d2} \\ \vdots \\ E''_{qn} + jE''_{dn} \end{bmatrix} \tag{5.34}$$

Pode-se definir um modelo computacional para as máquinas síncronas:

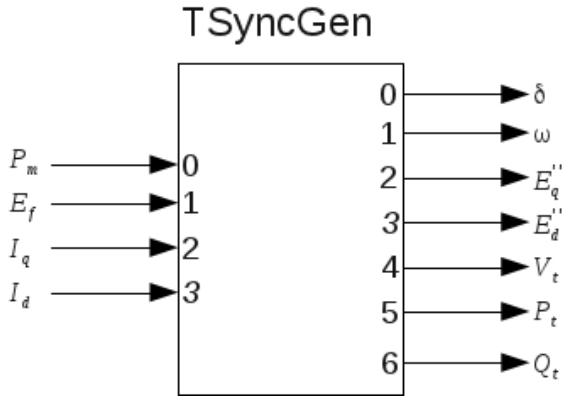

Figura 5-6 - Representação computacional de uma máquina síncrona

5.3.4 Cálculo das condições iniciais

As condições iniciais da máquina são calculadas a partir do resultado do fluxo de carga. Para cada barra do sistema elétrico de potência são calculados pelo fluxo de carga a potência ativa P_t , reativa Q_t o módulo V e a fase θ da tensão terminal.

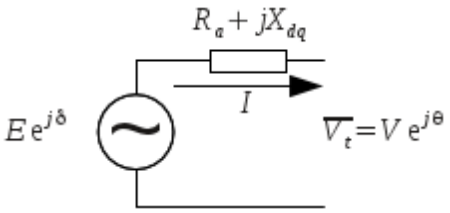

Figura 5-7 - Modelo clássico da máquina em regime permanente

O modelo da máquina em regime permanente é apresentado na Figura 5-7. Utilizando o índice *0* para representar que os valores das grandezas são valores iniciais (regime permanente).

O cálculo do fluxo de potência permite determinar os valores de regime permanente da potência complexa injetada pela máquina $\overline{S}$ e a tensão nos terminais da mesma $\overline{V_t}$, então:

$$\overline{S} = P_t + jQ_t \tag{5.35}$$

$$\overline{I} = \left(\frac{\overline{S}}{\overline{V_t}} \right)^* \tag{5.36}$$

$$\overline{E} = E\mathrm{e}^{j\delta_0} = \overline{V}_t + \left(R_a + jX_{dq}\right)\bar{I} \tag{5.37}$$

Com o ângulo do rotor calculado δ_0 pode-se fazer a conversão de $\bar{I}$ e $\overline{V}_t$ para a referência da máquina segundo a equação (7.50) na referência bibliográfica (Arrilaga & Arnold, 1990):

$$I_{q0} = \mathrm{real}\left(\bar{I}\right)\cos\left(\delta_0\right) + \mathrm{imag}\left(\bar{I}\right)\mathrm{sen}\left(\delta_0\right) \tag{5.38}$$

$$I_{d0} = -\mathrm{real}\left(\bar{I}\right)\mathrm{sen}\left(\delta_0\right) + \mathrm{imag}\left(\bar{I}\right)\cos\left(\delta_0\right) \tag{5.39}$$

$$V_{q0} = \mathrm{real}\left(\overline{V}_t\right)\cos\left(\delta_0\right) + \mathrm{imag}\left(\overline{V}_t\right)\mathrm{sen}\left(\delta_0\right) \tag{5.40}$$

$$V_{d0} = -\mathrm{real}\left(\overline{V}_t\right)\mathrm{sen}\left(\delta_0\right) + \mathrm{imag}\left(\overline{V}_t\right)\cos\left(\delta_0\right) \tag{5.41}$$

A partir das equações (5.31) e (5.32) pode-se determinar a tensão interna do gerador na referência *dq0*:

$$E''_{d0} = V_{d0} + X''_q I_{q0} + R_a I_{d0} \tag{5.42}$$

$$E''_{q0} = V_{q0} - X''_d I_{d0} + R_a I_{q0} \tag{5.43}$$

Na equação (5.29) quando a máquina está em regime permanente a derivada de $E_q^{''}$ é nula então se obtém:

$$E'_{q0} = E''_{q0} - \left(X'_d - X''_d\right) I_{d0} \tag{5.44}$$

Na equação (5.28) a derivada de $E_q^{'}$ é feita igual a zero então se obtêm o valor inicial da tensão de campo dada pela equação (5.45):

$$E_{f0} = E'_{q0} - \left(X_d - X'_d\right) I_{d0} \tag{5.45}$$

As equações de (5.33) a (5.35) podem ser utilizadas para calcular os valores iniciais das potências internas e terminais.

5.3.5 Implementação no Framework das classes para geradores

Para facilitar a apresentação, neste trabalho está sendo apresentada a modelagem de geradores síncronos, no entanto, este Framework foi desenvolvido para que qualquer modelo matemático de gerador possa ser representado. Para implementar esta característica foi criada uma classe abstrata que representa um gerador genérico com algumas poucas funcionalidades necessárias para que seja possível calcular as matrizes $\overline{Y}_{nn}$, $\overline{Y}_{rr}$, $\overline{Y}_{nr}$ e $\overline{Y}_{rn}$ na classe *TNetwork*. A classe *TNetwork* precisa da impedância série das máquinas para realizar os cálculos das suas matrizes e as máquinas precisam dos dados de barra sejam obtidos (potências ativas, reativas, módulo e fase das tensões) da barra ao qual está associada. Na Figura 5-8 é apresentada a declaração da classe

de máquina genérica, onde se pode verificar que apenas três métodos são definidos: um método chamado *GetSerieImpedance* que retorna a impedância série da máquina, o método chamada *SetTerminalBus* para definir qual a barra a máquina está associada e o método virtual *SetParameters* que define os valores dos parâmetros segundo a ordem a ser estabelecida nas classes derivadas.

```
class TMachine: public TBlock {
protected:
	TBus *bus;
	complex<double> ys;
public:
	TMachine();
	virtual ~TMachine();
	complex<double> GetSerieImpedance();
	void SetSerieImpedance(complex<double> y);
	void SetTerminalBus(TBus *b);
	virtual void SetParameters (std::vector<double> &param);
};
```

Figura 5-8 - Classe fundamental para geradores

A implementação de qualquer modelo de gerador deve ser feito derivando a classe *TMachine.*

O modelo de gerador síncrono no Framework deve ser feita derivando-se uma nova classe a partir da classe *TMachine*. A classe que representa um gerador síncrono recebeu a denominação *TSyncGen* cujos atributos para a classe são:

Atributo	Tipo (C++)	Descrição
Xd	Double	Reatância de regime permanente no eixo direto
Xq	Double	Reatância de regime permanente no eixo em quadratura
Xd_	Double	Reatância transitória no eixo direto
Xq_	Double	Reatância transitória no eixo em quadratura
Xd__	Double	Reatância sub - transitória no eixo direto

Xq__	Double	Reatância sub - transitória no eixo em quadratura
Ra	Double	Resistência de armadura.
Tdo_	Double	Constante de tempo transitória de circuito aberto no eixo direto
Tqo_	Double	Constante de tempo transitória de circuito aberto no eixo em quadratura
Tdo__	Double	Constante de tempo sub – transitória de circuito aberto no eixo direto
Tqo__	Double	Constante de tempo sub – transitória de circuito aberto no eixo em quadratura
Hg	Double	Constante de inércia.
Da	Double	Amortecimento.

Tabela 5-3 - Atributos da classe que representa as máquinas síncronas

Os métodos virtuais da classe *TBlock*: *Derivatives*, *Outputs*, *Initialize* e *Jacobian* devem ser substituídas na classe para geradores síncronos *TSyncGen*. Além disso, métodos para definir os parâmetros da máquina devem ser fornecidos.

5.3.5.1 Método de cálculo das derivadas

A criação da classe que representa um gerador síncrono é feita derivando-se a classe *TMachine*. Para escrever o método de cálculo das derivadas das variáveis de estado, deve-se levar em consideração a "*pinagem*" do modelo computacional apresentada na Figura 5-6.

O fragmento de código abaixo apresenta o detalhe da declaração da classe que representa os geradores síncronos *TSyncGen*:

```
#include <bcssd/core/include/bcssd.h> //Definições do Framework

namespace bcssd {                     //Espaço de nomes do Framework

class TSyncGen: public TMachine {     //Derivando a classe
      ..............................
};
```

Figura 5-9 - Declaração da classe TSyncGen

O modelo matemático utilizado neste trabalho para o gerador síncrono é definido pelo conjunto formado

pelas equações de (5.26) a (5.35), sendo que as equações (5.26) a (5.30) são diferenciais e definem cinco variáveis de estado como ilustrado na Tabela 5-4.

Número	Variável de estado	Equação diferencial	Descrição
0	δ	$\frac{d\delta}{dt} = 2\pi f_0(\omega - 1.0)$	Ângulo do rotor em relação ao ângulo de referência em radianos.
1	ω	$\frac{d\omega}{dt} = \frac{1}{2H_g}[P_m - P_e - D_a(\omega - 1.0)]$	Velocidade do rotor em pu.
2	E'_q	$\frac{dE'_q}{dt} = \frac{1}{T'_{d0}}[E_f + (X_d - X'_d)I_d - E'_q]$	Tensão interna transitória do eixo em quadratura.
3	E''_q	$\frac{dE''_q}{dt} = \frac{1}{T''_{d0}}[E'_q + (X'_d - X''_d)I_d - E''_q]$	Tensão interna sub - transitória do eixo em quadratura.
4	E''_d	$\frac{dE''_d}{dt} = \frac{1}{T''_{q0}}[-(X_q - X''_q)I_q - E''_d]$	Tensão interna sub - transitória do eixo direto.

Tabela 5-4 - Variáveis de estado do modelo do gerador síncrono

Ao fazer as definições das variáveis de estado apresentadas na Figura 5-10 pode-se escrever o método que vai realizar o cálculo destas variáveis de estado.

```
1  void TSyncGen::Derivatives (TVector<double> & Ysys,
2                              TVector<double> & Xstate,
3                              TVector<double> & Fstate,
4                              double T)
5  {
6      double pd,pW,pEq__,pEd__,pEq_,Ef,Pm,Vq,Vd,W,Eq__,Ed__,Eq_,Iq,Id,Pe;

7      Pm = Ysys[GetInput(0)];
8      Ef = Ysys[GetInput(1)];
9      Iq = Ysys[GetInput(2)];
```

```
10      Id = Ysys[GetInput(3)];

11      W    = Xstate[GetState(1)];
12      Eq_  = Xstate[GetState(2)];
13      Eq__ = Xstate[GetState(3)];
14      Ed__ = Xstate[GetState(4)];

15      Vq = Eq__-Ra*Iq+Xd__*Id;
16      Vd = Ed__-Ra*Id+Xq__*Iq;

17      Pe = Vq*Iq+Vd*Id+Ra*(Iq*Iq+Id*Id);
18      pW = (1.0/(2.0*Hg)) * (Pm - Pe - Da*(W-1.0));
19      pd = (W - 1.0) * (2.0*M_PI*fo);
20      pEq_ = (Ef + (Xd - Xd_)*Id - Eq_) / Tdo_;
21      pEq__ = (Eq_ + (Xd_ - Xd__)*Id - Eq__)/ Tdo__;
22      pEd__ = (-(Xq - Xq__)*Iq - Ed__) / Tqo__;

23      Fstate[GetState(0)] = pd;
24      Fstate[GetState(1)] = pW;
25      Fstate[GetState(2)] = pEq_;
26      Fstate[GetState(3)] = pEq__;
27      Fstate[GetState(4)] = pEd__;
28  }
```

Figura 5-10 - Cálculo das derivadas das variáveis de estado da máquina síncrona

Na Figura 5-10 nas linhas 7 a 10 são obtidos os valores das entradas do bloco que representa a máquina síncrona. Nas linhas de 11 a 14 são obtidos os valores atuais das variáveis de estado que serão usados nas linhas 15 a 22 para o cálculo das derivadas das variáveis de estado dadas pelas equações (5.26) a (5.32). Nas linhas 23 a 28 são atribuídos os valores das derivadas nas posições respectivas no vetor *Fstate* das derivadas das variáveis de estado do sistema dinâmico.

5.3.5.2 Método de cálculo das saídas

O método de cálculo das saídas é responsável por resolver as equações algébricas do modelo cujos resultados são as saídas. As equações de (5.31) a (5.35) são calculadas neste método e mais a equação do módulo da tensão terminal dada por:

$$V_t = \sqrt{\left(V_q^2 + V_q^2\right)} \tag{5.46}$$

Na Figura 5-11 é apresentado o código fonte deste método onde se pode observar como as equações são avaliadas.

```
1  void TSyncGen::Outputs (TVector<double> & Ysys,
2                          TVector<double> & Xstate,
3                          double T)
4  {
5     double W, delta,Vt, Pt,Qt,Vq,Vd,Eq__,Ed__,Iq,Id;

6     Iq = Ysys[GetInput(2)];
7     Id = Ysys[GetInput(3)];

8     delta = Xstate[GetState(0)];
9     W     = Xstate[GetState(1)];
10    Eq__  = Xstate[GetState(3)];
11    Ed__  = Xstate[GetState(4)];

12    Vq = Eq__-Ra*Iq+Xd__*Id;
13    Vd = Ed__-Ra*Id-Xq__*Iq;
14    Vt = sqrt(Vd*Vd+Vq*Vq);
15    Pt = Vd*Id+Vq*Iq;
16    Qt = Vd*Iq-Vq*Id;

17    Ysys[GetOutput(0)] = delta;
18    Ysys[GetOutput(1)] = W;
19    Ysys[GetOutput(2)] = Eq__;
20    Ysys[GetOutput(3)] = Ed__;
21    Ysys[GetOutput(4)] = Vt;
22    Ysys[GetOutput(5)] = Pt;
23    Ysys[GetOutput(6)] = Qt;
24 }
```

Figura 5-11 - Método para o cálculo das saídas

Na Figura 5-11 nas linhas 6 e 7 são obtidas as correntes injetadas na rede que correspondem às entradas

2 e 3. Nas linas de 8 a 11 são obtidas as variáveis de estado cujos valores já foram calculados pelo integrador numérico. Nas linhas 12 a 16 as equações algébricas são avaliadas. Nas linhas de 17 a 23 os valores respectivos das saídas das máquinas síncronas são atribuídos.

Incluindo o método de inicialização das variáveis da máquina (*initialize*) a classe está pronta para ser utilizada em simuladores onde os métodos de integração são explícitos, como, por exemplo, o método de Runge-Kutta.

Para utilizar esta classe em simuladores que utilizam o método de integração trapezoidal, é necessário que o método *Jacobian* seja implementado na classe.

5.3.5.3 Método de cálculo da matriz jacobiana para o método trapezoidal

O método *Jacobian* está definido na classe ancestral *TBlock* como um método sem implementação. Ele deve ser implementado nas classes concretas derivadas de *TBlock* para uso no método de integração trapezoidal.

Define-se como o vetor de variáveis de estado do gerador síncrono:

$$\overline{X} = \begin{bmatrix} \delta \\ \omega \\ E'_q \\ E''_q \\ E''_d \end{bmatrix} \tag{5.47}$$

Definindo-se a função matricial:

$$\bar{f}\left(\overline{X}\right) = \begin{bmatrix} 2\pi f_0(\omega - 1) \\ \frac{1}{2\mathrm{H}_g}\left[P_m - P_e - D_a(\omega - 1)\right] \\ \frac{1}{T'_{d0}}\left[E_f + (X_d - X'_d)I_d - E'_q\right] \\ \frac{1}{T''_{d0}}\left[E'_q + (X'_d - X''_d)I_d - E''_q\right] \\ \frac{1}{T''_{q0}}\left[-(X_q - X''_q)I_q - E''_d\right] \end{bmatrix} \tag{5.48}$$

Então o sistema de equações diferenciais que representa a dinâmica da máquina síncrona pode ser escrito como:

$$\frac{d\overline{X}}{dt} = \bar{f}\left(\overline{X}\right) \tag{5.49}$$

Como descrito no capítulo anterior, é necessário que sejam calculadas as contribuições de cada bloco na formação da matriz jacobiana total do sistema. Esta contribuição é composta pela jacobiana do sistema de equações (5.48). Portanto, sabendo que:

$$\overline{X} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \tag{5.50}$$

E que:

$$\bar{f}\left(\overline{X}\right) = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{bmatrix} \tag{5.51}$$

A jacobiana é dada por:

$$\bar{J}\left(\overline{X}\right) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} & \frac{\partial f_1}{\partial x_4} & \frac{\partial f_1}{\partial x_5} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} & \frac{\partial f_2}{\partial x_4} & \frac{\partial f_2}{\partial x_5} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} & \frac{\partial f_3}{\partial x_4} & \frac{\partial f_3}{\partial x_5} \\ \frac{\partial f_4}{\partial x_1} & \frac{\partial f_4}{\partial x_2} & \frac{\partial f_4}{\partial x_3} & \frac{\partial f_4}{\partial x_4} & \frac{\partial f_4}{\partial x_5} \\ \frac{\partial f_5}{\partial x_1} & \frac{\partial f_5}{\partial x_2} & \frac{\partial f_5}{\partial x_3} & \frac{\partial f_5}{\partial x_4} & \frac{\partial f_5}{\partial x_5} \end{bmatrix} \tag{5.52}$$

Então:

$$\bar{J}\left(\overline{X}\right) = \begin{bmatrix} 10^{-8} & 2\pi f_0 & 0 & 0 & 0 \\ 0 & -\frac{D_a}{2H_g} & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{T'_{d0}} & 0 & 0 \\ 0 & 0 & \frac{1}{T''_{d0}} & -\frac{1}{T''_{d0}} & 0 \\ 0 & 0 & 0 & 0 & -\frac{1}{T''_{q0}} \end{bmatrix} \tag{5.53}$$

O elemento 10^{-8} é usado no lugar do zero como primeiro elemento da matriz para evitar problemas

numéricos.

A partir da equação (5.53) pode-se escrever o método para cálculo da contribuição da máquina síncrona para a formação da matriz jacobiana, do sistema dinâmico, necessária para a aplicação do método de integração trapezoidal. Na Figura 5-12 é apresentado o método *Jacobian*.

```
1  tmpTMatrix<double>
2  TSyncGen::Jacobian (TVector<double> & Ysys,
3                      TVector<double> & Xstat,
4                      double T)
5  {
6     tmpTMatrix<double> J(5,5);
7     J(0,0) = 1e-8; J(0,1) = 1.0;
8     J(1,1) = -(Da/2.0*Hg);
9     J(2,2) = -1.0/Tdo_;
10    J(3,2) = 1.0/Tdo__; J(3,3) = -1.0/Tdo__;
11    J(4,4) = -1.0/Tqo__;
12    return J;
13 }
```

Figura 5-12 - Cálculo da contribuição da máquina síncrona para a formação da matriz jacobiana

5.3.5.4 Método de inicialização das variáveis da máquina

A inicialização das variáveis do bloco que representa a máquina síncrona é feita através da substituição do método *Initialize*. O cálculo das condições iniciais é feito utilizando a formulação matemática apresentada na seção 5.3.4, portanto, são necessários os resultados obtidos no cálculo do fluxo de carga.

As informações de tensões e potências estão armazenadas nas instâncias que representam as barras na rede elétrica, portanto, é necessário que o objeto que representa a máquina síncrona possua como um dos seus atributos uma referência a barra PV ou swing ao qual está associada.

A associação de uma máquina síncrona com uma determinada barra é feita utilizando o método da classe *TMachine*:

SetTerminalBus(bus)

Onde *bus* é um ponteiro para a barra onde os terminais da máquina estão conectados. Esta associação permite à classe de máquina síncrona encontrar os valores do módulo e da fase da tensão terminal e as potências ativa e reativa injetadas na barra. Ao chamar o método *SetTerminalBus* o atributo *bus* da classe conterá a referência à barra. Na Figura 5-13 é apresentado o código fonte do método.

```
1 void TSyncGen::Initialize (TVector<double> & Ysys, TVector<double> & Xstate)
```

```
2 {
3     complex<double> Vtc,I,E__,S,j=complex<double>(0.0,1.0);
4     double Vt, Vtang,  delta,Iq,Id,Vq,Vd,Ed__,Eq__,
5            Eq_,Ef,W,Pe,Pm,Pt,Qt,Sbase,Vbase;

6     Sbase = bus->getNetwork()->getSbase();
7     Vbase = bus->getNetwork()->getVbase();

8     Pt = bus->getPg()/Sbase;
9     Qt = bus->getQg()/Sbase;
10    S = Pt+j*Qt;
11    Vt = (bus->getVmod()*bus->getVbase())/Vbase;
12    Vtang = bus->getVang()*M_PI/180.0;
13    Vtc = Vt*exp(j*Vtang);
14    I = conj(S/Vtc);
15    E__ = Vt+(Ra+j*Xq)*I;
16    delta = arg(E__);
17    Iq=real(I)*cos(delta)+imag(I)*sin(delta);
18    Id=-real(I)*sin(delta)+imag(I)*cos(delta);
19    Vq=real(Vt)*cos(delta)+imag(Vt)*sin(delta);
20    Vd=-real(Vt)*sin(delta)+imag(Vt)*cos(delta);
21    Pt = Vd*Id+Vq*Iq;
22    Qt = Vd*Iq-Vq*Id;
23    Pe = Vq*Iq+Vd*Id+Ra*(Iq*Iq+Id*Id);

24    Ed__ = Vd+Xq__*Iq+Ra*Id;
25    Eq__ = Vq-Xd__*Id+Ra*Iq;

26    Eq_ = Eq__-(Xd_-Xd__)*Id;
27    Ef = Eq_-(Xd-Xd_)*Id;
28    W = 1.0;
29    Pm = Pe;

30    Ysys[GetInput(0)] = Pm;
```

```
31    Ysys[GetInput(1)] = Ef;
32    Ysys[GetInput(2)] = Iq;
33    Ysys[GetInput(3)] = Id;

34    Xstate[GetState(0)] = delta;
35    Xstate[GetState(1)] = W;
36    Xstate[GetState(2)] = Eq_;
37    Xstate[GetState(3)] = Eq__;
38    Xstate[GetState(4)] = Ed__;

39    Ysys[GetOutput(0)] = delta;
40    Ysys[GetOutput(1)] = W;
41    Ysys[GetOutput(2)] = Eq__;
42    Ysys[GetOutput(3)] = Ed__;
43    Ysys[GetOutput(4)] = Vt;
44    Ysys[GetOutput(5)] = Pt;
45    Ysys[GetOutput(6)] = Qt;
46 }
```

Figura 5-13 - Cálculo das condições iniciais da máquina

Na Figura 5-13 nas linhas 6 e 7 são obtidos a potência e a tensão base comum do sistema de potência. Nas linhas 8 e 9 as potências ativas e reativas são convertidas para valores em pu na base comum do sistema. Na linha 11 a tensão em pu obtida da barra está na base da barra então este valor de tensão é convertido na base comum do sistema. Nas linhas 14 até 29 são avaliadas as equações de (5.35) a (5.45) para em seguida seus resultados serem atribuídos aos vetores de variáveis de estado e de saída do sistema nas linhas de 30 até 45.

5.4 Conclusão

Com a implementação das classes que representam a rede elétrica e os geradores síncronos, pode-se desenvolver programas de simulação dinâmica de sistemas de potência. As classes que representam os controladores e dispositivos diversos devem ser desenvolvidas pelos usuários do Framework para suas aplicações específicas.

Com este capítulo encerra-se a apresentação do Framework. No próximo e último capítulo serão apresentados alguns resultados obtidos na simulação de um sistema de potência real.

6 Estudo de caso 1: aplicação e validação do Framework na UHE Tucuruí

6.1 Modelagem da Hidrelétrica Tucuruí

Um dos resultados obtidos com o uso do Framework apresentado neste documento foi o desenvolvimento de um simulador dinâmico para a Hidrelétrica (UHE) de Tucuruí que recebeu a denominação de "*Dynapower*" (Di Paolo Í. F., 2009).

A UHE de Tucuruí é composta por 23 geradores. Em 1984 a UHE entrou em operação com 12 geradores de *350 MW*, com máquinas de diferentes fabricantes, conhecidos como "geradores da primeira etapa" ou "geradores da primeira casa de força" (ver Figura 6-1).

Figura 6-1 - Máquinas da primeira etapa da construção da UHE de Tucuruí

Em 2002 começaram a operar mais 11 geradores de *390 MW*, também de diferentes fabricantes, conhecidos como "geradores da segunda etapa" ou "geradores da segunda casa de força", compondo uma potencia instalada de *8,49 GW* de energia gerada para o SIN (ver Figura 6-2).

Figura 6-2 - Máquinas da segunda etapa da construção da UHE de Tucuruí

O modelo desenvolvido neste trabalho objetivou o estudo da dinâmica de todas as UGHs da UHE de Tucuruí, e seus respectivos controles de forma detalhada.

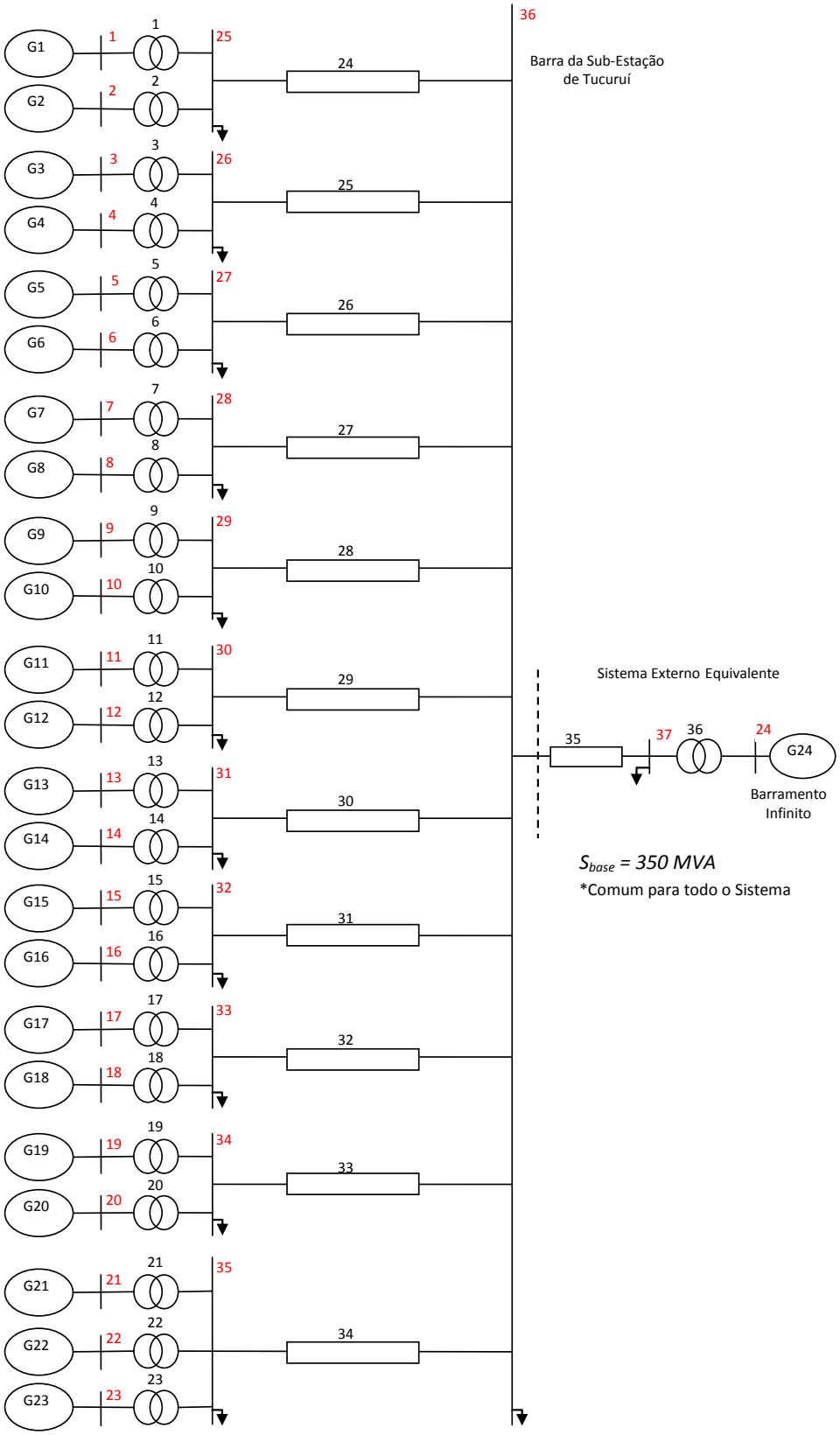

Figura 6-3 - Diagrama unifilar do sistema modelado

Através da Figura 6-3, é apresentado o diagrama unifilar da usina onde as máquinas estão interligadas ao barramento infinito que representa o restante do sistema interligado nacional (SIN). O barramento infinito é

modelado como uma grande máquina com tensão terminal constante. A interface com o barramento infinito é simbolizada por uma linha pontilhada passando pelo ramo *35*, sendo a barra *36* a subestação da UHE. As máquinas *G1* a *G12* são as máquinas na primeira etapa, e as *G13* a *G23* são da segunda etapa.

A base comum utilizada é de *350 MVA* de potência e *13,8 kV* de tensão. Os ramos *24* a *34* são linhas de transmissão que possuem todas a mesma impedância série de $3{,}612.10^{-5}+j4{,}5682.10^{-4}$ *p.u.*, e susceptância *shunt* de $3{,}6.10^{-3}$ *p.u.* Os ramos *1* a *23* são os transformadores das máquinas geradoras que possuem reatância série de *0,165 p.u.* O ramo *35* é uma linha de transmissão configurada com os mesmos valores de impedância série e susceptância *shunt* das demais linhas. O transformador do ramo 36 foi configurado com uma reatância muito pequena de $1{,}65.10^{-7}$ *p.u.*, eletricamente muito perto da barra *24*. A barra *37* é barramento infinito, que também é a referência deste sistema. Foram ainda representadas pequenas cargas de *1 MW* nas barras *25* a *35*, e de *10 MW* na barra *36*, modelando o consumo interno da UHE. Na barra *37* foi representada uma carga de *2800 MW*.

Nessas condições, a matriz admitância *Y* para as configurações apresentadas desta rede possui ordem 37 e com esparsidade igual a 99,92%.

6.1.1 Geradores

O modelo das máquinas síncronas utilizado é o modelo 4 definido na referência (Arrilaga & Arnold, 1990).

No banco de dados do Operador Nacional do Sistema Elétrico, a UHE Tucuruí é representada como um agregado de 5 geradores equivalentes. Neste estudo de caso, as 23 máquinas geradoras foram representadas separadamente. Em seguida serão apresentados os controles e atuadores presentes nas UGHs.

6.1.2 Reguladores de tensão

Na Figura 6-4 está representado o momento em que os ensaios no regulador automático de tensão, RAT, da UGH 8 estavam sendo realizados em fevereiro de 2009.

Figura 6-4 - Ensaios sendo realizados no regulador de tensão da UGH 8

6.1.2.1 Modelagem em Diagrama de Blocos

O modelo do RAT da primeira etapa foi implementado baseado principalmente nas referências (Eletronorte, 1988) e (Operador Nacional do Sistema, 2008). Na Figura 6-5 é apresentado o diagrama em blocos.

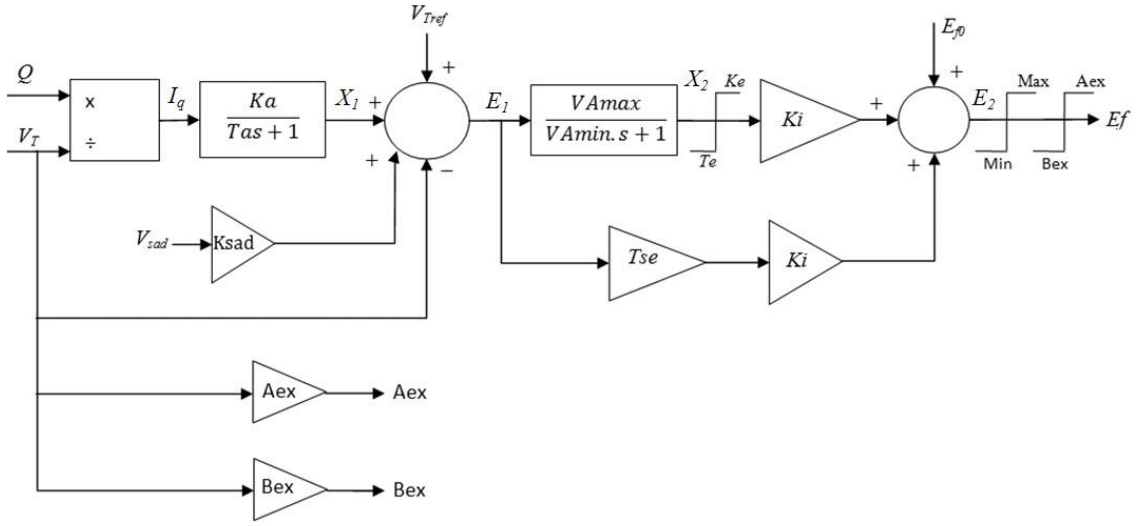

Figura 6-5 - Diagrama de Blocos do RAT da primeira etapa

O modelo do RAT da segunda etapa foi implementado baseado principalmente nas referências (Eletronorte, 2001), (Operador Nacional do Sistema, 2004) e (Operador Nacional do Sistema, 2008). O Diagrama na Figura 6-6 apresenta a malha principal do RAT da segunda etapa, e nas Figura 6-7, Figura 6-8, Figura 6-9, Figura 6-10, Figura 6-11 e Figura 6-12 todos os demais blocos.

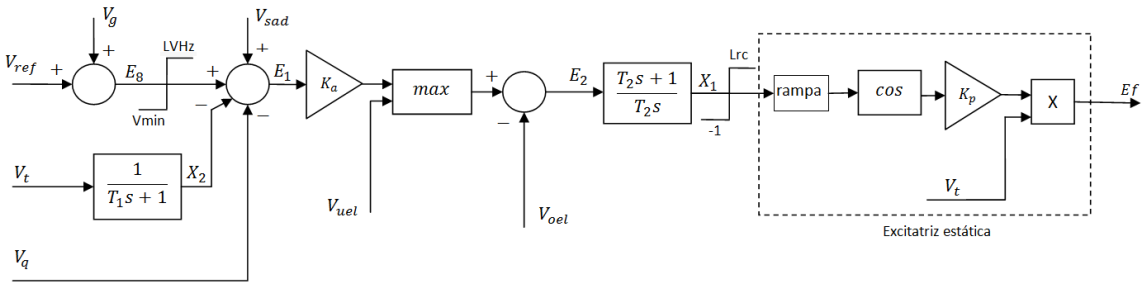

Figura 6-6 - Malha principal do RAT da segunda etapa

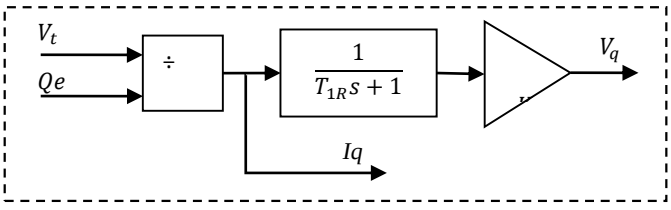

Figura 6-7 – Compensação de potência reativa da segunda etapa

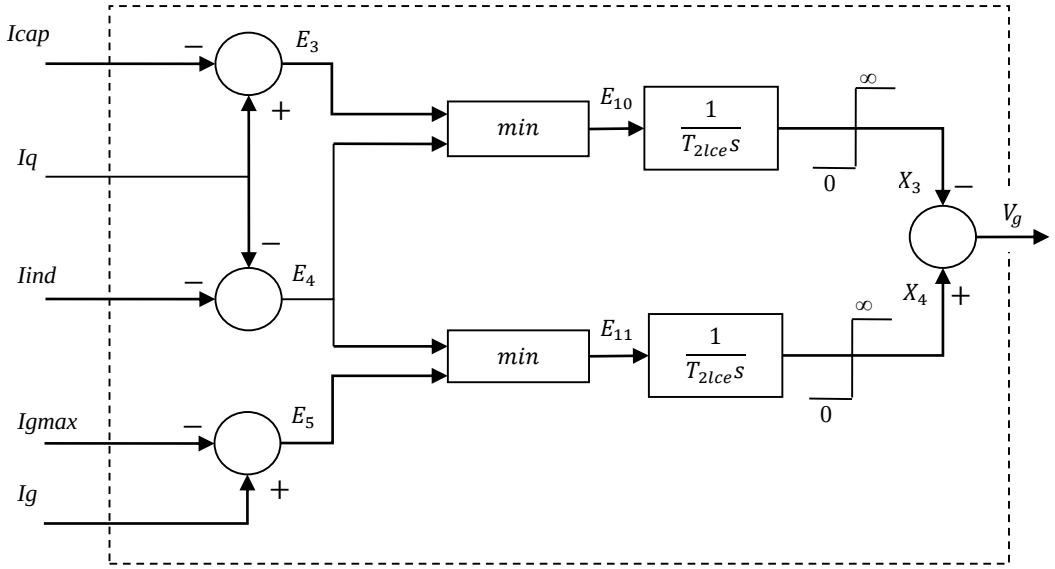

Figura 6-8 - Limitador de corrente de armadura da segunda etapa

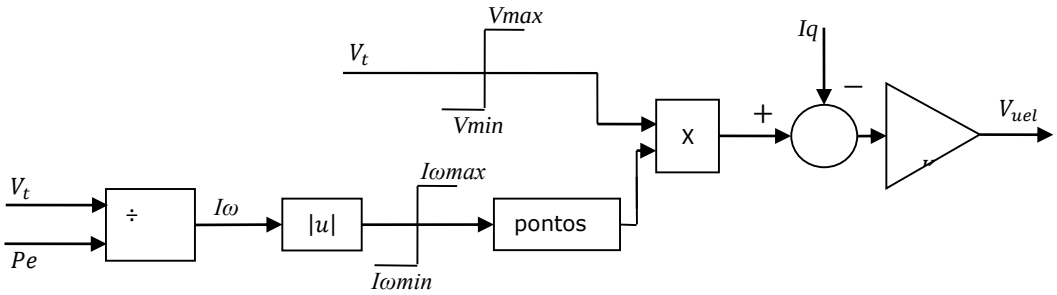

Figura 6-9 - Limitador de subexcitação da segunda etapa

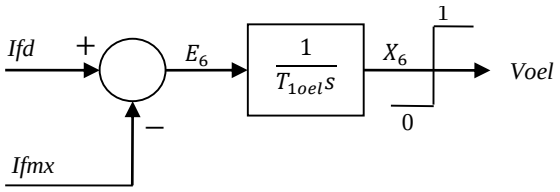

Figura 6-10 - Limitador de sobrexcitação da segunda etapa

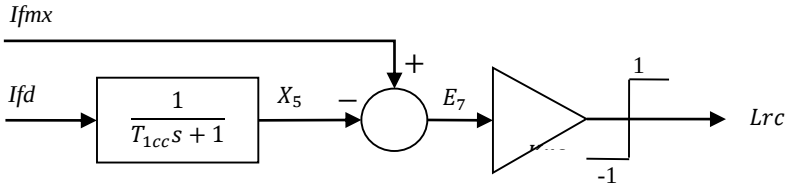

Figura 6-11 - Limitador de corrente de excitação da segunda etapa

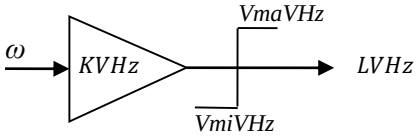

Figura 6-12 - Limitador de enlace de fluxo (Volt/Hz) da segunda etapa

6.1.2.2 Modelagem em Espaços de Estados

Um passo necessário para a implementação dos modelos no Framework é converter os modelos dinâmicos em sua representação equivalente em espaços de estados. Para o RAT da primeira etapa, escolheram-se as variáveis de estado X_1 e X_2, conforme apresenta na Figura 6-5, definindo-se as variáveis de estado que representam a dinâmica deste sistema em (6.11).

$$\begin{cases} \dfrac{dX_1}{dt} = \dfrac{1}{T_a}\left(K_a I_q - X_1\right) \\ \dfrac{dX_2}{dt} = \dfrac{1}{V_{A\min}}\left(V_{A\max} E_1 - X_2\right) \end{cases} \tag{6.11}$$

Sendo $I_q = Q/V_T$ e $E_1 = V_{ref} + X_1 + V_{sad}K_{sad} - V_T$.

As condições iniciais são determinadas pela expressão (6.12) e em (6.13) a equação da saída.

$$\begin{cases} X_1 = \dfrac{Q}{V_T} K_a \\ X_2 = 0 \\ V_{ref} = V_T - X_1 \end{cases} \tag{6.12}$$

$$E_f = E_2 = X_2 K_i + E_{f0} + E_1 T_{se} K_i \tag{6.13}$$

Para o RAT da segunda etapa, as variáveis de estado são apresentadas em (6.14).

$$\begin{cases} \dfrac{dX_1}{dt} = \dfrac{E_2}{T_2} \\ \dfrac{dX_2}{dt} = \dfrac{1}{T_1}\left(V_T - X_2\right) \\ \dfrac{dX_3}{dt} = \dfrac{E_{10}}{T_1 I_{ce}} \\ \dfrac{dX_4}{dt} = \dfrac{E_{11}}{T_2 I_{ce}} \\ \dfrac{dX_5}{dt} = \dfrac{1}{T_{1cc}}\left(I_{fd} - X_5\right) \\ V_{oel} = \dfrac{E_6}{T_{1oel}} \\ \dfrac{dV_q}{dt} = \dfrac{1}{T_1 R}\left(K_q I_q - V_q\right) \end{cases} \tag{6.14}$$

Em (6.15) são apresentadas as condições iniciais, e em (6.16) a equação de saída.

$$\begin{cases} L_{rc} = 1 \\ L_{VHz} = \omega K_{VHz} \\ E_9 = 1 - \dfrac{2}{\pi}\cos^{-1}\left(\dfrac{E_f}{V_T K_p}\right) \\ X_1 = E_9 \\ X_2 = V_T \\ X_3 = -I_{cap} \\ X_4 = -I_{ind} \\ X_5 = I_{fd} \\ V_{oel} = 0 \\ V_q = \dfrac{Q_e K_q}{V_T} \\ V_{ref} = V_T \end{cases} \tag{6.15}$$

$$E_f = V_T K_p \cos\left[\text{rampa}\left(E_9\right)\right] \tag{6.16}$$

6.1.2.3 Implementação da classe que representa o regulador de tensão

Para exemplificar a implementação de um controlador no Framework, é descrita nesta seção o desenvolvimento da classe para os reguladores de tensão da primeira etapa da UHE Tucuruí.

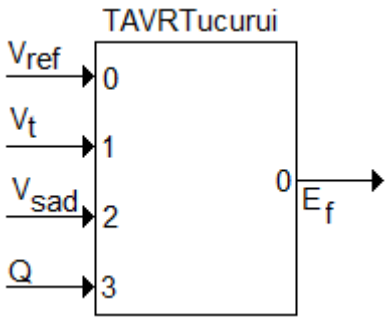

Figura 6-13 - Representação do regulador de tensão no Framework

Ao observar a Figura 6-5 pode-se verificar que este tipo de regulador de tensão possui como atributos: K_a, V_{Amax}, V_{Amin}, K_e, K_{sad}, T_{se}, K_i, A_{ex}, B_{ex}, T_a, T_e, Max, Min e E_{f0}. A numeração das entradas e das saídas é apresentada através da Figura 6-13. A classe que representa o regulador deve ser uma classe derivada de *TBlock* e sobrecarregar os quatro métodos fundamentais que definem o comportamento do dispositivo:

1. Método para o cálculo das condições iniciais: *Initialize*;
2. Método para o cálculo das derivadas das variáveis de estado: *Derivatives*;
3. Método para o cálculo da parcela da matriz jacobiana: *Jacobian*;
4. Método para o cálculo das variáveis de saída e aplicação de não linearidades: *Outputs*.

A declaração da classe *TAVRTucurui* é apresentada na Figura 6-14.

```
1 #ifndef TAVRTUCURUI_H_
2 #define TAVRTUCURUI_H_

3 #include <bcssd/core/include/bcssd.h>

4 namespace bcssd {
5 class TAVRTucurui: public TBlock {
6 private:
7    double Ka, VAmax, Ke, Ksad, Tse, Ki, Aex, Bex;
8    double Ta, VAmin;
9    double Te, Max, Min;
10   double Ef0;
11 public:
12   TAVRTucurui();
13   virtual ~TAVRTucurui();
```

```
14   virtual void
15   Derivatives (TVector<double> & Ysys, TVector<double> & Xstate,
16                TVector<double> & Fstate, double T);

17   virtual tmpTSparseMatrix<double>
18   Jacobian (TVector<double> & Ysys, TVector<double> & Xstat,
19             double T);

20   virtual void
21   Outputs (TVector<double> & Ysys, TVector<double> & Xstate,
22            double T);

23   virtual void
24   Initialize (TVector<double> & Ysys, TVector<double> & Xstate);
25};
26}
27 #endif /* TAVRTUCURUI_H_ */
```

Figura 6-14 - Declaração da classe que representa o regulador de tensão

Na Figura 6-14, nas linhas 7 a 10 estão declarados os atributos da classe. Nas linhas de 14 a 24 são declaradas as sobrecargas dos métodos referidos anteriormente.

No Framework, para definir o número de entradas, saídas e estados e inicializar as estruturas de dados internas deve-se chamar o método *MemInit* no construtor da classe *TAVTTucurui*. Os argumentos deste método são o número de entradas, número de estados e número de saídas respectivamente. Para a classe *TAVRTucurui*, são quatro entradas (V_{ref}, V_t, V_{sad} e Q respectivamente), duas variáveis de estado (X_1 e X_2) e uma saída (E_f). O código fonte do construtor da classe é apresentado na Figura 6-15.

```
TAVRTucurui::TAVRTucurui() {
      MemInit(3,2,1);
}
```

Figura 6-15 - Construtor da classe TAVRTucurui

O cálculo das derivadas das variáveis de estado do regulador de tensão é feito no método sobrecarregado *Derivatives*, onde o sistema de equações (6.11) deve ser avaliado. O código fonte deste método é apresentado na Figura 6-16.

```
1  void TAVRTucurui
2  ::Derivatives (TVector<double> & Ysys,TVector<double> & Xstate,
3                 TVector<double> & Fstate, double T)
4  {
5     // Sinais de entrada
6     double Vref = Ysys[GetInput(0)],
7            VT   = Ysys[GetInput(1)],
8            Vsad = Ysys[GetInput(2)],
9            Q    = Ysys[GetInput(3)];

10    // Variáveis de estado
11    double X1 = Xstate[GetState(0)],
12           X2 = Xstate[GetState(1)];

13    double Iq = Q/VT,
14           E1 = Vref+X1+Vsad*Ksad-Vt;

15    //Derivadas das variáveis de estado
16    double dX1 = (1.0/Ta)*(Ka*Iq-X1),
17           dX2 = (1.0/VAmin)*(VAmax*E1-X2);

18    //Atribuição das derivadas ao vetor de derivadas
19    Fstate[GetState(0)] = dX1;
20    Fstate[GetState(1)] = dX2;
21 }
```

Figura 6-16 - Método para o cálculo das derivadas das variáveis de estado

Na Figura 6-16, o argumento *Ysys* do método *Derivatives,* contem os valores de todas as variáveis do sistema dinâmico que está sendo simulado, portanto, para obter os valores das entradas do bloco deve-se indexar este vetor de forma apropriada. Por exemplo, para descobrir qual a posição no vetor *Ysys* corresponde a entrada *0* do bloco deve-se empregar o método *GetInput(0)* que retornará a posição dentro do *Ysys* correspondente a entrada *0* do bloco. Desta forma, nas linhas 6 até 9 são obtidos os valores das entradas do bloco.

Na Figura 6-16, o argumento *Xstate* contêm os valores de todas as variáveis de estado do sistema dinâmico que está sendo simulado, portanto, para identificar quais as posições correspondentes das variáveis de

estado do bloco utiliza-se o método *GetState*. Nas linhas 11 e 12 são obtidos os valores atuais das duas variáveis de estado do bloco.

Na Figura 6-16, as linhas 13 até 17 avaliam o sistema de equações (6.11). O vetor *Fstate* contem as derivadas de todas as variáveis de estado do sistema dinâmico que está sendo simulado. A indexação de *Fstate* é feita empregando-se o método *GetState* de forma análoga a indexação feita no vetor *Xstate*, desta forma, nas linhas 19 e 20 são inseridos os novos valores calculados das derivadas das variáveis de estado do bloco.

Caso haja necessidade do emprego do método de integração numérica trapezoidal ou método análogo é necessário sobrecarregar o método *Jacobian*. O método *Jacobian* retorna uma matriz esparsa com a contribuição do bloco para a formação da matriz jacobiana do sistema dinâmico inteiro. Através do sistema de equações (6.11) pode-se formar o seguinte vetor de funções:

$$\bar{f}(X)=\begin{bmatrix} f_1(X_1,X_2) \\ f_2(X_1,X_2) \end{bmatrix}=\begin{bmatrix} \frac{1}{T_a}\left(K_a I_q - X_1\right) \\ \frac{1}{V_{A\min}}\left[V_{A\max}\left(V_{ref}+X_1+V_{sad}K_{sad}-V_T\right)-X_2\right] \end{bmatrix} \tag{6.17}$$

$$J=\begin{bmatrix} \frac{\partial}{\partial X_1} f_1(X_1,X_2) & \frac{\partial}{\partial X_2} f_2(X_1,X_2) \\ \frac{\partial}{\partial X_1} f_2(X_1,X_2) & \frac{\partial}{\partial X_2} f_2(X_1,X_2) \end{bmatrix}=\begin{bmatrix} -\frac{1}{T_a} & 0 \\ \frac{V_{A\max}}{V_{A\min}} & -\frac{1}{V_{A\min}} \end{bmatrix} \tag{6.18}$$

Utilizando a equação (6.18) pode-se implementar o método *Jacobian*. O código fonte do método é apresentado na Figura 6-17. Assim como o método *Derivatives*, a implementação do método *Jacobian* é muito simples.

```
1  tmpTSparseMatrix<double>
2  TAVRTucurui::Jacobian (TVector<double> & Ysys,
3  TVector<double> & Xstat, double T)
4  {
5     tmpTSparseMatrix<double> J(2,2,3);
6     J(0,0,-1.0/Ta);
7     J(1,0,VAmax/VAmin); J(1,1,-1.0/VAmin);
8     return J;
9  }
```

Figura 6-17 - Implementação do método Jacobian

Na Figura 6-17, a matriz jacobiana é montada nas linhas 6 e 7. Deve-se notar que o método deve retornar uma matriz esparsa do tipo *tmpTSparseMatrix*, portanto, foi usada a declarada na linha 5.

Os métodos *Derivatives* e *Jacobian* são métodos *callbacks* chamados nas classes de integração numérica do Framework desenvolvido nesta tese, onde são calculadas as variáveis de estado do sistema dinâmico sendo simulado. As variáveis de saída são calculadas a partir dos valores das variáveis de estado, para isso, o Framework invoca os métodos *Outputs* de cada bloco existente no sistema dinâmico.

No caso da classe *TAVRTucurui*, o seu método *Outputs* avalia a equação (6.13) e aplica as não linearidades presentes no modelo (limites). Na Figura 6-18 é apresentado o código fonte do método *Outputs*.

```
1  void TAVRTucurui
2  ::Outputs (TVector<double> & Ysys, TVector<double> & Xstate,
3             double T)
4  {
5     double Vref = Ysys[GetInput(0)],
6            VT   = Ysys[GetInput(1)],
7            Vsad = Ysys[GetInput(2)],
8            Q    = Ysys[GetInput(3)];

9     double X1 = Xstate[GetState(0)],
10           X2 = Xstate[GetState(1)];

11    double E1 = Vref+X1+Vsad*Ksad-VT;

12    if (X2>=Ke) X2 = Ke;
13    if (X2<=Te) X2 = Te;

14    double Ef = X2*Ki+Ef0+E1*Tse*Ki;
15    if (Ef>=Max) Ef = Max;
16    if (Ef<=Min) Ef = Min;

17    if (Ef>=Aex*Vt) Ef = Aex*Vt;
18    if (Ef<=Bex*Vt) Ef = Bex*Vt;

19    Ysys[GetOutput(0)] = Ef;
20 }
```

Figura 6-18 - Implementação do método Outputs

Na Figura 6-18, nas linhas de 5 a 8 são obtidos os valores das entradas, nas linhas de 9 e 10 são obtidos os valores das variáveis de estado, nas linhas de 11 a 18 é calculada a saída do bloco e são aplicadas as não linearidades. Na linha 19 é atribuído o valor da saída do bloco, onde se deve observar que o vetor *Ysys*, que contêm todas as variáveis do sistema dinânico, é indexado pelo valor de retorno do método *GetOutput* para atribuir o valor da saída *0*, E_f, do bloco à posição correspondente no vetor *Ysys*.

O outro método sobrecarregado na classe *TAVRTucurui* é o método *Initialize* que será chamado pelo Framework para inicializar as suas variáveis de estado. Este método avalia o sistema de equações (6.12) para inicializar as suas variáveis de estado. Além das variáveis de estado, este método vai calcular os valores iniciais de V_{ref} e V_{sad}. Os valores iniciais da tensão terminal V_T, tensão de campo E_f e potência reativa já foram determinados no cálculo das condições iniciais das máquinas síncronas (classe *TSyncGen*).

```
1  void TAVRTucurui
2  ::Initialize (TVector<double> & Ysys,
3                TVector<double> & Xstate)
4  {
5     double VT0 = Ysys[GetInput(1)],
6            Q0 = Ysys[GetInput(3)];
7            Ef0 = Ysys[GetOutput(0)];
8     double Vsad0 = 0.0,
9            Iq0 = Q0/VT0,
10           X10 = Ka*Iq0,
11           X20 = 0,
12           Vref0 = VT0-X10;
13    Ysys[GetInput(0)] = Vref0;
14    Ysys[GetInput(2)] = Vsad0;
15    Xstate[GetState(0)] = X10;
16    Xstate[GetState(1)] = X20;
17 }
```

Figura 6-19 - Inicialização das variáveis de estado de TAVRTucurui

Na Figura 6-19, nas linhas de 5 a 8 são obtidos os valores iniciais já conhecidos. Nas linhas de 9 a 12 é avaliado o sistema de equações (6.12). Nas linhas de 13 a 16 são atribuídos os valores iniciais aos seus respectivos vetores para serem usados nas rotinas internas do Framework.

No exemplo apresentado nesta seção, pode ser verificado que a criação de uma classe para representar um dispositivo dinâmico utilizando o Framework é muito simples. Para utilizar a classe criada, deve-se instanciá-la e fazer as devidas conexões com os outros blocos a serem usados na simulação. Portanto, a partir das equações de estado é uma tarefa muito simples desenvolver uma classe que o implemente, basta seguir os procedimentos ilustrados nesta seção. Nas seções seguintes são apresentados os diagramas em blocos e as equações de estado dos dispositivos empregados na simulação da UHE Tucuruí, portanto, contendo as informações suficientes para o desenvolvimento das classes que foram omitidas nesta tese devido a limitação de espaço no documento.

6.1.3 Estabilizadores de Sistemas de Potência

Através da Figura 6-20 é apresenta a gaveta em que se localiza o Estabilizador de Sistemas de Potência, ESP, na UGH 8, em ensaios realizados em fevereiro de 2009.

Figura 6-20 - Gaveta onde se encontra o ESP

6.1.3.1 Modelagem em Diagrama de Blocos

O modelo do ESP da primeira etapa foi baseado na referência (Operador Nacional do Sistema, 2008), cujo Diagrama em blocos é apresentado através da Figura 6-21.

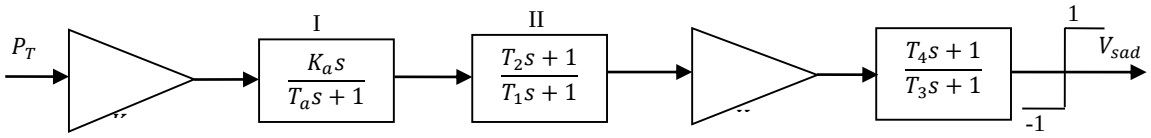

Figura 6-21 - Diagrama de Blocos do ESP da primeira etapa

O modelo do ESP da segunda etapa foi baseado no modelo das referências (Operador Nacional do Sistema, 2004) e (Operador Nacional do Sistema, 2008), e na Figura 6-22 é apresentado o diagrama em blocos.

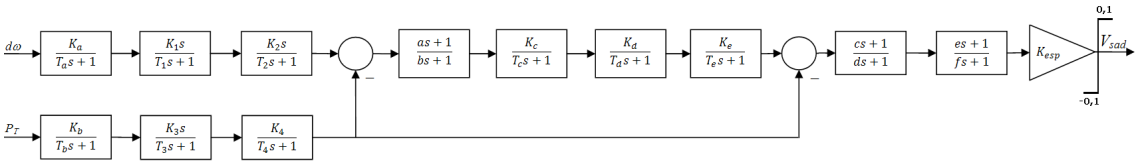

Figura 6-22 - Diagrama de Blocos do ESP da segunda etapa

6.1.3.2 Modelagem em Espaços de Estados

As equações de estado com a dinâmica do ESP da primeira etapa são dadas pelo sistema de equações (6.19).

$$\begin{cases} \dfrac{dX_1}{dt} = \dfrac{1}{T_a}\left[K_1 P_T\left(-\dfrac{K_a}{T_a} - X_1\right)\right] \\ \dfrac{dX_2}{dt} = \dfrac{1}{T_1}\left[\left(1-\dfrac{T_2}{T_1}\right)E_1 - X_2\right] \\ \dfrac{dX_3}{dt} = \dfrac{1}{T_3}\left[\left(1-\dfrac{T_4}{T_3}\right)K_2 E_2 - X_3\right] \end{cases} \tag{6.19}$$

A expressão (6.20) é usada para determinar as condições inicias e na expressão (6.21) o cálculo da saída V_{sad} do ESP da primeira etapa.

$$\begin{cases} X_1 = K_1 P_T\left(-\dfrac{K_a}{T_a}\right) \\ X_2 = 0 \\ X_3 = 0 \end{cases} \tag{6.20}$$

$$\begin{cases} E_1 = K_1 P_T\left(\dfrac{K_a}{T_a}\right) + X_1 \\ E_2 = E_1\left(\dfrac{T_2}{T_1}\right) + X_2 \\ V_{sad} = E_2 K_2\left(\dfrac{T_4}{T_3}\right) + X_3 \end{cases} \tag{6.21}$$

Para o ESP da segunda etapa as equações de estado são dadas por (6.22).

$$
\begin{cases}
\dfrac{dX_1}{dt} = \dfrac{1}{T_a}\left(d\omega K_a - X_1\right) \\
\dfrac{dX_2}{dt} = \dfrac{1}{T_1}\left[X_1\left(-\dfrac{K_1}{T_1}\right) - X_2\right] \\
\dfrac{dX_3}{dt} = \dfrac{1}{T_2}\left[E_1\left(-\dfrac{K_2}{T_2}\right) - X_3\right] \\
\dfrac{dX_4}{dt} = \dfrac{1}{T_b}\left(P_T K_b - X_4\right) \\
\dfrac{dX_5}{dt} = \dfrac{1}{T_3}\left[X_4\left(-\dfrac{K_3}{T_3}\right) - X_5\right] \\
\dfrac{dX_6}{dt} = \dfrac{1}{T_4}\left(E_3 K_4 - X_6\right) \\
\dfrac{dX_7}{dt} = \dfrac{1}{b}\left[E_2\left(1-\dfrac{a}{b}\right) - X_7\right] \\
\dfrac{dX_8}{dt} = \dfrac{1}{T_C}\left(E_4 K_C - X_8\right) \\
\dfrac{dX_9}{dt} = \dfrac{1}{T_d}\left(X_8 K_d - X_9\right) \\
\dfrac{dX_{10}}{dt} = \dfrac{1}{T_e}\left(X_9 K_e - X_{10}\right) \\
\dfrac{dX_{11}}{dt} = \dfrac{1}{d}\left[E_5\left(1-\dfrac{c}{d}\right) - x_{11}\right] \\
\dfrac{dX_{12}}{dt} = \dfrac{1}{f}\left[E_6\left(1-\dfrac{e}{f}\right) - X_{12}\right]
\end{cases}
\tag{6.22}
$$

As condições iniciais são calculadas utilizando a expressão (6.23), e a saída V_{sad} é dada pela expressão (6.24).

$$
\begin{cases}
\Delta\omega = \omega - \omega_0 \\
E_1 = X_1 \dfrac{K_1}{T_1} \\
E_2 = E_1 \dfrac{K_2}{T_2} + X_3 + X_6 \\
E_3 = X_4 \dfrac{K_3}{T_3} + X_5 \\
E_4 = X_2 \dfrac{a}{b} + X_7 \\
E_5 = X_{10} - X_6 \\
E_6 = E_5 \dfrac{c}{d} + X_{11} \\
X_1 = \Delta\omega K_a \\
X_2 = X_1 \dfrac{K_3}{T_3} \\
X_3 = -E_1 \dfrac{K_2}{T_2} \\
X_4 = P_T K_b \\
X_5 = -X_4 \dfrac{K_3}{T_3} \\
X_6 = E_3 K_4 \\
X_7 = E_2\left(1-\dfrac{a}{b}\right) \\
X_8 = E_4 K_c \\
X_9 = X_8 K_d \\
X_{10} = X_9 K_e \\
X_{11} = E_5\left(1-\dfrac{c}{d}\right) \\
X_{12} = E_6\left(1-\dfrac{e}{f}\right)
\end{cases}
\tag{6.23}
$$

$$\begin{cases} E_7 = E_6 \dfrac{e}{f} + X_{12} \\ V_{sad} = E_7 K_{esp} \end{cases} \tag{6.24}$$

6.1.4 Turbinas

Através da Figura 6-23 é apresenta uma ilustração de turbina hidráulica Francis, do tipo das instaladas na UHE de Tucuruí.

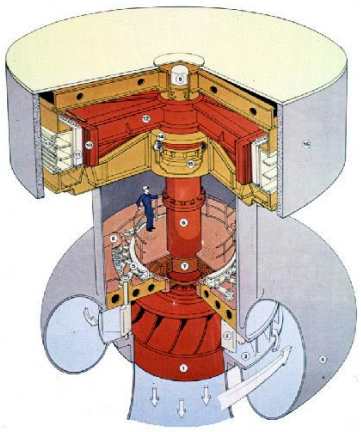

Figura 6-23 - Turbina Francis, similar às instaladas na UHE de Tucuruí (Wikipédia: A enciclopédia livre)

6.1.4.1 Modelagem em Diagrama de Blocos

O modelo utilizado para implementação é não linear e apresentado através do diagrama na Figura 6-24, os quais também servem de referência para os estudos do ONS.

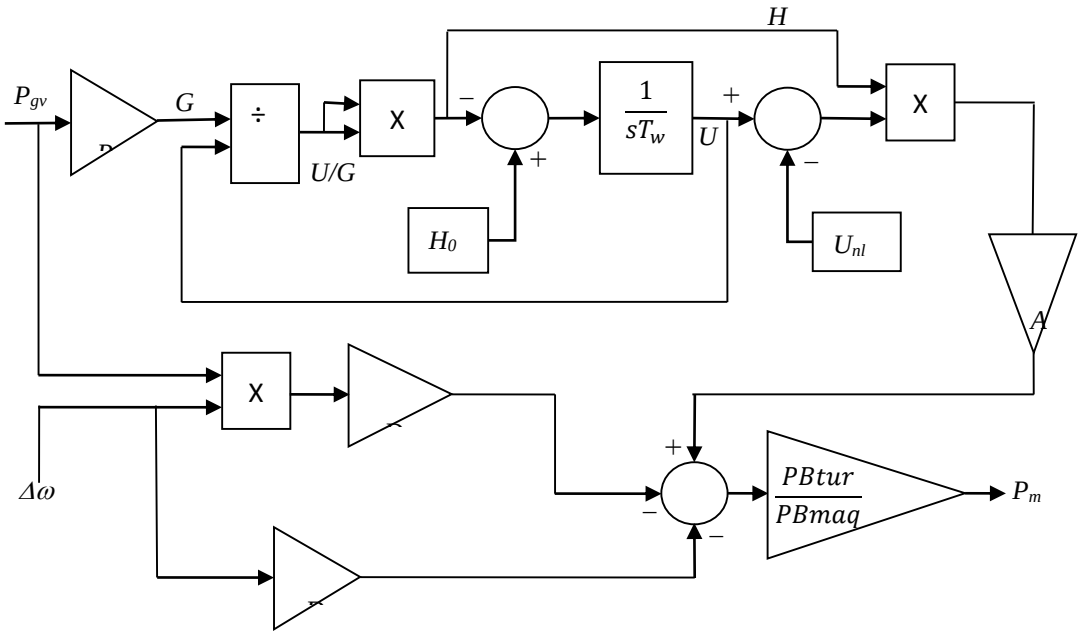

Figura 6-24 - Modelo das turbinas empregado

6.1.4.2 Modelagem em Espaços de Estados

Ambos os modelos não lineares de turbina apresentados são sistemas de primeira ordem, portando, possuindo apenas uma equação de estado. A equação de estado é apresentada em (6.25).

$$\begin{cases} G = gP_b \\ H = \left(\dfrac{U}{G}\right)^2 \\ \dfrac{dU}{dt} = \dfrac{H_0 - H}{T_W} \end{cases} \tag{6.25}$$

As condições iniciais são apresentadas em (6.26), e a saída em (6.27).

$$\begin{cases} U = \dfrac{P_m}{A_t \dfrac{PB_{tur}}{PB_{maq}}} + U_{nl} \\ G = \dfrac{U}{\sqrt{H_0}} \\ g = P_{gv} = \dfrac{G}{P_b} \end{cases} \tag{6.26}$$

$$\begin{cases} G = gP_b \\ \Delta\omega = \omega - \omega_0 \\ H = \left(\dfrac{U}{G}\right)^2 \\ P_m = \dfrac{PB_{tur}}{PB_{maq}} A_t H\left(U - U_{nl}\right) - \Delta\omega\left(D + D_t g\right) \end{cases} \tag{6.27}$$

6.1.5 Reguladores de Velocidade

Através da Figura 6-25 é apresentado o armário em que se localiza o regulador de velocidade, RV, da UGH 8, em ensaios realizados em fevereiro de 2009.

Figura 6-25 - Ensaios sendo realizados no RV da UGH 8

6.1.5.1 Modelagem em Diagrama de Blocos

O modelo do RV da primeira etapa foi baseado em (Operador Nacional do Sistema, 2003a), (Operador Nacional do Sistema, 2003b) e (Operador Nacional do Sistema, 2008) cujo diagrama em blocos é apresentado através da Figura 6-26. Nota-se que foram modelados a válvula atuadora e servo-motor de comando, a válvula distribuidora e servo-motor principal, o servo-motor principal e os estatismos permanente e transitório.

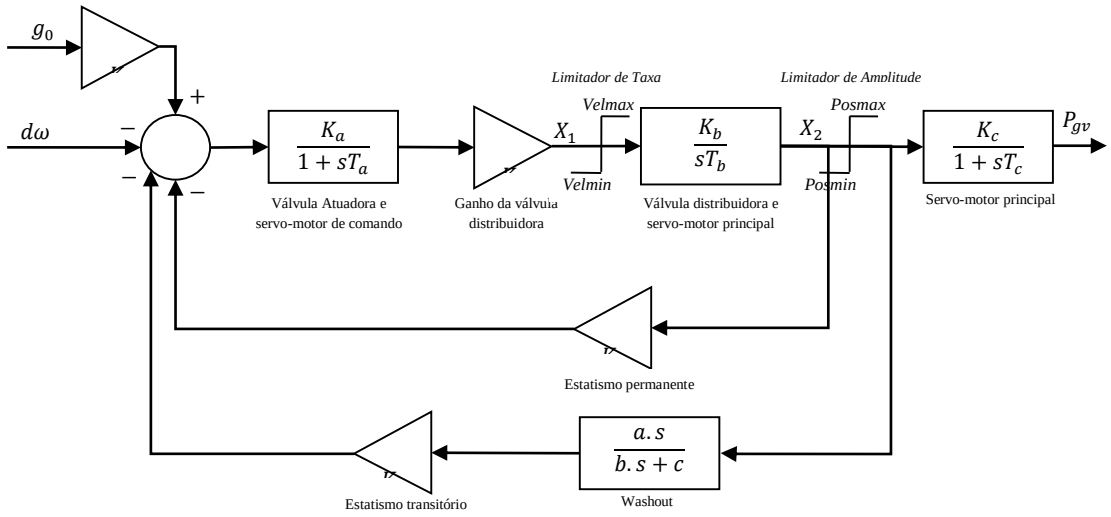

Figura 6-26 - Diagrama de Blocos do RV da primeira etapa

O modelo do RV da segunda etapa foi baseado em (Operador Nacional do Sistema, 2006a), (Operador Nacional do Sistema, 2006b) e (Operador Nacional do Sistema, 2008), e através da Figura 6-27 é apresentado o diagrama em blocos onde se pode observar a modelagem do regulador e do sistema de acionamento hidráulico.

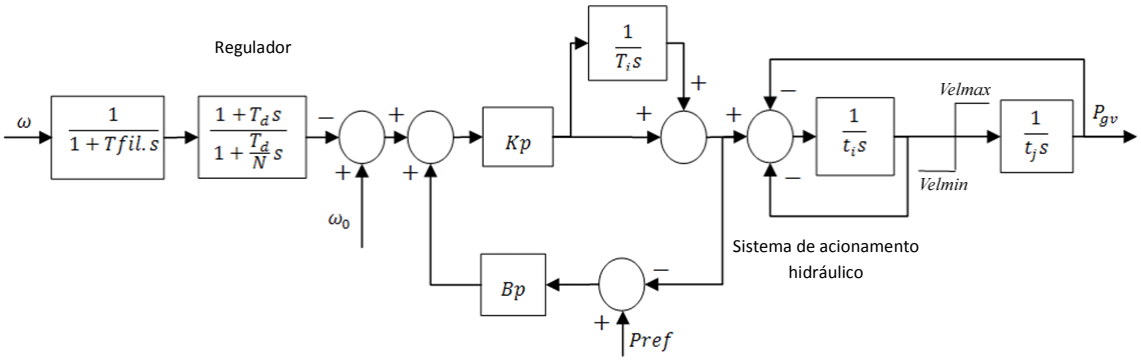

Figura 6-27 - Diagrama de Blocos do RV da segunda etapa

6.1.5.2 Modelagem em Espaços de Estados

As equações de estado com a dinâmica do RV da primeira etapa estão apresentadas em (6.28).

$$\begin{cases} E_1 = K_1 g_0 - \Delta\omega - K_3 X_2 - K_4 X_2 \dfrac{a}{b} + K_4 X_3 \\ \dfrac{dP_{gv}}{dt} = \dfrac{1}{T_c}\left(K_c X_2 - P_{gv}\right) \\ \dfrac{dX_1}{dt} = \dfrac{1}{T_a}\left(K_2 K_a E_1 - X_1\right) \\ \dfrac{dX_2}{dt} = \dfrac{1}{T_b} X_1 \\ \dfrac{dX_3}{dt} = \dfrac{ac}{b^2} X_2 - \dfrac{c}{b} X_3 \end{cases} \tag{6.28}$$

Em (6.29) são apresentadas as condições inicias e em (6.30) o cálculo da saída, devendo-se considerar também que X_2 é limitado em amplitude entre *Posmin* e *Posmax*.

$$\begin{cases} X_1 = 0 \\ X_2 = \dfrac{P_{gv}}{K_c} = \dfrac{g}{K_c} \\ X_3 = \dfrac{a}{b} X_2 \end{cases} \tag{6.29}$$

$$P_{gv} = g \tag{6.30}$$

Para o RV da segunda etapa, e seguindo também o procedimento para retirada dos zeros, as equações de estado são apresentadas em (6.31).

$$\begin{cases} E_1 = X_2 + NX_1 \\ E_2 = \dfrac{1}{1+K_pB_p}\left[\omega_0 - E_1 + B_p\left(g_0 - X_3\right)\right] \\ g_1 = X_3 + E_2K_p \\ E_3 = g_1 - P_{gv} - X_4 \\ \dfrac{dP_{gv}}{dt} = \dfrac{1}{t_j}X_4 \\ \dfrac{dX_1}{dt} = \dfrac{1}{t_{fil}}\left(\omega - X_1\right) \\ \dfrac{dX_2}{dt} = \dfrac{N}{T_d}\left[X_1\left(1-N\right) - X_2\right] \\ \dfrac{dX_3}{dt} = \dfrac{K_p}{T_i}E_2 \\ \dfrac{dX_4}{dt} = \dfrac{1}{t_i}E_3 \end{cases} \tag{6.31}$$

As condições iniciais são apresentadas em (6.32), e a saída em (6.33), considerando que sua amplitude é limitada entre os valores *Lmax* e *Lmin*.

$$\begin{cases} X_1 = \omega \\ X_2 = \left(1-N\right)X_1 \\ X_3 = P_{gv} = g \\ X_4 = 0 \\ P_{ref} = P_{gv} \end{cases} \tag{6.32}$$

$$P_{gv} = g \tag{6.33}$$

6.2 Comparações entre as simulações e medições em campo

Nesta seção serão abordados os resultados obtidos com a implementação das UGHs da UHE de Tucuruí no ambiente de simulação do *Framework* e alguns ensaios de campo na usina para validação dos resultados obtidos.

6.2.1 Simulações implementadas

Foi descrito no capítulo "Fluxo de carga" que as rotinas de cálculo do fluxo de carga são parte integrante do Framework, portanto, não é necessário o emprego de aplicativos separados com este objetivo.

Na Tabela 6-1 é apresentado o resultado de um fluxo de carga calculado utilizando-se o Framework, para um dado ponto de operação da UHE de Tucuruí, utilizando o método de Newton-Raphson. Para este caso, foi utilizada uma tolerância de 10^{-6}, o fluxo de carga convergiu em *4* iterações.

Barra	P_t (MW)	Q_t (MVAr)	V_t (p.u.)	Θ (Grau)	P_L (MW)	Q_L (MVAr)
1	315	23,5298	1	9,1309	0	0
2	315	23,5298	1	9,1309	0	0
3	315	23,5298	1	9,1309	0	0
4	315	23,5298	1	9,1309	0	0
5	315	23,5298	1	9,1309	0	0
6	315	23,5298	1	9,1309	0	0
7	315	23,5298	1	9,1309	0	0
8	315	23,5298	1	9,1309	0	0
9	315	23,5298	1	9,1309	0	0
10	315	23,5298	1	9,1309	0	0
12	315	23,5298	1	9,1309	0	0
13	315	23,5298	1	9,1309	0	0
14	315	23,5298	1	9,1309	0	0
15	315	23,5298	1	9,1309	0	0
16	315	23,5298	1	9,1309	0	0
17	315	23,5298	1	9,1309	0	0
18	315	23,5298	1	9,1309	0	0
19	315	23,5298	1	9,1309	0	0
20	315	23,5298	1	9,1309	0	0
21	315	23,5298	1	9,1309	0	0
22	315	23,5298	1	9,1309	0	0
23	315	23,5298	1	9,1309	0	0
24	-4418,08	611,249	1	7,74944e-013	0	0
25	0	0	0,999995	0,590849	1	0
26	0	0	0,999995	0,590849	1	0

27	0	0	0,999995	0,590849	1	0
28	0	0	0,999995	0,590849	1	0
29	0	0	0,999995	0,590849	1	0
30	0	0	0,999995	0,590849	1	0
31	0	0	0,999995	0,590849	1	0
32	0	0	0,999995	0,590849	1	0
33	0	0	0,999995	0,590849	1	0
34	0	0	0,999995	0,590849	1	0
35	0	0	0,999996	0,614545	1	0
36	10	0	0,999992	0,543533	0	0
37	0	0	1	1,19336e-04	2800	0

Tabela 6-1 - Resultado do fluxo de carga

Foram implementadas as faltas:

1. Curto-circuito trifásico, informando-se a admitância complexa de curto-circuito;
2. Degrau na referência do RAT, informando-se um número real que será multiplicado pela referência durante um tempo determinado, e
3. Degrau na referência do RV, informando-se um número real que será multiplicado pela referência durante um tempo determinado.

Na Figura 6-28 são apresentadas duas simulações de *4s* de um curto-circuito trifásico executadas com o uso do Framework, na barra 13. Esta falta foi aplicada em *0,5s*, com duração de *10ms*, e com uma admitância de curto-circuito de $10^5(1+j)$ *p.u.* Em azul a potência ativa com todos os controles ligados, porém, com o ESP da máquina desligado, e em vermelho a mesma simulação com o ESP da máquina ligado. Com o ESP ligado, percebe-se um maior amortecimento das oscilações, como era de se esperar.

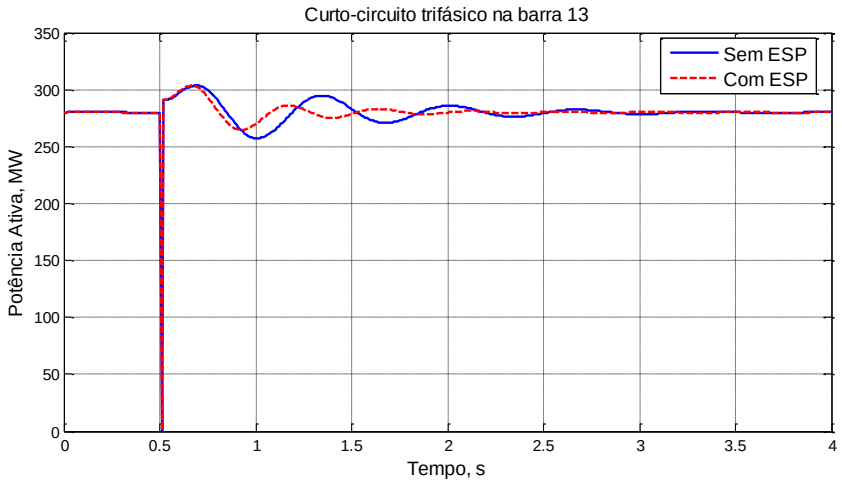

Figura 6-28 - Curto-circuito trifásico aplicado na máquina 13

Através da Figura 6-29 são apresentadas duas simulações de um degrau de +10% na referência do RAT da máquina 13, também com e sem a atuação do ESP.

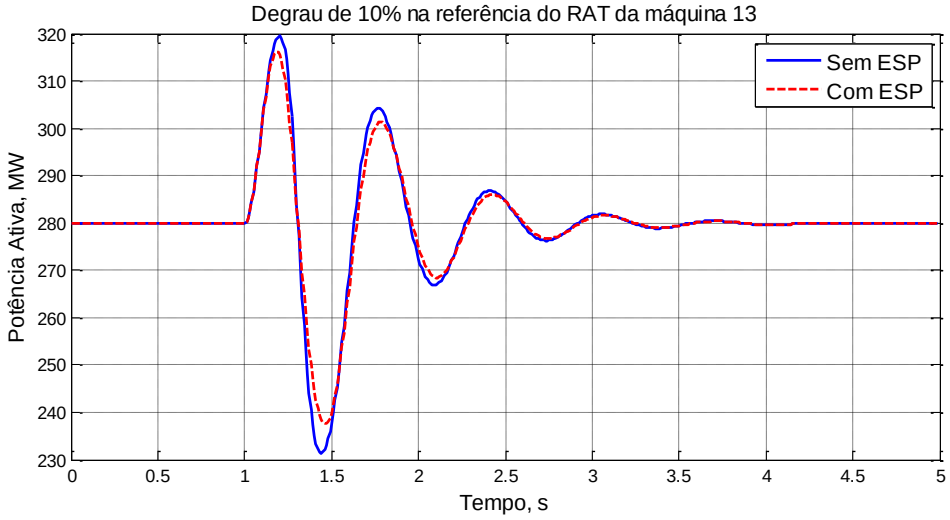

Figura 6-29 - Degrau de +10% aplicado na referência do RAT da máquina 13

Através da Figura 6-30 é apresenta a potência ativa de uma simulação de *40s*, com a aplicação de um degrau de +10% na referência do RV da máquina 13. Observa-se o comportamento esperado de fase não mínima exibido pelo sistema.

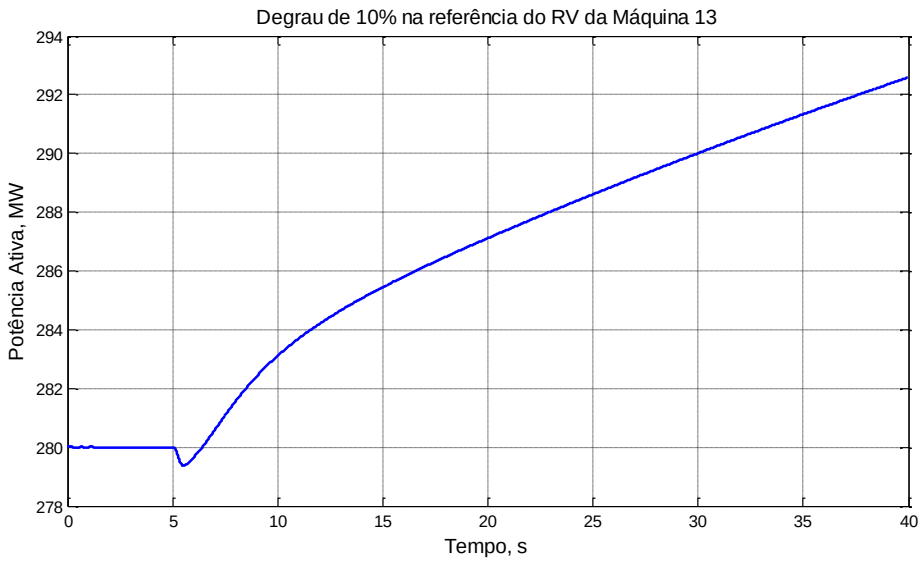

Figura 6-30 - Degrau de +10% aplicado referência do RV da máquina 13

6.2.2 Comparações com medições obtidas em campo

Através da Figura 6-31 é apresentado o resultado de medição e simulação apresentados em (Operador Nacional do Sistema, 2003a), com o modelo do SIN do ano de 2003. O ensaio e as simulações foram realizados com a máquina sincronizada com o sistema injetando uma potência de 300MW quando um degrau de potência de +10% é aplicado. Em vermelho o resultado da simulação, feita no ANATEM, e em azul a medição em campo.

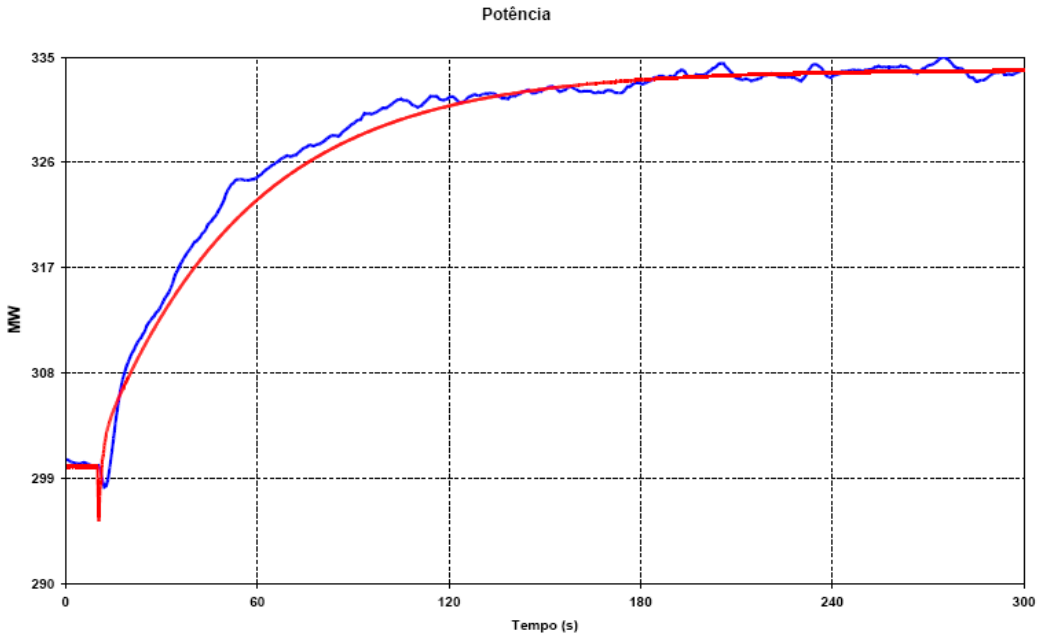

Figura 6-31 - Degrau de +10% aplicado na referência do RV de uma máquina da primeira etapa, obtido pelo ONS, em azul dados medidos e em vermelho a simulação

A Figura 6-32 apresenta os resultados deste mesmo cenário obtidos com uso do Framework. Nota-se, perfeitamente, a similaridade do comportamento dinâmico.

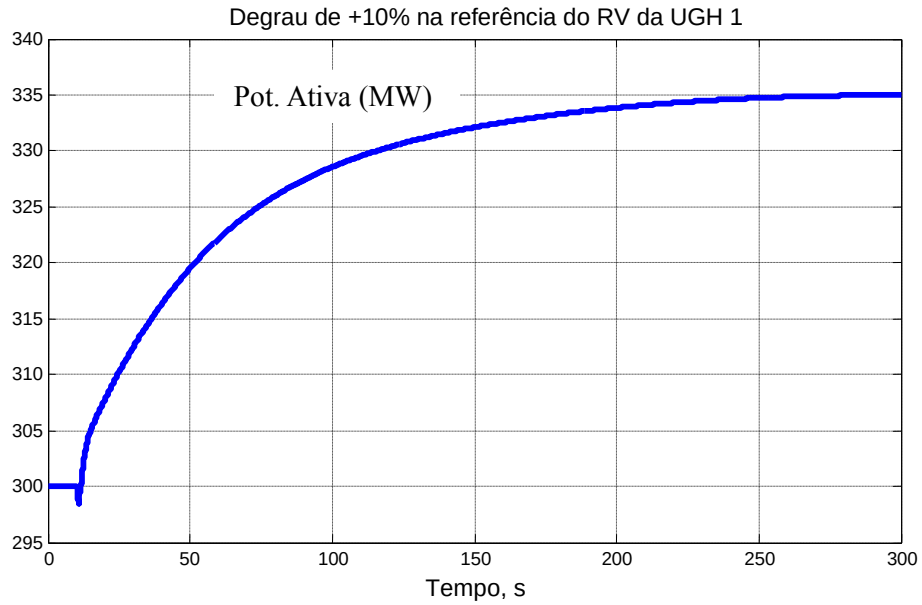

Figura 6-32 - Degrau de +10% aplicado na referência do RV de uma máquina da primeira etapa, utilizando Framework

Em (Operador Nacional do Sistema, 2003a), é descrito um ensaio em que a máquina opera em 330MW e é aplicado um degrau de -30MW (ou -9,09% da potência atual), monitorando a potência do gerador. Na Figura 6-33, em vermelho o resultado da simulação, feita no ANATEM, e em azul a medição em campo.

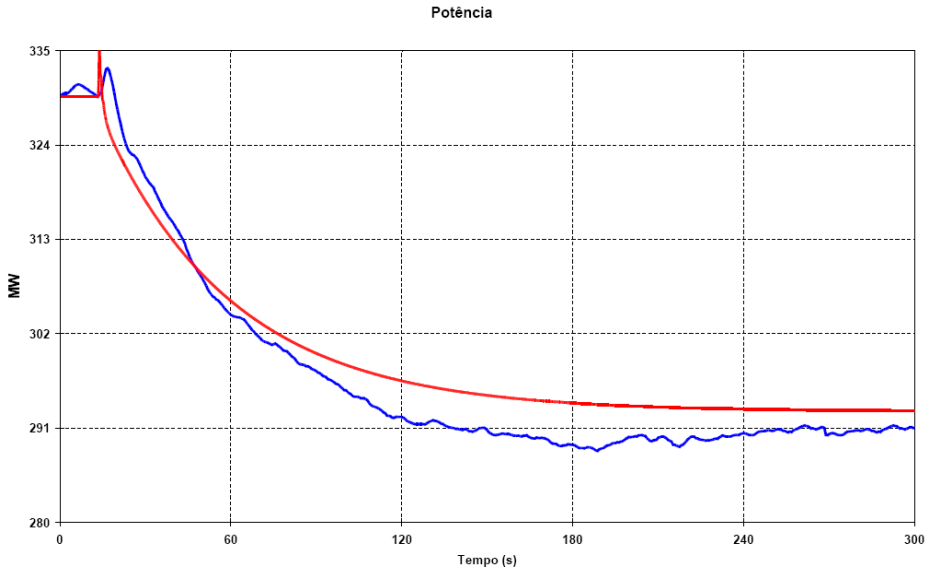

Figura 6-33 - Degrau de -9,09% aplicado na referência do RV de uma máquina da primeira etapa, obtido pelo ONS (Operador Nacional do Sistema, 2003a)

Esta mesma situação foi simulada com uso do Framework, cujo resultado é apresentado através da Figura 6-34. Pode-se verificar a similaridade do comportamento dinâmico.

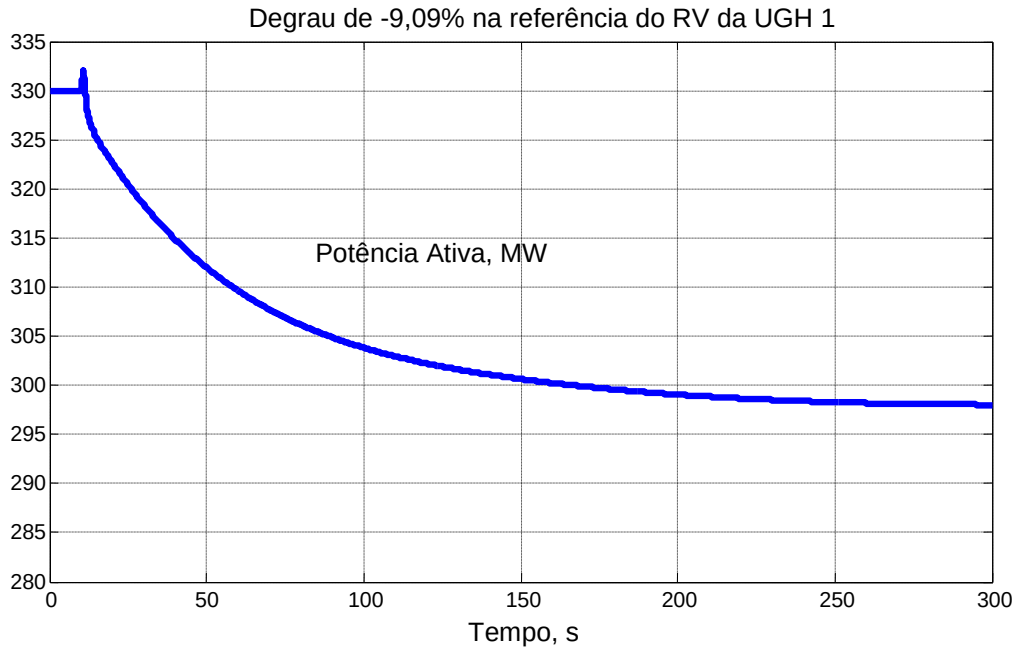

Figura 6-34 - Degrau de -9,09% aplicado na referência do RV de uma máquina da primeira etapa, utilizando o Framework

Em ensaios realizados pelo grupo de sistemas de controle da UFPA e a equipe de regulação da UHE Tucuruí em abril de 2009 na UGH 8, a qual operava em 0,733 p.u., ou 256,5 MW, foi aplicado um degrau de +2,9% em 35 s na referência do RV, com duração de 200 s, e tempo total de 450 s. Através da Figura 6-35 é apresenta a comparação das curvas de potência ativa medida neste ensaio, comparados aos resultados de simulação no Framework.

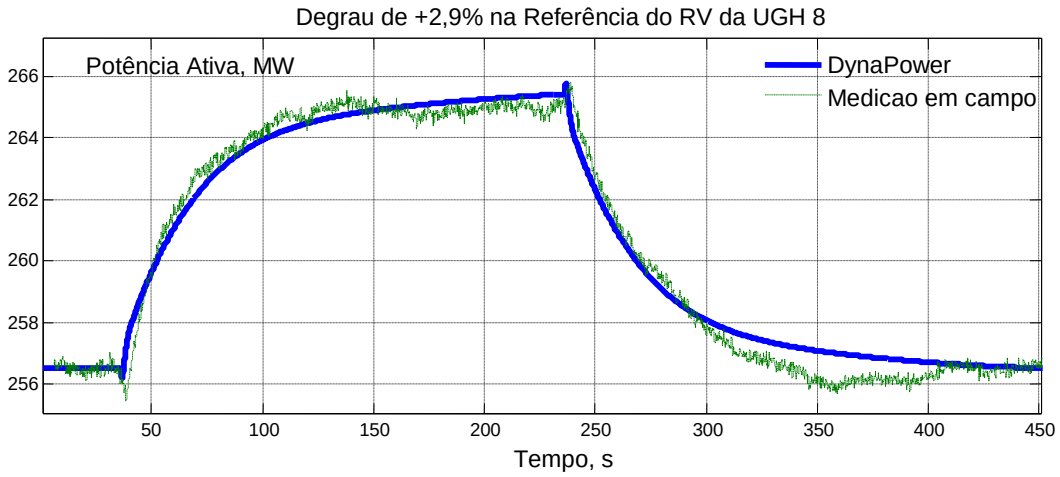

Figura 6-35 - Degrau de +2,9% aplicado na referência do RV da UGH 8, medido em campo e comparado com simulações utilizando o Framework

Através da Figura 6-36 são apresentados resultados de medição e simulação em ANATEM, realizados pelo ONS e descritos em (Operador Nacional do Sistema, 2006b) com uma máquina da segunda etapa, operando

a 0,68 p.u. ou 265 MW. Foi aplicado um degrau de aproximadamente -25% em 90 s e retirado em 190 s. Em vermelho está o resultado da medição em campo e em azul o resultado obtido pela simulação em ANATEM, com um modelo do SIN de 2006.

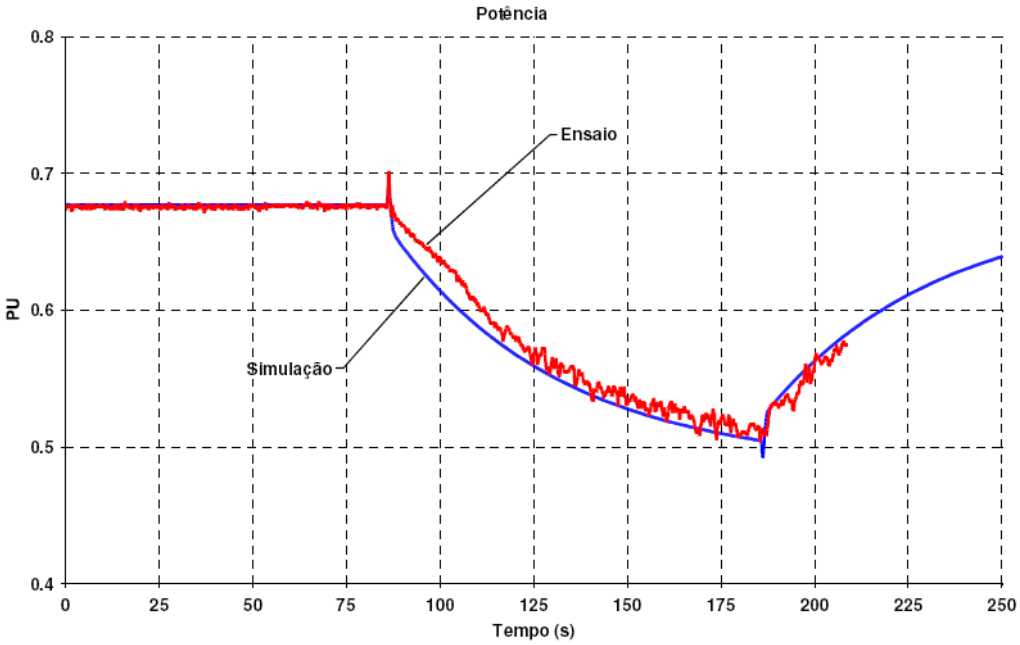

Figura 6-36 - Degrau de -25% aplicado na referência do RV de uma máquina da segunda etapa, obtido pelo ONS

A mesma situação foi simulada no Framework, cujo resultado está na Figura 6-37.

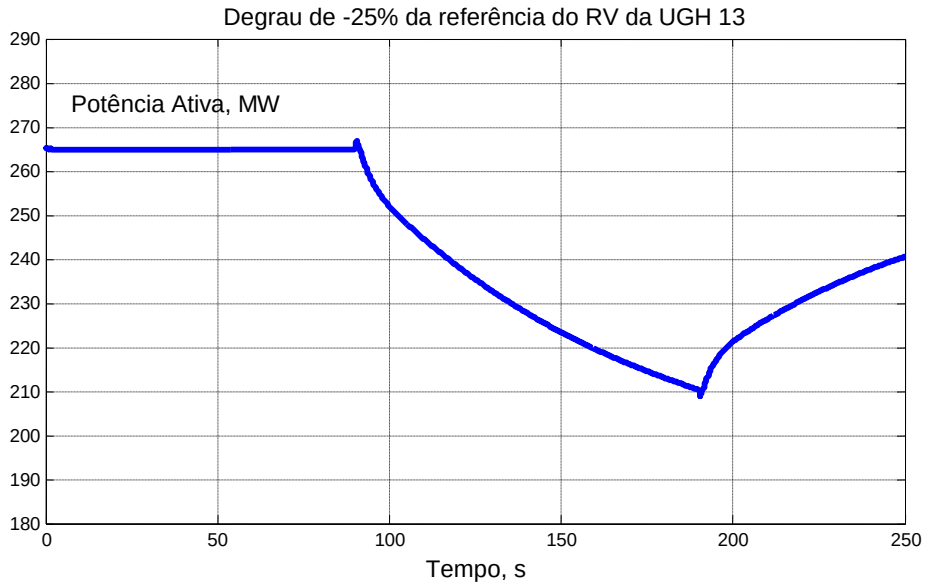

Figura 6-37 - Degrau de -25% aplicado na referência do RV de uma máquina da segunda etapa, utilizando o Framework

6.3 Conclusão

Neste capítulo foi apresentado um estudo de caso em que a maior usina hidrelétrica brasileira teve os modelos dos componentes, que compõe suas unidades de geração, obtidos e avaliados através de ensaios. Foi desenvolvido um simulador, empregando o Framework apresentado nesta tese, cujos resultados foram comparados aos resultados dos ensaios correspondentes.

Ao realizar as comparações entre os resultados obtidos com o Framework e os resultados dos ensaios de campo, pode-se verificar que o simulador representou, com grande precisão, o comportamento eletromecânico da usina hidrelétrica.

Com os resultados apresentados neste capítulo espera-se ter demonstrado que o Framework desenvolvido nesta tese apresenta resultados confiáveis. Ao comparar as simulações realizadas com o Framework e os resultados de medições de campo pode-se verificar que tanto o Framework como os modelos empregados são válidos.

Neste capítulo pode-se verificar que o Framework fornece ferramentas que simplificam o desenvolvimento de simuladores de sistemas elétricos de potência convencionais, no entanto, a principal vantagem do emprego deste Framework está no desenvolvimento de simuladores onde dispositivos de controle e sistemas de geração não-convencionais estão presentes.

No próximo capítulo da tese, é apresentado um estudo de caso onde um sistema de potência hipotético composto por quatro máquinas síncronas e um parque fotovoltaico são simulados, com o objeto de demonstrar o emprego do Framework na simulação de unidades de geração não-convencionais conectados a um sistema de transmissão.

7 Estudo de caso 2: aplicação do Framework em estudo da interconexão de parques fotovoltaicos em sistemas multimáquinas

7.1 Introdução

Neste estudo de caso é investigado o desempenho dinâmico de um sistema de geração Fotovoltaico (PV) integrado a um sistema de potência hipotético com quatro máquinas síncronas, de modo a atender a conversão adequada da corrente contínua (CC), gerada pelos painéis fotovoltaicos, em corrente alternada (CA) de saída, a ser injetada na rede. No sistema de geração fotovoltaico foram incluídos os controles de corrente, controle de carga da capacitância do link CC e um rastreador de potência máxima (MPPT) digital. A análise do desempenho dos controladores é avaliada utilizando-se técnicas baseadas em análise de resposta no domínio do tempo. O estudo é baseado em um programa simulador, desenvolvido utilizando o Framework apresentado nesta tese. Através da utilização deste estudo de caso é possível observar que a utilização do Framework permite a realização de testes de diversos cenários dinâmicos e de regime permanente em sistemas de potência convencionais conectados geradores distribuídos inseridos na rede através de inversores.

Os modelos dos componentes do sistema de potência foram formulados e analisados em uma referência síncrona girando na frequência elétrica de cada unidade de geração (referência *dq*). O conversor CC/CA foi modelado utilizando-se o comportamento médio das tensões e correntes em um período de um ciclo de chaveamento, seguindo a mesma metodologia de (Fonseca, Gomes, Barra Jr, & Sena, 2012).

Ao apresentar este estudo de caso, o autor espera ter demonstrado a aplicabilidade do Framework na implementação de modelos computacionais para os mais diversos tipos de equipamentos de geração de energia elétrica que existam ou venham a existir no futuro bem como a implementação de dispositivos digitais em uma simulação mista.

7.2 Descrição do sistema de potência simulador

O sistema de potência hipotético utilizado neste estudo de caso é baseado no sistema da referência (Kundur, 1994). Foram feitas alterações nos parâmetros das máquinas e dos reguladores com o objetivo de manter o sistema estável mesmo na ocorrência de curtos-circuitos trifásicos sem a necessidade do emprego de estabilizadores de sistemas de potência. O diagrama unifilar do sistema é apresentado através da Figura 7-1.

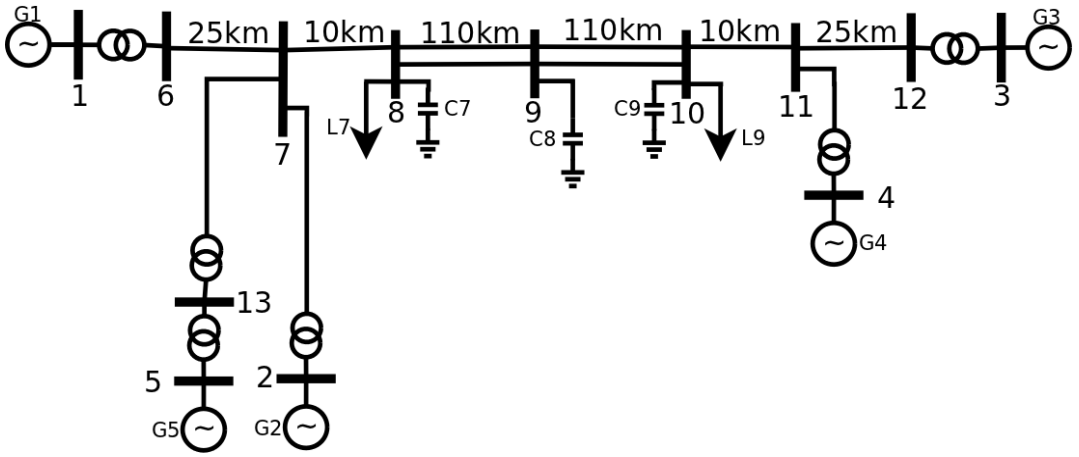

Figura 7-1 Diagrama unifilar do sistema utilizado

As barras foram numeradas de acordo com o tipo de barra. A barra 1 é a barra de referência, as barras de 2 a 5 são barras PV e as demais são as barras PQ.

A barra 5 corresponde a barra onde o gerador fotovoltaico (PV) está conectado na baixa tensão, sendo conectado na barra 13 através de um transformador elevador para posteriormente ser conectado a rede de transmissão. A tensão nos terminais do gerador PV, para esta simulação, é de 600V (na barra 5). O gerador PV está conectado a uma central coletora onde existe um transformador elevador, conectando as barras 5 e 13, que eleva a tensão de 600V para 13,8kV. Na barra 13 está o primário de um transformador que eleva a tensão de 13,8kV para o nível de tensão de transmissão de 230kV que injeta a potência de 2,45MW produzida pelo gerador fotovoltaico no sistema de transmissão.

É desejável que o gerador PV opere com fator de potência o mais próximo possível de 100%, para obter isto, em um primeiro momento, no fluxo de carga, deve-se classificar a barra 5 como uma barra PQ com potência de carga negativa igual a -2,45MW (potência a ser gerada) e potência reativa de carga igual a zero, então o módulo da tensão calculada, através deste primeiro fluxo de carga nesta barra, será usado no fluxo de carga do sistema que se deseja simular, cujo resultado é apresentado através da Tabela 7-1.

# Barra	Tensão (pu)	Fase (°)	P gerada (MW)	Q gerada (MVAr)	P carga (MW)	Q carga (MVAr)
1	1,03	20,2	301	164	0	0
2	1,01	20,1	700	355	0	0
3	1,03	31,6	719	238	0	0
4	1,01	21,2	700	378	0	0
5	0,958	15,5	2,45	0,0238	0	0
6	1	17,4	0	0	0	0
7	0,958	13,2	0	0	0	0

8	0,91	6,88	0	0	967	300
9	0,825	6,61	0	0	0	500
10	0,917	5,23	0	0	1.400	100
11	0,955	14,3	0	0	0	0
12	0,998	24,9	0	0	0	0
13	0,958	13,2	0	0	0	0

Tabela 7-1 - Dados de barras após o fluxo de carga

As impedâncias série e susceptâncias shunt das linhas são respectivamente: na base de 100MVA e 230kV e na base 900MVA e 230kV.

Os transformadores das máquinas de 1 a 5 possuem as reatâncias série de 0,15 pu na base de 900MVA. O transformador entre as barras 5 e 13 possui uma reatância de 1,5 pu na base de 100MVA e o transformador entre as barras 13 e 7 possui uma reatância de 0,017 pu na base de 100MVA.

7.3 Implementando a rede elétrica

Para montar a rede elétrica deve-se criar uma instância da classe *TNetwork* onde devem ser passados a potência base, a tensão base, os erros admissíveis e o número máximo de iterações do fluxo de carga, como apresentado no fragmento de código apresentado através da Figura 7-2.

```
using namespace std;
using namespace bcssd;
int main() {
 TNetwork *rede = new TNetwork(100, //Potência base em MVA
                               230, //Tensão base em kV
                               1e-6,//Erro de potência ativa em pu
                               1e-6,//Erro de potência reativa em pu
                               20); //Num. máximo de iterações
.....................................................................
```

Figura 7-2 - Declaração de uma rede elétrica

O passo seguinte é cadastrar as várias barras do sistema. As barras receberão numerações sequencias de acordo com a ordem em que são cadastradas, portanto, deve-se cadastrar em primeiro lugar a barra de referência, em seguida as barras PV e, por último, as barras PQ. Para realizar este cadastro é utilizado o método *AddBus* da classe *TNetwork*.

(0) Neste estudo de caso, é utilizado uma sintaxe simplificada dos métodos estáticos *CreateRefBus*,

CreatePVBus e *CreatePQBus*. A sintaxe usada para cadastrar uma barra de referência é:

```
rede->AddBus(TBus::CreateRefBus(
            "Nome", Bsh, Vmag, Vang, Pg, Qg, Pl, Ql, Sbase, Vbase));
```

onde:

Argumento	Descrição
“Nome”	String com o nome da barra.
Bsh	Banco de capacitores/reatores shunts da barra em MVAr.
Vmag	Magnitude da tensão na barra em pu.
Vang	Ângulo de fase da tensão na barra em graus.
Pg	Potência ativa gerada em MW.
Qg	Potência reativa gerada em MVAr.
Pl	Potência ativa de carga em MW.
Ql	Potência reativa de carga em MVAr.
Sbase	Potência base em MVA.
Vbase	Tensão base em kV

Tabela 7-2 - Argumentos usados no cadastro de barras

Os outros métodos para cadastro de barras possuem a mesma sintaxe.

No fragmento de código na Figura 7-3 é apresentado o cadastramento das barras do sistema de potência em estudo.

```
rede->AddBus(TBus::CreateRefBus(
            "Barra 1", 0.0, 1.03, 20.2, 0, 0, 0, 0, 100, 230));  //1
rede->AddBus(TBus::CreatePVBus(
            "Barra 2", 0.0, 1.01, 0.0, 700, 0, 0, 0, 100, 230)); //2
rede->AddBus(TBus::CreatePVBus(
            "Barra 3", 0.0, 1.03, 0.0, 719, 0, 0, 0, 100, 230)); //3
rede->AddBus(TBus::CreatePVBus(
```

```
            "Barra 4", 0.0, 1.01, 0.0, 700, 0, 0, 0, 100, 230)); //4
rede->AddBus(TBus::CreatePVBus(
            "Barra 5", 0.0,0.958, 0.0,2.45, 0, 0, 0, 100, 230)); //5
rede->AddBus(TBus::CreatePQBus(
            "Barra 6", 0.0,  1.0, 0.0, 0, 0, 0, 0, 100, 230));   //6
rede->AddBus(TBus::CreatePQBus(
            "Barra 7", 0.0,  1.0, 0.0, 0, 0, 0, 0, 100, 230));   //7
rede->AddBus(TBus::CreatePQBus(
            "Barra 8", 0.0,  1.0, 0.0, 0, 0,967,300,100,230));   //8
rede->AddBus(TBus::CreatePQBus(
            "Barra 9", 0.0,  1.0, 0.0, 0, 0,0,500,100,230));     //9
rede->AddBus(TBus::CreatePQBus(
           "Barra 10", 0.0, 1.0, 0.0, 0, 0,1400,100,100,230));   //10
rede->AddBus(TBus::CreatePQBus(
           "Barra 11", 0.0, 1.0, 0.0, 0, 0, 0, 0, 100, 230));    //11
rede->AddBus(TBus::CreatePQBus(
            "Barra 12", 0.0, 1.0, 0.0, 0, 0, 0, 0, 100, 230));   //12
rede->AddBus(TBus::CreatePQBus(
            "Barra 13", 0.0, 1.0, 0.0, 0, 0, 0, 0, 100  230));   //13
```

Figura 7-3 - Cadastrando barras no sistema

Cadastradas as barras no sistema, deve-se agora cadastrar os ramos de circuito que, neste estudo de caso, podem ser transformadores ou linhas de transmissão. Para cadastrar ramos, utiliza-se o método *AddBranch* da classe *TNetwork*. Neste estudo de caso, é empregada uma sintaxe simplificada para o método *CreateBranch* como ilustrada no fragmento de código:

```
rede->AddBranch(TBranch::CreateBranch(
                "Nome",rede->GetBus(de),rede->GetBus(para),
                akm, phikm, rkm, xkm, bsh, Sbase, Vbase));
```

onde os argumentos são apresentados na Tabela 7-3.

Argumento	Descrição
“Nome”	Nome do ramo.
De	Número da barra de origem.
Para	Número da barra de destino
Akm	Módulo do TAP de transformadores, para linhas de transmissão deve ser 1,0.
Phikm	Fase em graus do TAP de transformadores, para linhas de transmissão deve ser zero.
Rkm	Resistência série do ramo em pu.
Xkm	Reatância série do ramo em pu.
Bsh	Susceptância shunt de linha de transmissão em MVAr, para transformadores deve ser zero.
Sbase	Potência base em MVA.
Vbase	Tensão base em kV.

Tabela 7-3 - Argumentos do método de cadastro de ramos

```
rede->AddBranch(TBranch::CreateBranch(
      "Trafo 1",rede->GetBus(1),rede->GetBus(6),
      1.0, 0, 0, 0.15/9.0, 0, 100, 230));
rede->AddBranch(TBranch::CreateBranch(
      "Trafo 2",rede->GetBus(2),rede->GetBus(7),
      1.0, 0, 0, 0.15/9.0, 0, 100, 230));
rede->AddBranch(TBranch::CreateBranch(
      "Trafo 3",rede->GetBus(3),rede->GetBus(12),
      1.0, 0, 0, 0.15/9.0, 0, 100, 230));
rede->AddBranch(TBranch::CreateBranch(
      "Trafo 4",rede->GetBus(4),rede->GetBus(11),
      1.0, 0, 0, 0.15/9.0, 0, 100, 230));
rede->AddBranch(TBranch::CreateBranch(
      "Linha 1",rede->GetBus(6),rede->GetBus(7),
```

```
      1.0, 0, 1e-4*25, 1e-3*25, 900*0.00175*25, 100, 230));
.....................................................
```

Figura 7-4 - Cadastro de ramos da rede elétrica

Após o cadastro dos ramos, deve-se empregar o método *Arrange* da classe *TNetwork,* para informar à rede elétrica que todos os elementos foram cadastrados e as estruturas de dados para o cálculo do fluxo de carga podem ser inicializadas. Pode-se agora realizar o cálculo do fluxo de carga através do método *LoadFlow* da classe *TNetwork*. O método *LoadFlow* retorna verdadeiro se o fluxo de carga convergir e falso caso não haja a convergência.

Na Figura 7-5 é apresentado um fragmento de código em que o fluxo de carga é calculado e, caso convergir, o resultado é apresenta na tela do computador.

```
1  rede->Arrange();
2  if (rede->LoadFlow())
3  {
4     int i;
5     for (i=1;i<=rede->GetNumBus();i++)
6     {
7           cout<<*rede->GetBus(i)<<endl;
8     }
9     cout<<endl;
10    for (i=1;i<=rede->GetNumBranches();i++)
11    {
12          cout<<*rede->GetBranch(i)<<endl;
13    }
14 }
15 else
16 {
17    cout<<"Não houve convergência\n";
18 }
```

Figura 7-5 - Cálculo do fluxo de carga

Na Figura 7-5 na linha 1 é chamado o método que prepara as estruturas de dados para o fluxo de carga. Na linha 2 é chamado o método que calcula o fluxo de carga e testa se houve convergência. Caso o calculo do fluxo de carga seja bem sucedido, as linhas de 4 a 13 exibem os resultados do fluxo de carga.

7.4 Modelagem das máquinas síncronas e seus controladores

7.4.1 Modelagem da máquina síncrona

O modelo das máquinas síncronas é implementado através da classe *TSyncGen*. O código utilizado para criar as instâncias dos geradores síncronos é apresentado através da Figura 7-6.

```
1  TSyncGen *G1, *G2, *G3, *G4;
2  TVector<double> param(16);
   // Parametrização dos geradores
3  param[XQ] =    1.7;    // Xq
4  param[XD] =    1.8;    // Xd
5  param[XQ_] =   0.55;   // Xq'
6  param[XD_] =   0.3;    // Xd'
7  param[XQ__] =  0.25;   // Xq''
8  param[XD__] =  0.25;   // Xd''
9  param[RA] =    0.0025; // Ra
10 param[TQO_] =  0.4;    // Tq0'
11 param[TDO_] =  8;      // Td0'
12 param[TQO__] = 0.05;   // Tq0''
13 param[TDO__] = 0.03;   // Td0''
14 param[HG] =    8.5;    // Hg
15 param[DA] =    0;      // Da
16 param[FO] =    60.0;   // fo
17 param[SBASE] = 900;    // Sbase
18 param[VBASE] = 230;    // Vbase
19 G1 = new TSyncGen(param, rede->GetBus(1));
20 G2 = new TSyncGen(param, rede->GetBus(2));
21 param[HG] = 8.3;
22 G3 = new TSyncGen(param, rede->GetBus(3));
23 G4 = new TSyncGen(param, rede->GetBus(4));
```

Figura 7-6 - Criação das instâncias dos geradores síncronos

Na linha 1, da Figura 7-6, são declarados os ponteiros usados nas criações das instâncias das quatro máquinas síncronas. Na linha 2 é criado um objeto para um vetor de dezesseis posições que conterá os parâmetros das máquinas. Nas linhas de 3 a 18 são definidos os valores dos parâmetros das máquinas, onde podem ser observados os símbolos que representam as posições no vetor de cada parâmetro. As máquinas *G1* e

G2 possuem o mesmo conjunto de parâmetros bem como as máquinas *G3* e *G4*. Nas linhas 19 e 20 são criadas as instâncias das duas máquinas, onde no construtor o primeiro argumento é o vetor de parâmetros e o segundo é a barra onde está localizada a máquina, no caso das máquinas *G1* e *G2* estão localizadas nas barras 1 e 2 respectivamente. Nas linhas 22 e 23 são criadas as instâncias das máquinas *G3* e *G4* que estão localizadas nas barras 3 e 4 respectivamente.

7.4.2 Modelagem dos reguladores de tensão

Neste estudo de caso foi empregado um modelo de regulador de tensão muito simples que representa apenas a dinâmica do controlador. Através da Figura 7-7 é apresentado o diagrama em blocos deste controlador.

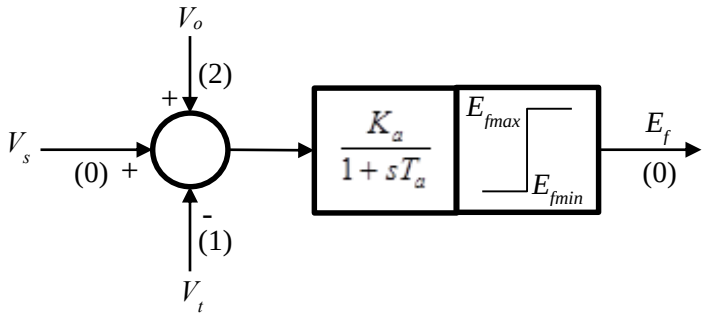

Figura 7-7 - Diagrama em blocos do regulador de tensão

No diagrama de blocos da Figura 7-7, V_s (numerada como entrada zero) é a entrada da tensão de referência (em p.u.). A entrada V_t (numerada como entrada um) é o módulo da tensão medida nos terminais do gerador (em p.u.). A entrada V_o (numerada como entrada dois) é uma entrada auxiliar (em p.u.). A saída do regulador é a tensão terminal E_f (numerada como saída zero). No Framework apresentado nesta tese, não há implementação pronta de controladores, portanto, qualquer controlador específico deve ser implementado pelo usuário.

A declaração da classe que representa um regular de tensão é apresentada através da Figura 7-8.

```
1  class TVoltageReg: public TBlock {
2  protected:
3      double Ka, Ta, Efmax, Efmin;
4  public:
5      TVoltageReg();
6      TVoltageReg(double, double, double, double);
7      void SetParam(double, double, double, double);
8      virtual void
9      Derivatives (TVector<double> & Ysys, TVector<double> & Xstate,
```

```
10                TVector<double> & Fstate, double T);


11     virtual void
12     Initialize (TVector<double> & Ysys, TVector<double> & Xstate);


13     virtual void
14     Outputs (TVector<double> & Ysys, TVector<double> & Xstate,
15              double T);


16 };
```

Figura 7-8 - Declaração da classe que representa o regulador de tensão

Na linha 5 da Figura 7-8 está a declaração do construtor default. Na linha 6 está a declaração do construtor onde é passado os quatro parâmetros do regulador de velocidade: o ganho, a constante de tempo e os limites superior e inferior da tensão de campo. Das linhas 8 até 15 estão declarados os métodos que serão redefinidos nesta classe e que implementam seu comportamento dinâmico

Para implementar o método *Derivatives* é necessário que seja obtida a expressão para a derivada das variáveis de estado do regulador automático de tensão.

Considerando-se que:

$$u = V_s - V_t + V_o \tag{7.1}$$

Então a função de transferência do regulador é:

$$E_f = \frac{K_a}{1 + sT_a} \tag{7.2}$$

o que resulta em uma equação diferencial:

$$\frac{dE_f}{dt} = \frac{1}{T_a}\left(K_a u - E_f\right) \tag{7.3}$$

Pode-se então definir que a variável de estado é:

$$x = E_f \tag{7.4}$$

e

$$\frac{dx}{dt} = \frac{1}{T_a}\left(K_a u - x\right) \tag{7.5}$$

Pode-se então escrever o método que calcula a derivada da variável de estado. O código fonte do método é apresentado através da Figura 7-9.

```
1  void TVoltageReg::Derivatives (TVector<double> & Ysys,
2                                 TVector<double> & Xstate,
3                                 TVector<double> & Fstate,
4                                 double T)
5  {
6     double Vt, Vs, Vo, u, px, x;
7     Vs = Ysys[GetInput(0)];
8     Vt = Ysys[GetInput(1)];
9     Vo = Ysys[GetInput(2)];
10    x = Xstate[GetState(0)];
11    u = Vs-Vt+Vo;
12    px = (1.0/Ta)*(Ka*u-x);
13    Fstate[GetState(0)] = px;
14 }
```

Figura 7-9 - Implementação do método Derivatives para o regulador de tensão

Outro método que deve ser implementado é o responsável por calcular as condições iniciais das variáveis do regulador de tensão. A tensão terminal V_t inicial é conhecida pois ela é um dos resultados do fluxo de carga. O valor inicial da entrada de referência V_s é igual a V_t. O valor inicial da tensão de campo da máquina é conhecido. Portanto, a única variável desconhecida é a tensão da entrada auxiliar V_o, desta forma:

$$\frac{1}{T_a}\left(K_a u - x\right) = 0 \tag{7.6}$$

$$u = \frac{1}{K_a} x \tag{7.7}$$

como

$$u = V_s - V_t + V_o \tag{7.8}$$

então:

$$u = V_o \tag{7.9}$$

portanto:

$$V_o = \frac{1}{K_a} x \tag{7.10}$$

sendo que $x = E_f$

Portanto o código que implementa o método de inicialização das variáveis de estado é apresentado na Figura 7-10.

```
1  void TVoltageReg::Initialize (TVector<double> & Ysys,
2                                TVector<double> & Xstate)
3  {
4     double Vt, Vs, Vo, u, px, x, Ef;
5     Vt = Ysys[GetInput(1)];
6     Ef = Ysys[GetOutput(0)];
7     x = Ef;
8     Vs = Vt;
9     Vo = x/Ka;
10    Ysys[GetInput(0)] = Vs;
11    Ysys[GetInput(2)] = Vo;
12    Xstate[GetState(0)] = x;
13 }
```

Figura 7-10 - Cálculo das condições iniciais do regulador de tensão

A tensão terminal é igual a variável de estado do regulador de tensão, portanto, a implementação do método *Outputs* é dada na Figura 7-11.

```
1  void TVoltageReg::Outputs (TVector<double> & Ysys,
2                             TVector<double> & Xstate, double T)
3  {
4     double x, Ef;
5     x  = Xstate[GetState(0)];
6     Ef = x;
7     if (Ef >= Efmax) Ef = Efmax;
8     if (Ef <= Efmin) Ef = Efmin;
9     Ysys[GetOutput(0)] = Ef;
10 }
```

Figura 7-11 - Cálculo da saída do regulador de tensão

O limitador da tensão de campo que aparece no diagrama em blocos da Figura 7-7 é implementado através das linhas 7 e 8 da Figura 7-11.

O fragmento de código da Figura 7-12, apresenta a criação das instâncias dos reguladores de tensão, onde $K_a = 5$, $T_a = 0{,}2$, $P_{max} = 10$ e $P_{min} = -10$.

```
1  TVoltageReg *rat1 = new TVoltageReg(5,0.2,10,-10),
2              *rat2 = new TVoltageReg(5,0.2,10,-10),
3              *rat3 = new TVoltageReg(5,0.2,10,-10),
4              *rat4 = new TvoltageReg(5,0.2,10,-10);

5  TSource     *Vo1=new TSource(), *Vo2=new TSource(),
6              *Vo3=new TSource(), *Vo4=new Tsource();

7  TSource     *Vs1=new TSource(), *Vs2=new TSource(),
8              *Vs3=new TSource(), *Vs4=new Tsource();
```

Figura 7-12 - Criação das instâncias dos reguladores de tensão

Na Figura 7-12, nas linhas de 1 a 4 são criadas as instâncias dos reguladores de tensão. Além das instâncias dos reguladores de tensão, é necessário criar instâncias de fontes de sinal (classe *TSource*) para as entradas auxiliares dos regulares (entrada V_o) nas linhas 5 e 6. Nas linhas 7 e 8 são criadas as instâncias das fontes de sinais para as entradas de referência do regulador de tensão.

7.4.3 Modelagem dos reguladores de velocidade

O modelo de regulador de velocidade empregado neste estudo de caso é proposto na referência (Arrilaga & Arnold, 1990). O diagrama em blocos deste regulador é apresentado na Figura 7-13.

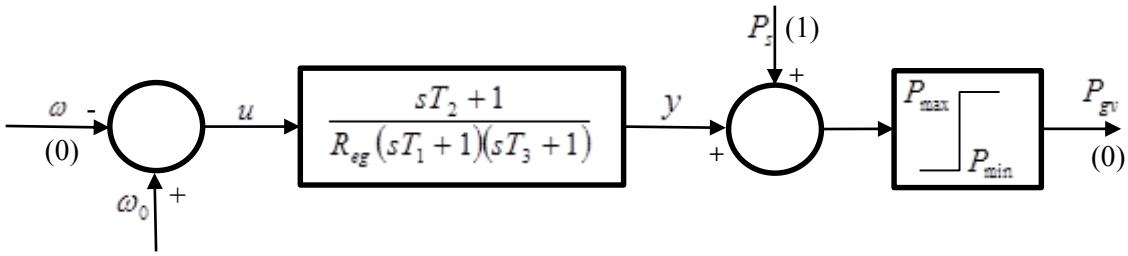

Figura 7-13 - Modelo de regulador de velocidade proposto em (Arrilaga & Arnold, 1990)

Na Figura 7-13, a entrada (0) corresponde a velocidade de rotação da máquina em pu ω. A velocidade de referência ω_0é sempre igual a 1.0pu, neste estudo de caso, então ele é gerada internamente ao bloco. A potência de referência P_s, em pu, é a entrada (1) do bloco. A saída (0) é a potência mecânica de referência para os servo - mecanismos que acionam o distribuidor P_{gv} em pu.

Para implementar este modelo no Framework, a função de transferência do regulador será convertida para o formato de espaço de estados, então é necessário que sejam criados os atributos *A*, *B*, *C*, *D* que representam as matrizes do espaço de estados. Na Figura 7-14 é apresentada a declaração da classe que representa este regulador de velocidade.

```
1  namespace bcssd {

2  typedef enum {GOVT1=0, GOVT2, GOVT3, GOVREG, GOVPMAX, GOVPMIN}
3                TParamGov;

4  class THydroGov: public TBlock {
5  protected:
6     double Reg, T1, T2, T3, Pmax, Pmin;
7     TMatrix<double> A, B, C, D;
8  private:
9     THydroGov();
10 public:

11    THydroGov(TVector<double> &param);
12    void SetParam(double, double, double, double, double, double);
13    void SetParam(TVector<double> &param);

14    virtual void
15    Derivatives (TVector<double> & Ysys, TVector<double> & Xstate,
                   TVector<double> & Fstate, double T);

16    virtual void
17    Initialize (TVector<double> & Ysys, TVector<double> & Xstate);

18    virtual void
19    Outputs (TVector<double> & Ysys, TVector<double> & Xstate,
20             double T);
21 };
```

Figura 7-14 - Declaração da classe que representa o regulador de velocidade

A classe fundamental *TBlock* fornece o método *tf2ss* que converte para o espaço de estados uma função

de transferência expressa por dois vetores, um com os coeficientes do numerado e outro com os coeficientes do denominador. A sintaxe do método é:

```
tf2ss(num, den, A, B, C, D)
```

Onde *num* é um vetor com os coeficientes do numerador da função de transferência em ordem decrescente de potências de *s*, *den* contém os coeficientes do denominador da função de transferência e as matrizes *A*, *B*, *C* e *D* são as matrizes do modelo em espaço de estados do regulador de velocidade.

Pode-se obter da Figura 7-13 a função de transferência do regulador dada pela expressão:

$$\frac{Y(s)}{U(s)} = \frac{sT_2 + 1}{R_{eg}(sT_1 + 1)(sT_3 + 1)} \tag{7.11}$$

Agora expandindo o denominador obtém-se:

$$\frac{Y(s)}{U(s)} = \frac{sT_2 + 1}{(T_1 T_3 R_{eg})s^2 + [(T_1 + T_3)R_{eg}]s + R_{eg}} \tag{7.12}$$

Obtendo-se o valor de *Y(s)* a partir da função de transferência (7.12) pode calcular a saída do regulador de velocidade dada pela equação (7.13).

$$P_{gv}(s) = Y(s) + P_s(s) \tag{7.13}$$

A partir da equação 7.12, pode-se utilizar a função *tf2ss* no construtor da classe, onde as matrizes *A*, *B*, *C* e *D* são calculadas e as estruturas de dados internas são inicializadas. O construtor da classe é apresentado na Figura 7-15.

```
1  THydroGov::THydroGov(TVector<double> &param)
2  {
3     TVector<double> num(2),den(3);
4     SetParam(param);
5     num[0] = T2; num[1] = 1.0;
6     den[0] = T1*T3*Reg; den[1] = (T1+T3)*Reg; den[2] = Reg;
7     tf2ee(num,den,A,B,C,D);
8     MemInit(2,A.getNumRow(),1);
9  }
```

Figura 7-15 - Construtor da classe que representa o regulador de velocidade

Nas linhas 5 e 6 da Figura 7-15, pode-se observar a montagem dos vetores dos coeficientes do numerador e do denominador da função de transferência expressa através da equação (7.12). Na linha 7 as matrizes do espaço de estados são calculadas e na linha 8 as estruturas de dados internas são inicializadas.

A dinâmica do regulador de velocidade agora é expressa pelas equações:

$$\frac{dx}{dt} = Ax + Bu \tag{7.14}$$

$$y = Cx + Du \tag{7.15}$$

A equação (7.14) é usada para calcular as derivadas das variáveis de estado, portanto, o método *Derivatives* é implementado como apresentado na Figura 7-16.

```
1  void THydroGov::Derivatives (TVector<double> & Ysys,
2                               TVector<double> & Xstate,
3                               TVector<double> & Fstate, double T)
4  {
5     TVector<double> x(A.getNumRow()),u(1),px;
6     double W, Ps;
7     int i;
8     W  = Ysys[GetInput(0)];
9     Ps = Ysys[GetInput(1)];
10    u[0] = 1.0-W;
11    for (i=0;i<A.getNumRow();i++) x[i] = Xstate[GetState(i)];
12    px = A*x+B*u;
13    for (i=0;i<A.getNumRow();i++) Fstate[GetState(i)] = px[i];
14 }
```

Figura 7-16 - Método para o cálculo da derivada das variáveis de estado

Na Figura 7-16, as linhas 8 e 9 são usadas para obter o valor das entradas do regulador de velocidade, a velocidade do rotação da máquina e a potência de referência respectivamente, ambos em pu. Na linha 11 são obtidos os valores das variáveis de estado do regulador de velocidade. Na linha 12 a equação (7.14) é avaliada. Na linha 13 as derivadas das variáveis de estado calculadas são atribuídas ao vetor das derivadas de todas as variáveis de estado do sistema em simulação, *Fstate*.

O cálculo das condições iniciais do regulador de velocidade é feito considerando $\frac{dx}{dt} = 0$ (o sistema está em seu regime permanente). No cálculo das condições iniciais das máquinas e das turbinas, a valor inicial da velocidade ω é conhecido, assim como a potência de saída que o regulador deve fornecer P_{gv}, portanto, deve-se determinar o valor inicial da potência de referência P_s. Em regime permanente, a equação (7.14) torna-se:

$$Ax + Bu = 0 \tag{7.16}$$

A equação (7.16) pode ser reescrita como:

$$Ax = -Bu \tag{7.17}$$

Onde a equação (7.17) é um sistema de equações lineares onde a incógnita é *x*. A solução deste sistema de equações é feita através do emprego da função utilitária *solve* que resolve um sistema de equações lineares do tipo *Fx* = *G* e possui a seguinte sintaxe:

```
x = solve (F, G)
```

Conhecido o valor inicial de *x(t)* pode-se empregar a equação (7.15) para calcular o vetor *y(t)* e, utilizando a equação (7.13) no domínio do tempo, pode-se determinar o valor inicial de *Ps(t)*. Na Figura 7-17 é apresentado o método que calcula as condições iniciais do regulador de velocidade seguindo a formulação apresentada.

```
1  void THydroGov::Initialize (TVector<double> & Ysys,
2                              TVector<double> & Xstate)
3  {
4     TVector<double> x,u(1),px,y;
5     double W, Ps, Pgv;
6     int i;
7     W    = Ysys[GetInput(0)];
8     Pgv  = Ysys[GetOutput(0)];
9     u[0] = 1.0-W;
10    x = solve(A,-1.0*B*u);
11    y = C*x+D*u;
12    Ps = Pgv-y[0];
13    Ysys[GetInput(1)] = Ps;
14    for (i=0;i<A.getNumRow();i++) Xstate[GetState(i)] = x[i];
15 }
```

Figura 7-17 - Método de cálculo das condições iniciais do regulador de velocidade

Na Figura 7-17 nas linhas 7 e 8 são obtidos os valores iniciais conhecidos. Na linha 10 são calculados os valores iniciais das variáveis de estado. Na linha 11 é calculado o valor inicial de *y* e na linha 12 o valor inicial da potência de referência. Na linha 13 é atribuído ao vetor das variáveis do sistema o valor inicial da potência de referência e na linha 14 são atribuídos os valores iniciais das variáveis de estado.

O método que calcula a saída do regulador de velocidade *Outputs* é implementado para realizar a avaliação das equações (7.13) e (7.15) e aplicar a não linearidade. O código fonte deste método é apresentado na Figura 7-18.

```
1  void THydroGov::Outputs (TVector<double> & Ysys,
2                           TVector<double> & Xstate, double T)
3  {
4     double Pgv, Ps, W;
5     TVector<double> x(A.getNumRow()), u(1), y;
6     int i;
7     W    = Ysys[GetInput(0)];
8     Ps   = Ysys[GetInput(1)];
9     u[0] = 1.0-W;
10    for (i=0;i<A.getNumRow();i++) x[i] = Xstate[GetState(i)];
11    y = A*x+B*u;
12    Pgv = y[0]+Ps;
13    if (Pgv>=Pmax) Pgv = Pmax;
14    if (Pgv<=Pmin) Pgv = Pmin;
15    Ysys[GetOutput(0)] = Pgv;
16 }
```

Figura 7-18 - Implementação do método Outputs do regulador de velocidade

Na Figura 7-18, as linhas 7 e 8 são usadas para obter as entradas do regulador de velocidade. Na linha 10 são obtidos os valores calculados das variáveis de estado. Na linha 11 é calculado o vetor *y* da equação (7.15) e na linha 12 é calculado a saída do bloco que logo em seguida sofre uma limitação nas linhas 13 e 14. Na linha 15 a saída do regulador de velocidade é disponibilizada ao sistema.

Ao implementar estes três métodos, o regulador de velocidade está pronto para o uso. Deve-se observar na apresentação da implementação do modelo do regulador de velocidade bem como no modelo do regulador de tensão, foi necessário escrever pouco código.

As criações das instâncias dos reguladores de velocidade e das suas fontes de sinal para as potências de referência P_s são apresentadas na Figura 7-19.

```
1  TSource *Ps1=new TSource(), *Ps2=new TSource(),
2          *Ps3=new TSource(), *Ps4=new Tsource();
3  param[GOVREG]   = 0.05; // Reg
4  param[GOVT1]    = 1.0;  // T1
5  param[GOVT2]    = 1.0;  // T2
6  param[GOVT3]    = 2.0;  // T3
7  param[GOVPMAX]  = 100;  // Pmax
```

```
8  param[GOVPMIN]  = -100; // Pmin
9  THydroGov *gov1 = new THydroGov(param),
10           *gov2 = new ThydroGov(param),
11           *gov3 = new THydroGov(param),
12           *gov4 = new THydroGov(param);
```

Figura 7-19 - Criação das instâncias dos reguladores de velocidade

Na Figura 7-19, nas linhas 1 e 2 são criadas as instâncias das fontes de sinal de referência de potência. Nas linhas de 3 a 8 são definidos os parâmetros do regulador de velocidade no vetor *param* que é usado nas linhas 9 a 12 para criar as instâncias dos reguladores de velocidade.

7.4.4 Implementação do modelo das turbinas

O modelo de turbina usado neste estudo de caso é apresentado na referência (Arrilaga & Arnold, 1990). O diagrama em blocos do modelo da turbina empregado é apresentado na Figura 7-20.

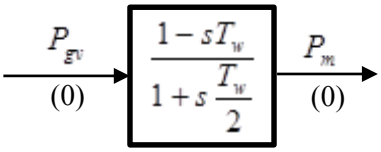

Figura 7-20 - Diagrama em blocos da turbina

Para o caso das turbinas não foi necessário criar uma nova classe, foi utilizada uma classe existente no framework para implementar uma função de transferência – a classe *TTransFunc*.

No construtor da classe deve-se passar os vetores com os coeficientes do numerador e do denominador da função de transferência. O fragmento de código na Figura 7-21 ilustra como é feita a criação das instâncias das turbinas, onde é utilizado $T_w = 1{,}0s$.

```
1  TVector<double> num(2), den(2);
2  num[0] = -1;  num[1] = 1;
3  den[0] = 0.5; den[1] = 1;
4  TTransFunc *tur1 = new TTransFunc(num,den),
5             *tur2 = new TTransFunc(num,den),
6             *tur3 = new TTransFunc(num,den),
7             *tur4 = new TTransFunc(num,den);
```

Figura 7-21 - Criação das instâncias das turbinas

Na Figura 7-21, nas linhas 2 e 3 são montados os vetores dos coeficientes do denominador e do numerador da função de transferência. Estes mesmos vetores são usados para criar as instâncias das quatro turbinas nas linhas 4 até 7.

7.4.5 Conexão das máquinas síncronas à rede elétrica e aos seus controladores

Antes das conexões entre as instâncias dos dispositivos, é necessário que os mesmos sejam registrados em um sistema dinâmico. Um sistema dinâmico é uma instância da classe *TSystem*. Deve-se criar também uma instância do integrador numérico que se deseja utilizar, neste estudo de caso será o *Runge-Kutta* de quarta ordem. O fragmento de código na Figura 7-22 demonstra a criação de um sistema que poderá ter no máximo 50 blocos na linha 1 e do integrador numérico na linha 2.

```
1  TSystem      sistema(50);
2  TRungeKutta4 integrador;
```

Figura 7-22 - Criação das instâncias das classes TSystem e TrungeKutta4

Todas as instâncias criadas para os diversos dispositivos devem ser cadastrados no sistema dinâmico através do método *PutBlock* da classe *TSystem*. No fragmento de código na Figura 7-23, é ilustrado o cadastramento da rede elétrica e de uma máquina síncrona e seus controladores.

```
1  sistema.PutBlock(rede);
2  sistema.PutBlock(G1);
3  sistema.PutBlock(tur1);
4  sistema.PutBlock(gov1);
5  sistema.PutBlock(Ps1);
6  sistema.PutBlock(rat1);
7  sistema.PutBlock(Vs1);
8  sistema.PutBlock(Vo1);
```

Figura 7-23 - Cadastro dos blocos no sistema dinâmico

Após o cadastro dos blocos, a conexão entre os blocos poderá ser efetuada. As conexões são realizadas através do método *InputConnect* da classe *TBlock*. O digrama em blocos que indica as conexões entre os blocos da primeira unidade geradora (barra 1) é apresentado através da Figura 7-24.

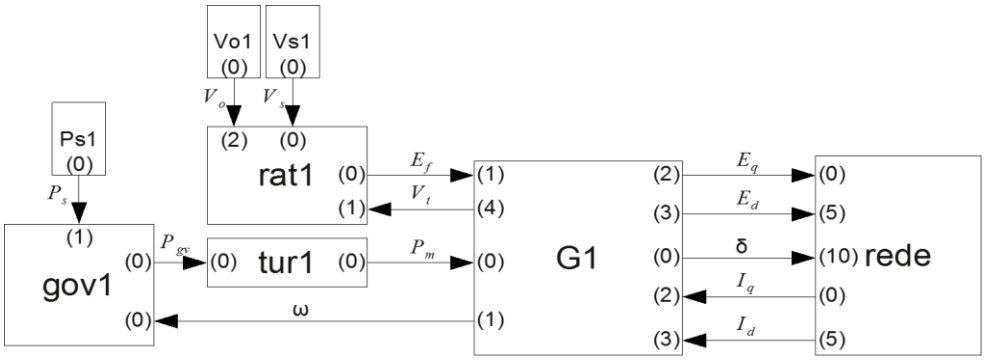

Figura 7-24 - Diagrama em blocos de uma unidade de geração

Na Figura 7-24 são apresentadas as conexões entre os vários blocos instanciados. A conexão entre as entradas da rede elétrica e a máquina *G1* é expressa através do fragmento de código da Figura 7-25.

```
1  rede->InputConnect(0,2,G1);  //Eq''
2  rede->InputConnect(5,3,G1);  //Ed''
3  rede->InputConnect(10,0,G1); //delta
```

Figura 7-25 - Conexão das entradas da rede nas saídas da máquina síncrona

Após a conexão das entradas da rede elétrica na máquina *G1* pode-se conectar as entradas da máquina nas saídas correspondentes da rede elétrica, como mostrado na Figura 7-26.

```
1  G1->InputConnect(2,0,rede); //Iq
2  G1->InputConnect(3,5,rede); //Id
```

Figura 7-26 - Conexão das entradas de G1 nas saídas da rede elétrica

Procedendo de forma análoga, pode-se conectar os outros dispositivos com a máquina *G1* e com as suas referências, como demonstrado no fragmento de código da Figura 7-27.

```
1  G1->InputConnect(0,0,tur1);
2  tur1->InputConnect(0,0,gov1);
3  gov1->InputConnect(0,1,G1);
4  gov1->InputConnect(1,0,Ps1);
5  G1->InputConnect(1,0,rat1);
6  rat1->InputConnect(1,4,G1);
7  rat1->InputConnect(0,0,Vs1);
8  rat1->InputConnect(2,0,Vo1);
```

Figura 7-27 - Fragmento de código que completa as conexões da unidade de geração G1

Para as outras três máquinas síncronas, o procedimento de conexão é o mesmo. A quinta unidade de geração não é uma máquina síncrona é um conversor CC/CA (corrente contínua para corrente alternada) alimentado por uma corrente CC (corrente contínua) fornecida por uma matriz de células fotovoltaicas, além de seus controles associados.

7.5 Modelagem do Sistema Fotovoltaico Conectado a Rede Elétrica

Na representação do sistema Fotovoltaico (PV), realizada neste estudo de caso, são implementados os modelos dos dispositivos físicos do sistema, como: matriz de células PV, conversor CC/CA de interconexão com a rede e o link CC de modo a realizar estudos de estabilidade e controle de sistemas PV interligados a rede de transmissão.

7.5.1 Modelagem dos módulos PV

Através da Figura 7-28 é apresentado um circuito equivalente que modela uma matriz *PV*, muito utilizado na literatura e que representa com bastante precisão as matrizes PV reais. No circuito, as células são modeladas como uma junção *PN* que converte diretamente a energia luminosa em energia elétrica. A fonte de corrente I_{ph} representa a célula PV, o diodo *D* representa as características não lineares da junção *PN*, os resistores R_p e R_s representam as resistências em paralelo e em série, respectivamente, intrínsecas das células (R_s representa a resistência equivalente de todas as resistências estruturais do dispositivo enquanto R_p está relacionado às resistências fuga das junções PN). Normalmente, R_s é muito pequeno e R_p é muito grande, portanto, estes resistores podem ser desconsiderados para simplificar o modelo (Fonseca, Gomes, Barra Jr, & Sena, 2012).

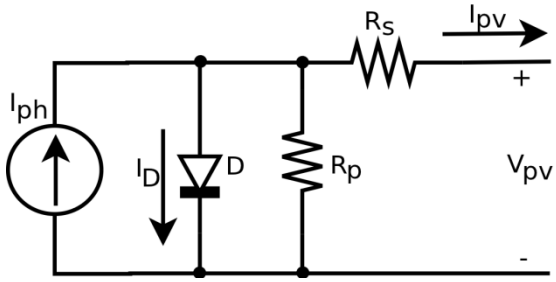

Figura 7-28 - Circuito equivalente das células PV

As células PV são agrupadas em unidades maiores que constituem matrizes de células PV com um número de células em série n_s e um número de células em paralelo n_p. Estas matrizes de células PV são chamadas de *Módulos PV*.

A corrente fornecida pelo módulo PV I_{pv} é dada pela expressão (7.18):

$$I_{pv} = n_p I_{ph} + n_p I_{rs}\left(e^{\frac{qV_{pv}}{kTAn_s}} - 1\right) \tag{7.18}$$

Onde I_{pv} é a corrente de saída do módulo PV em Ampères, V_{pv} é a tensão nos terminais do módulo PV em Volts, *q* é a carga de um elétron, *k* é a constante de Boltzmann, *A* é o fator de idealidade da junção *PN*, *T* é a temperatura da junção em K e I_{rs} é a corrente de saturação reversa das células em Ampères.

A corrente de saturação reversa I_{rs} varia com a temperatura de acordo com a equação (7.19).

$$I_{rs} = I_{rr}\left(\frac{T}{T_r}\right)^3 e^{\frac{qE_g}{kA}\left(\frac{1}{T_t} - \frac{1}{T}\right)} \tag{7.19}$$

Onde T_r é a temperatura de referência, I_{rr} é a corrente de saturação na temperatura de referência e E_g é a energia da banda proibida do semicondutor empregado na construção das células.

A corrente produzida através da exposição das células à luz I_{ph} depende da radiação luminosa *S,* ao qual está exposta, e à temperatura da junção semicondutora *T* das células. Estas dependências são expressas através da equação (7.20).

$$I_{ph} = \left[I_{scr} + k_i\left(T - T_r\right)\right]\frac{S}{100} \tag{7.20}$$

Onde I_{scr} é a corrente de curto-circuito da célula na temperatura de referência, k_i é o coeficiente de temperatura da corrente de curto circuito e *S* é a radiação solar em mW/cm^2.

Através do modelo matemático do módulo PV apresentado, pode-se definir quais os atributos a classe a ser desenvolvida deve possuir, quais as variáveis de entrada, quais as variáveis de saída e quais as funcionalidades deve-se implementar.

Os atributos constituem as características físicas da célula. As constantes físicas *q* e *k* serão declaradas como constantes estáticas da classe a ser criada. Todos os outros parâmetros físicos deverão ser informados quando a criação da instância for realizada. A classe criada para representar a matriz fotovoltaica recebeu o nome de *TPVArray* e a sua declaração é apresentada na Figura 7-29.

```
1  namespace bcssd {
2  typedef enum {NP=0, NS, TR, IRR, EG, KI,
3                ISCR, FA, TC, SI, NI, VPVO} TParamPVArray;

4  class TPVArray: public bcssd::TBlock {
5  protected:
6     static const
7     double q = 1.60217646e-19, //carga do elétron
8            K =  1.3806503e-23; //Constante de Boltzmann

9     double np,   //Número de células conectadas em paralelo
10           ns,   //Número de células conectadas em série
11           Tr,   //Temperatura de referência em K
12           Irr,  //Corrente de saturação reversa em Tr
13           Eg,   //Energia da banda proibida do semicondutor usado
14           ki,   //Coefic. de temp. da corrente de curto circuito
15           Iscr, //Corrente de curto circuito em Tr
16           A,    //Fator de idealidade da junção PN
17           S0,   //Chute inicial do cálc. das cond. iniciais
```

```
18          T0,   //Temperatura inicial da junção
19          NM,   //Número máximo de iterações
20          Vpv0; //Tensão especificada para o painel
21    bool  conv; //Convergiu ?
22 private:
23    TPVArray();
24 public:
25    TPVArray(TVector<double> &param);

26    virtual void
27    Outputs (TVector<double> & Ysys,
28             TVector<double> & Xstate, double T);

29    virtual void
30    Initialize (TVector<double> & Ysys, TVector<double> & Xstate);

31    bool isSuccess(){return conv;};
32 };

33 }
```

Figura 7-29 - Declaração da classe que representa um módulo PV

Na Figura 7-29 pode-se verificar que nas linhas de 6 a 8 as constantes físicas foram declaradas como constantes estáticas. Nas linhas de 9 a 16 são declarados os parâmetros do modelo do módulo PV.

A implementação do modelo possui como entradas a tensão nos terminais do módulo V_{pv}, a temperatura da junção T e a radiação solar S. A saída da implementação do modelo é a corrente injetada pelo módulo I_{pv}.

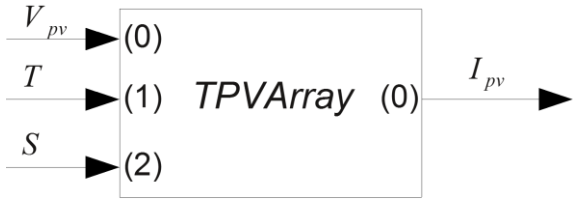

Figura 7-30 - Representação computacional do módulo PV

Como não existe equações diferenciais no modelo utilizado, não há variáveis de estado, então somente os métodos *Outputs* e *Initialize* devem ser redefinidos como apresentado nas linhas 26 a 30 da Figura 7-29.

Empregando as equações (7.18) a (7.20) pode-se escrever a implementação do método *Outputs* apresentado na Figura 7-31.

```
1  void TPVArray::Outputs (TVector<double> & Ysys,
2                          TVector<double> & Xstate, double t)
3  {
4     double Vpv, T, S, Ipv, Iph, Irs;
5     Vpv = Ysys[GetInput(0)];
6     T   = Ysys[GetInput(1)];
7     S   = Ysys[GetInput(2)];
8     Iph =(Iscr+ki*(T-Tr))*(S/100.0);
9     Irs = Irr*pow(T/Tr,3.0)*exp(((q*Eg)/(K*A))*(1.0/Tr-1.0/T));
10    Ipv = np*Iph-np*Irs*(exp((q*Vpv)/(K*T*A*ns))-1);
11    Ysys[GetOutput(0)] = Ipv;
12 }
```

Figura 7-31 - Implementação do método Outputs da classe que representa os módulos PV

Nas linhas de 5 a 7 na Figura 7-31, são obtidas as entradas para modelo V_{pv}, T e S. A equação (7.20) é avaliada na linha 8. A equação (7.19) é avaliada na linha 9. A saída do modelo, I_{pv}, é obtida através da equação (7.18) avaliada na linha 10. Na linha 11 a saída do modelo é disponibilizada ao sistema sendo simulado.

No cálculo das condições iniciais, é necessário encontrar a raiz de uma equação transcendental, portanto, o método de Newton deverá ser empregado. Sendo *f(x)* uma função transcendental, então para encontrar as raízes da equação:

$$f(x) = 0 \tag{7.21}$$

Utiliza-se a equação recursiva:

$$x^{k+1} = x^k - \frac{f\left(x^k\right)}{\frac{\partial f\left(x^k\right)}{\partial x}} \tag{7.22}$$

Em regime permanente, a corrente que o módulo deve injetar I_{pv} é conhecida. a tensão nos terminais do módulo é um dado de projeto, V_{pv0} e a temperatura na junção também é um dado conhecido, então a incógnita é a radiação solar S.

Para aplicar a fórmula de Newton na equação (7.22) deve-se reescrever a equação (7.18) como:

$$f\left(I_{pv},V_{pv},T,S\right)=I_{pv}-\left[n_pI_{ph}+n_pI_{rs}\left(e^{\frac{qV_{pv}}{kTAn_s}}-1\right)\right] \tag{7.23}$$

Uma vez que se deseja determinar *S* então:

$$\frac{\partial f}{\partial S}=-n_p\frac{\partial I_{ph}}{\partial S} \tag{7.24}$$

Substituindo a equação (7.20) na equação (7.24) obtém-se:

$$\frac{\partial f}{\partial S}=-n_p\left[I_{scr}+k_i\left(T-T_r\right)\right]\frac{1}{100} \tag{7.25}$$

Desta forma, pode-se aplicar as equações (7.23) e (7.25) na equação (7.22), para determinar o valor inicial da radiação solar *S*. Por se tratar de um processo iterativo, pode haver ou não a convergência, portanto, para evitar laço infinito deve-se limitar o número máxima de iterações, *NM*, que está definido na linha 19 da Figura 7-29 e de uma estimativa inicial para a radiação solar representado pelo atributo *S0* declarado na linha 17 são informados no construtor da classe. O valor inicial da temperatura da junção é representado pelo atributo *T0*, declarado na linha 18 da Figura 7-29, e é informado como parâmetro do modelo no construtor da classe. Através da Figura 7-32 é apresentado a implementação do método *Initialize* para a classe *TPVArray*.

```
1  void TPVArray::Initialize (TVector<double> & Ysys,
2                             TVector<double> & Xstate)
3  {
4     double Ipv, Vpv, T, S, Iph, Irs, Df, f, df, Sold, err;
5     int i;
6     conv = false;
7     Ipv = Ysys[GetOutput(0)];
8     Vpv = Vpv0;
9     Ysys[GetInput(0)] = Vpv;
10    T   = T0;
11    S = S0;
12    for (i=0;i<NM;i++)
13    {
14     Iph = (Iscr+ki*(T-Tr))*(S/100.0);
15     Irs = Irr*pow(T/Tr,3.0)*exp(((q*Eg)/(K*A))*(1.0/Tr-1.0/T));
16     f = Ipv-(np*Iph-np*Irs*(exp((q*Vpv)/(K*T*A*ns))-1));
17     df = -np*((Iscr+ki*(T-Tr))*(1.0/100.0));
```

```
18      Sold = S;
20      S = S - f/df;
21      err = S-Sold;
22      if (fabs(err)<=1e-13)
23      {
24            conv = true;
25            break;
26      }
27     }
28     if (i>=NM)
29     {
30            conv = false;
31            return;
32     }
33     Ysys[GetInput(1)] = T;
34     Ysys[GetInput(2)] = S;
35 }
```

Figura 7-32 - Método Initialize da classe TPVArray

Nas linhas de 7 a 11 da Figura 7-32 os valores conhecidos a serem empregados no cálculo de *S* são obtidos. O processo iterativo para obtenção da solução é composto pelas linhas 12 a 27. Caso o processo iterativo convirja, o atributo *conv* torna-se verdadeiro indicando que a solução foi encontrada. Caso o processo não convirja, o atributo *conv* torna-se falso. Para verificar se o processo iterativo foi bem sucedido ou não, utiliza-se o método *TPVArray::isSuccess()* que retorna o conteúdo de *conv*. Nas linhas 33 e 34 são disponibilizados ao sistema os valores iniciais da temperatura *T* e da radiação solar *S*.

7.5.2 Inversor CC/CA

O inversor CC/CA é o dispositivo responsável pela conexão do módulo PV a rede elétrica. Este dispositivo converte a corrente contínua gerada pelo módulo PV em uma corrente alternada para a injeção na rede elétrica. Basicamente, o inversor consiste em um modulador por largura de pulso polifásico conectado a um filtro, que neste estudo de caso foi empregado um *LC*, para a eliminação das componentes de alta frequência. Para cada fase existe um modulador com ciclos ativos m_a, m_b e m_c. O modelo empregado neste estudo de caso foi o modelo da média convertido para a referência *dq0* onde o ângulo desta referência é fixo, pois a dinâmica do dispositivo responsável pela sincronização do inversor com o sistema elétrico é considerado muito rápido e, portanto, seu comportamento dinâmico é desprezado.

Através da Figura 7-33, é apresentado um circuito equivalente do modelo da média do inversor. O

modelo é constituído de um link CC que é modelado através de um capacitor com um resistor em série. O resistor em série modela as perdas ôhmicas, em contatos e/ou emendas diversas na fiação, caso existam. O bloco com um símbolo de um transistor representa os circuitos de chaveamento presentes no inversor. O filtro LC de saída é representado em regime permanente senoidal por uma impedância $R_t + jX_t$.

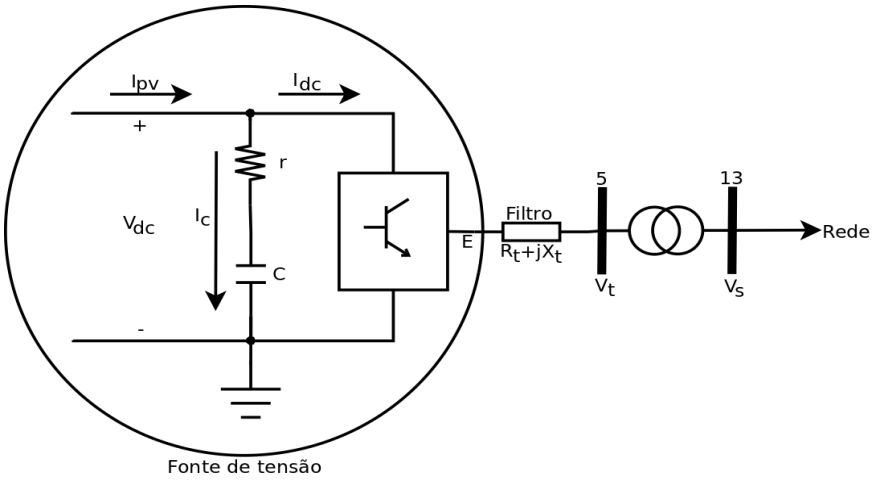

Figura 7-33 - Circuito equivalente do modelo da média para o inversor CC/CA

Ao analisar a Figura 7-33, pode-se verificar que o inversor pode ser representado através do modelo clássico de uma fonte de tensão controlada atrás de uma impedância série, onde *E* é a tensão do nó interno e $R_t + jX_t$ a impedância série, portanto, pode-se empregar nos inversores CC/CA a mesma formulação, em regime permanente, feita em sistemas multimáquinas com máquinas síncronas. São empregados valores em p.u., sendo que as tensões serão valores de linha RMS.

Sendo m_q e m_d as componentes na referência *dq0* dos índices de modulação dos circuitos de chaveamento do inversor. Então o valor médio da tensão interna do inversor é dado pelas equações (7.25) e (7.26):

$$E_q = \left(\frac{\sqrt{6}}{4}\right) m_q V_{dc} \tag{7.25}$$

$$E_d = \left(\frac{\sqrt{6}}{4}\right) m_d V_{dc} \tag{7.26}$$

A tensão do link CC V_{dc}é dado por:

$$V_{dc} = I_C r + V_C \tag{7.27}$$

A corrente do capacitor é dada pela equação:

$$I_C = C\frac{dV_C}{dt} \tag{7.28}$$

Substituindo (7.28) em (7.27):

$$V_{dc} = rC\frac{dV_C}{dt} + V_C \tag{7.29}$$

Definindo a constante de tempo $T = rC$ a equação (7.29) torna-se:

$$V_{dc} = T\frac{dV_C}{dt} + V_C \tag{7.30}$$

A relação entre as correntes no link CC é dada pela equação:

$$I_{pv} = I_C + I_{dc} \tag{7.31}$$

A corrente I_{dc} é dada pela equação:

$$I_{dc} = \left(\frac{\sqrt{6}}{4}\right)\left(m_q I_q + m_d I_d\right) \tag{7.32}$$

A corrente I_{pv} é conhecida, então a tensão V_{dc} pode ser dada por:

$$V_{dc} = r\left(I_{pv} - I_{dc}\right) + V_C \tag{7.33}$$

Substituindo a (7.33) na equação (7.30) obtém-se:

$$\frac{dV_C}{dt} = \frac{1}{T}\left(I_{pv} - I_{dc}\right)r \tag{7.34}$$

Portanto, a única variável de estado do modelo é a tensão no capacitor V_C e as equações empregadas na implementação do modelo são: (7.32), (7.34) e (7.33) empregadas nesta ordem. O resistor *r* deve ser utilizado com seu valor convertido para p.u. .

Uma variável importante para os controladores empregados em conjunto com o inversor é a tensão no secundário do transformador do modelo V_s. Ela é calculada em função da tensão interna *E* e da corrente injetada na rede. Para o cálculo de V_s, primeiro é necessário converter as grandezas utilizadas de *dq0* para a referência da rede e então obter a tensão V_s na referência da rede elétrica e, finalmente, converter V_s para a referência *dq0*. A conversão das grandezas é apresentada nas equações

$$E_r = \cos(\delta)E_q - \text{sen}(\delta)E_d \tag{7.35}$$

$$E_m = \text{sen}(\delta)E_q + \cos(\delta)E_d \tag{7.36}$$

$$I_r = \cos(\delta)I_q - \text{sen}(\delta)I_d \tag{7.37}$$

$$I_m = \text{sen}(\delta)I_q + \cos(\delta)I_d \tag{7.38}$$

Pode-se agora calcular a tensão no secundário do transformador:

$$E = E_r + jE_m \tag{7.39}$$

$$I = I_r + jI_m \tag{7.40}$$

$$V_t = E - \left(R_t + jX_t\right) \tag{7.41}$$

$$V_s = V_{sr} + jV_{sm} = V_t - \left(jX_{trafo}\right)I \tag{7.41}$$

Onde X_{trafo} é a reatância série do transformador em pu. Convertendo para a referência *dq0*:

$$V_{sq} = V_{sr}\cos(\delta) + V_{sm}\,\mathrm{sen}(\delta) \tag{7.42}$$

$$V_{sd} = -V_{sr}\mathrm{sen}(\delta) + V_{sm}\cos(\delta) \tag{7.43}$$

Para criar a classe que representa o inversor, *TPVInverter*, são necessários os seguintes atributos:

Atributo	Descrição
r	Resistência série do capacitor em pu.
T	Constante de tempo *rC*.
Rt	Resistência série do filtro *LC* em pu.
Xt	Reatância série do filtro em pu.
Xtrafo	Reatância do transformador.

Tabela 7-4 - Atributos da classe criada

A declaração da classe *TPVInverter* é apresentada através da Figura 7-34.

```
1  namespace bcssd {

2  typedef enum {IRT=0, IXT, IC, IR,
3                IVDC, IVBASE, IXTRAFO} TParamPVInverter;

4  class TPVInverter: public TMachine {
5  protected:
6     double Rt, Xt, r, T, Vdc0, Xtrafo;
7  private:
```

```
8     TPVInverter();
9   public:

10    TPVInverter(TVector<double> &param, TBus *b);

11    virtual void
12    Derivatives (TVector<double> & Ysys,
                   TVector<double> & Xstate,
                   TVector<double> & Fstate, double T);

13    virtual void
14    Outputs (TVector<double> & Ysys,
              TVector<double> & Xstate, double T);

15    virtual void
16    Initialize (TVector<double> & Ysys,
                  TVector<double> & Xstate);
17  };

18  }
```

Figura 7-34 - Declaração da classe TPVInverter

Na perspectiva da rede elétrica, o inversor é o dispositivo que está injetando corrente na rede, ou seja, está funcionando como um gerador de fato, portanto, a classe do qual a classe do inversor *TPVInverter* é derivada é a classe *TMachine*.

As informações trocadas entre a instância da rede elétrica e a instância do inversor são as mesmas quando a rede elétrica é ligada a uma máquina síncrona, portanto, a classe *TPVInverter* (inversor) deverá fornecer para a classe *TNetwork* (rede elétrica) a tensão interna e o seu ângulo de referência e a classe *TNetwork* fornecerá as correntes injetadas em pu. Através da Figura 7-35 é apresentado o bloco que representa o inversor com suas entradas e saídas.

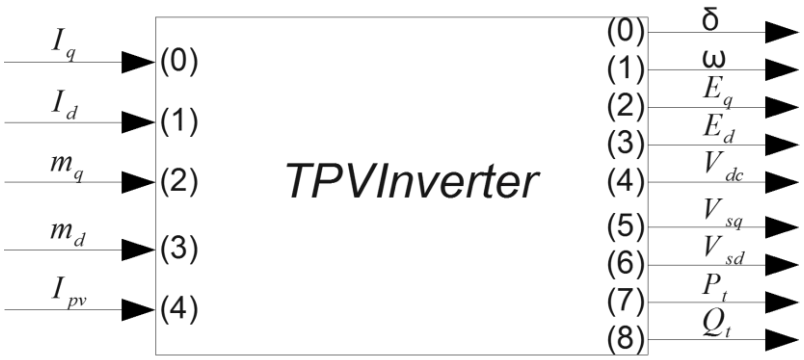

Figura 7-35 - Representação para diagrama de blocos da classe TPVInverter

Na Figura 7-35, as saídas (0) e (1) são sempre constantes, até que em uma futura versão desta classe, eles passem a ser calculados pelo modelo do PLL.

O método que calcula as saídas do modelo, *Outputs*, é apresentado na Figura 7-36.

```
1  void
2  TPVInverter::Outputs (TVector<double> & Ysys,
3                        TVector<double> & Xstate, double T)
4  {
5     double Vdc, Idc, Id, Iq, Ed, Eq, Vd, Vq, md, mq,
6            delta, W, Pt, Qt, Er, Em, Ir, Im,Ipv,Vc,Vsq,Vsd;
7     complex<double> j(0.0,1.0),Vt,St,E,I,Vs;
8
9     Iq =  Ysys[GetInput(0)]; //pu
10    Id =  Ysys[GetInput(1)]; //pu
11    mq =  Ysys[GetInput(2)]; //pu
12    md =  Ysys[GetInput(3)]; //pu
13    Ipv = Ysys[GetInput(4)]; //pu

14    Vc = Xstate[GetState(0)];

15    Idc = (sqrt(6.0)/4.0)*(md*Id+mq*Iq);
16    Vdc = r*(Ipv-Idc)+Vc;

17    Ed = (sqrt(6.0)/4.0)*(md*Vdc);
18    Eq = (sqrt(6.0)/4.0)*(mq*Vdc);
```

```
19    delta = Ysys[GetOutput(0)];

20    Er = cos(delta)*Eq-sin(delta)*Ed;
21    Em = sin(delta)*Eq+cos(delta)*Ed;

22    Ir = cos(delta)*Iq-sin(delta)*Id;
23    Im = sin(delta)*Iq+cos(delta)*Id;

24    E = Er+j*Em;
25    I = Ir+j*Im;
26    W = 1.0;
27    Vt = E-(Rt+j*Xt)*I;
28    Vs = Vt-(j*Xtrafo)*I;
29    Vsq=(real(Vs)*cos(delta)+imag(Vs)*sin(delta));
30    Vsd=(-real(Vs)*sin(delta)+imag(Vs)*cos(delta));
31    St = Vt*conj(I);
32    Pt = real(St);
33    Qt = imag(St);

34    Ysys[GetOutput(0)] = delta; //rad
35    Ysys[GetOutput(1)] = W;     //pu
36    Ysys[GetOutput(2)] = Eq;    //pu
37    Ysys[GetOutput(3)] = Ed;    //pu
38    Ysys[GetOutput(4)] = Vdc;   //pu
39    Ysys[GetOutput(5)] = Vsq;   //pu
40    Ysys[GetOutput(6)] = Vsd;   //pu
41    Ysys[GetOutput(7)] = Pt;    //pu
42    Ysys[GetOutput(8)] = Qt;    //pu
43 }
```

Figura 7-36 - Método Outputs da classe TPVInverter

Nas linhas de 9 a 13 da Figura 7-36 as entradas do modelo do inversor são obtidas. Na linha 15 é avaliada a expressão (7.32), na linha 16 a expressão (7.33) é avaliada, na linha 17 a equação (7.26) e na linha 18 a equação (7.25). Nas linhas de 20 a 33 são calculadas as potências e a tensão no secundário do transformador. Nas linhas de 34 a 42 as saídas são disponibilizadas ao sistema.

O cálculo da derivada da variável de estado é feito no método *Derivatives* apresentado através da Figura 7-37.

```
1  void
2  TPVInverter::Derivatives (TVector<double> & Ysys,
3                            TVector<double> & Xstate,
4                            TVector<double> & Fstate, double t)
5  {
6     double Vdc, Idc, Id, Iq, md, mq, Ipv, pVc, Vc;

8     Iq =  Ysys[GetInput(0)];
9     Id =  Ysys[GetInput(1)];
10    mq =  Ysys[GetInput(2)];
11    md =  Ysys[GetInput(3)];
12    Ipv = Ysys[GetInput(4)];

13    Vc = Xstate[GetState(0)];

14    Idc = (sqrt(6.0)/4.0)*(md*Id+mq*Iq);
15    Vdc = r*(Ipv-Idc)+Vc;
16    pVc = (1.0/T)*(Ipv-Idc)*r;

17    Fstate[GetState(0)] = pVc;
18 }
```

Figura 7-37 - Cálculo da derivada da variável de estado

Na linha 16 da Figura 7-37 é calculado a derivada da variável de estado.

O cálculo das condições iniciais do inversor é obtido a partir do resultado do fluxo de carga. No resultado do fluxo de carga está disponibilizado o módulo da tensão terminal $|V_t|$em pu e sua fase αem graus e a potência ativa P_te reativa $Q_t$em pu, então:

$$V_t = |V_t| e^{j\alpha\left(\frac{\pi}{180}\right)} \tag{7.44}$$

$$S_t = P_t + jQ_t \tag{7.45}$$

A corrente injetada pelo inversor na referência da rede elétrica é dada por:

$$I = \left(\frac{S_t}{V_t} \right)^* \tag{7.46}$$

Dispondo da corrente injetada calculada pela expressão (7.46) pode-se calcular a tensão interna *E* e a tensão no secundário do transformador V_s, através das equações (7.47) e (7.48):

$$E = V_t + (R_t + jX_t)I \tag{7.47}$$

$$V_s = V_t - jX_{trafo}I \tag{7.48}$$

Onde X_{trafo} é a reatância série do transformador. Assim como nas máquinas síncronas, o ângulo de fase de *E* é igual ao ângulo da referência *dq0* do inversor δ.

$$\delta = \angle E \tag{7.49}$$

Convertendo para a referência *dq0* as grandezas E, V_se Ientão:

$$I_q = \text{real}(I)\cos(\delta) + \text{imag}(I)\text{sen}(\delta) \tag{7.50}$$

$$I_d = -\text{real}(I)\text{sen}(\delta) + \text{imag}(I)\cos(\delta) \tag{7.51}$$

$$E_q = \text{real}(E)\cos(\delta) + \text{imag}(E)\text{sen}(\delta) \tag{7.52}$$

$$E_d = -\text{real}(E)\text{sen}(\delta) + \text{imag}(E)\cos(\delta) \tag{7.53}$$

$$V_{sq} = \text{real}(V_s)\cos(\delta) + \text{imag}(V_s)\text{sen}(\delta) \tag{7.54}$$

$$V_{sd} = -\text{real}(V_s)\text{sen}(\delta) + \text{imag}(V_s)\cos(\delta) \tag{7.55}$$

O valor inicial da tensão V_{dc} é especificado como um dado de projeto, então os valores iniciais dos índices de modulação são obtidos a partir das equações (7.25) e (7.26) explicitando m_q e m_d:

$$m_q = \frac{E_q}{\frac{\sqrt{6}}{4}V_{dc}} \tag{7.56}$$

$$m_d = \frac{E_d}{\frac{\sqrt{6}}{4}V_{dc}} \tag{7.57}$$

O valor inicial da corrente do link CC é calculada através da equação (7.32).

Em regime permanente a derivada na (7.34) é igual a zero, isto resulta em:

$$I_{pv} = I_{dc} \tag{7.58}$$

A partir da equação (7.30), fazendo a derivada igual a zero (regime permanente), conclui-se que:

$$V_C = V_{dc} \tag{7.59}$$

Na Figura 7-38 está o código fonte do método que calcula os valores iniciais do modelo do inversor. Como comentário está a indicação de cada uma das equações utilizadas. Nas linhas de 5 a 8 são obtidos os resultados do fluxo de carga, onde *bus* é o atributo cujo valor é passado no construtor da classe e corresponde a barra terminal do inversor e que está ligado ao primário do transformador. Nas linhas de 29 a 43 os valores iniciais são disponibilizados ao sistema.

```
1  void
2  TPVInverter::Initialize (TVector<double> & Ysys,
3                           TVector<double> & Xstate)
4  {
5   double Pt = (bus->getPg()/bus->getNetwork()->getSbase()),
6          Qt = (bus->getQg()/bus->getNetwork()->getSbase()),
7          alfa = bus->getVang(),
8          modVt= bus->getVmod();

9   complex<double> j(0.0,1.0),
10                  Vt = modVt*exp(j*alfa*M_PI/180.0),//eq. 7.44
11                  St = Pt+j*Qt,                     //eq. 7.45
12                  I = conj(St/Vt),                  //eq. 7.46
13                  E = Vt+(Rt+j*Xt)*I,               //eq. 7.48
14                  Vs = Vt-(j*Xtrafo)*I;             //eq. 7.49

15  double delta = arg(E), Iq, Id, Eq, Ed, Ipv,
16         Vdc, Vc, mq, md, Idc, Vsq, Vsd;

17  Iq= real(I)*cos(delta)+imag(I)*sin(delta);        //eq. 7.50
18  Id= -real(I)*sin(delta)+imag(I)*cos(delta);       //eq. 7.51

19  Eq= real(E)*cos(delta)+imag(E)*sin(delta);        //eq. 7.52
20  Ed= -real(E)*sin(delta)+imag(E)*cos(delta);       //eq. 7.53

21  Vsq= real(Vs)*cos(delta)+imag(Vs)*sin(delta);     //eq. 7.54
```

```
22  Vsd= -real(Vs)*sin(delta)+imag(Vs)*cos(delta);    //eq. 7.55

23  Vdc = Vdc0;

24  mq = Eq/((sqrt(6.0)/4.0)*Vdc);                    //eq. 7.56
25  md = Ed/((sqrt(6.0)/4.0)*Vdc);                    //eq. 7.57

26  Idc = (sqrt(6.0)/4.0)*(md*Id+mq*Iq);              //eq. 7.32
27  Ipv = Idc;                                        //eq. 7.58
28  Vc = Vdc;                                         //eq. 7.59

29  Xstate[GetState(0)] = Vc;

30  Ysys[GetInput(0)] = Iq;     //pu
31  Ysys[GetInput(1)] = Id;     //pu
32  Ysys[GetInput(2)] = mq;     //pu
33  Ysys[GetInput(3)] = md;     //pu
34  Ysys[GetInput(4)] = Ipv;    //pu

35  Ysys[GetOutput(0)] = delta; //rad
36  Ysys[GetOutput(1)] = 1.0;   //pu
37  Ysys[GetOutput(2)] = Eq;    //pu
38  Ysys[GetOutput(3)] = Ed;    //pu
39  Ysys[GetOutput(4)] = Vdc;   //pu
40  Ysys[GetOutput(5)] = Vsq;   //pu
41  Ysys[GetOutput(6)] = Vsd;   //pu
42  Ysys[GetOutput(7)] = Pt;    //pu
43  Ysys[GetOutput(8)] = Qt;    //pu
44 }
```

Figura 7-38 - Calculo das condições iniciais do inversor

7.5.3 Controle de corrente

O controle da potência ativa e reativa no inversor no modo corrente é mais adequado para o sistema de geração distribuída em comparação com o controle em modo tensão. Sua sistemática de regulação de corrente

protege o inversor contra condições de sobrecorrente, além de possuir robustez contra variações nos parâmetros do sistema PV e do sistema CA.

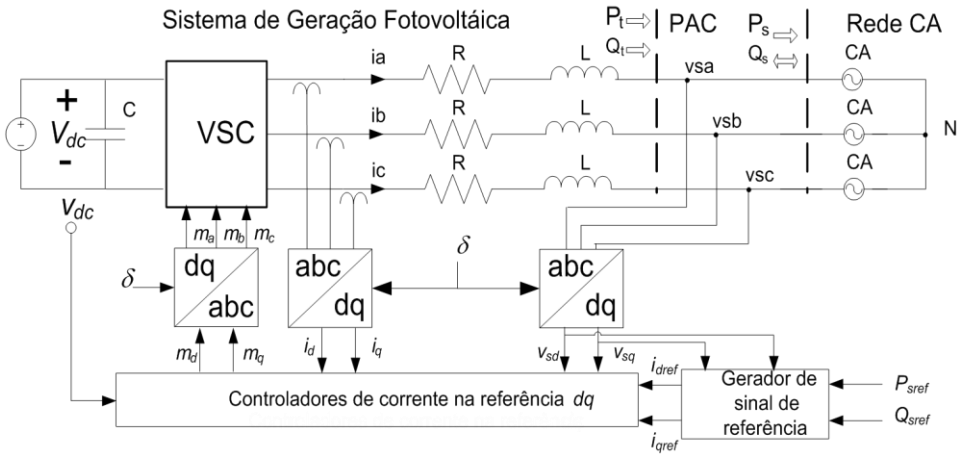

Figura 7-39 - Diagrama em blocos do sistema de controle de corrente

Através da Figura 7-39 é apresentado um diagrama esquemático do sistema de geração fotovoltaica e inversor com a agregação da estrutura de controle dos fluxos de potência ativa e reativa em modo corrente, expressos nas componentes *dq*.

Onde as variáveis m_a, m_b, m_c representam os índices de modulação trifásicas e m_q e m_d, são as componentes *dq*. O bloco denominado "gerador de sinal de referência" realiza um desacoplamento entre as componentes *d* e *q* das correntes de saída do inversor e as potências ativa e reativa despachada para o sistema interligado, explicitando uma dependência única da corrente I_d com a potência ativa (P_s) e I_q com a potência reativa (Q_s), e determinando os sinais de referência para cada corrente, como mostrado nas equações (7.60) e (7.61). Para o caso deste trabalho, o "gerador de sinal de referência" não é empregado.

$$I_{qref} = \frac{2}{3V_{sq}} Q_{sref} \tag{7.60}$$

$$I_{dref} = \frac{2}{3V_{sd}} P_{sref} \tag{7.61}$$

Para a simulação dinâmica é necessária a informação do ângulo δ da referência do barramento da unidade de geração PV com o sistema CA. Tal função normalmente é realizada pelo PLL (do inglês, phase locked loop). Entretanto, para a modelagem abordada este trabalho foi necessário somente uma transformação estática de eixos, com base na nova referência δ, de acordo com as equações (7.50) a (7.55).

O bloco dos controladores de corrente representados nas componentes *dq* da Figura 7-40 implementam as técnicas de controle PI, os ramos de desacoplamento para cada componente de corrente (I_q e I_d) e a pré-alimentação das componentes da tensão da barra de acoplamento comum (V_{sq} e V_{sd}). O diagrama de bloco para

o sistema de controle da unidade de geração distribuída que realiza tais funções é apresentado na Figura 7-40. Neste estudo de caso a realimentação de V_{sq} e V_{sd} não foi utilizada, pois ela estava introduzindo instabilidade ao sistema.

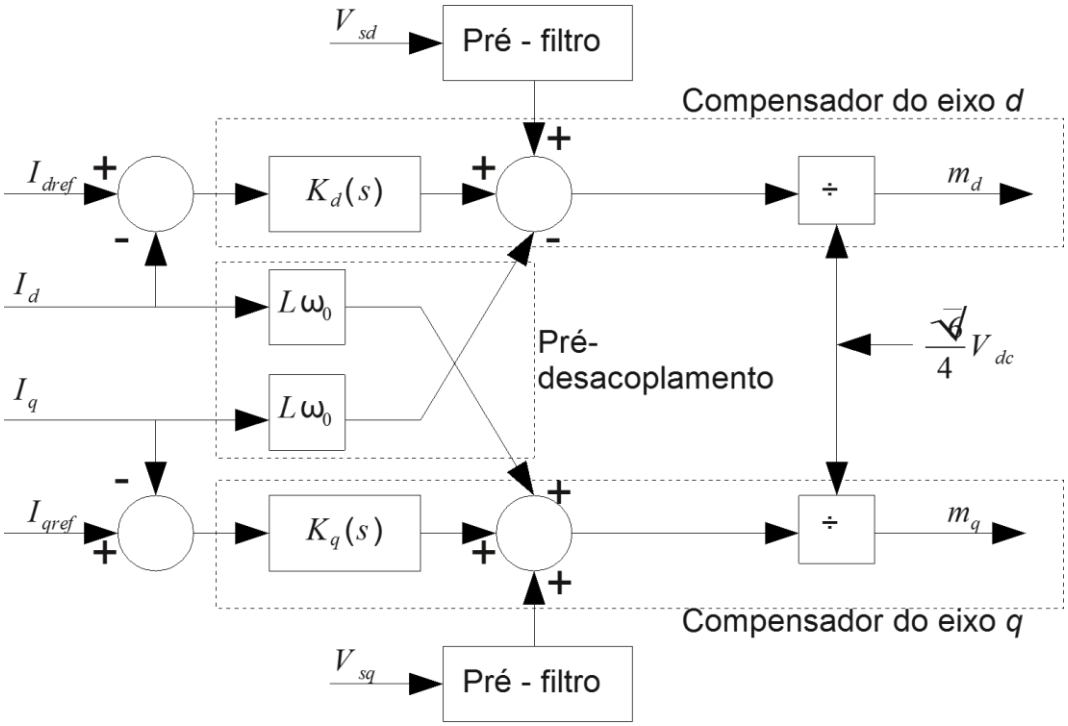

Figura 7-40 - Compensadores de corrente do eixo *d* e *q*

Na Figura 7-40, $K_d(s)$ e $K_q(s)$ são os controladores PI (proporcional mais integral) clássicos para cada componente de corrente e são projetados com objetivos de aumentar a velocidade da planta e eliminar o erro de regime permanente. O pré-desacoplador apresentado através da Figura 7-40, permite que o projeto de cada controlador seja realizado independentemente, de forma desacoplada como dois sistemas SISO (com uma entrada e uma saída). O diagrama em blocos equivalente SISO do compensador do eixo *q* é apresentado através da Figura 7-41 e do eixo *d* na Figura 7-42.

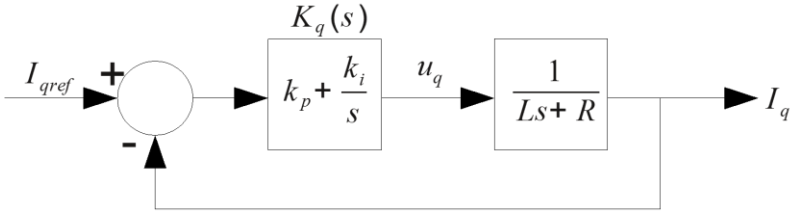

Figura 7-41 - Equivalente SISO do compensador do eixo *q*

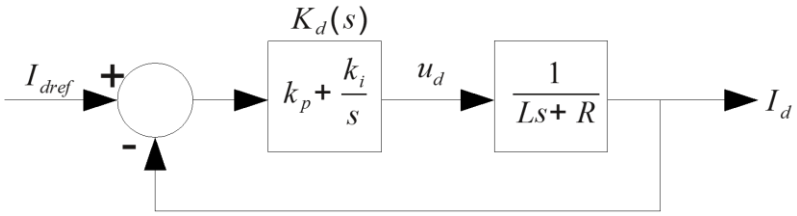

Figura 7-42 - Equivalente SISO do compensador do eixo *d*

Onde *L* e *R* são os parâmetros do filtro de saída do inversor CC/CA (indutância e capacitância). Para o procedimento de obtenção de $K_q(s)$ e $K_d(s)$é empregada a metodologia de cancelamento de polos e zeros. Tal procedimento é aplicável se a planta possui polos e zeros situados no semiplano esquerdo do plano complexo, como neste trabalho, pois a dinâmica predominante do inversor CC/CA é determinada pelo seu filtro de saída que possui uma função de transferência equivalente de primeira ordem $(1/(Ls+R))$.

Para facilitar a implementação, o compensador de corrente da Figura 7-40 foi divido em dois dispositivos: um denominado de compensador do eixo *q* e o outro de compensador do eixo *d*. O compensador do eixo *q* é apresentado no diagrama em blocos da Figura 7-43 e o compensador do eixo *d* no diagrama da Figura 7-44.

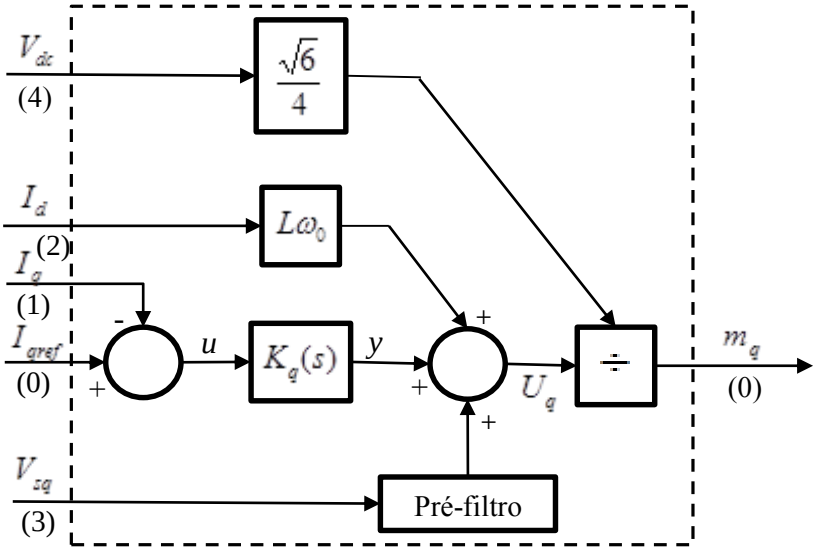

Figura 7-43 - Diagrama em blocos do compensador do eixo *q*

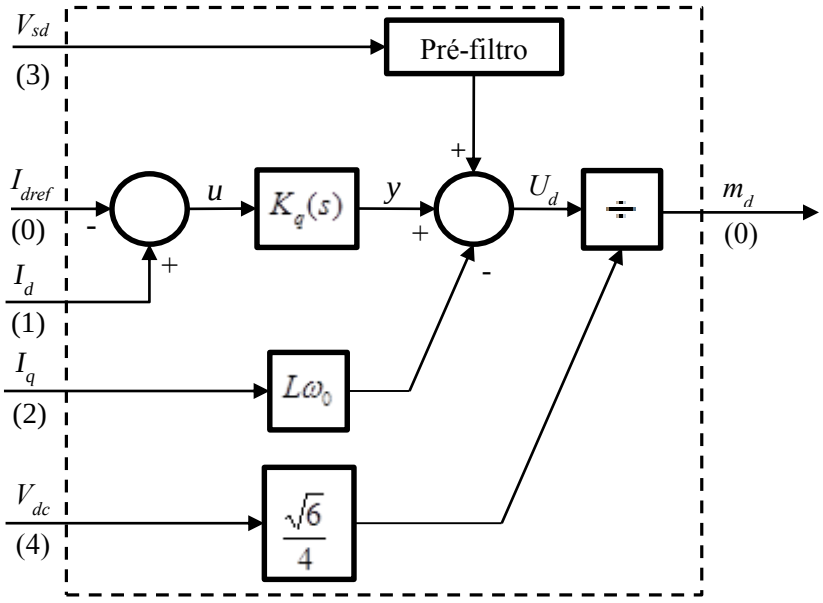

Figura 7-44 - Diagrama em blocos do compensador do eixo *d*

Em Figura 7-43 e Figura 7-44, os números entre parênteses são as numerações das entradas e das saídas.

No compensador do eixo q o produto $X_t = L\omega_0$é a reatância do indutor do filtro de saída do inversor CC/CA e deve ser utilizado o valor desta reatância em pu dividindo a mesma pela impedância base (Z_{base}). Para obter as equações matemáticas que descrevem o comportamento do compensador, deve-se expandir o controlador PI, como apresentado na Figura 7-45.

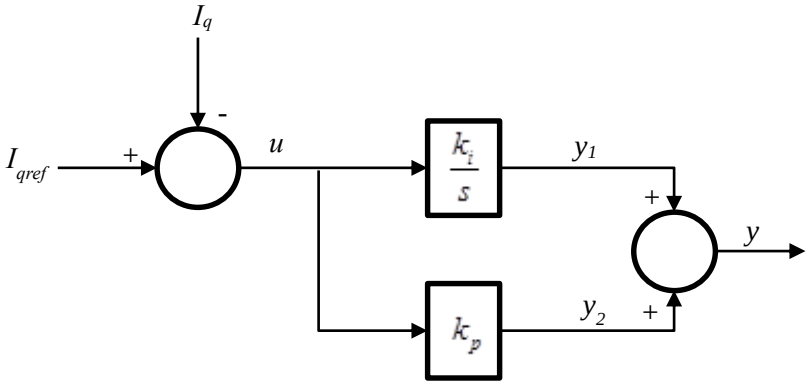

Figura 7-45 - Controle PI expandido do eixo *q*

Ao observar Figura 7-45, pode-se verificar que a única variável de estado é:

$$x = y_1 \tag{7.62}$$

Cuja derivada é dada pela expressão:

$$\frac{dx}{dt} = k_i u \tag{7.63}$$

Onde u:

$$u = I_{qref} - I_q \tag{7.64}$$

Para o cálculo da saída têm-se:

$$y_1 = x \tag{7.65}$$

$$y_2 = k_p u \tag{7.66}$$

$$y = y_1 + y_2 \tag{7.67}$$

As demais equações que compõe o modelo do compensador do eixo q são:

$$U_q = y + I_d X_t \tag{7.68}$$

$$m_q = \frac{U_q}{\frac{\sqrt{6}}{4} V_{dc}} \tag{7.69}$$

A classe que representa o compensador do eixo q recebeu a denominação de *TPVCompQ* e é uma classe derivada da classe fundamental *TBlock*. A declaração da classe é apresentada através da Figura 7-46.

```
1  namespace bcssd {

2  typedef enum {WOCQ=0, LCQ, KPCQ,
3                KICQ, SBASECQ, VBASECQ} TPVCompQParam;

4  class TPVCompQ: public bcssd::TBlock {
5  private:
6     TPVCompQ(){};
7  protected:
8     double Wo,      //Frequencia síncrona em rad/s
9            Lt,      //Indutância do filtro RL do inversor
10           Kp,      //Ganho proporcional do compensador
11           Ki,      //Ganho integral do compensador
```

```
12          Sbase,   //Potência base comum
13          Vbasecc, //Tensão base do lado CC
14          Xt;      //Reatância do indutor do filtro em pu
15 public:
16    TPVCompQ(TVector<double> &param);

17    virtual void
18    Derivatives (TVector<double> & Ysys,
19                 TVector<double> & Xstate,
20                 TVector<double> & Fstate, double T);

21    virtual void
22    Outputs (TVector<double> & Ysys,
23             TVector<double> & Xstate, double T);

24    virtual void
25    Initialize (TVector<double> & Ysys, TVector<double> & Xstate);

26 };

27 }
```

Figura 7-46 - Declaração da classe que representa o compensador do eixo *q*

Na Figura 7-46, na linha 16 está a declaração do construtor da classe onde é passado um vetor com os parâmetros do controlador, onde cada posição corresponde a um parâmetro, como demonstrado através da Tabela 7-5.

Posição no vetor de parâmetros	Parâmetro
W0CQ	Velocidade síncrona em rad/s
LCQ	Indutância do filtro do conversor em H
KPCQ	Ganho proporcional
KICQ	Ganho integral
SBASECQ	Potência base comum em MVA

VBASECQ	Tensão base do lado CC em kV

Tabela 7-5 - Lista de parâmetros a ser passado para a classe no construtor

Para implementar o método *Derivatives* deve-se empregar as equações (7.63) e (7.64). O código do método é apresentado na Figura 7-47.

```
1  void TPVCompQ::Derivatives (TVector<double> & Ysys,
2                              TVector<double> & Xstate,
3                              TVector<double> & Fstate, double T)
4  {
5     double u, x, dx, Iqref, Iq;
6     Iqref = Ysys[GetInput(0)]; //pu
7     Iq    = Ysys[GetInput(1)]; //pu
8     u  = Iqref-Iq;             //eq. 7.64
9     dx = Ki*u;                 //eq. 7.63
10    Fstate[GetState(0)] = dx;
11 }
```

Figura 7-47 - Implementação do método *Derivatives* da classe *TPVCompQ*

Na Figura 7-47 as linhas 8 e 9 mostram a utilização das equações (7.64) e (7.63).

A implementação do compensador do eixo *d* é muito semelhante ao do eixo *q* e a classe que o representa é denominada de *TPVCompD*. As mesmas considerações feitas ao compensador do eixo *q* são válidas ao compensador do eixo *d* e, portanto, sua implementação não será apresentada neste texto.

7.5.4 Conversor CC/CC Boost e o controle de carga do link CC

Uma matriz de células fotovoltaicas possui o comportamento muito semelhante a uma fonte de corrente controlada, onde as variáveis de controle são a temperatura da junção *PN* e a radiação solar. Esta característica de fonte de corrente pode provocar um aumento contínuo da tensão no capacitor do link CC que pode levar ao rompimento da isolação e a consequente destruição do link CC. Este aumento contínuo da tensão ocorre em função da parcela de corrente direcionada ao capacitor cuja tensão é expressa através da equação:

$$V_C = \frac{1}{C}\int I_C dt \tag{7.70}$$

Quando a tensão no capacitor atingir o valor especificado, a corrente injetado no mesmo deve ser zerada. Isto é conseguido através do controle da corrente injetada no inversor CC/CA. Este controle é conseguido através de um conversor CC/CC boost cujo índice de modulação *D* é controlado através de controlador *PI*. O controlador *PI* mede a tensão no capacitor e compara com a tensão de referência e, assim, determina qual o

índice de modulação será usado pelo conversor boost. Através da Figura 7-48 é apresentado um diagrama em blocos deste sistema de controle.

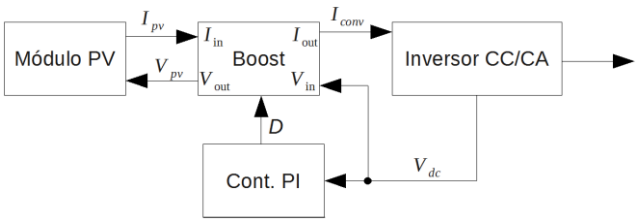

Figura 7-48 - Sistema de controle de carga

O comportamento do conversor boost é descrito por suas equações algébricas:

$$I_{out} = \left(\frac{1}{D-1}\right) I_{in} \tag{7.71}$$

$$V_{out} = \left(\frac{1}{D-1}\right) V_{in} \tag{7.72}$$

Como apenas equações algébricas são empregadas, deve-se substituir apenas os métodos *Outputs* e *Initialize*. Na Figura 7-49 é apresentada a declaração da classe que representa o conversor boost, onde nas linhas 11 a 17 são declarados os métodos substituídos.

```
1  namespace bcssd {

2  class TPVConvBoost: public bcssd::TBlock {
3  protected:
4     double Sbase,  //Potência base comum em MVA
5            Vbase,  //Tensão base do lado CC
6            V0;     //Tensão especificada na saída
7  private:
8     TPVConvBoost();
9  public:
10    TPVConvBoost(double S, double V, double Vout0);

11    virtual void
12    Outputs (TVector<double> & Ysys,
13             TVector<double> & Xstate,
14             double T);
```

```
15    virtual void
16    Initialize (TVector<double> & Ysys,
17                TVector<double> & Xstate);
18 };

19 }
```

Figura 7-49 - Declaração da classe que implementa o conversor boost

A implementação dos métodos *Outputs* e *Initialize* são muito simples, pois avaliam apenas duas equações algébricas. A representação deste dispositivo em um diagrama em blocos é apresentado através da Figura 7-50.

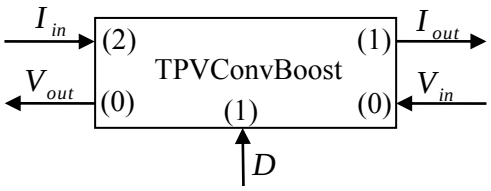

Figura 7-50 - Representação em diagrama de blocos do conversor boost

O controlador *PI* possui o diagrama em blocos apresentado através da Figura 7-51. A classe que o representa no Framework recebeu a denominação *TPVBoostCrtl*.

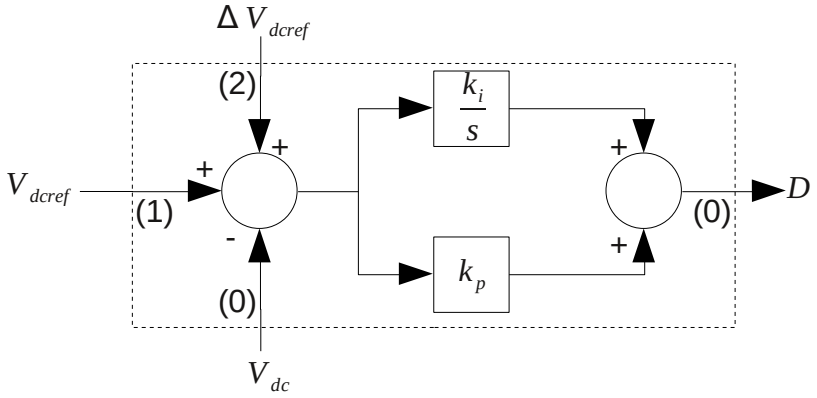

Figura 7-51 - Diagrama em blocos do controlador PI

Na Figura 7-51 a entrada $\boldsymbol{V_{dc}}$ (entrada 0) é a tensão medida no link CC, $\boldsymbol{V_{dcref}}$ é a entrada de referência e $\boldsymbol{\Delta V_{dcref}}$ é uma entrada auxiliar para uso com outros controladores. Os números entre parênteses são as identificações das entradas e das saídas. O controlador *PI* do conversor boost possui uma única variável de estado e sua implementação é análoga ao realizado na implementação dos compensadores de corrente, portanto,

precisa substituir os métodos *Outputs*, *Derivatives* e *Initialize* e cuja declaração é apresentada através da Figura 7-52.

```
namespace bcssd {

class TPVBoostCrtl: public bcssd::TBlock {
protected:
      double Ki,    //Ganho integral
             Kp,    //Ganho proporcional
             Vbase; //Tensão base em kV no lado CC
private:
      TPVBoostCrtl();
public:
      TPVBoostCrtl(double gi, double gp, double V);

      virtual void
      Outputs (TVector<double> & Ysys,
               TVector<double> & Xstate, double T);

      virtual void
      Initialize (TVector<double> & Ysys,
                  TVector<double> & Xstate);

      virtual void
      Derivatives (TVector<double> & Ysys,
                   TVector<double> & Xstate,
                   TVector<double> & Fstate, double T);

};

}
```

Figura 7-52 - Declaração do controlador PI do conversor boost

7.5.5 Rastreador de máxima potência

O rastreamento de máxima potência é um componente muito importante na implementação de uma unidade geradora fotovoltaica. Seu objetivo é encontrar uma corrente I_{MPP} ou uma tensão V_{MPP} que resulta na máxima potência P_{MPP} que a matriz de células PV pode fornecer. O rastreador utilizado neste estudo de caso é digital com período de amostragem de 0,1s.

Através da Figura 7-53 é apresentada a curva da potência versus tensão de um sistema de geração fotovoltaico.

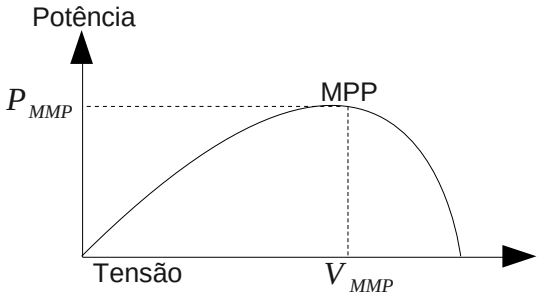

Figura 7-53 - Curva da potência versus tensão de um sistema fotovoltaico

Na Figura 7-53 pode-se verificar que o ponto de máxima potência é denominado de *MMP* e corresponde ao ponto onde a inclinação da curva é zero. Pode-se definir que a esquerda do ponto *MMP* a inclinação da curva é positiva e a direita esta inclinação é negativa, portanto:

$$\frac{dP}{dV} = 0 \text{, para o MPP} \tag{7.73}$$

$$\frac{dP}{dV} > 0 \text{, a esquerda do MPP} \tag{7.74}$$

$$\frac{dP}{dV} < 0 \text{, a direita do MPP} \tag{7.75}$$

Sabendo que:

$$\frac{dP}{dV} = \frac{d(IV)}{dV} = I + \frac{dI}{dV} \cong I + \frac{\Delta I}{\Delta V} \tag{7.76}$$

Aplicando as expressões (7.73), (7.74) e (7.75) em (7.76) então pode-se reescrever as condições como:

$$\frac{\Delta I}{\Delta V} = -\frac{I}{V} \text{, para o MPP} \tag{7.77}$$

$$\frac{\Delta I}{\Delta V} > -\frac{I}{V} \text{, à esquerda do MPP} \tag{7.78}$$

$$\frac{\Delta I}{\Delta V} < -\frac{I}{V}\text{, à direita do MPP} \tag{7.79}$$

Este método de rastreamento baseado na condutância instantânea e nos incrementos da condutância é chamado de *Condutância Incremental* (Esram & Chapman, 2007). O fluxograma que descreve a técnica é apresentado através da Figura 7-54.

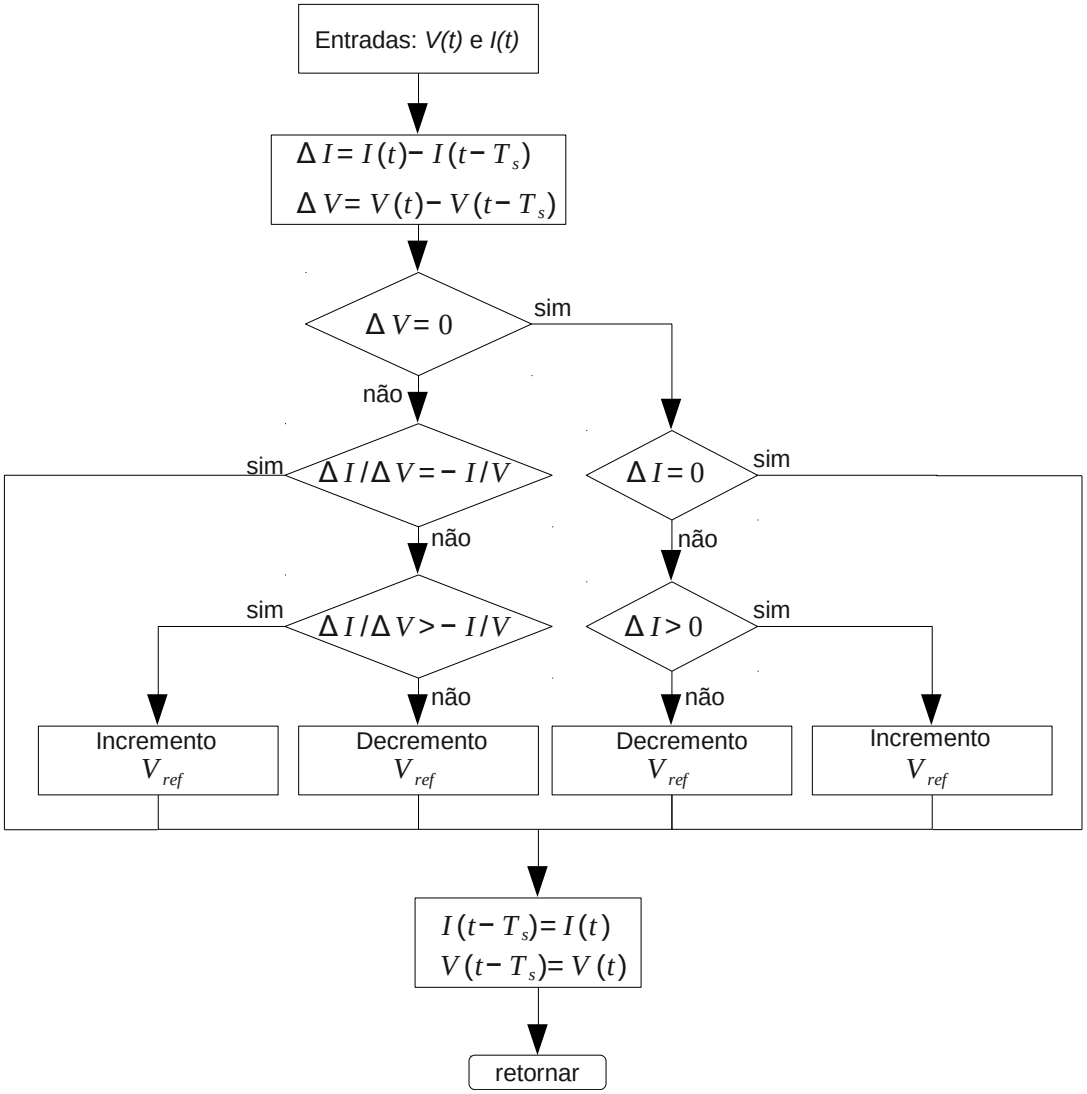

Figura 7-54 - Fluxograma do método de condutância incremental

A variável $\boldsymbol{V_{ref}}$ corresponde a tensão de referência do controlador de carga. A classe desenvolvida para representar o MPPT por condutância incremental recebeu a denominação de *TMpptCond* e o diagrama em blocos é apresentado através da Figura 7-55 e a declaração da classe na Figura 7-56.

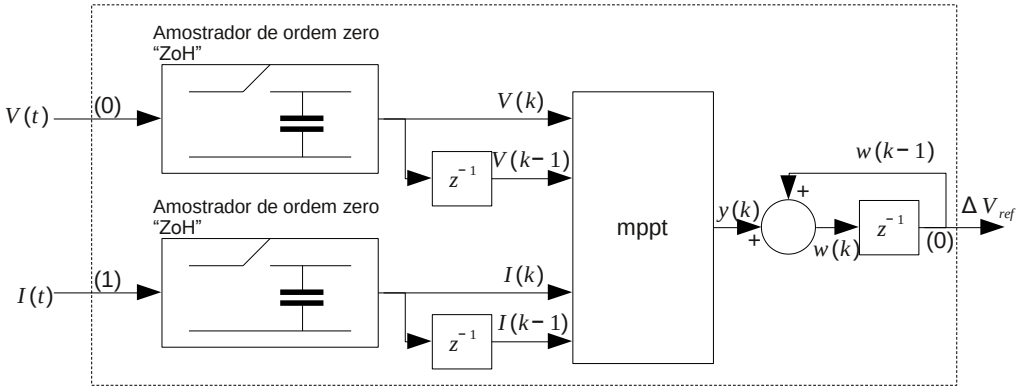

Figura 7-55 - Diagrama em blocos do MPPT

Na Figura 7-55, pode-se observar que as entradas *V(t)* (entrada 0) e *I(t)* (entrada 1) são encaminhadas a dois seguradores de ordem zero para realizar a amostragem do sinal (foi empregado um período de amostragem de 0,1s). Em seguida são aplicados os valores atuais e da amostragem anterior da tensão e da corrente ao bloco *mppt* responsável por implementar o algoritmo da técnica *MPPT* empregada. A declaração da classe que representa o rastreador é apresentada através da Figura 7-56.

```
1  namespace bcssd {

2  class TMpptCond: public bcssd::TBlock {
3  protected:
4     double Ts;       //Período de amostragem
5     double Vk,       //Tensão na amostragem atual
6            Vk_1,     //Tensão na amostragem anterior
7            Ik,       //Corrente na amostragem atual
8            Ik_1,     //Corrente na amostragem anterior
9            wk,       //Valor da amostragem atual da saída do MPPT
10           wk_1;     //Valor da amostragem anterior da saída do MPPT

11    bool ZoH (double In,
12              double &Out,
13              double T);          //Amostrador de ordem zero

14    double mppt(double Vk,        //Tensão na amostragem atual
15                double Vk_1,      //Tensão na amostragem passada
16                double Ik,        //Corrente na amostragem atual
17                double Ik_1);     //Corrente na amostragem passada
```

```
18 private:
19    TMpptCond();
20 public:
21    TMpptCond(double T);

22    virtual void
23    Outputs (TVector<double> & Ysys,
24             TVector<double> & Xstate, double T);

25    virtual void
26    Initialize (TVector<double> & Ysys,
27                TVector<double> & Xstate);
28 };

29 }
```

Figura 7-56 - Declaração da classe que representa o MPPT

Na Figura 7-56, a declaração do segurador de ordem zero encontra-se nas linhas 11 a 13. Este método retorna verdadeiro caso ocorrer a amostragem. O parâmetro *In* contêm o sinal de entrada e *Out* o sinal de saída que seguirá a entrada apenas quando ocorrer uma amostragem. Nas linhas 14 a 17 encontra-se a declaração do método *mppt* que realiza a implementação do fluxograma na Figura 7-54.

A implementação de um dispositivo digital normalmente é composta por equações algébricas, portanto apenas os métodos *Outputs* e *Initialize* são substituídos. Na Figura 7-57 é apresentada a implementação do método *Outputs*.

```
1  void TMpptCond::Outputs (TVector<double> & Ysys,
2                           TVector<double> & Xstate, double T)
3  {
4     double Vpv, Ipv, V, I, yk, DVpvref;
5     Vpv = Ysys[GetInput(0)];
6     Ipv = Ysys[GetInput(1)];
7     if (ZoH (Vpv, V,T) && ZoH (Ipv, I,T)) //Houve amostragem?
8     {
9           Vk_1 = Vk;
```

```
10            Ik_1 = Ik;
11            Vk = V;
12            Ik = I;
13            wk_1 = wk;
14            yk = mppt(Vk, Vk_1, Ik, Ik_1);
15            wk = yk+wk_1;
16      }
17      DVpvref = wk_1;
18      Ysys[GetOutput(0)] = DVpvref;
19  }
```

Figura 7-57 - Implementação do método *Outputs* do MPPT

Nas linhas 5 e 6 na Figura 7-57 são obtidos os valores de tensão e corrente de entrada. Na linha 7 é realizado o teste se uma amostragem ocorreu ou não. Caso haja uma amostragem *V* e *I* conterão os valores amostrados da tensão e da corrente e as linhas de 9 a 15 são executadas e o valor de *wk_1* é atualizado. Caso contrário, *wk_1* permanece constante. Nas linhas 17 e 18 os resultados são disponibilizados ao sistema. Deve-se relembrar que o método *Outputs* é invocado pelo sistema quando o processo de integração numérica está concluído. A saída do MPPT, *DVpvref*, é aplicado na entrada auxiliar do controlador *PI* do conversor boost.

O cálculo das condições iniciais é muito simples, os valores iniciais da corrente e da tensão são conhecidos assim como o valor inicial da saída do *MPPT*, então basta fazer os valores das amostragens anteriores (*k-1*) iguais aos valores das amostragens atuais (*k*). Através da Figura 7-58 é apresentado o método de inicialização da classe.

```
1  void TMpptCond::Initialize (TVector<double> & Ysys,
2                              TVector<double> & Xstate)
3  {
4      Vk = Vk_1 = Ysys[GetInput(0)];
5      Ik = Ik_1 = Ysys[GetInput(1)];
6      wk = wk_1 = Ysys[GetOutput(0)];
7  }
```

Figura 7-58 - Substituição do método *Initialize* para a classe do MPPT

Nas linhas de 4 a 6 na Figura 7-58 é realizada a inicialização das variáveis da classe que representa o *MPPT*.

7.5.6 Conexões entre os dispositivos que compõe o gerador fotovoltaico

Uma vez que as classes que representam os dispositivos presentes em um sistema de geração fotovoltaico criadas, pode-se criar as instâncias de cada dispositivo e realizar as conexões entre as instâncias. O diagrama em blocos do sistema de geração fotovoltaico conectado a rede de transmissão é apresentado através da Figura 7-59.

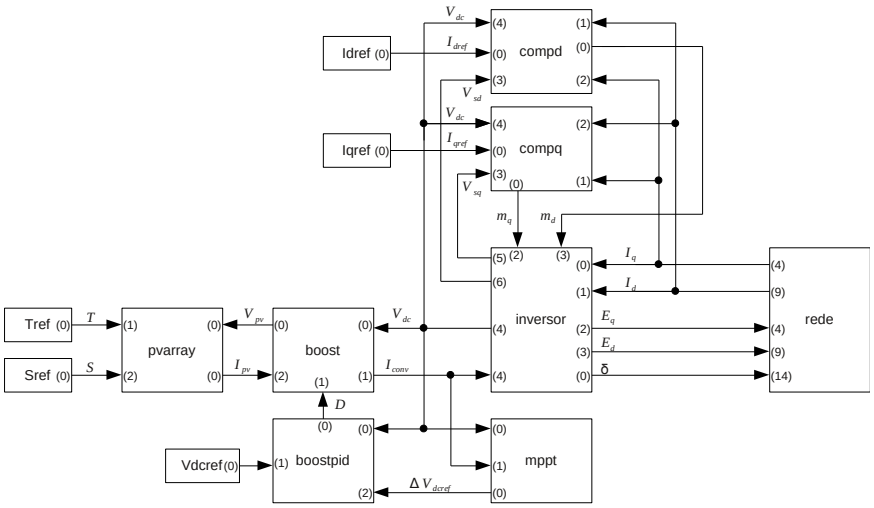

Figura 7-59 - Diagrama em blocos de uma unidade de geração PV

Na Figura 7-59, os objetos *Tref*, *Sref*, *Iqref* e *Idref* representam a temperatura, a radiação solar, a referência de corrente do eixo *q* e a referência de corrente do eixo *d* respectivamente. São ponteiros para instâncias da classe *TSource* que representa uma fonte de sinal constante. A criação destas instâncias é apresentada através da Figura 7-60.

```
TSource *Tref  = new TSource(),
        *Sref  = new TSource(),
        *Iqref = new TSource(),
        *Idref = new TSource();
```

Figura 7-60 - Criação das instâncias que representam as fontes

A criação das instâncias dos outros dispositivos é realizada utilizando os construtores descritos nas seções anteriores. Após a criação de cada instância deve-se utilizar o método *PutBlock* da classe *TSystem* para cadastrar cada instância. As conexões entre as instâncias são realizadas através do método *InputConnect* da classe *TBlock* de acordo com o diagrama em blocos da Figura 7-59. No fragmento de código apresentado através da Figura 7-61 como as conexões entre o inversor e a rede são executadas.

```
1  rede->InputConnect(4,2,inversor);  //Eq''
2  rede->InputConnect(9,3,inversor);  //Eq''
```

```
3  rede->InputConnect(14,0,inversor); //delta
4  inversor->InputConnect(0,4,rede);  //Iq
5  inversor->InputConnect(1,9,rede);  //Id
```

Figura 7-61 - Conexões do inversor com a rede elétrica

As conexões entre as demais instâncias seguem o mesmo princípio e por questão de objetividade não serão mostradas neste texto.

7.6 Simulações

7.6.1 Desacoplamento dos compensadores de corrente

Nestas simulações é demonstrado o desacoplamento entre os compensadores do eixo *q* e do eixo *d*. Nesta simulações são aplicados degraus de 20% na magnitude das correntes de referência do eixo *q* e do eixo *d* no instante de 3s.

O primeiro degrau é aplicado na corrente de referência do eixo *q* para verificar se a corrente no eixo *d* voltará a seu valor inicial. As curvas obtidas para esta primeira simulação são apresentadas na Figura 7-62.

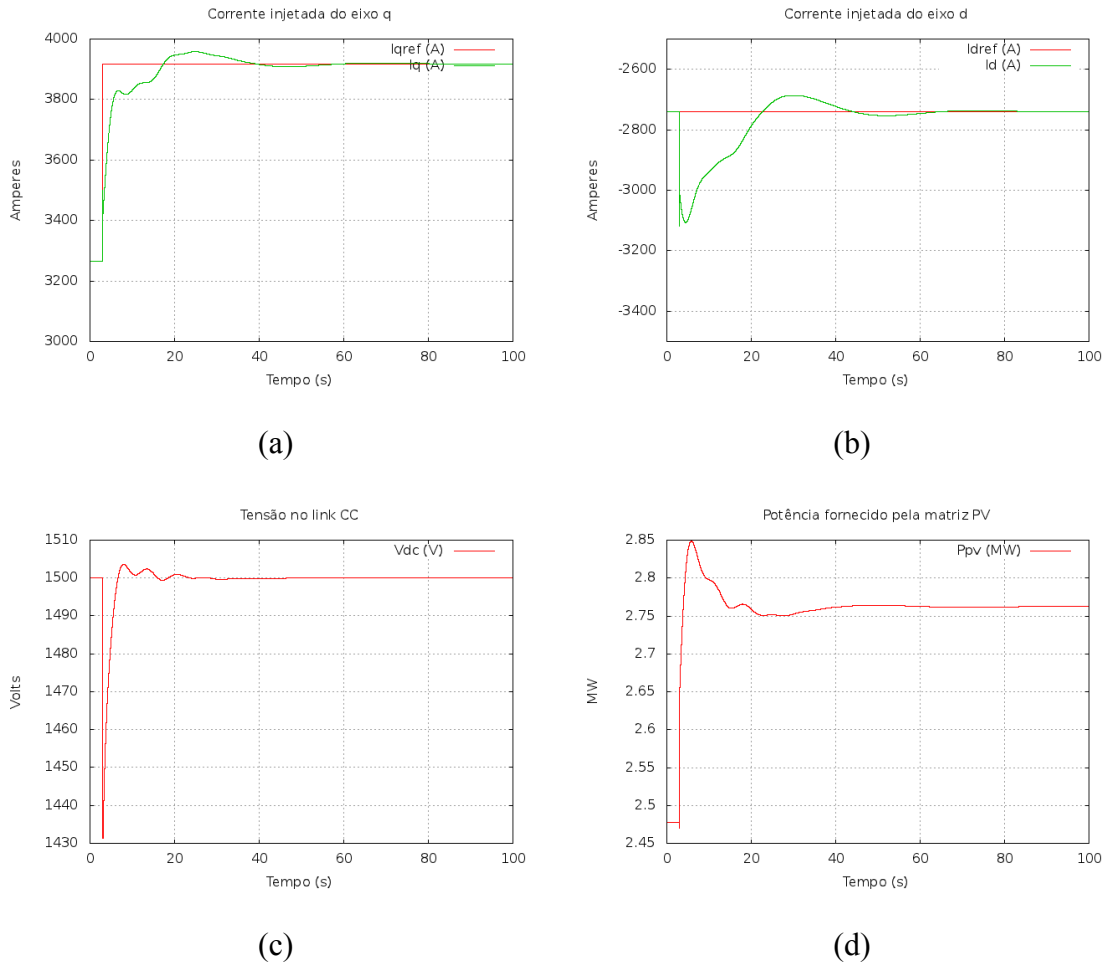

Figura 7-62 - Simulação de um degrau na referência de corrente no eixo *q*

Ao observar a Figura 7-62, pode-se notar que na parte (a) a corrente do eixo *q* (verde) segue a sua referência (vermelho). Na parte (b) da figura a corrente do eixo *d* sofre uma perturbação mas retorna ao seu valor original o que mostra que o comportamento das duas correntes estão desacoplados. Na parte (c) é mostrado o comportamento da tensão do link CC que está ligado a carga do capacitor do link, onde pode-se verificar que esta tensão sofreu uma perturbação mas o controle de carga fez com que a tensão retornasse ao seu valor original. Na parte (d) da figura é apresentado o comportamento da potência fornecida pela matriz fotovoltaica, onde pode-se notar que foi requisitado mais potência.

Na simulação anterior foi demonstrado que o controle da corrente do eixo *q* está desacoplado do eixo *d*, precisa-se agora demonstrar se o sistema de controle de corrente consegue desacoplar o comportamento da corrente do eixo *d* com relação ao eixo *q*. Para isso é aplicado um degrau de 20% na magnitude da corrente do eixo *d* e cujos resultados são apresentados através das curvas na Figura 7-63.

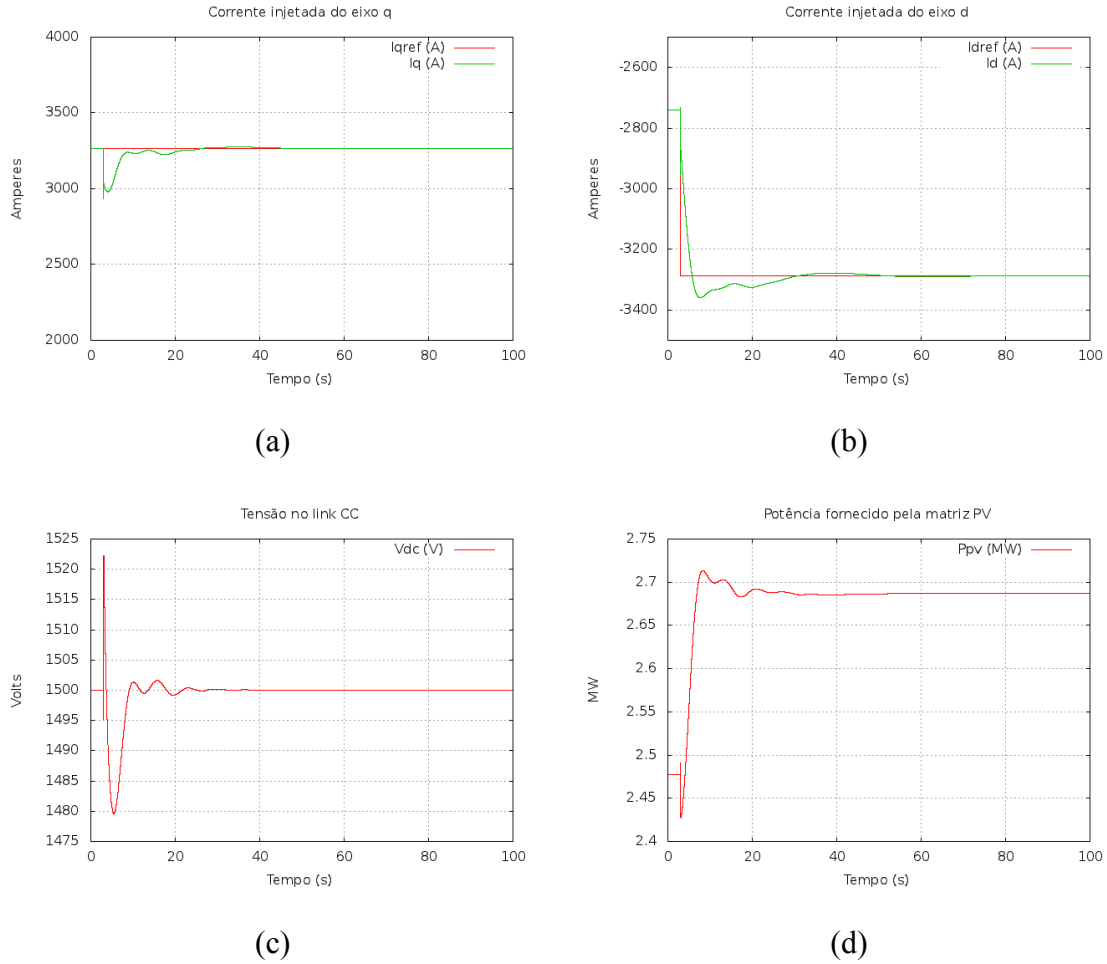

Figura 7-63 - Simulação de um degrau na referência de corrente no eixo *d*

Observa-se na parte (b) da Figura 7-63 que a corrente do eixo *d* segue sua referência e que a corrente do eixo *q* sofre uma perturbação mas retorna ao seu valor original. Observa-se na parte (c) que a tensão do link

CC sofre uma perturbação mas volta ao seu valor original e na parte (d) a potência solicitada à matriz PV aumenta.

7.6.2 Simulação do efeito do rastreamento de máxima potência

Nas simulações anteriores o *MPPT* estava desligado, portanto, as condições de operação do sistema fotovoltaico não estavam na condição de máxima potência. Na simulação apresentada nesta seção o *MPPT* é ativado no instante de três segundos e o sistema rastreia e se acomoda em uma condição onde o *PPT* (ponto de máxima potência) é atingido. As curvas são apresentadas através da Figura 7-64.

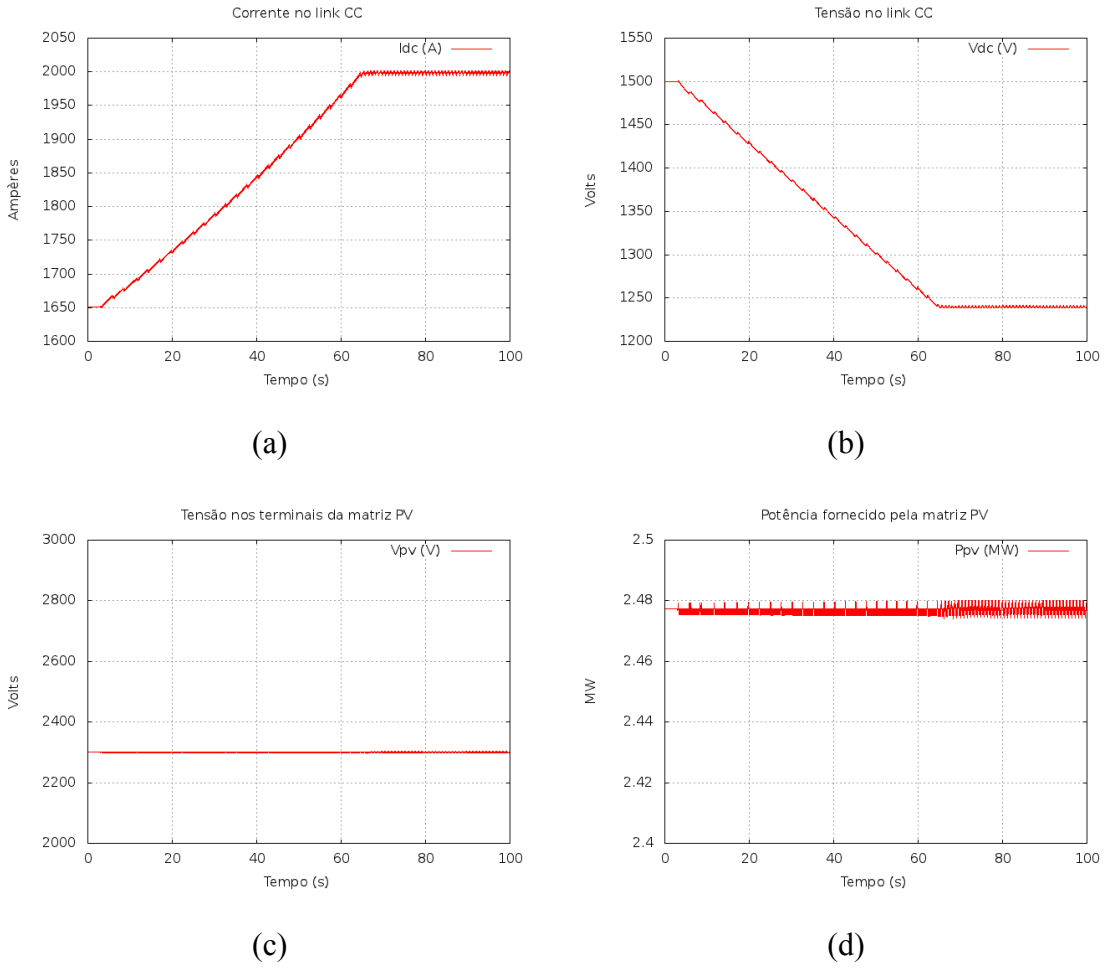

Figura 7-64 - Curvas da simulação em que o *MPPT* é ativado

Ao observar as partes (a) e (b) da Figura 7-64, pode-se observar a ação do *MPPT* nas correntes injetadas no inversor e na tensão do link CC, onde o *MPPT* está diretamente conectado. O conversor boost introduz um nível de isolamento entre a matriz fotovoltaica e o inversor o que reduz o efeito do *MPPT* sobre a matriz fotovoltaica. O impacto do *MPPT* é "absorvido" pelo conversor boost através da alteração do índice de modulação pelo controlador *PI* ligado ao conversor, o que pode ser observado através da Figura 7-65.

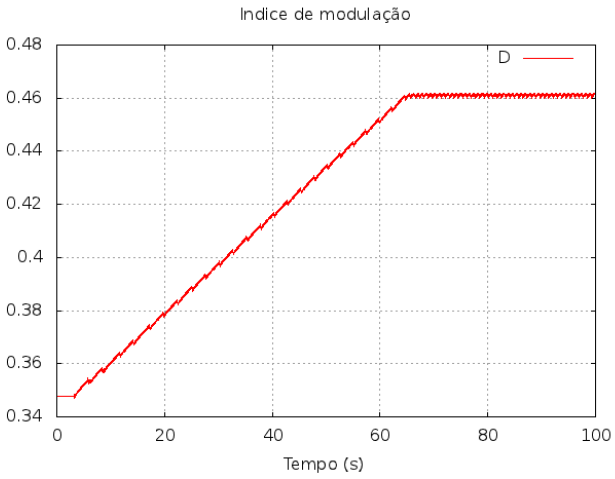

Figura 7-65 - Comportamento do índice de modulação do conversor boost

Uma consequência do emprego do *MPPT* é o surgimento de uma modulação nas variáveis do sistema como pode ser observado através da Figura 7-64 e da Figura 7-65.

7.7 Conclusão

Neste capítulo foi apresentado um estudo de caso onde uma unidade de geração não convencional (fotovoltaica) e seus controles associados foram representados no Framework conectados a um sistema de transmissão composto por unidades de geração síncronas.

O objetivo deste estudo de caso é demonstrar a aplicabilidade do Framework na simulação de sistemas de geração não convencionais onde novos modelos matemáticos são necessários. É apresentado também a implementação de um dispositivo discreto e sua utilização em uma simulação mista, ou seja, em uma simulação onde dispositivos digitais e analógicos estão presentes. Pode-se concluir, então, que o Framework possui uma versatilidade que torna possível a implementação de componentes de geração distribuída, conectados ao sistema elétrico convencional, de forma simples e coerente.

8 Conclusão

Ao longo deste texto foi apresentado o Framework desenvolvido que provê ferramentas essenciais para o desenvolvimento de programas para simulação dinâmica de sistemas de potência.

Esta ferramenta permite que o pesquisador ou estudante escreva seu próprio programa para testar novas topologias de controle ou novos dispositivos cujas implementações em um pacote de software comercial são de difícil execução.

As classes orientadas a objetos que representam matrizes e vetores trazem para o C++ as facilidades disponíveis no Matlab para tratamento de matrizes e vetores (tanto não esparsos como esparsos).

As classes que representam blocos e sistemas permitem que os sistemas dinâmicos, que se deseja simular, possam ser implementados através de conexões entre os blocos e o registro dos mesmos em classe *contêiner*. As conexões entre estes blocos podem ser alteradas em tempo de execução, permitindo que mudanças nas topologias do sistema sofram alterações de forma dinâmica.

As classes que representam blocos não são apenas blocos dinâmicos fechados, a classe ancestral *TBlock* permite que novos blocos representando a dinâmico de novos dispositivos sejam implementados escrevendo uma quantidade mínima de código de programação. Podem ser implementados blocos de tempo contínuo e de tempo discreto, pode-se implementar blocos que implementem inteligência artificial ou qualquer técnica não convencional de controle.

As classes que representam os sistemas dinâmicos são desacoplados das classes que implementam os métodos de integração numérica. Neste trabalho foram implementados dois métodos de integração numérica, o explícito Runge-Kutta de quarta ordem e o implícito trapezoidal, mas outros métodos também podem ser implementados e utilizados sem qualquer modificação no restante do Framework.

Foram apresentados dois blocos fundamentais para simulação dinâmica de sistemas de potência: gerador síncrono e rede elétrica. Os demais dispositivos podem ser implementados pelo próprio usuário do Framework. Para exemplificar o uso do Framework, foram desenvolvidos vários blocos que representam os dispositivos presentes na UHE Tucuruí para o desenvolvimento de um simulador que permitiu validar o Framework ao comparar os resultados obtidos com ensaios experimentais e com resultados fornecidos por outros softwares.

Neste trabalho foi apresentado o desenvolvimento de uma infraestrutura de software que permitirá o desenvolvimento de muitas aplicações. Utilizando os recursos fornecidos pelo Framework, pode-se propor futuros trabalhos na implementação de modelos de inversores, geradores eólicos, geradores fotovoltaicos e outras formas de geração de energia elétrica que venham a surgir.

O novo cenário delineado pela geração distribuída, impõe a necessidade de uma reavaliação dos modelos matemáticos, e suas implementações numéricas, dos dispositivos utilizados em sistemas elétricos de

potência o que resulta em novos desafios no desenvolvimento de simuladores. Neste novo cenário, as redes de distribuição (redes bastantes malhadas) não se comportam mais como entidades passivas que apenas absorvem a energia transmitida até os centros de consumo, com a massificação da geração distribuída, as redes de distribuição poderão, inclusive exportar seus excedentes de produção para os sistemas de transmissão e subtransmissão.

A geração distribuída está provocando uma revolução no setor elétrico, onde os simuladores possuem um papel fundamental. Neste contexto de reinvenção dos simuladores, o Framework apresentado nesta tese contribui fornecendo novos conceitos e metodologias que poderão ajudar neste processo.

É necessário que o Framework seja acrescido de novas classes que representem dispositivos polifásicos e novos algoritmos de fluxo de carga trifásicos sejam inclusos. Ferramentas como fluxo de carga contínuo e fluxo de carga ótimo e análise de pequenos sinais deverão ser adicionados. A estabilidade numérica deve ser avaliada na representação de sistemas com redes malhadas, no entanto, o foco principal dos futuros trabalhos baseados no Framework aqui apresentado, consistirão no desenvolvimento de novas classes derivadas de *TBlock* que representem a dinâmica dos mais diversos dispositivos.

Referências Bibliográficas

Agostini, M. N., Decker, I. C., & Silva, A. S. (15 de Janeiro de 2007). A New Approach for the Design of Electric Power System Software Using Object Oriented Modeling. *Electrical Power & Energy Systems* , pp. 505-513.

Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J. W., Dongarra, J. J., et al. (1999). *LAPACK User's guide (third ed.).* Society for Industrial and Applied Mathematics.

Anderson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Croz, J. D., et al. (1990). LAPACK: A portable linear algebra library for high-performance computers. *Supercomputing '90. Proceedings.* November: IEEE.

Anderson, P. M., & Fouad, A. A. (2002). *Power System Control and Stability.* Wiley IEEE Press.

Aoyama, M. (1994). Evolutionary Patterns of Design and Design Patterns. *IEEE Transactions on Power Systems* , pp. 1045-1051.

Aoyama, M. (2000). Evolutionary Patterns of Design and Design Patterns. *Proceedings International Symposium on Principles of Software Evolution.* Kanazawa, Japan: IEEE.

Arrilaga, J., & Arnold, C. P. (1990). *Computer Analysis of Power Systems.* John Wiley e Sons.

Blackford, S., Dongarra, J., Hammarling, S., Kaufman, L., Maany, Z., Henry, J. D., et al. (1996-2000). *Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard.* University of Tennessee.

Blackfordy, L. S., Choiz, J., Cleary, A., Demmel, J., Dhillon, I., Dongarra, J., et al. (1996). ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing.* Knoxville, TN: IEEE.

Cellier, F., & Elmqvist, H. (Abril de 1993). Automated formula manipulation supports object-oriented continuous-system modeling. *IEEE Control Systems* , pp. 28 -38.

Choi, J., Dongarra, J. J., & Walker, D. W. (1994). PB-BLAS: A Set of Parallel Block Basic Linear Algebra Subprograms. *Proceedings of the Scalable High-Performance Computing Conference.* Knoxville, TN: IEEE.

Chow, J. H., & Cheung, K. W. (Novembro de 1992). A Toolbox for Power System Dynamics and Control Engineering Education and Research. *IEEE Transaction on Power Systems* , pp. 1559-1564.

Crow, M. (2003). *Computacional Methods for Electric Power Systems.* CRC Press.

Dangorra, J. J., Croz, J. D., Duff, I. S., & Hammarling, S. (1990). A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.* , pp. 1-17.

Dangorra, J. J., Croz, J. D., Duff, I. S., & Hammarling, S. (1990). Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.* , pp. 18-28.

Dangorra, J. J., Croz, J. D., Hammarling, S., & Hanson, R. J. (1988). Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.* , pp. 18-32.

Deitel, H. M., & Deitel, P. J. (2001). *C++ How to program: Third edition.* Prentice-Hall.

Di Paolo, Í. F. (2009). Aplicação de técnicas de padrões de projeto orientados a objeto na construção de framework para modelagem e simulação dinâmica de geradores síncronos. *Dissertação de Mestrado* . Belém, PA: Universidade Federal do Pará.

Di Paolo, Í. F., Cordeiro, T. D., Sena, J. A., Fonseca, M. C., & Barra, W. (3 de Agosto de 2010). Creational Object - Oriented Design Pattern Applied to the Development of Software Tools for Electric Power Systems Dynamic Simulations. *IEEE Latin America Transactions* , pp. 287-295.

Di Paolo, Í. F., Fonseca, M. C., Sena, J. A., Nogueira, F. G., Damasceno, T., & Barra, W. (2011). Investigação de Fenômenos Dinâmicos na Usina Hidrelétrica de Tucuruí Através de uma Metodologia Orientada a Objeto. *Anais so IX Congresso Latino Americano de Geração e Transmissão de Energia Elétrica.* Mar del Plata.

digSILENT. (2012). *DigSILENT Corporation*. Acesso em 11 de 06 de 2012, disponível em Products: PowerFactory: htpp://www.digsilent.de/index.php/products-powerfactory.html

Dingle, A., & Hildebrandt, T. H. (1998). Improving C++ Performance Using Temporaries. *IEEE - Computer* , 31-41.

Dongarra, J. J., Pozo, R., & Walker, D. W. (1993). LAPACK++: A Design Overview of Object-Oriented Extensions for High Performance Linear Algebra. *Supercomputing '93. Proceedings.* IEEE.

Duff, I. S., Heroux, M. A., & Pozo, R. (Junho de 2002). An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *Transactions on Mathematical Software (TOMS)* .

Eletronorte. (1988). *Determinação da impedância equivalente do sistema Eletronorte na barra-500 KV da SE Presidente Dutra – Geração Máxima.* Belém: Eletronorte.

Eletronorte. (2001). *Sistema de excitação estática digital – diagrama em blocos do regulador de tensão.* Belém: Eletronorte.

Esram, T., & Chapman, P. L. (Junho de 2007). Comparison of Phototovoltaic Array Maximum Power Point Tracking Techiques. *IEEE Transactions of Energy Conversion* , pp. 439-449.

Ferreira, A. M. (2005). *Amortecimento de oscilações eletromecânicas em sistemas elétricos de potência utilizando controle robusto adaptativo em dispositivos FACTS TCSC.* Belém: PPGEE/UFPA.

Fonseca, M. C., Gomes, M. C., Barra Jr, W., & Sena, J. A. (20 de Maio de 2012). Desenvolvimento de um Simulador para Estudos de Estabilidade Dinâmica de Sistemas Fotovoltaicos Conectados a Rede Elétrica - Um Estudo de Caso para o Sistema Hidrotérmico do Estado do Amapá. *Anais do XII SEPOPE* . Rio de Janeiro, Rio de Janeiro, Brasil: Cigrè Brasil.

Fritzson, P., & Bunus, P. (2002). Modelica - a general object-oriented language for continuous and discrete-event system modeling and simulation. *Symposium, 2002. Proceedings. 35th Annual* (pp. 365 - 380). IEEE.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns - Elements of Reusable Object-Oriented Softwares.* Addison - Wesley.

Hakavik, B., & Holen, A. T. (1994). Power System Modelling and Sparse Matrix Operations Using Object-Oriented Programming. *IEEE Transactions on Power Systems* , pp. 1045-1051.

Hebel, Z., Kajgnanié, B., & Delimar, M. (2000). On Sparse Matrices in C++ using Templates and Collection Classes Part I - Modelling Power Systems Matrices. *22 Inf. Conf. Information Technology Interfaces.* Pula, Croatia: IEEE.

Intel Corporation. (2012). *Math Kernel Library*. Acesso em 14 de 06 de 2012, disponível em Intel Software Network: http://software.intel.com/en-us/articles/intel-mkl/

Krogh, F. T. (1973). On Testing a Subroutine for the Numerical Integration of Ordinary Differential Equations. *Journal of the Association for Computing Machinery* , pp. 545-562.

Kundur, P. (1994). *Power System Stability and Control.* McGraw-Hill.

Larman, C. (1999). *Applying UML and Patters - An Introduction to Object-Orientd Analysis and Design.* Prentice-Hall.

Larsson, M. (Fevereiro de 2004). Objectstab - An Educational Tool for Power System Stability Studies. *IEEE Transaction on Power Systems* , pp. 56-63.

Lawson, C. L., Hanson, R. J., Kincaid, D., & Krogh, F. T. (1979). Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Trans. Math. Soft.* , pp. 308-323.

Leelaruji, R., & Vanfretti, L. (2012). Detailed Modelling, Implementation and Simulation of an "all-in-one" Stability Test System Including Power System Protective Devices. *Simulation Modelling Practice and Theory* , pp. 36-59.

Manitoba - HVDC RESEARCH CENTRE. (2012). *Manitoba - Research Centre*. Acesso em 11 de 06 de 2012, disponível em Products: PSCAD: https://pscad.com/products/pscad

Manzoni, A., Silva, A. S., & Decker, I. C. (Fevereiro de 1999). Power Systems Dynamics Simulations Using Object-Oriented Programming. *IEEE Transactions on Power Systems* .

McMorran, A. W., Ault, G. W., Morgan, C., Elders, I. M., & McDonald, J. R. (Fevereiro de 2006). A Common Information Model (CIM) Toolkit Framework Implemented in Java. *IEEE Transaction on Power Systems* , pp. 194-201.

Milano, F. (Novembro de 2005). An Open Source Power System Analysis Toolbox. *IEEE Transaction on Power Systems* , pp. 1199-1206.

Monticelli, A. J. (1983). *Fluxo de carga em redes elétricas.* São Paulo: Edgard Blücher.

Neyer, A. F., Wu, F. F., & Imhof, K. (Agosto de 1990). Object-oriented programming for flexible software: example of a load flow. *IEEE Transactions on Power Systems* , pp. 689-696.

Nogueira, F. G., Sena, J. A., Fonseca, M. C., & Barra, W. (2012). Design and field tests of a digital control system to damping electromechanical oscillations between large diesel generators. In: Intech, *Diesel Engines.* Intech.

Oliveira, S. E., Rangel, R. D., Thomé, L. M., Baitelli, R., & Guimarães, C. H. (1994). Programa ANATEM para Simulação do Desempenho Dinâmico dos Sistemas Elétricos de Potência. *IV SEPOPE - Simpósio de Planejamento e Operação de Sistemas Elétricos de Potência.* Foz do Iguaçu - PR: CIGRÈ - Brasil.

Operador Nacional do Sistema. (2006a). *Avaliação da Regulação de Velocidade da UHE de Tucuruí na Ocorrência do Dia 20/12/2005 às 12h40min.* Rio de Janeiro: ONS.

Operador Nacional do Sistema. (2008). *Casos de referência – estabilidade – transitórios eletromecânicos. bdadosMAR2008.zip: base de dados do sistema interligado nacional para uso no ANATEM e ANAT0.* Rio de Janeiro: ONS.

Operador Nacional do Sistema. (2003a). *Novo Modelo do Regulador de Velocidade da UHE Tucuruí.* Rio de Janeiro: ONS.

Operador Nacional do Sistema. (2003b). *Otimização do Regulador de Velocidade da UHE Tucuruí.* Rio de Janeiro: ONS.

Operador Nacional do Sistema. (2004). *Reajuste e Validação do Regulador de Tensão e PSS da UHE Tucuruí II.* Rio de Janeiro: ONS.

Operador Nacional do Sistema. (2006b). *Validação e Otimização do Regulador de Velocidade da UHE Tucuruí - Segunda Etapa.* Rio de Janeiro: ONS.

Pandit, S., Soman, S. A., & Khaparde, S. A. (Novembro de 2001). Design of Generic Sparse Linear System Solver in C++ for Power System Analysis. *IEEE Transactions on Power Systems* , pp. 647-652.

Pöller, M., Schmid, C., & Schmieg, M. (1997). Object Oriented Modeling of Power System Devices. *Electric Machines and Drives Conference Record* (pp. TC3/5.1 - TC3/5.3). IEEE International.

Sauer, P. W., & Pai, M. A. (1998). *Power System Dynamics and Stability.* Prentice-Hall.

Sena, J. A., Barra, W. J., Barreiros, J. A., Fonseca, M. d., Campos, B. M., & Costa, C. T. (2005). Estratégias de Padrões de Projeto do tipo Bridge para o Desenvolvimento de Softwares para Simulação de Sistemas Dinâmicos de Grande Porte. *Sixth Latin-American Congress on Electricity Generation and Transmission.* Mar del Plata, Argentina: IEEE.

Sena, J. A., Campos, B. M., Fonseca, M. C., Barra, W., Barreiros, J. A., & Costa, C. T. (2006). Framework para Simulação de Sistemas Dinâmicos de Grande Porte: Aplicação em Sistemas Elétricos de Potência. *Anais do XVI Congresso Brasileiro de Automática.* Salvador: Sociedade Brasileira de Automática.

Sena, J. A., Fonseca, M. C., Di Paolo, Í. F., & Barra, W. (2011). An Object-Oriented Framework Applied to the Study of Electromechanical oscillations at Tucurui Hydroelectric Power Plant. *Electrical Power System Research* , pp. 2081 - 2087.

Shalloway, A., & Trott, J. R. (2002). *Design Patters Explained: A New Perspective on Objected-Oriented Design.* Addison-Wesley.

Soman, S. A., Khaparde, S. A., & Pandit, S. (2002). *Computational Methods for Large Sparse Power Systems Analysis - An Object Oriented Approach.* KLUWER ACADEMIC PUBLISHERS.

Stubbe, M., Bihain, A., Deuse, J., & Baader, J. C. (Fevereiro de 1989). EUROSTAG - A New Unified Software Program for the Study of the Dynamic Behaviour of Electrical Power Systems. *IEEE Transactions on Power Systems* , pp. 129-138.

Thomasma, T., & Madsen, J. (1990). Object Oriented Programming Languages for Developing Simulation-Related Software. *Proceedings of the 1990 Winter Simulation Conference.* IEEE.

Whaley, R. C., & Dongarra, J. J. (1998). Automatically tuned linear algebra software. *Supercomputing '98 Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM).* Washington, DC: IEEE.

Wikipédia: A enciclopédia livre. (s.d.). *Turbina hidráulica*. Acesso em 25 de junho de 2012, disponível em Wikipédia em português: http://pt.wikipedia.org/wiki/Turbina_hidráulica

Zhou, E. Z. (Fevereiro de 1996). Object-oriented Programming, C++ and Power System Simulation. *IEEE Transaction on Power Systems* , pp. 206-212.

Zhu, J., & Jossman, P. (Maio de 1999). Application of Design Patterns for Object-Oriented Modeling of Power Systems. *IEEE Transaction on Power Systems* , pp. 532-537.

Zhu, J., & Lubkeman, D. L. (Maio de 1997). Object-Oriented Development of Software Systems for Power System Simulations. *IEEE Transaction on Power Systems* , pp. 1002-1007.