



UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Civil, Arquitetura e Urbanismo

ANDRÉ FELIPE APARECIDO DE MELLO

**SPMTOOL: UMA FERRAMENTA COMPUTACIONAL PARA ANÁLISE DE
ESTRUTURAS DE CONCRETO ARMADO PELO MÉTODO BIELA PAINEL**

CAMPINAS
2022

ANDRÉ FELIPE APARECIDO DE MELLO

**SPMTOOL: UMA FERRAMENTA COMPUTACIONAL PARA ANÁLISE DE
ESTRUTURAS DE CONCRETO ARMADO PELO MÉTODO BIELA PAINEL**

Tese de Doutorado apresentada à Faculdade de Engenharia Civil, Arquitetura e Urbanismo da UNICAMP, para obtenção do título de Doutor em Engenharia Civil, na área de Estruturas e Geotécnica.

Orientador: Leandro Mouta Trautwein

ESTE EXEMPLAR CORRESPONDE À VERSÃO FINAL DA TESE DEFENDIDA PELO ALUNO ANDRÉ FELIPE APARECIDO DE MELLO E ORIENTADO PELO PROF. DR. LEANDRO MOUTA TRAUTWEIN.

**CAMPINAS
2022**

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Rose Meire da Silva - CRB 8/5974

M789s Mello, André Felipe Aparecido de, 1991-
SPMtool: uma ferramenta computacional para análise de estruturas de concreto armado pelo Método Biela Painel / André Felipe Aparecido de Mello. – Campinas, SP : [s.n.], 2022.

Orientador: Leandro Mouta Trautwein.
Tese (doutorado) – Universidade Estadual de Campinas, Faculdade de Engenharia Civil, Arquitetura e Urbanismo.

1. Concreto armado. 2. Análise não-linear. 3. Método dos elementos finitos. 4. Análise estrutural (Engenharia). 5. Estruturas de concreto. I. Trautwein, Leandro Mouta, 1977-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Civil, Arquitetura e Urbanismo. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: SPMtool: a computational tool for the analysis of reinforced concrete structures using Stringer-Panel Method

Palavras-chave em inglês:

Reinforced concrete

Nonlinear analysis

Finite element method

Structural analysis (Engineering)

Concrete structures

Área de concentração: Estruturas e Geotécnica

Titulação: Doutor em Engenharia Civil

Banca examinadora:

Leandro Mouta Trautwein [Orientador]

Thomaz Eduardo Teixeira Buttignol

Wayne Santos de Assis

Rafael Alves de Souza

João da Costa Pantoja

Data de defesa: 01-07-2022

Programa de Pós-Graduação: Engenharia Civil

Identificação e informações acadêmicas do(a) aluno(a)

- ORCID do autor: <https://orcid.org/0000-0003-0627-1904>

- Currículo Lattes do autor: <http://lattes.cnpq.br/3912140825419452>

UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Civil, Arquitetura e Urbanismo

**SPMTOOL: UMA FERRAMENTA COMPUTACIONAL PARA ANÁLISE DE
ESTRUTURAS DE CONCRETO ARMADO PELO MÉTODO BIELA PAINEL**

ANDRÉ FELIPE APARECIDO DE MELLO

Tese de Doutorado aprovada pela Banca Examinadora, constituída por:

Prof. Dr. Leandro Mouta Trautwein

Presidente e Orientador

Universidade Estadual de Campinas

Prof. Dr. Thomaz Eduardo Teixeira Buttignol

Universidade Estadual de Campinas

Prof. Dr. Wayne Santos de Assis

Universidade Federal de Alagoas

Prof. Dr. Rafael Alves de Souza

Universidade Estadual de Maringá

Prof. Dr. João da Costa Pantoja

Universidade de Brasília

A Ata da defesa com as respectivas assinaturas dos membros encontra-se no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 01 de julho de 2022

Agradecimentos

À minha família, em especial aos meus pais, Valdir e Neuza, que, apesar de todas as dificuldades, acreditaram e apoiaram a realização dessa jornada.

À minha esposa Nayza, pela paciência empenhada ao longo desses anos de curso, por estar ao meu lado e tornar o caminho menos árduo.

Ao meu caro orientador e amigo Dr. Leandro Mouta Trautwein, por acreditar no potencial deste estudo e pelas valiosas contribuições oferecidas.

Ao prezado Dr. Rafael Alves de Souza, meu amigo e orientador de Mestrado, pelas críticas construtivas que contribuíram diretamente para eu chegar até aqui.

Aos membros da banca examinadora, pela disponibilidade de avaliação e pela notória contribuição ao conteúdo deste trabalho.

Aos professores e técnicos da FECFAU/UNICAMP que tive o prazer de conhecer durante minha formação, pelo auxílio direto e indireto à conclusão deste processo.

À Universidade Federal da Grande Dourados (UFGD), da qual orgulhosamente faço parte do corpo docente, pelo apoio financeiro concedido por meio de afastamento integral.

Aos colegas e amigos, pela companhia nos momentos de descontração, pelos conselhos e trocas de experiência.

Resumo

Uma estrutura de concreto armado é formada por duas regiões distintas: as regiões B , onde as deformações por cisalhamento são desprezíveis e se pode aplicar a simplificação de que as seções permanecem planas após a deformação, e as regiões D , onde as deformações por cisalhamento são significativas. Nestas últimas, a análise é normalmente feita através do Método das Bielas (MB) ou do Método dos Elementos Finitos (MEF). O Método Biela-Painel (MBP), um modelo intermediário entre o MB e o MEF, teve suas primeiras aplicações na engenharia civil na década de 1960, seguindo o conceito de dividir a estrutura em dois tipos distintos de elementos: as bielas, que carregam tensões normais, e os painéis, que carregam tensões cisalhantes. No final da década de 1990, a formulação do MBP foi utilizada no desenvolvimento do programa *SPanCAD* (HOOGENBOOM, 1998). Infelizmente esse programa se tornou obsoleto e, atualmente, não existe um programa computacional atualizado para aplicação deste método. Assim, este trabalho visa desenvolver a ferramenta computacional *SPMTool*, para análise de estruturas de concreto armado utilizando o MBP. Este programa se trata de um complemento para as versões mais recentes do *AutoCAD*, no qual é possível executar análises elásticas que auxiliam no dimensionamento de elementos estruturais, bem como análises não-lineares para previsão de deslocamentos, aberturas de fissuras e capacidade de carga. Em conjunto com a biblioteca *RCMembrane*, também desenvolvida neste trabalho, o programa utiliza os modelos mais desenvolvidos de análise de elementos de membrana, como a Teoria Modificada do Campo de Compressão (TMCC), o Modelo de Campo de Tensão Perturbado (MCTP) e o Modelo de Membrana Atenuado (MMA). Os modelos foram validados a partir de resultados de ensaios experimentais de painéis, vigas-parede, vigas com aberturas e consolos, disponíveis na literatura. Foram desenvolvidos exemplos de dimensionamento a partir de análises elásticas do *SPMTool*, e os resultados das simulações não-lineares foram comparadas aos resultados do software comercial *ATENA 2D*. De modo geral, os modelos do *SPMTool* se deformaram mais que os do *ATENA*, porém, conseguiu-se notáveis aproximações de carregamentos de fissuração e ruptura, com custo computacional muito inferior. Por fim, mostrou-se que o *SPMTool* é uma ferramenta de uso intuitivo, que utiliza apenas parâmetros usuais dos materiais, e, por se tratar de um complemento para *AutoCAD*, torna-se atrativo para utilização no dia-a-dia dos escritórios de cálculo estrutural e aplicação educacional.

Palavras-chave: Método Biela-Painel, Análise não-linear, Concreto armado, Regiões D.

Abstract

A reinforced concrete structure is formed by two distinct regions: B regions, where shear strains are negligible and the simplification that cross-sections remain plane after deformation can be applied, and D regions, where shear strains are significant. For these last, analysis is generally done by Strut and Tie Method (STM) or Finite Element Method (FEM). The Stringer-Panel Method (SPM), an intermediate model between STM and FEM, was first applied in civil engineering in the 1960s, and its concept is to divide a structure in two distinct elements: stringers, which carry normal stresses, and panels, which carry shear stresses. In the late 1990s, the SPM's formulation was used in the development of *SPanCAD* (HOOGENBOOM, 1998). Unfortunately, this program became obsolete and, nowadays, there is no updated computer program for the application of this method. So, this research aims to develop the software *SPMTool*, for analysis of reinforced concrete structures by SPM. This program is a plugin for the most recent versions of *AutoCAD*, in which it's possible to execute elastic analysis that assist the design of structural elements, as well non-linear analyses for predicting displacements, crack openings and load capacity. Together with the *RCMembrane* library, also developed in this research, the program uses the most advanced membrane element analysis models, such as the Modified Compression Field Theory, the Disturbed Stress Field Model and the Softened Membrane Model. The models were validated from experimental results of panels, deep beams, beams with openings and corbels, available in the literature. Design examples were developed from *SPMTool* elastic analyses, and the nonlinear simulations' results were compared to the commercial software *ATENA 2D*. In general, *SPMTool*'s models deformed more than *ATENA*'s, however, remarkable approximations were achieved for cracking and ultimate loads, with much lower computational cost. Finally, it was showed that *SPMTool* is an intuitive tool that uses only materials' usual parameters, and, for being a plugin for *AutoCAD*, it becomes attractive for the day-to-day use in structural design offices and educational use.

Keywords: Stringer-Panel Method, Nonlinear analysis, Reinforced concrete, D regions.

Lista de ilustrações

Figura 1.1 – Definição das regiões B e D numa estrutura de concreto armado	22
Figura 2.1 – Estruturas idealizadas como um conjunto de elementos de membrana .	26
Figura 2.2 – Equipamento de teste de painéis de concreto armado	27
Figura 2.3 – Deslizamento da fissura (<i>crack slip</i>)	28
Figura 2.4 – Elemento de membrana no estado plano de tensões	29
Figura 2.5 – Círculo de Mohr para tensões médias	30
Figura 2.6 – Deformações num elemento de membrana	31
Figura 2.7 – Círculo de Mohr para deformações no concreto	32
Figura 2.8 – Definição do módulo secante	34
Figura 2.9 – Relação tensão-deformação para o concreto de acordo com a TMCC .	43
Figura 2.10–Relação tensão-deformação para o aço	44
Figura 2.11–Distribuição de tensões num elemento de concreto armado fissurado .	45
Figura 2.12–Tensões atuantes no elemento de membrana	45
Figura 2.13–Transmissão de tensões na fissura através do engrenamento dos agregados	46
Figura 2.14–Procedimento de verificação da fissura para um tirante de concreto .	47
Figura 2.15–Deformações no elemento de membrana de acordo com o MCTP	50
Figura 2.16–Relação tensão-deformação para o concreto na compressão de acordo com o MCTP	52
Figura 2.17–Relações tensão-deformação para o concreto à tração de acordo com o MCTP	54
Figura 2.18–Tensões principais no elemento de membrana conforme o MMA	57
Figura 2.19–Círculo de Mohr para tensões e deformações no elemento de membrana conforme o MMA	58
Figura 2.20–Dados experimentais dos coeficientes de Poisson em painéis de concreto armado	60
Figura 2.21–Relação tensão-deformação para o concreto de acordo com o MMA .	61
Figura 3.1 – Modelo biela-painel para uma asa de aeronave	63
Figura 3.2 – <i>Stringer Method</i> aplicado a uma parede com orifício: (a) modelo ideal- izado e (b) esforços nas bielas	64
Figura 3.3 – Idealização de uma viga fissurada em concreto armado	64
Figura 3.4 – Modelo <i>Stringer and Wall</i>	65
Figura 3.5 – Modelo Biela-Painel para uma viga biapoiada	68
Figura 3.6 – Comportamento linear das forças normais nas bielas	69
Figura 3.7 – Viga-parede com furo na alma investigada pelo MBP	69
Figura 3.8 – Esforços atuantes num modelo de viga-parede	70
Figura 3.9 – Exemplo de modelagem de uma viga-parede pelo MBP	71

Figura 3.10–Exemplo de modelo biela-painel	73
Figura 3.11–Método de Newton-Raphson	78
Figura 3.12–Método de Newton-Raphson Modificado	79
Figura 3.13–Método Secante Incremental	80
Figura 3.14–Método de Força Incremental	81
Figura 3.15–Método do Comprimento de Arco	82
Figura 4.1 – Criação do objeto <i>Membrane</i> pela biblioteca <i>Reinforced Concrete Mem-</i> <i>brane Library (RCMembrane)</i>	85
Figura 4.2 – Fluxograma de análise pelo MEF utilizando a biblioteca <i>RCMembrane</i>	86
Figura 4.3 – Fluxograma da rotina <i>Membrane.Calculate</i>	86
Figura 4.4 – Diagrama de dispersão dos resultados obtidos	88
Figura 4.5 – Diagramas de caixa dos resultados obtidos	88
Figura 4.6 – Diagrama de deformação $\times$ tensão de cisalhamento para alguns painéis	90
Figura 5.1 – Interface do <i>Stringer-Panel Modelling Tool (SPMTool)</i> como um com- plemento do <i>AutoCAD</i>	97
Figura 5.2 – Janelas de configuração	98
Figura 5.3 – Janela de edição de geometria e armadura dos elementos	99
Figura 5.4 – Exemplo de modelo biela-painel para uma viga-parede, criado no <i>SPMTool</i>	100
Figura 5.5 – Janela de configuração dos parâmetros do concreto	101
Figura 5.6 – Janela de configuração dos parâmetros de análise não-linear	102
Figura 5.7 – Exemplo de resultados do <i>SPMTool</i>	103
Figura 5.8 – Exemplo de gráfico Fator de carga $\times$ deslocamento plotado pelo <i>SPMTool</i>	105
Figura 5.9 – Diagramas de dispersão para os modelos analisados	106
Figura 5.10–Diagramas de caixa para os modelos analisados	107
Figura 5.11–Gráficos força $\times$ deslocamento para as vigas de Ismail, Guadagnini e Pilakoutas (2016)	109
Figura 5.12–Geometria dos modelos analisados no <i>SPMTool</i>	113
Figura 6.1 – Equilíbrio na fissura para um elemento de membrana submetido ao estado de cisalhamento puro	120
Figura 6.2 – Esforços atuantes numa biela	121
Figura 6.3 – Painel de concreto armado submetido ao estado de cisalhamento puro .	123
Figura 6.4 – Viga-parede contínua	128
Figura 6.5 – Viga-parede contínua: tensões no concreto	130
Figura 6.6 – Viga-parede contínua: detalhamento das armaduras	131
Figura 6.7 – Viga-parede contínua: modelos desenvolvidos no <i>ATENA 2D</i>	132
Figura 6.8 – Viga-parede contínua: diagramas de comportamento	133
Figura 6.9 – Viga-parede com aberturas	136
Figura 6.10–Viga-parede com aberturas: tensões no concreto	138
Figura 6.11–Viga-parede com aberturas: detalhamento das armaduras	139

Figura 6.12–Viga-parede com aberturas: modelos desenvolvidos no <i>ATENA 2D</i> . . .	139
Figura 6.13–Viga-parede com aberturas: diagramas de comportamento	140
Figura 6.14–Consolo curto	143
Figura 6.15–Consolo curto: detalhamento das armaduras	145
Figura 6.16–Consolo curto: modelos desenvolvidos no <i>ATENA 2D</i>	145
Figura 6.17–Consolo curto: diagramas de comportamento	146
Figura A.1–Relações entre os parâmetros da biela	158
Figura A.2–Biela carregada com força distribuída	159
Figura A.3–Deslocamentos de uma biela num sistema global de coordenadas	159
Figura A.4–Elemento quadrático em coordenadas ξ	160
Figura A.5–Funções de forma G_1 , G_2 e G_3	161
Figura A.6–Interpolação quadrática do campo de deslocamentos	161
Figura A.7–Biela carregada com força normal constante	164
Figura B.1 –Painel: modelo linear	166
Figura B.2–Relações entre os parâmetros do painel	167
Figura B.3–Painel: componentes de deslocamento no sistema de coordenadas global	168
Figura B.4–Equilíbrio de forças no painel	169
Figura B.5–Painel linear: deslocamentos	172
Figura B.6–Geometria de um painel retangular	176
Figura B.7–Painel: modelo não-linear com esforços normais	177
Figura B.8–Modos independentes do campo de tensões do painel	178
Figura B.9–Modos de deformação do painel em concordância com os modos de tensão	179
Figura B.10–Painel não-linear	179
Figura B.11–Pontos de integração do painel e distribuição linear de tensões em x e y	181
Figura B.12–Painel não-linear: tensões e forças nodais	186
Figura B.13–Dimensões relativas entre o painel e as bielas adjacentes	186

Lista de tabelas

Tabela 2.1 – Tensão de cisalhamento na fissura condicionada às tensões nas armaduras	49
Tabela 4.1 – Resultados obtidos em painéis carregados por cisalhamento puro	91
Tabela 4.2 – Resultados obtidos para painéis submetidos a solicitações combinadas .	94
Tabela 5.1 – Resultados não-lineares do <i>SPMTool</i>	110
Tabela 5.2 – Geometria e armaduras dos modelos biela-painel analisados no <i>SPMTool</i>	114
Tabela 6.1 – Propriedades dos materiais utilizados nos dimensionamentos	126
Tabela 6.2 – Viga-parede contínua: dimensionamento das armaduras das bielas . . .	129
Tabela 6.3 – Viga-parede contínua: processo iterativo de dimensionamento do painel P2	129
Tabela 6.4 – Viga-parede contínua: dimensionamento das armaduras dos painéis . .	130
Tabela 6.5 – Viga-parede contínua: verificação das tensões no concreto	131
Tabela 6.6 – Viga-parede contínua: comparação entre os modelos não-lineares	134
Tabela 6.7 – Viga-parede com aberturas: dimensionamento das armaduras das bielas	137
Tabela 6.8 – Viga-parede com aberturas: processo iterativo de dimensionamento do painel P2	137
Tabela 6.9 – Viga-parede com aberturas: dimensionamento das armaduras dos painéis	138
Tabela 6.10–Viga-parede com aberturas: verificação das tensões no concreto	138
Tabela 6.11–Viga-parede com aberturas: comparação entre os modelos não-lineares	141
Tabela 6.12–Consolo curto: dimensionamento das armaduras das bielas	143
Tabela 6.13–Consolo curto: processo iterativo de dimensionamento dos painéis . . .	144
Tabela 6.14–Consolo curto: dimensionamento das armaduras dos painéis	144
Tabela 6.15–Consolo curto: verificação das tensões no concreto	144
Tabela 6.16–Consolo curto: comparação entre os modelos não-lineares	146

Lista de Abreviações e Siglas

<i>RCMembrane</i>	<i>Reinforced Concrete Membrane Library</i>
<i>SPMTool</i>	<i>Stringer-Panel Modelling Tool</i>
ABNT	Associação Brasileira de Normas Técnicas
CF	Combinação Frequente
CQP	Combinação Quase-Permanente
ELS	Estado Limite de Serviço
ELS-DEF	Estado Limite de Deformações Excessivas
ELS-W	Estado Limite de Abertura de Fissuras
ELU	Estado Limite Último
MB	Método das Bielas
MBP	Método Biela-Painel
MCTP	Modelo de Campo de Tensão Perturbado
MEF	Método dos Elementos Finitos
MMA	Modelo de Membrana Atenuado
MPT	Modelo Plástico de Treliça
MTA	Modelo de Treliça Atenuado
MTCM	Modelo de Treliça com Compatibilidade de Mohr
TMCC	Teoria Modificada do Campo de Compressão

Lista de símbolos

Nomenclatura Básica

A	Área
E	Módulo de elasticidade
G	Módulo de elasticidade transversal
N	Força normal
V	Força cortante
e	Deformação generalizada
β	Tensão generalizada
ε	Deformação normal
$\varepsilon_{1;2}$	Deformações principais
γ	Deformação de cisalhamento
σ	Tensão normal
$\sigma_{1;2}$	Tensões principais
τ	Tensão cisalhante
$\theta_{1;2}$	Ângulo do estado principal de tensão/deformação

Nomenclatura Específica

A_c	Área de concreto
A_s	Área de aço
E_c	Módulo de elasticidade inicial do concreto
E_s	Módulo de elasticidade do aço
G_c	Módulo de elasticidade transversal do concreto
G_f	Energia de fraturamento do concreto
f'_c	Resistência do concreto à compressão
f_{cd}	Resistência de cálculo concreto à compressão
f_{ck}	Resistência característica do concreto à compressão
f_{cr}	Tensão crítica de fissuração do concreto
f_p	Tensão de pico do concreto à compressão

f'_t	Resistência do concreto à tração
f_y	Tensão de escoamento do aço
f_{yd}	Tensão de escoamento de cálculo do aço
f_{yk}	Tensão de escoamento característica do aço
s_m	Espaçamento médio entre fissuras
w	Abertura média de fissura
δ_s	Deslizamento da fissura
ε'_c	Deformação de pico do concreto
ε_{cr}	Deformação de fissuração do concreto
ε_p	Deformação de pico do concreto à compressão
$\varepsilon_{c1;2}$	Deformações principais no concreto
ε_{su}	Deformação última do aço
ε_y	Deformação de escoamento do aço
ϕ_a	Diâmetro máximo do agregado
ϕ_s	Diâmetro da barra de aço
ρ_s	Taxa de armadura
$\sigma_{c1;2}$	Tensões principais no concreto
σ_{ci}	Tensão normal de compressão no concreto, na fissura
σ_{cr}	Tensão de fissuração do concreto
σ_s	Tensão no aço
σ_c	Tensão no concreto
σ_{scr}	Tensão local no aço, na fissura
τ_{ci}	Tensão de cisalhamento no concreto, na fissura
τ_{cimax}	Tensão de cisalhamento máxima no concreto, na fissura
$\theta_{c1;2}$	Ângulo do estado principal de tensão/deformação no concreto

Matrizes e Vetores

$[B]$	Matriz de forma
$[D]$	Matriz de rigidez secante
$[D_c]$	Matriz de rigidez secante do concreto
$[D_s]$	Matriz de rigidez secante das armaduras
$[F]$	Matriz de flexibilidade
$[K]$	Matriz de rigidez
$[T]$	Matriz de transformação
$\{\beta\}$	Vetor de tensões generalizadas

$\{ e \}$	Vetor de deformações generalizadas
$\{ \varepsilon \}$	Vetor de deformações
$\{ \varepsilon_c \}$	Vetor de deformações no concreto
$\{ \varepsilon_s \}$	Vetor de deformações nas armaduras
$\{ \varepsilon_{sl} \}$	Vetor de deformações de deslizamento da fissura
$\{ f \}$	Vetor de forças
$\{ \sigma \}$	Vetor de tensões
$\{ \sigma_0 \}$	Vetor de pseudo-pré-tensão
$\{ \sigma_c \}$	Vetor de tensões no concreto
$\{ \sigma_s \}$	Vetor de tensões nas armaduras
$\{ u \}$	Vetor de deslocamentos

Sumário

1	INTRODUÇÃO	20
1.1	Justificativas	23
1.2	Objetivos	23
1.2.1	Objetivo Geral	23
1.2.2	Objetivos Específicos	24
1.3	Organização do trabalho	24
2	ANÁLISE DE ELEMENTOS DE MEMBRANA EM CONCRETO ARMADO	26
2.1	Histórico	26
2.2	Condições de equilíbrio	29
2.3	Relações de compatibilidade	31
2.4	Critério de fissuração	32
2.5	Matriz de rigidez secante dos materiais	33
2.5.1	Concreto	34
2.5.2	Armaduras	35
2.6	Modelo Plástico de Treliza (MPT)	35
2.6.1	Caso de cisalhamento puro	36
2.6.1.1	Condição balanceada	37
2.6.1.2	Elementos subarmados	38
2.6.1.3	Elementos superarmados	38
2.6.1.4	Casos de condição balanceada	38
2.6.1.5	Armadura mínima	39
2.6.2	Caso genérico de carregamento	39
2.7	Teoria Modificada do Campo de Compressão (TMCC)	40
2.7.1	Modelos constitutivos	42
2.7.1.1	Concreto comprimido	42
2.7.1.2	Concreto tracionado	42
2.7.2	Aço	43
2.7.3	Transmissão de esforços através da fissura	44
2.7.4	Verificação das tensões na fissura	47
2.8	Modelo de Campo de Tensão Perturbado (MCTP)	49
2.8.1	Modelos constitutivos	51
2.8.1.1	Concreto comprimido	51
2.8.1.2	Concreto tracionado	53

2.8.2	Limitação da tensão principal de tração	54
2.8.3	Tensões locais nas fissuras	54
2.8.4	Deslizamento da fissura	55
2.9	Modelo de Membrana Atenuado (MMA)	57
2.9.1	Consideração do Efeito de Poisson	58
2.9.1.1	Concreto	58
2.9.1.2	Armaduras	59
2.9.1.3	Cálculo dos coeficientes	59
2.9.2	Modelos constitutivos	60
2.9.2.1	Concreto comprimido	60
2.9.2.2	Concreto tracionado	61
2.9.2.3	Cisalhamento	61
3	O MÉTODO BIELA-PAINEL (MBP)	63
3.1	Histórico	63
3.2	Descrição Geral do MBP	66
3.3	Modelagem utilizando o MEF	71
3.3.1	Vetor de forças	73
3.3.2	Vetor de deslocamentos	73
3.3.3	Matriz de rigidez global	74
3.3.4	Condições de contorno	75
3.3.5	Solução do sistema	76
3.3.6	Solução de equações não-lineares	76
3.3.6.1	Método de Newton-Raphson	77
3.3.6.2	Método de Newton-Raphson Modificado	78
3.3.6.3	Método Secante Incremental	79
3.3.6.4	Método de Força Incremental	80
3.3.6.5	Método do Comprimento de Arco	81
4	DESENVOLVIMENTO DE UMA BIBLIOTECA PARA ANÁLISE DE ELEMENTOS DE MEMBRANA	84
4.1	Processo de análise	85
4.2	Proposta de atualização do fator C_s para o MCTP	87
4.3	Validação	87
5	O PROGRAMA <i>SPMTOOL</i>	96
5.1	Pré-processamento	97
5.1.1	Definição do modelo	98
5.1.2	Definição dos materiais	101
5.1.3	Parâmetros de análise não-linear	102

5.2	Pós-processamento	102
5.3	Validação dos modelos não-lineares	105
6	APLICAÇÃO DO MBP AO DIMENSIONAMENTO DE ESTRUTURAS DE CONCRETO	118
6.1	Limitação da tensão de compressão no concreto	118
6.2	Consideração da contribuição do concreto ao cisalhamento	119
6.3	Dimensionamento das bielas	121
6.3.1	Força de tração	121
6.3.2	Força de compressão	122
6.4	Dimensionamento dos painéis	123
6.5	Exemplos de aplicação	125
6.5.1	Propriedades dos materiais	126
6.5.2	Viga-parede contínua	127
6.5.3	Viga-parede com aberturas	135
6.5.4	Consolo curto	142
6.5.5	Resumo das análises não-lineares	147
7	CONCLUSÃO	148
7.1	Sugestões para trabalhos futuros	150
	REFERÊNCIAS	151
	APÊNDICE A – FORMULAÇÃO DAS BIELAS	158
A.1	Geometria	158
A.2	Sistema de referência local	159
A.3	Funções de forma quadráticas	160
A.4	Matriz de rigidez	162
A.5	Determinação das forças nodais	163
A.6	Modelo linear	165
	APÊNDICE B – FORMULAÇÃO DOS PAINÉIS	166
B.1	Modelo linear	166
B.1.1	Relações	166
B.1.2	Transformação de coordenadas	168
B.1.3	Equilíbrio	169
B.1.4	Cinemática	172
B.1.5	Matriz de rigidez	175
B.1.6	Painel retangular	176
B.2	Modelo não-linear	177

B.2.1	Relações	178
B.2.2	Deformações generalizadas	180
B.2.3	Deformações nos pontos de integração	180
B.2.4	Tensões nos pontos de integração	182
B.2.5	Tensões Generalizadas	183
B.2.6	Forças nodais	184

APÊNDICE C – CÓDIGO-FONTE DAS BIBLIOTECAS DESENVOLVIDAS 188

C.1	<i>Material</i>	188
C.1.1	Parâmetros dos materiais	188
C.1.2	Materiais uniaxiais	208
C.1.3	Materiais biaxiais	226
C.2	<i>RCMembrane</i>	256
C.2.1	Classe <i>Membrane</i> e derivadas	256
C.2.2	Classe <i>MembraneSolver</i>	271
C.3	<i>FEMAnalysis</i>	283
C.3.1	Interfaces	283
C.3.2	Entrada de dados	287
C.3.3	Análise	289
C.3.4	Saída de dados	310
C.3.5	Extensões	313
C.4	<i>SPMElements</i>	317
C.4.1	Classe base	317
C.4.2	Elemento nó	321
C.4.3	Elemento biela	325
C.4.4	Elemento painel	340

1 Introdução

Entende-se por concreto armado o material formado por concreto e barras de aço dispostas em seu interior, sendo que a vinculação entre esses dois materiais se dá por aderência entre as armaduras e o material cimentício por rugosidade. Esses dois materiais trabalham solidariamente: o aço é necessário nas regiões onde existem tensões de tração e o concreto onde existe compressão. Como este último material possui baixa resistência à tração, não consegue acompanhar as grandes deformações do aço nas regiões tracionadas. Consequentemente, existirá o processo de fissuração do concreto, fazendo com que os esforços de tração sejam resistidos exclusivamente pelas armaduras (LEONHARDT; MÖNNIG, 1988).

A primeira teoria de dimensionamento de vigas de concreto armado data do final do século XIX, quando Wilhelm Ritter propôs o modelo clássico de treliça, que posteriormente foi melhorado por Emil Mörsch. Esse modelo considera que a viga fissurada de concreto armado atua como uma treliça, constituída por banzos longitudinais paralelos, escoras inclinadas de concreto e tirantes transversais. Com aplicação de carregamento, as escoras diagonais e o banzo superior se comprimem, e os tirantes transversais e banzo inferior se tracionam (VECCHIO; COLLINS, 1988). Esta analogia se tornou o primeiro método de cálculo para dimensionamento de vigas de concreto armado submetidas à flexão e força cortante, sendo aplicada até hoje.

Mais tarde, a teoria da flexão da Resistência dos Materiais foi adaptada para a análise de seções de concreto armado (HOGNESTAD; HANSON; MCHENRY, 1955; HOGNESTAD, 1956). Esta teoria considera a Hipótese de Bernoulli, que admite que as seções permanecem planas após a deformação, ou seja, considera uma distribuição linear de deformações ao longo da seção transversal e despreza as deformações devidas às forças cortantes, devido ao fato destas serem muito pequenas. Desse modo, o dimensionamento pode ser feito considerando o equilíbrio na seção e as relações constitutivas simplificadas dos materiais, para uma grande variedade de carregamentos e geometrias de seção transversal (MITCHELL; COOK, 1991).

Segundo Souza (2016), embora o modelo de treliça fosse de grande significância ao entendimento do concreto armado, ele não era capaz de explicar os mecanismos que ocorrem de forma complementar ao comportamento de treliça, como a transferência de esforços de cisalhamento por meio do engrenamento dos agregados e o efeito de pino das armaduras. Além disso, os modelos até então não consideravam a perda de resistência à compressão do concreto devido à influência de tensões transversais de tração, efeito esse conhecido como abrandamento (*softening*).

Com isso, as estruturas de concreto passaram a ser analisadas não mais como uma treliça, mas sim como uma junção de elementos de membrana, para os quais as condições de equilíbrio e compatibilidade podem ser adaptadas da Resistência dos Materiais. A influência da transmissão de tensões por meio do engrenamento dos agregados e o efeito de abrandamento do concreto foram considerados, posteriormente, por Vecchio e Collins (1986), no desenvolvimento da Teoria Modificada do Campo de Compressão (TMCC) (*Modified Compression Field Theory*), e por Hsu (1988) através do Modelo de Treliça Atenuado (MTA) (*Softened Truss Model*).

Mais tarde, os modelos citados foram aprimorados pelo Modelo de Campo de Tensão Perturbado (MCTP) (*Disturbed Stress Field Model*) (VECCHIO, 2000), para a consideração das deformações por deslizamento das fissuras (*crack slip*), e pelo Modelo de Membrana Atenuado (MMA) (*Softened Membrane Model*) (HSU; ZHU, 2002), com a consideração do efeito de Poisson no estado de deformação do elemento.

Modelos de treliça ainda são abordagens simples e úteis ao dimensionamento de estruturas de concreto armado até hoje. Contudo, apesar de a hipótese das seções planas e a analogia de treliça simplificarem o dimensionamento de elementos de concreto armado, elas não podem ser consideradas para qualquer parte de uma estrutura. Regiões localizadas próximas a pontos de aplicação de cargas concentradas ou pontos de mudança brusca da seção transversal tendem a gerar um campo de tensões perturbado que, de acordo com o Princípio de Saint-Venant, tende a se normalizar numa distância que é, no mínimo, igual a maior dimensão da seção transversal (HIBBELER, 2010). Nessas regiões, as deformações devidas às forças cortantes são significativas, não sendo adequado assumir que as seções permaneçam planas e que as tensões cisalhantes sejam uniformes (MITCHELL; COOK, 1991).

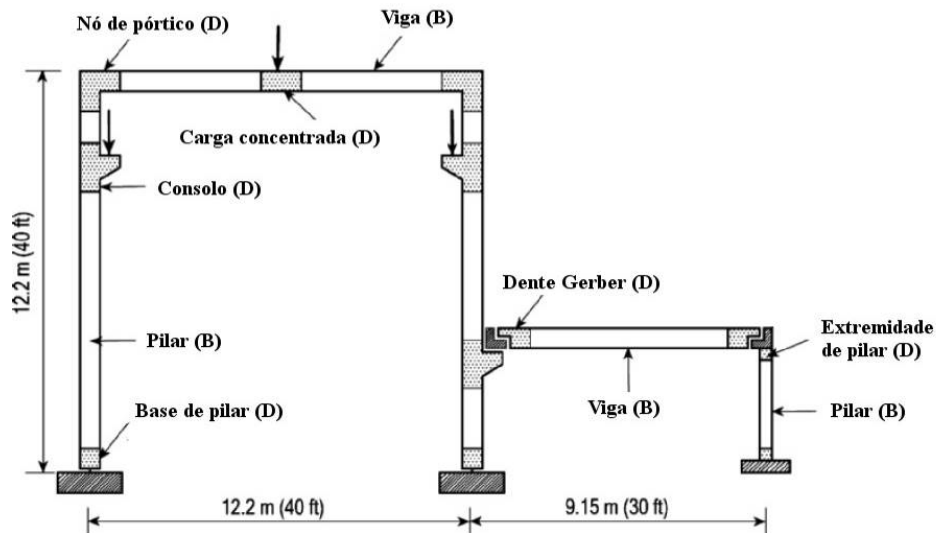
As regiões para as quais é possível aplicar a Hipótese de Bernoulli são denominadas simplesmente por regiões *B* (*Bernoulli* ou *Beam*), enquanto as regiões que não seguem essa teoria são chamadas de regiões *D* (*Disturbed* ou *Descontinuity*). A Figura 1.1 apresenta exemplos de regiões *B* e *D* numa estrutura convencional de concreto armado.

O dimensionamento racional das regiões *D* foi possível através de uma generalização da analogia de treliça proposta por Schlaich, Schäfer e Jennewein (1987). Denominado Método das Bielas (MB) ou Método de Escoras e Tirantes (*Strut and Tie Method*), o modelo divide a estrutura em escoras de concreto, tirantes de aço e regiões nodais, podendo ser aplicado às regiões *D* através de modelos definidos pelo próprio engenheiro estrutural. De modo a garantir a segurança do modelo, as tensões nas escoras e nas regiões nodais devem ser verificadas e não devem ultrapassar valores baseados nos limites plásticos do concreto.

Aliado ao MB, o Método dos Elementos Finitos (MEF) (*Finite Element Method*) tem grande potencial para análise de estruturas constituídas por regiões *D*. O MEF nada

mais é do que uma solução aproximada ao problema real, por meio da divisão do elemento estrutural em vários elementos interconectados, chamados elementos finitos (RAO, 2011). A partir de análises elásticas, é possível obter um entendimento do fluxo de tensões dentro da estrutura e, com isso, propor a posição das escoras e dos tirantes (SOUZA, 2004). Além disso, o MEF possui grande aplicabilidade em análises não-lineares de estruturas em concreto armado.

Figura 1.1 – Definição das regiões *B* e *D* numa estrutura de concreto armado



Fonte: adaptado de Hsu e Mo (2010).

Paralelamente ao MB, foi desenvolvido um método intermediário entre as duas abordagens descritas. Inicialmente utilizado na engenharia mecânica para a modelagem da fuselagem de aeronaves (ARGYRIS; KELSEY, 1960), o Método Biela-Painel (MBP) (*Stringer-Panel Method*) foi aplicado na engenharia civil primeiramente em paredes de concreto armado por Kaern (1979).

O modelo divide a estrutura em dois elementos distintos: as bielas, que trabalham em regime axial, e os painéis, que trabalham em regime de membrana de cisalhamento puro. Assim como para o MB, as tensões no concreto devem ser verificadas de acordo com a Teoria da Plasticidade, sendo que a resistência é normalmente reduzida por um fator de eficiência (SIMONE, 1998; NIELSEN; HOANG, 2011).

Os maiores avanços no desenvolvimento do MBP foram atingidos na década de 1990, a partir de pesquisas desenvolvidas no Politécnico de Milão e na Universidade Tecnológica de Delft, sendo que estas focaram na análise matricial do modelo (HOOGENBOOM, 1993; SIMONE, 1998). Nesse período foi desenvolvido o programa *SPanCAD* (HOOGENBOOM, 1998), uma ferramenta que funcionava como uma extensão do *AutoCAD* (v. *Release* 14, 1997) e era capaz de efetuar análises elásticas e não-lineares de modelos biela-painel, por meio da aplicação da TMCC.

O MBP já foi aplicado com sucesso em análises de vigas (WANG; HOOGENBOOM, 2004; SILVA, 2004), vigas-parede (HOOGENBOOM, 1998; MELLO, 2015), consolos (TARQUINI; SGAMBI, 2003) e paredes estruturais (HAUKSDÓTTIR, 2007; REFER, 2012). Além disso, através dos trabalhos de Souza (2012), Mello (2015) e Mello e Souza (2016), mostrou-se que o método tem fácil aplicação por abordagens manuais, sendo que, em estruturas mais simples, os esforços são determinados por equações de equilíbrio.

1.1 Justificativas

Na rotina de um escritório de cálculo estrutural, muitas vezes é necessário dar atenção especial a certos elementos de uma estrutura, principalmente na existência de regiões sujeitas a descontinuidades. Atualmente, não existe uma ferramenta computacional simples para análise de elementos isolados de concreto armado, os quais, na maioria das vezes, são analisados por programas comerciais de elementos finitos, que demandam muito tempo e conhecimento do usuário.

O programa *SPanCAD*, infelizmente, teve o desenvolvimento descontinuado no início dos anos 2000 e se tornou obsoleto, já que o *AutoCAD Release 14* não é compatível com as versões mais recentes do *Microsoft Windows*. Dessa forma, modelos de membrana que surgiram desde então, como o MMA e o MCTP, não foram estudados em aplicações do MBP.

Além disso, observa-se o MBP não é um método muito difundido ao redor do mundo, existindo apenas uma recomendação na norma dinamarquesa (DK NA:2013) e uma pequena menção no CEB-FIP Model Code 1990 (1993). No caso brasileiro, a ABNT NBR 6118:2014, que regulamenta o projeto de estruturas de concreto, recomenda, em sua seção 22, que os elementos especiais, assim denominadas as estruturas sujeitas à descontinuidades, sejam analisadas por aplicação do MB ou do MEF.

1.2 Objetivos

1.2.1 Objetivo Geral

Este trabalho tem como principal objetivo desenvolver o programa computacional de código aberto *SPMTool*, para análises elásticas e não-lineares pelo MBP. Esta ferramenta foi pensada como uma evolução do *SPanCAD* e, de modo a avançar no conhecimento científico, serão aplicados modelos de elementos de membrana mais recentes, como MCTP e o MMA, além da já difundida TMCC. Além disso, busca-se que o *SPMTool* esteja

disponível como uma alternativa simples no dia-a-dia dos escritórios de engenharia, de modo que se obtenham respostas rápidas para o dimensionamento e verificação de elementos especiais de concreto armado.

1.2.2 Objetivos Específicos

Este trabalho tem como objetivos específicos:

- apresentar o roteiro de análise e dimensionamento de estruturas de concreto armado utilizando modelos biela-painel;
- discutir a aplicação de modelos de elementos de membrana para análise de tensões e deformações;
- desenvolver uma biblioteca de código aberto para modelos constitutivos para concreto e aço, e de análise de elementos de membrana utilizando a TMCC, o MCTP e o MMA;
- aplicar os diferentes modelos de membrana já citados à formulação não-linear do MBP;
- desenvolver o programa *SPMTool*, como um complemento para as versões mais recentes do *AutoCAD*, para análise de elementos estruturais de concreto armado pelo MBP;
- validar os resultados numéricos do *SPMTool* a partir de resultados experimentais disponíveis na literatura, comparando com os resultados do *SPanCAD*;
- criar um processo iterativo de dimensionamento das armaduras dos painéis, considerando a contribuição do concreto pela formulação da TMCC;
- apresentar exemplos de aplicação do *SPMTool* no dimensionamento e verificação de elementos especiais de concreto armado, comparando com os resultados não lineares do software comercial *ATENA 2D* (v. 5.7.0, 2019).

1.3 Organização do trabalho

A primeira parte da pesquisa apresenta um estudo bibliográfico dos conceitos necessários ao entendimento do assunto. O Capítulo 2 se trata de um estudo dos elementos de membrana em concreto armado, onde são apresentadas as condições de equilíbrio e compatibilidade, e os modelos constitutivos utilizados neste trabalho. No Capítulo 3 são apresentados a origem e o conceito do MBP, sua aplicação nas estruturas de concreto armado e sua utilização em conjunto com o MEF. A formulação específica dos elementos finitos é apresentada nos Apêndices A e B.

Na segunda parte do trabalho, são apresentadas as aplicações desenvolvidas. O Capítulo 4 mostra a implementação dos modelos de análise de elementos de membrana estudados, com resultados validados a partir de ensaios experimentais de painéis existentes na literatura. O Capítulo 5 apresenta o programa *SPMTool* e descreve objetivamente como montar os modelos, ajustar as configurações e executar os diferentes tipos de análise. São também apresentados resultados dos modelos numéricos não-lineares, validados a partir de resultados experimentais obtidos previamente por outros pesquisadores. O código-fonte das bibliotecas desenvolvidas é apresentado no Apêndice C.

O Capítulo 6 mostra o processo de dimensionamento dos elementos do modelo biela-painel, juntamente com exemplos de aplicação do *SPMTool* no dimensionamento e verificação de elementos especiais de concreto armado. Estes elementos também são simulados e os resultados são comparados com o software comercial *ATENA 2D* (v. 5.7.0, 2019).

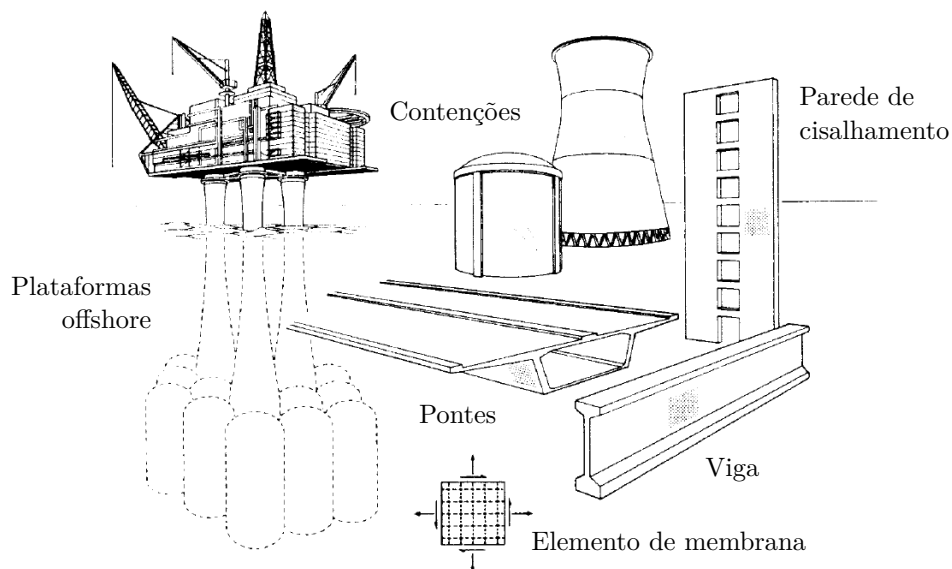
Finalmente, o Capítulo 7 apresenta um resumo dos resultados do trabalho e as conclusões obtidas. São também propostas sugestões para pesquisadores que, futuramente, queiram desenvolver trabalhos nesta linha de pesquisa.

2 Análise de elementos de membrana em concreto armado

2.1 Histórico

Por volta da década de 1960, as estruturas começaram a ser modeladas como uma junção de elementos bidimensionais, comumente denominados de elementos de membrana, para estruturas submetidas a força cortante e torção. De acordo com Vecchio e Collins (1986), elementos de membrana devem ser utilizados na modelagem de estruturas para as quais o carregamento é transportado através de tensões no plano. Como exemplo, podem ser citadas as vigas, paredes e pontes conforme ilustrado na Figura 2.1.

Figura 2.1 – Estruturas idealizadas como um conjunto de elementos de membrana



Fonte: adaptado de Vecchio e Collins (1986).

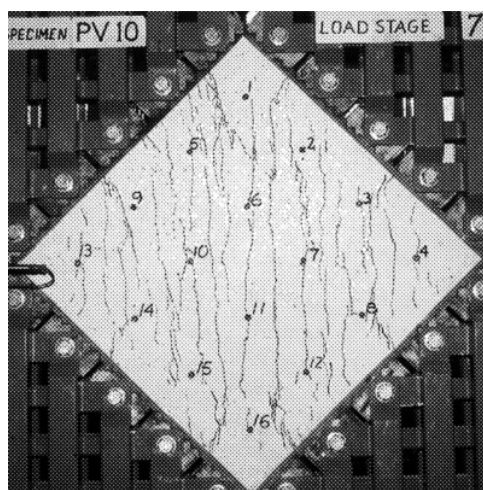
Esses elementos foram idealizados de modo que os três princípios da mecânica dos materiais de Navier (equilíbrio de tensões, compatibilidade de deformações e Lei de Hooke) seriam válidos (HSU, 1998b). A partir destes três princípios, surgiram o Modelo Plástico de Treliça (MPT) e o Modelo de Treliça com Compatibilidade de Mohr (MTCM). Segundo Hsu (1998b), apesar de não utilizar elementos lineares, esses modelos ainda são chamados de “modelos de treliça” por considerarem que o coeficiente de Poisson seja nulo, ou seja, as tensões principais somente causam deformações em suas direções de aplicação.

O MPT, originalmente *Equilibrium Plasticity Truss Model*, foi primeiramente aplicado por Nielsen (1967 apud HSU, 1998b) Lampert e Thürlimann (1968), sendo que esses autores chegaram nas equações de equilíbrio para elementos submetidos a tensões normais e cisalhantes. Posteriormente, Elfgrén (1972) aprimorou o modelo existente, com a interação entre tensões normais e cisalhantes com momentos torçores.

O MTCM, ou *Mohr Compatibility Truss Model*, foi obtido por Baumann (1972 apud HSU, 1998b) e Collins e Mitchell (1980) a partir das equações de equilíbrio do MPT combinadas com equações de compatibilidade simplificadas a partir da Lei de Hooke. O nome desse modelo se deve ao fato de que as condições de deformações médias devem satisfazer o Círculo de Mohr (HSU, 1998a).

Embora os dois modelos pudessem servir de base para um dimensionamento seguro de estruturas de concreto, não era possível prever o comportamento estrutural até a ruína. Isso somente se tornou possível após a realização de programas experimentais ao redor do mundo (COLLINS; VECCHIO; MEHLHORN, 1985), que procuraram prever o comportamento de painéis de concreto armado submetidos a carregamentos nas quatro faces, que geravam um comportamento de cisalhamento puro, através do equipamento ilustrado na Figura 2.2.

Figura 2.2 – Equipamento de teste de painéis de concreto armado



Fonte: Vecchio et al. (2001).

Estes estudos experimentais serviram de base para a formulação da Teoria Modificada do Campo de Compressão (TMCC) (*Modified Compression Field Theory*) (VECCHIO; COLLINS, 1986), modelo que considera a contribuição do concreto a partir das tensões de tração nas regiões de concreto que se encontram entre fissuras e também a capacidade de transmissão de tensões cisalhantes através de fissuras por meio do engrenamento dos agregados. Além disso, o modelo trata o concreto fissurado como um novo material, com suas próprias relações tensão-deformação, sendo capaz de prever, através de relações

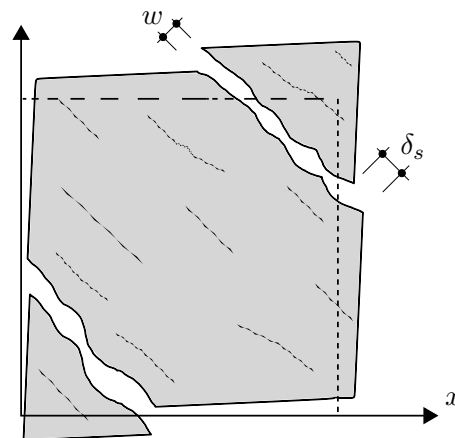
não-lineares, o comportamento estrutural desde o início do carregamento até a ruptura (SOUZA, 2010a).

Paralelamente à TMCC, um modelo bastante similar era desenvolvido por Hsu (1988), o Modelo de Treliça Atenuado (MTA) (*Softened Truss Model*), que considerava a redução de resistência à compressão do concreto através de um coeficiente de abrandamento. Mais tarde, esse modelo foi aprimorado para incorporar o efeito de Poisson no estado de deformação do elemento, sendo então denominado de Modelo de Membrana Atenuado (MMA) (*Softened Membrane Model*) (HSU; ZHU, 2002).

Deve-se observar que todos os modelos citados anteriormente se baseiam na hipótese de que os ângulos das deformações e das tensões principais são iguais, o que não é fundamentalmente correto. Após a fissuração, os planos principais tendem a se distanciar conforme o carregamento aumenta (SOUZA, 2016).

Este comportamento em parte se deve ao deslizamento da fissura (*crack slip*), fenômeno que ocorre com o aumento da deformação de cisalhamento e que faz com que surja um deslocamento δ_s , paralelo às interfaces das fissuras (Figura 2.3). Este efeito foi considerado por Vecchio (2000) no desenvolvimento do Modelo de Campo de Tensão Perturbado (MCTP) (*Disturbed Stress Field Model*), modelo considerado como um aprimoramento da TMCC.

Figura 2.3 – Deslizamento da fissura (*crack slip*)



Fonte: adaptado de Vecchio e Lai (2004).

Apesar da limitação citada, estudos apontam que, do ponto de vista prático, estas teorias propiciam boas previsões de capacidade de carga (VECCHIO; COLLINS, 1986; VECCHIO; COLLINS; ASPIOTIS, 1994; VECCHIO et al., 2001; HSU; ZHU, 2002).

2.2 Condições de equilíbrio

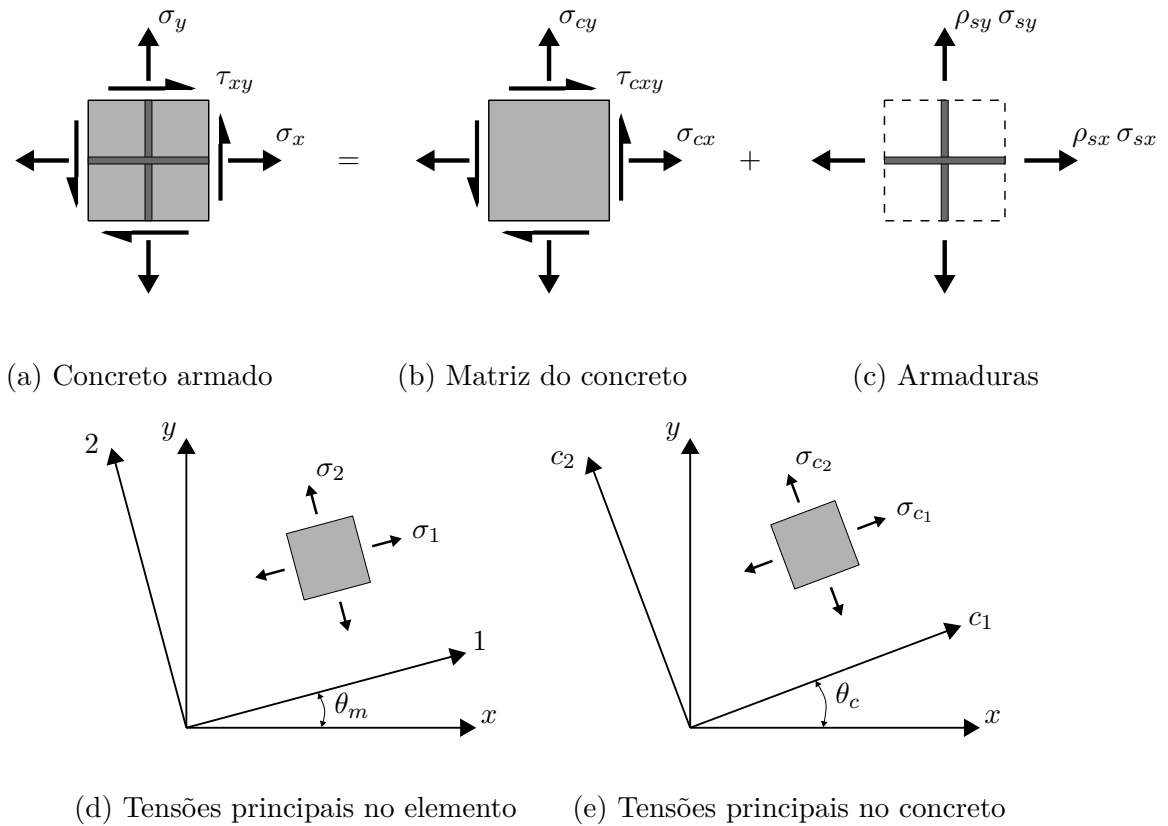
Considera-se um elemento de membrana submetido a um estado de tensão definido por tensões normais nas direções x e y (σ_x e σ_y) e uma tensão de cisalhamento (τ_{xy}), conforme ilustrado na Figura 2.4a. Considerando que as armaduras não ofereçam rigidez ao cisalhamento, pode-se dizer que essas tensões serão o somatório de uma parcela resistida pelo concreto (σ_{cx} , σ_{cy} e τ_{cxy}) e outra resistida pelas armaduras ($\rho_{sx} \sigma_{sx}$ e $\rho_{sy} \sigma_{sy}$), conforme as figuras 2.4b e 2.4c, ou seja:

$$\sigma_x = \sigma_{cx} + \rho_{sx} \sigma_{sx} \quad \sigma_y = \sigma_{cy} + \rho_{sy} \sigma_{sy} \quad \tau_{xy} = \tau_{cxy} \quad (2.1)$$

sendo ρ_{sx} e ρ_{sy} as taxas de armadura das direções x e y , respectivamente.

As tensões principais no elemento de concreto armado são representadas por σ_1 e σ_2 (Figura 2.4d), sendo que σ_1 corresponde à maior tensão e σ_2 , à menor. É visível que a orientação do estado principal das tensões médias não coincide com o estado principal do concreto puro (Figura 2.4e), ou seja: $\theta_m \neq \theta_c$.

Figura 2.4 – Elemento de membrana no estado plano de tensões



Fonte: adaptado de Simone (1998).

As relações entre as tensões no concreto podem ser escritas a partir do Círculo de Mohr, ilustrado na Figura 2.5. Assim, pode-se definir o estado principal de tensão como:

$$\sigma_1 = \sigma_m + R \quad \sigma_2 = \sigma_m - R \quad (2.2)$$

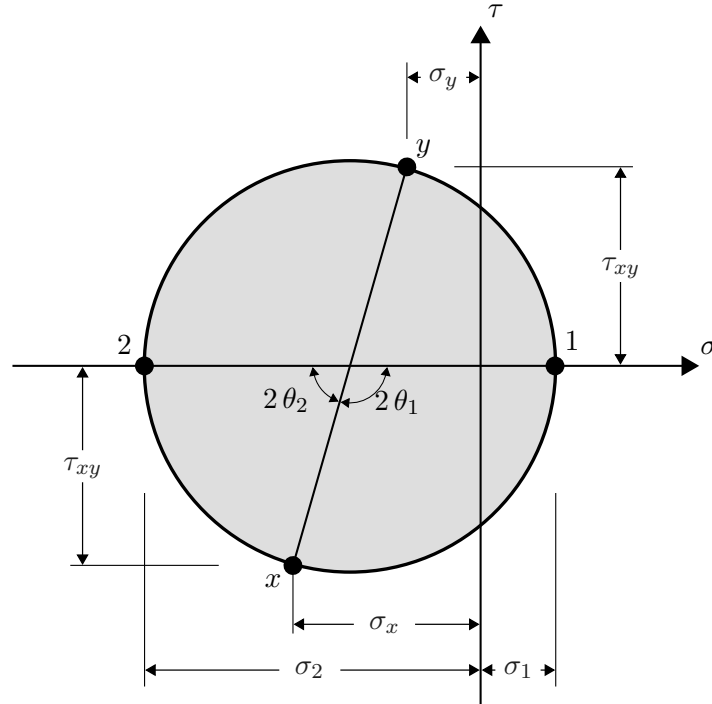
com:

$$\sigma_m = \frac{\sigma_x + \sigma_y}{2} \quad R = \sqrt{\left(\frac{\sigma_y - \sigma_x}{2}\right)^2 + \tau_{xy}^2}$$

Os ângulos dos planos principais são dados por:

$$\theta_2 = \frac{1}{2} \arcsen\left(\frac{\tau_{cxy}}{R}\right) \quad \theta_1 = \frac{\pi}{2} - \theta_2 \quad (2.3)$$

Figura 2.5 – Círculo de Mohr para tensões médias



Fonte: adaptado de Vecchio (2000)

Equações similares podem ser escritas para encontrar as tensões principais no concreto substituindo as tensões e direções correspondentes à matriz de concreto.

Os estados de tensão podem, ainda, ser relacionados matricialmente através da matriz de transformação:

$$\{\sigma\}_{xy} = [T]^T \{\sigma\}_{12} \quad (2.4)$$

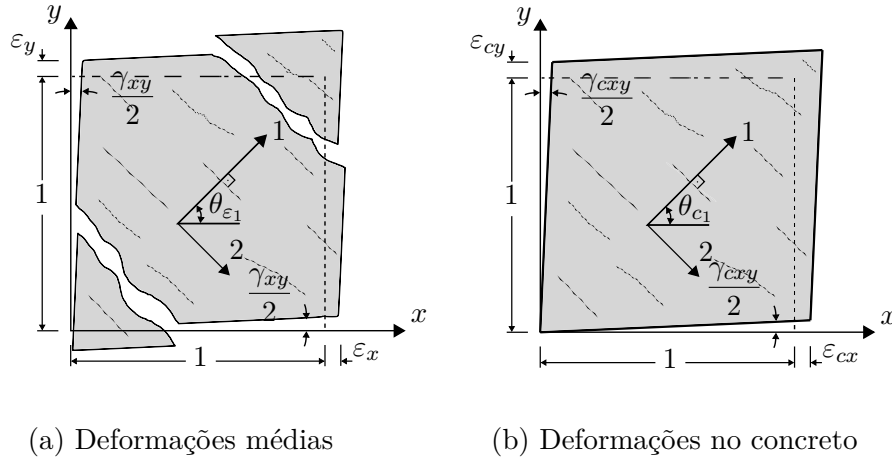
sendo $[T]^T$ a matriz transposta da matriz de transformação, definida por:

$$[T] = \begin{bmatrix} \cos^2 \theta_1 & \sin^2 \theta_1 & \cos \theta_1 \sin \theta_1 \\ \sin^2 \theta_1 & \cos^2 \theta_1 & -\cos \theta_1 \sin \theta_1 \\ -2 \cos \theta_1 \sin \theta_1 & 2 \cos \theta_1 \sin \theta_1 & \cos^2 \theta_1 - \sin^2 \theta_1 \end{bmatrix} \quad (2.5)$$

2.3 Relações de compatibilidade

As deformações médias no elemento de membrana são representadas por deformações normais nas direções x e y (ε_x e ε_y) e uma deformação por cisalhamento (γ_{xy}), visualizadas na Figura 2.6.

Figura 2.6 – Deformações num elemento de membrana



Fonte: adaptado de Vecchio (2000)

As armaduras estarão sujeitas a deformações normais de mesma intensidade que as deformações médias, isto é:

$$\varepsilon_{sx} = \varepsilon_x \qquad \varepsilon_{sy} = \varepsilon_y \qquad (2.6)$$

Já as deformações no concreto dependerão da consideração ou não do deslizamento da fissura (Figura 2.3), que introduzem as deformações adicionais ε_{sl} ao elemento de membrana. Assim, as deformações no concreto serão iguais à diferença entre as deformações médias e as deformações por deslizamento:

$$\varepsilon_{cx} = \varepsilon_x - \varepsilon_{slx} \qquad \varepsilon_{cy} = \varepsilon_y - \varepsilon_{sly} \qquad \gamma_{cxy} = \gamma_{xy} - \gamma_{slxy} \qquad (2.7)$$

Quando estas deformações não são consideradas, como é o caso das análises feitas a partir da TMCC, as deformações no concreto serão iguais às deformações nas armaduras, conforme será discutido posteriormente.

As deformações principais no elemento de membrana podem ser escritas a partir do Círculo de Mohr ilustrado na Figura 2.7:

$$\varepsilon_1 = \varepsilon_m + R \qquad \varepsilon_2 = \varepsilon_m - R \qquad (2.8)$$

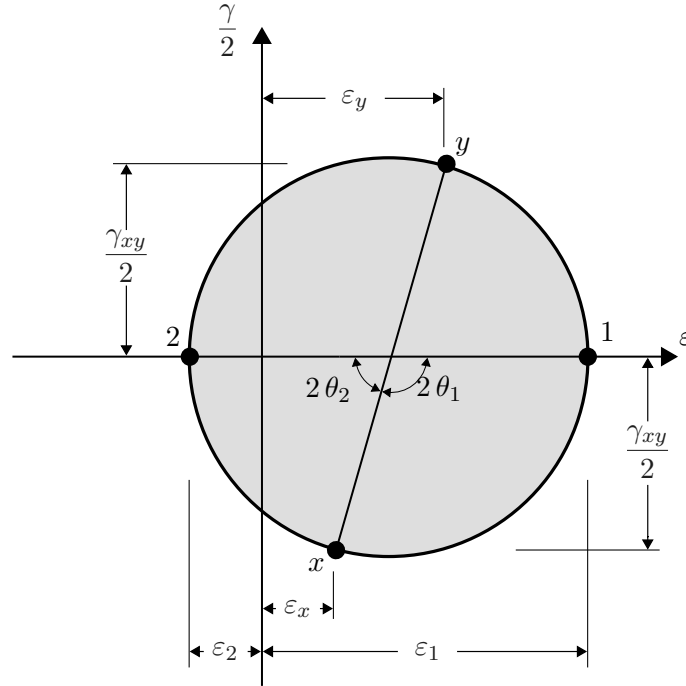
com:

$$\varepsilon_m = \frac{\varepsilon_x + \varepsilon_y}{2} \qquad R = \frac{1}{2} \sqrt{(\varepsilon_y - \varepsilon_x)^2 + \gamma_{xy}^2}$$

Os ângulos dos planos principais são dados por:

$$\theta_2 = \frac{1}{2} \arcsen \left(\frac{\gamma_{xy}}{2R} \right) \quad \theta_1 = \frac{\pi}{2} - \theta_2 \quad (2.9)$$

Figura 2.7 – Círculo de Mohr para deformações no concreto



Fonte: adaptado de Vecchio (2000)

Para o concreto, pode-se escrever as mesmas relações substituindo pelas correspondentes deformações específicas.

Pode-se ainda relacionar estados de deformação matricialmente por:

$$\{\varepsilon\}_{xy} = [T]^T \{\varepsilon\}_{12} \quad (2.10)$$

sendo $[T]^T$ a matriz transposta da matriz de transformação, definida na Equação (2.5).

É importante destacar que os ângulos de inclinação dos estados principais de tensão e de deformação são considerados iguais nos modelos que serão considerados a seguir.

2.4 Critério de fissuração

Segundo Wong, Vecchio e Trommels (2013), a tensão de fissuração do concreto (f_{cr}) depende de vários fatores, como tamanho do espécime e resistência à compressão e, em geral, diminui quando existe uma tensão de compressão transversal atuando. Dessa forma, o valor de f_{cr} somente será igual à resistência à tração do concreto (f'_t) para o estado uniaxial de tensões:

$$\sigma_{cr} = f'_t \quad (\text{Estado uniaxial}) \quad (2.11)$$

Para o estado biaxial, Gupta (1998), após analisar vários experimentos, concluiu que a resistência ao cisalhamento do concreto está correlacionada à tensão de fissuração assumida, sendo que essa tensão reduz à medida que a tensão transversal de compressão aumenta. O autor propõe a seguinte relação para a tensão crítica de fissuração:

$$\sigma_{cr} = f'_t \left(1 - \frac{\varepsilon_{c2}}{\varepsilon'_c} \right), \quad 0,25 f'_t \leq \sigma_{cr} \leq f'_t \quad (2.12)$$

sendo ε_{c2} a deformação principal de compressão no concreto e ε'_c a deformação de pico do concreto.

Portanto, pode-se observar que, para um elemento de membrana, a tensão de fissuração não dependerá apenas da resistência à tração do concreto, mas também da deformação transversal existente.

2.5 Matriz de rigidez secante dos materiais

Segundo Vecchio (1989), as formulações lineares de elementos finitos podem ser adaptadas para incorporar as relações tensão-deformação não-lineares dos materiais. Uma dessas formas é a utilização de uma matriz de rigidez secante para os materiais, de modo que o vetor de tensões $\{\sigma\}$ é relacionado ao vetor de deformações $\{\varepsilon\}$ por meio da matriz de rigidez secante $[D]$:

$$\{\sigma\} = [D] \{\varepsilon\} \quad (2.13)$$

sendo que os vetores de tensão e deformação são compostos por:

$$\{\sigma\} = \begin{Bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{Bmatrix} \quad \{\varepsilon\} = \begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{Bmatrix}$$

No caso de um material elástico isotrópico, a matriz $[D]$ é dada por:

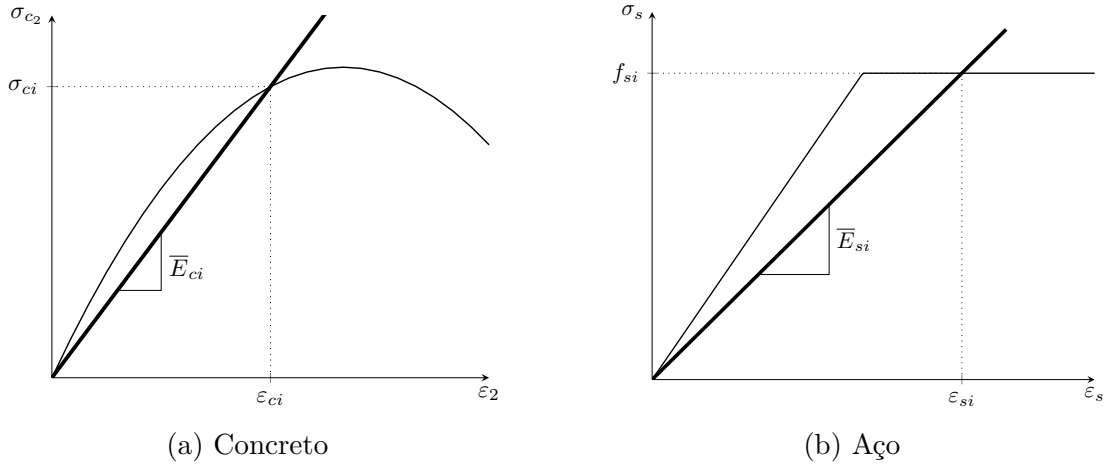
$$[D] = \frac{E}{(1 - \nu^2)} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{(1 - \nu)}{2} \end{bmatrix} \quad (2.14)$$

Para um elemento de concreto armado, $[D]$ será a soma da matriz de rigidez secante do concreto $[D_c]$ e das armaduras $[D_s]$:

$$[D] = [D_c] + [D_s] \quad (2.15)$$

As matrizes de rigidez secante são formadas a partir de um caso particular de tensão/deformação. Os módulos secantes são obtidos a partir da reta que liga a origem do diagrama ao estado em questão, conforme se visualiza na Figura 2.8.

Figura 2.8 – Definição do módulo secante



Fonte: adaptado de Vecchio (2001).

A seguir, serão apresentadas as formulações para obtenção das matrizes do concreto e das armaduras.

2.5.1 Concreto

Para tratar do comportamento não-linear do concreto, a matriz $[D_c]$ deve ser modificada para considerar apropriadamente o modelo constitutivo. O concreto fissurado é tratado como um material ortotrópico com as direções das tensões principais correspondentes às direções das deformações principais. Além disso, o efeito de Poisson pode ser desconsiderado (VECCHIO, 1990).

A matriz de rigidez secante $[D_c]'$, que relaciona o estado principal de tensão com o estado principal de deformação é definida por:

$$[D_c]' = \begin{bmatrix} \bar{E}_{c1} & 0 & 0 \\ 0 & \bar{E}_{c2} & 0 \\ 0 & 0 & \bar{G}_c \end{bmatrix} \quad (2.16)$$

sendo: $\bar{E}_{c1}$: módulo de elasticidade secante para o concreto tracionado;
 $\bar{E}_{c2}$: módulo de elasticidade secante para o concreto comprimido;
 $\bar{G}_c$: módulo transversal secante para o concreto.

Os módulos secantes são calculados para cada estado de tensão e deformação em particular:

$$\bar{E}_{c1} = \frac{\sigma_{c1}}{\varepsilon_{c1}} \quad \bar{E}_{c2} = \frac{\sigma_{c2}}{\varepsilon_{c2}} \quad \bar{G}_c = \frac{\bar{E}_{c1} \bar{E}_{c2}}{\bar{E}_{c1} + \bar{E}_{c2}} \quad (2.17)$$

A matriz de rigidez secante do concreto deve ser então transformada para o plano xy por:

$$[D_c] = [T_c]^T [D_c]' [T_c] \quad (2.18)$$

sendo $[T_c]$ a matriz de transformação para o concreto, que pode ser calculada pela Equação (2.5), com $\theta_1 = \theta_{c1}$.

2.5.2 Armaduras

A matriz de rigidez secante das armaduras $[D_s]$ deve ser calculada para cada componente de armadura (VECCHIO, 1990). Genericamente, a relação é expressa por:

$$[D_s] = \sum_{i=1}^n [T_s]_i^T [D_s]_i' [T_s]_i \quad (2.19)$$

A matriz $[D_s]_i'$ é a matriz de rigidez secante calculada para cada componente de armadura:

$$[D_s]_i' = \begin{bmatrix} \rho_{si} \bar{E}_{si} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (2.20)$$

sendo ρ_{si} a taxa de armadura na direção considerada e $\bar{E}_{si}$ o módulo de elasticidade secante, calculado para cada estado de tensão e deformação por:

$$\bar{E}_{si} = \frac{\sigma_{si}}{\varepsilon_{si}} \quad (2.21)$$

A matriz de transformação $[T_s]_i$ pode ser obtida a partir da Equação (2.5), substituindo θ_1 pelo ângulo de inclinação do componente de armadura considerado.

Para o caso usual de armaduras dispostas nas direções x e y , a matriz $[D_s]$, já considerando o somatório das duas direções, é dada por:

$$[D_s] = \begin{bmatrix} \rho_{sx} \bar{E}_{sx} & 0 & 0 \\ 0 & \rho_{sy} \bar{E}_{sy} & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (2.22)$$

2.6 Modelo Plástico de Treliça (MPT)

De acordo com Thürlimann (1979), a análise plástica oferece um método racional, simples, econômico e seguro para calcular a resistência estática de elementos estruturais em concreto armado. Para o concreto, normalmente se considera um fator de eficiência para a redução da sua resistência, que leva em conta a redistribuição de tensões a partir das primeiras fissuras até o colapso que, acompanhada pela evolução da fissuração, irá diminuir a resistência. Este tipo de análise é normalmente aplicada na teoria dos estados limites, principalmente ao Estado Limite Último (ELU).

As condições de compatibilidade e as relações constitutivas dos materiais não são consideradas no equacionamento do MPT, não sendo possível a obtenção das deformações de cisalhamento devidas à força cortante e, também, às deformações no aço e no concreto presentes num elemento de membrana, o que dificulta a representação do elemento em serviço. Apesar dessas deficiências, através do MPT é possível efetuar um dimensionamento racional de elementos de concreto submetidos a esforços de membrana (SOUZA, 2010b).

A análise despreza a resistência do concreto à tração, assumindo que somente as armaduras serão responsáveis por absorver esse esforço. Além disso, considera-se que as armaduras não absorvam as tensões cisalhantes, sendo o concreto responsável por absorvê-las, assim como as tensões de compressão (SOUZA, 2016).

Considerando as condições de equilíbrio (Equação 2.1) e as equações de transformação de tensões para o concreto (Equação 2.4), pode-se escrever as tensões no elemento em função das tensões principais no concreto e da parcela das armaduras:

$$\sigma_x = \sigma_{c1} \cos^2 \theta_{c1} + \sigma_{c2} \sin^2 \theta_{c1} + \rho_{sx} \sigma_{sx} \quad (2.23a)$$

$$\sigma_y = \sigma_{c1} \sin^2 \theta_{c1} + \sigma_{c2} \cos^2 \theta_{c1} + \rho_{sy} \sigma_{sy} \quad (2.23b)$$

$$\tau_{xy} = (\sigma_{c1} - \sigma_{c2}) \cos \theta_{c1} \sin \theta_{c1} \quad (2.23c)$$

Desprezando as tensões de tração no concreto, as equações acima são reescritas como:

$$\sigma_x = \sigma_{c2} \sin^2 \theta_{c1} + \rho_{sx} \sigma_{sx} \quad (2.24a)$$

$$\sigma_y = \sigma_{c2} \cos^2 \theta_{c1} + \rho_{sy} \sigma_{sy} \quad (2.24b)$$

$$\tau_{xy} = -\sigma_{c2} \cos \theta_{c1} \sin \theta_{c1} \quad (2.24c)$$

Reordenando (2.24c), encontra-se a tensão principal de compressão no concreto:

$$\sigma_{c2} = -\frac{\tau_{xy}}{\cos \theta_{c1} \sin \theta_{c1}} = -\tau_{xy} (\operatorname{tg} \theta_{c1} + \operatorname{cotg} \theta_{c1}) \quad (2.25)$$

Pode-se então reescrever (2.23) a partir da tensão de cisalhamento:

$$\sigma_x = \tau_{xy} \operatorname{tg} \theta_{c1} + \rho_{sx} \sigma_{sx} \quad (2.26a)$$

$$\sigma_y = \tau_{xy} \operatorname{cotg} \theta_{c1} + \rho_{sy} \sigma_{sy} \quad (2.26b)$$

A seguir serão definidos os casos de carregamento, nos quais a análise deve ser conduzida de maneira distinta.

2.6.1 Caso de cisalhamento puro

Neste caso, o elemento de membrana está submetido somente a uma tensão cisalhante, não existindo tensões normais nas direções x e y ($\sigma_x = \sigma_y = 0$). As equações de

equilíbrio podem ser escritas por:

$$\rho_{sx} \sigma_{sx} = \sigma_{c2} \sin^2 \theta_{c1} \quad (2.27a)$$

$$\rho_{sy} \sigma_{sy} = \sigma_{c2} \cos^2 \theta_{c1} \quad (2.27b)$$

com σ_{c2} calculado em (2.25).

Tais equações também podem ser expressas a partir da tensão de cisalhamento, substituindo (2.25) em (2.27):

$$\rho_{sx} \sigma_{sx} = \tau_{xy} \operatorname{tg} \theta_{c1} \quad (2.28a)$$

$$\rho_{sy} \sigma_{sy} = \tau_{xy} \operatorname{cotg} \theta_{c1} \quad (2.28b)$$

Adicionando as equações vistas em (2.27) e sabendo que: $\cos^2 \theta_{c1} + \sin^2 \theta_{c1} = 1$, tem-se:

$$\rho_{sx} \sigma_{sx} + \rho_{sy} \sigma_{sy} = \sigma_{c2} \quad (2.29)$$

Segundo Hsu e Mo (2010), para garantir que a ruptura dúctil do elemento de membrana aconteça neste caso de cisalhamento puro, os elementos podem ser classificados em dois tipos de acordo com a taxa de armadura: subarmados e superarmados. No primeiro caso, as armaduras de ambas as direções escoarão antes do esmagamento do concreto, satisfazendo a condição de ruptura do modelo plástico. No segundo, ocorre o contrário, situação esta que deve ser evitada por violar o critério de ruptura dúctil.

2.6.1.1 Condição balanceada

Esta condição é intermediária entre os casos sub e super armados. Nesse caso, uma das armaduras atinge o escoamento enquanto o concreto se rompe a uma tensão efetiva $\zeta f'_c$. Este coeficiente de abrandamento ζ varia de 0,2 a 0,6 (HSU; MO, 2010) e normalmente é encontrado nos códigos de projeto.

Considerando que ambas as armaduras escoarão enquanto o concreto atinge a tensão $\zeta f'_c$, isto é: $\sigma_{sx} = f_{yx}$ e $\sigma_{sy} = f_{yy}$, reescreve-se a Equação (2.29) como:

$$\rho_{sx} f_{yx} + \rho_{sy} f_{yy} = \zeta f'_c \quad (2.30)$$

sendo f_{yx} e f_{yy} as tensões de escoamento das armaduras nas direções x e y , respectivamente, e f'_c a resistência à compressão do concreto.

A condição balanceada pode ser escrita de forma adimensional como:

$$\omega_x + \omega_y = 1 \quad (2.31)$$

com:

$$\omega_x = \frac{\rho_{sx} f_{yx}}{\zeta f'_c} \quad \omega_y = \frac{\rho_{sy} f_{yy}}{\zeta f'_c}$$

2.6.1.2 Elementos subarmados

De acordo com Hsu e Mo (2010), este caso é definido quando as armaduras das duas direções escoam antes do esmagamento do concreto, isto é: $\omega_x + \omega_y < 1$. Substituindo as definições de $\sin^2 \theta_{c1}$ de (2.27a) e $\cos^2 \theta_{c1}$ de (2.27b) em (2.24c), é possível encontrar a tensão de cisalhamento (τ_{xy}) que provocará o escoamento das armaduras:

$$\tau_{xy} = \sqrt{(\rho_{sx} f_{yx})(\rho_{sy} f_{yy})} \quad (2.32a)$$

Dividindo (2.32a) pela tensão de pico efetiva do concreto ($\zeta f'_c$), obtém-se a taxa da tensão de cisalhamento:

$$\frac{\tau_{xy}}{\zeta f'_c} = \sqrt{\omega_x \omega_y} \quad (2.32b)$$

O ângulo θ_{c1} para este caso é encontrado dividindo (2.27a) por (2.27b):

$$\operatorname{tg} \theta_{c1} = \sqrt{\frac{\rho_{sx} f_{yx}}{\rho_{sy} f_{yy}}} = \sqrt{\frac{\omega_x}{\omega_y}} \quad (2.33)$$

2.6.1.3 Elementos superarmados

Segundo Hsu e Mo (2010), o elemento é superarmado quando: $\omega_x + \omega_y > 1$, sendo que o concreto é esmagado antes do escoamento das armaduras. Como este tipo de ruína viola a hipótese básica do MPT, o dimensionamento não pode ser feito para este caso.

2.6.1.4 Casos de condição balanceada

Ainda segundo Hsu e Mo (2010), podem existir três casos da condição balanceada ($\omega_x + \omega_y = 1$):

Caso 1 ($\omega_x = \omega_y$):

Neste caso, as armaduras em x e y escoarão simultaneamente ao esmagamento do concreto. Esta condição é dada por:

$$\omega_x = \omega_y = 0,5 \quad \frac{\tau_{xy}}{\zeta f'_c} = 0,5 \quad \theta_{c1} = 45^\circ \quad (2.34)$$

Caso 2 ($\omega_y < 0,5$):

Neste caso, a armadura em y já escoou e a armadura em x escoará simultaneamente ao esmagamento do concreto. A condição balanceada será definida por:

$$\omega_x = 1 - \omega_y \quad \frac{\tau_{xy}}{\zeta f'_c} = \sqrt{\omega_y (1 - \omega_y)} \quad \theta_{c1} > 45^\circ \quad (2.35)$$

Caso 3 ($\omega_x < 0,5$):

Neste caso, a armadura em x já escoou e a armadura em y escoará simultaneamente ao esmagamento do concreto. A condição balanceada será:

$$\omega_y = 1 - \omega_x \quad \frac{\tau_{xy}}{\zeta f'_c} = \sqrt{\omega_x (1 - \omega_x)} \quad \theta_{c1} < 45^\circ \quad (2.36)$$

2.6.1.5 Armadura mínima

Antes de o concreto fissurar, existirá uma tensão τ_{xy} que produzirá uma tensão principal de tração no concreto. Esta tensão será: $\sigma_{c1} = \tau_{xy}$, inclinada a 45° do eixo horizontal. No momento em que o concreto fissura, a tensão σ_{c1} atinge seu valor crítico (f_{cr}). Substituindo $\tau_{xy} = f_{cr}$ em (2.28):

$$\rho_{sx} \sigma_{sx} = f_{cr} \operatorname{tg} \theta_{c1} \quad \rho_{sy} \sigma_{sy} = f_{cr} \operatorname{cotg} \theta_{c1} \quad (2.37)$$

Somando as equações:

$$\rho_{sx} \sigma_{sx} + \rho_{sy} \sigma_{sy} = f_{cr} (\operatorname{tg} \theta_{c1} + \operatorname{cotg} \theta_{c1}) \quad (2.38)$$

A quantidade mínima de armadura para garantir que o aço não escoa no momento em que o concreto fissura é obtida fazendo: $f_{sx} = f_{sy} = f_y$. Nesta situação, o ângulo θ_{c1} será aproximadamente igual a 45° . Portanto, pode-se definir o critério de obtenção das taxas mínimas de armadura como:

$$\rho_{sx} + \rho_{sy} = \frac{2 f_{cr}}{f_y} \quad (2.39)$$

2.6.2 Caso genérico de carregamento

Considera-se um elemento de membrana sujeito ao estado de tensão ilustrado na Figura 2.4, com tensões normais não-nulas, isto é ($\sigma_x \neq 0$ e $\sigma_y \neq 0$). Segundo Souza (2016), podem existir quatro casos distintos de carregamento, conforme descrito a seguir.

Caso 1

Este caso é obtido quando as tensões em x e y , na (2.26) resultam em tração ($\sigma_x > 0$ e $\sigma_y > 0$), isto é: $\sigma_x \geq -\tau_{xy} \operatorname{tg} \theta_{c1}$ e $\sigma_y \geq -\tau_{xy} \operatorname{cotg} \theta_{c1}$.

O ângulo θ_{c1} pode ser calculado a partir da (2.33) e, considerando que as armaduras estejam escoando, as taxas podem ser obtidas reordenando (2.26):

$$\rho_{sx} f_{yx} = \sigma_x + \tau_{xy} \operatorname{tg} \theta_{c1} \quad \rho_{sy} f_{yy} = \sigma_y + \tau_{xy} \operatorname{cotg} \theta_{c1} \quad (2.40)$$

A tensão de compressão no concreto pode ser calculada a partir de (2.25).

Caso 2

Este caso é possível quando existe tensão de compressão em x e esta seja superior (em valor absoluto) à tensão cisalhante, isto é: $\sigma_x < 0$ e $|\sigma_x| > |\tau_{xy}|$. Caso a última afirmação seja falsa, a intensidade da tensão cisalhante fará com que seja necessária armadura na direção x (SOUZA, 2016).

Considerando que não exista armadura na direção x ($\rho_{sx} = 0$), pode-se obter ângulo θ_{c1} e a taxa ρ_{sy} a partir de (2.26a). A tensão no concreto é encontrada aplicando (2.25):

$$\operatorname{tg} \theta_{c1} = -\frac{\tau_{xy}}{\sigma_x} \quad \rho_{sy} f_{yy} = \sigma_y - \frac{\tau_{xy}^2}{\sigma_x} \quad \sigma_{c2} = -\left(\sigma_x + \frac{\tau_{xy}^2}{\sigma_x}\right) \quad (2.41)$$

Caso 3

Este caso é o inverso do Caso 2, sendo possível quando: $\sigma_y < 0$ e $|\sigma_y| > |\tau_{xy}|$. Como não existe armadura na direção y ($\rho_{sy} = 0$), têm-se:

$$\operatorname{tg} \theta_{c1} = -\frac{\sigma_y}{\tau_{xy}} \quad \rho_{sx} f_{yx} = \sigma_x - \frac{\tau_{xy}^2}{\sigma_y} \quad \sigma_{c2} = -\left(\sigma_y + \frac{\tau_{xy}^2}{\sigma_y}\right) \quad (2.42)$$

Caso 4

Este caso é uma combinação dos casos 2 e 3, isto é: $\sigma_x < 0$, $|\sigma_x| > |\tau_{xy}|$, $\sigma_y < 0$ e $|\sigma_y| > |\tau_{xy}|$.

Neste caso, não são necessárias armaduras nas duas direções, devendo apenas ser efetuada a verificação da tensão principal de compressão no concreto. Esta tensão pode ser encontrada a partir do Círculo de Mohr (Figura 2.5), aplicando a Equação (2.2).

Ainda é possível a existência de um caso específico onde uma das tensões normais é menor que a tensão cisalhante ($|\sigma_x| < |\tau_{xy}|$ ou $|\sigma_y| < |\tau_{xy}|$). Nesse caso, pode existir tração numa direção das armaduras.

2.7 Teoria Modificada do Campo de Compressão (TMCC)

A TMCC (VECCHIO; COLLINS, 1986) foi a primeira formulação a considerar o concreto fissurado como um novo material, com uma curva tensão-deformação definida após a análise de diversos resultados experimentais. Este comportamento considera que a tensão de pico do concreto (σ_{c2max}) é reduzida pela existência de uma deformação transversal de tração (ε_{c1}), conforme será visto em 2.7.1.

Segundo Souza (2016), as deformações impostas ao elemento de membrana consistem em deformações médias, já que consideram as deformações locais nas fissuras e entre as

fissuras. Consequentemente, as tensões também serão médias, ou seja, incluem as tensões nas fissuras e entre fissuras.

Para que estas relações possam ser utilizadas de forma eficiente e segura, as tensões devem ser corrigidas por um procedimento de verificação das fissuras (*"Crack Check"*) (BENTZ, 2000), que consiste em limitar a tensão principal de tração no elemento de membrana de modo que as armaduras resistam a esse esforço e o concreto seja capaz de transmiti-lo através do engrenamento dos agregados.

As hipóteses básicas das relações tensão-deformação da TMCC são relacionadas a seguir, de acordo com Souza (2016):

- (a) para cada estado de deformação existe um único estado de tensão, sendo que não são tratados casos de histórico de carregamento;
- (b) tensões e deformações podem ser consideradas em seus valores médios, quando atuantes em regiões suficientemente largas, contendo várias fissuras;
- (c) aderência perfeita entre concreto e armaduras nas fronteiras do elemento, não existindo deslizamento das armaduras;
- (d) armaduras distribuídas uniformemente ao longo do elemento;
- (e) tensões e deformações de tração são consideradas positivas, enquanto tensões e deformações de compressão, negativas.

A consideração da aderência perfeita entre o concreto e as armaduras e a desconsideração do deslizamento das fissuras leva à hipótese de que ambos os materiais se deformarão igualmente, ou seja, as deformações normais no concreto e no aço serão iguais à deformação média no elemento:

$$\varepsilon_x = \varepsilon_{cx} = \varepsilon_{sx} \qquad \varepsilon_y = \varepsilon_{cy} = \varepsilon_{sy} \qquad \gamma_{xy} = \gamma_{cxy} \qquad (2.43)$$

De acordo com Vecchio e Collins (1986), conhecendo o estado de tensão num elemento de membrana, o processo de análise pela TMCC se inicia pela estimativa das deformações médias e pelo cálculo das deformações principais, através das relações de compatibilidade definidas em 2.2. A partir destas deformações, calculam-se as tensões principais no concreto, aplicando o modelo constitutivo definido em 2.7.1, com limitação da tensão de tração pela verificação da fissura (BENTZ, 2000).

As tensões nas armaduras são obtidas diretamente pelas deformações médias no elemento e por aplicação das equações definidas em 2.7.2. A partir das deformações e tensões obtidas, pode-se utilizar o modelo de rigidez secante (seção 2.5), ou as condições de equilíbrio definidas em (2.1), para o cálculo das tensões médias no elemento. O resultado é então comparado ao estado conhecido de tensões e esse cálculo é repetido até atingir a convergência necessária, por meio de uma tolerância da ordem de 0,001.

Segundo Vecchio (1990), se a formulação da TMCC estiver sendo utilizada em conjunto a uma modelagem por elementos finitos, as tensões finais no elemento devem ser utilizadas para o cálculo dos esforços nodais, para os quais será verificada a convergência. Caso essa convergência não seja atingida, deve-se utilizar a matriz de rigidez global da estrutura para o cálculo dos deslocamentos nodais, que serão transformados numa nova estimativa das deformações por meio das funções de forma utilizadas.

A seguir, será apresentada a formulação dos modelos constitutivos dos materiais, bem como para cálculo e verificação das tensões nas fissuras.

2.7.1 Modelos constitutivos

2.7.1.1 Concreto comprimido

A relação tensão-deformação para o concreto na compressão, de acordo com a TMCC, não é dependente unicamente da deformação de compressão (ε_2), mas também da deformação de tração (ε_1) perpendicular à primeira. Desse modo, o concreto submetido à tensões transversais de tração se torna menos resistente do que esse mesmo concreto submetido a um ensaio de compressão de corpo-de-prova cilíndrico (VECCHIO; COLLINS, 1986), efeito este conhecido como abrandamento.

Assim, a tensão de compressão no concreto (σ_{c2}) é definida por:

$$\sigma_{c2} = \sigma_{c2max} \left[2 \frac{\varepsilon_2}{\varepsilon'_c} - \left(\frac{\varepsilon_2}{\varepsilon'_c} \right)^2 \right] \quad (2.44a)$$

com:

$$\frac{\sigma_{c2max}}{f'_c} = \frac{1}{0,8 - 0,34 \varepsilon_1 / \varepsilon'_c} \leq 1,0 \quad (2.44b)$$

sendo: f'_c : resistência à compressão do concreto;
 ε'_c : deformação de pico, na qual se atinge a tensão f'_c .

Esta relação pode ser vista graficamente na Figura 2.9a, sendo que o traço pontilhado representa o comportamento padrão do concreto e o traço contínuo, o comportamento considerando o abrandamento.

É importante destacar que tanto σ_{c2max} quanto ε'_c têm valores negativos na (2.44a) por se tratarem de uma tensão de compressão e uma deformação de encurtamento, respectivamente.

2.7.1.2 Concreto tracionado

O comportamento à tração considerado pela TMCC é linear até o concreto atingir a deformação de fissuração (ε_{cr}). Após a fissuração, o modelo considera o efeito de

enrijecimento do concreto (*tension stiffening*), dado pela transferência de tensões de tração entre o concreto e as armaduras nas regiões entre fissuras (VECCHIO; COLLINS, 1986). Portanto:

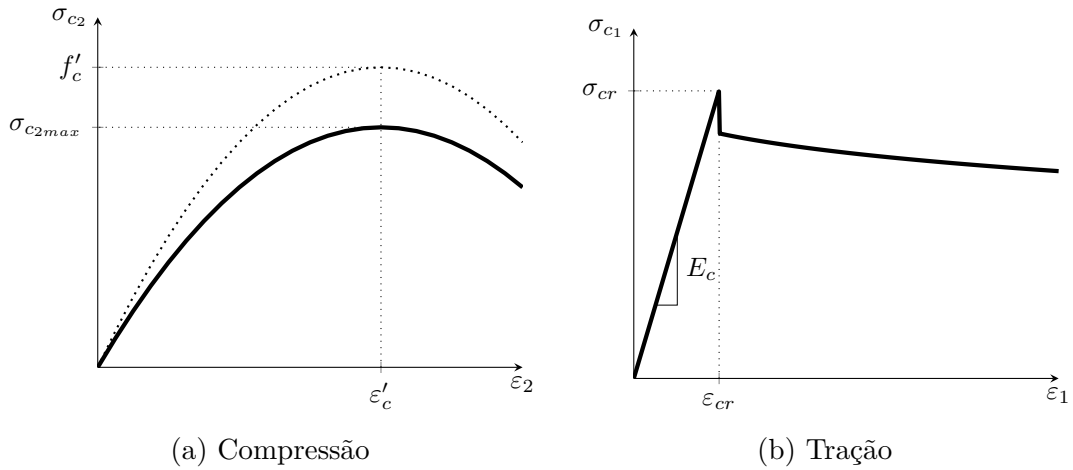
$$\sigma_{c1} = \begin{cases} E_c \varepsilon_1, & \text{se } \varepsilon_1 \leq \varepsilon_{cr}, \\ \frac{\sigma_{cr}}{1 + \sqrt{500 \varepsilon_1}}, & \text{se } \varepsilon_1 > \varepsilon_{cr}. \end{cases} \quad (2.45)$$

sendo: σ_{cr} : tensão de fissuração do concreto;

ε_{cr} : deformação de fissuração do concreto (f_{cr}/E_c).

O comportamento à tração do concreto, de acordo com a TMCC, pode ser visualizado na Figura 2.9b.

Figura 2.9 – Relação tensão-deformação para o concreto de acordo com a TMCC



Fonte: Adaptado de Vecchio e Collins (1986).

2.7.2 Aço

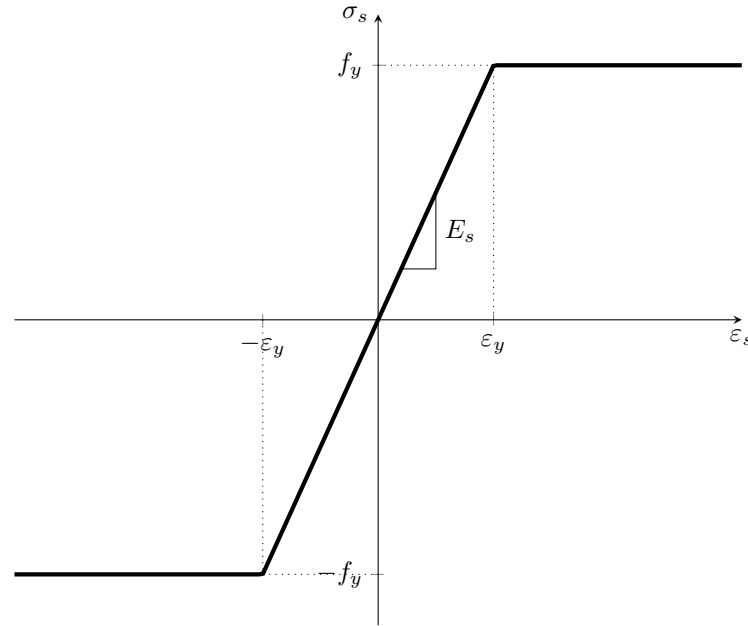
Neste trabalho será considerada a relação tensão-deformação bilinear para o aço. A tensão no aço (σ_s) é calculada a partir da deformação (ε_s) por:

$$\sigma_s(\varepsilon_s) = \begin{cases} -f_y, & \text{se } \varepsilon_s \leq -\varepsilon_y, \\ E_s \varepsilon_s, & \text{se } -\varepsilon_y < \varepsilon_s < \varepsilon_y, \\ f_y, & \text{se } \varepsilon_s \geq \varepsilon_y. \end{cases} \quad (2.46)$$

sendo E_s o módulo de elasticidade do aço.

Esta relação é apresentada graficamente na Figura 2.10.

Figura 2.10 – Relação tensão-deformação para o aço



Fonte: Adaptado de Vecchio e Collins (1986).

2.7.3 Transmissão de esforços através da fissura

De acordo com Vecchio e Collins (1986), numa fissura, as tensões nas armaduras serão maiores que seus valores médios, enquanto nas regiões entre essas fissuras, serão menores. Desse mesmo modo, a tensão principal de tração no concreto será nula na fissura e maior que seu valor médio entre as fissuras. Este efeito pode ser visualizado graficamente na Figura 2.11, sendo que, para compensar a redução da tensão de tração no concreto, as tensões nas armaduras aumentam (VECCHIO, 2000). O cálculo explícito dessas tensões é importante, pois a ruína de um elemento pode ser governada pela capacidade das armaduras de transmitir as tensões através das fissuras.

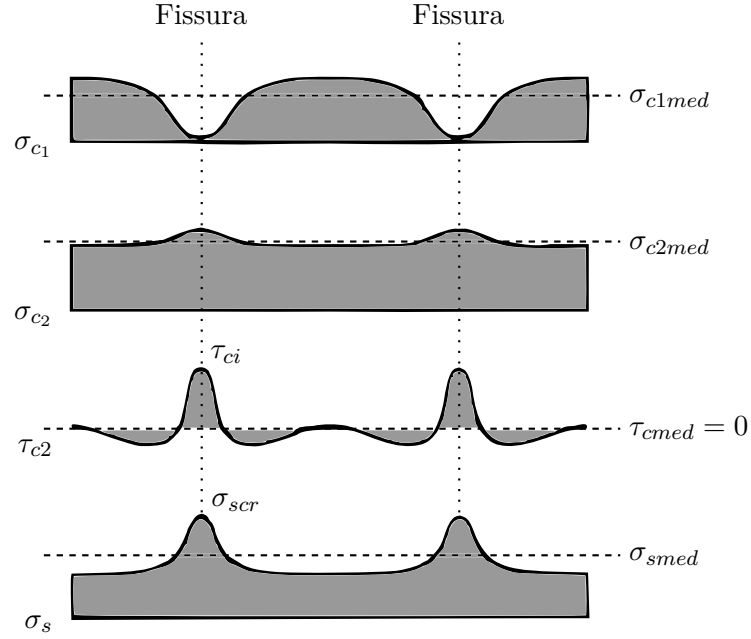
A Figura 2.12a apresenta as tensões médias num elemento de membrana, enquanto a Figura 2.12b apresenta as tensões atuantes na fissura. Observa-se que as tensões nas armaduras são dadas por σ_{sxcr} e $\sigma_{sy cr}$, que são superiores aos valores médios (σ_{sx} e σ_{sy}). Além disso, ainda atua uma tensão de cisalhamento concreto (τ_{ci}) acompanhada de uma pequena tensão de compressão (σ_{ci}), para as quais se considera a mesma direção da tensão principal média (θ_{c1}).

Já que os estados de tensão ilustrados na Figura 2.12 são equivalentes (VECCHIO; COLLINS, 1986), pode-se escrever:

$$\rho_{sx} \sigma_{sx} \cos \theta_{c1} + \sigma_{c1} \cos \theta_{c1} = \rho_{sx} \sigma_{sxcr} \cos \theta_{c1} - \sigma_{ci} \cos \theta_{c1} - \tau_{ci} \sin \theta_{c1} \quad (2.47a)$$

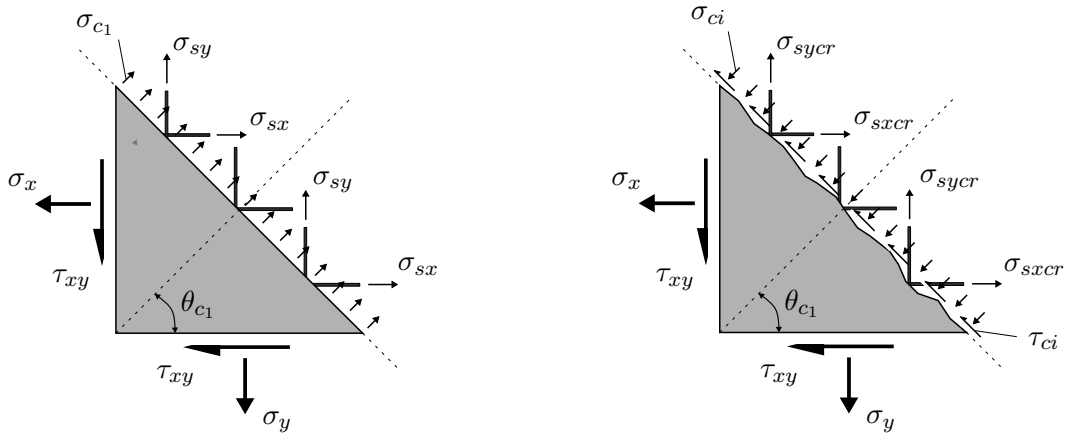
$$\rho_{sy} \sigma_{sy} \sin \theta_{c1} + \sigma_{c1} \sin \theta_{c1} = \rho_{sy} \sigma_{sy cr} \sin \theta_{c1} - \sigma_{ci} \sin \theta_{c1} + \tau_{ci} \cos \theta_{c1} \quad (2.47b)$$

Figura 2.11 – Distribuição de tensões num elemento de concreto armado fissurado



Fonte: adaptado de Vecchio (2000).

Figura 2.12 – Tensões atuantes no elemento de membrana



(a) Tensões médias

(b) Tensões na fissura

Fonte: adaptado de Vecchio e Collins (1986).

Dividindo (2.47a) por $\cos \theta_{c1}$ e (2.47b) por $\sin \theta_{c1}$, tem-se:

$$\rho_{sx} (\sigma_{sxcr} - \sigma_{sx}) = \sigma_{c1} + \sigma_{ci} + \tau_{ci} \operatorname{tg} \theta_{c1} \quad (2.48a)$$

$$\rho_{sy} (\sigma_{sycr} - \sigma_{sy}) = \sigma_{c1} + \sigma_{ci} - \tau_{ci} \operatorname{cotg} \theta_{c1} \quad (2.48b)$$

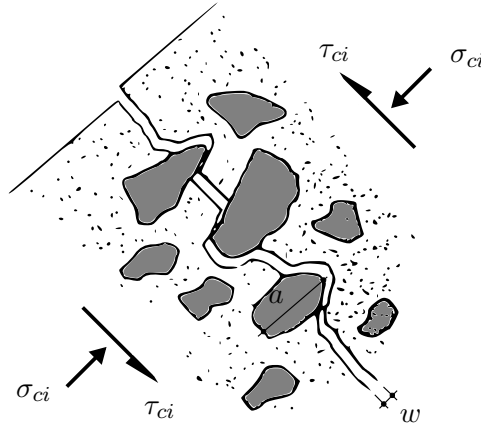
Portanto, o equilíbrio na fissura só é satisfeito sem tensões de cisalhamento e compressão se:

$$\rho_{sx} (\sigma_{sxcr} - \sigma_{sx}) = \rho_{sy} (\sigma_{sycr} - \sigma_{sy}) = \sigma_{c1} \quad (2.49)$$

sendo que as tensões nas armaduras não podem exceder a tensão de escoamento das armaduras ($\sigma_{sxcr} \leq f_{yx}$ e $\sigma_{sycr} \leq f_{yy}$).

Por outro lado, se as tensões nas armaduras forem altas, a Equação (2.48) só pode ser satisfeita com a existência de tensões de cisalhamento (τ_{ci}) na fissura. Já que as fissuras normalmente se formam nas interfaces entre a pasta de cimento e os agregados, esta tensão será transmitida por meio do engrenamento dos agregados (*aggregate interlock*), conforme se observa na Figura 2.13 (VECCHIO; COLLINS, 1986).

Figura 2.13 – Transmissão de tensões na fissura através do engrenamento dos agregados



Fonte: adaptado de Vecchio e Collins (1986).

Baseando-se nos estudos experimentais conduzidos por Walraven (1980), as tensões locais na fissura, transmitidas pelo engrenamento dos agregados, são relacionadas por:

$$\tau_{ci} = 0,18 \tau_{cimax} + 1,64 \sigma_{ci} - 0,82 \frac{\sigma_{ci}^2}{\tau_{cimax}} \quad (2.50)$$

A máxima tensão de cisalhamento na fissura (τ_{cimax}) que pode ser transmitida pelo concreto é calculada por:

$$\tau_{cimax} = \frac{\sqrt{|f'_c|}}{0,31 + 24 \frac{w}{\phi_a + 16}} \quad (2.51)$$

sendo: f'_c : resistência à compressão do concreto, em MPa;
 w : abertura média de fissura, em mm;
 ϕ_a : diâmetro máximo do agregado, em mm.

A abertura média de fissura (w) pode ser calculada, de acordo com Vecchio e Collins (1986), por:

$$w = \varepsilon_1 s_{m\theta} \quad (2.52)$$

sendo $s_{m\theta}$ o distanciamento médio entre fissuras, medido no eixo inclinado em θ_{c1} :

$$s_{m\theta} = \frac{1}{\frac{\cos \theta_{c1}}{s_{mx}} + \frac{\sin \theta_{c1}}{s_{my}}} \quad (2.53)$$

Os distanciamentos médios nas direções x e y (s_{mx} e s_{my}) podem ser estimados a partir da equação proposta por Kaklauskas, Ramanauskas e Ng (2019), que apresentam

resultados condizentes com ensaios experimentais:

$$s_{mx} = 21 \text{ mm} + 0.155 \frac{\phi_{sx}}{\rho_{sx}} \quad s_{my} = 21 \text{ mm} + 0.155 \frac{\phi_{sy}}{\rho_{sy}} \quad (2.54)$$

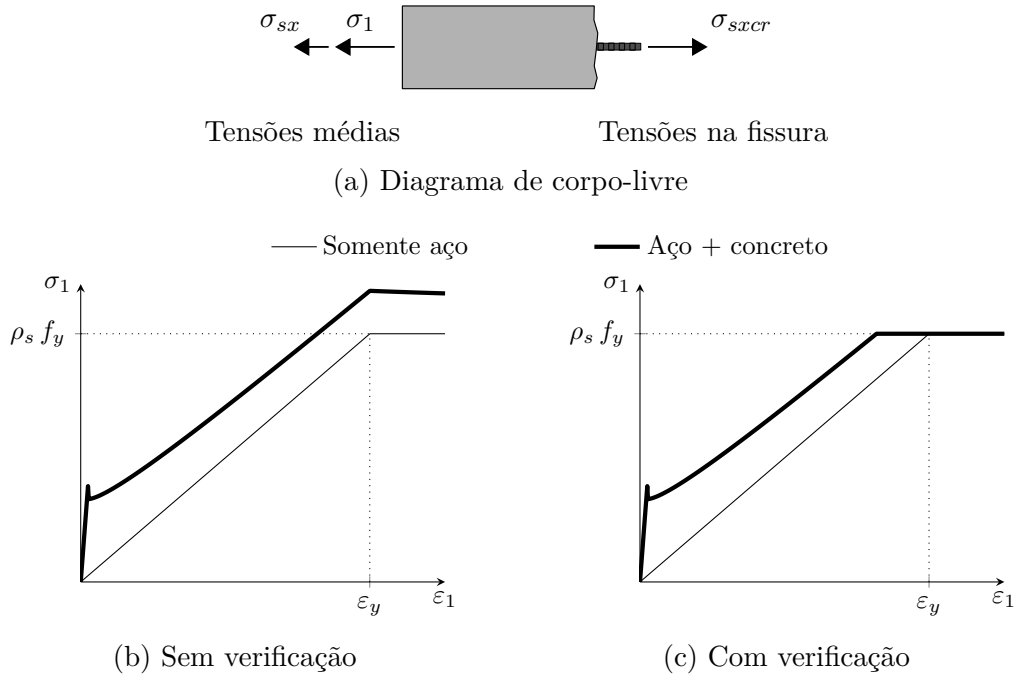
sendo ϕ_{sx} e ϕ_{sy} os diâmetros das barras de aço que constituem as armaduras dispostas em x e y , respectivamente.

2.7.4 Verificação das tensões na fissura

Segundo Bentz (2000), a verificação da fissura consiste num processo explícito que garante que as tensões médias sejam transmitidas localmente. A análise pela TMCC, sem a consideração deste procedimento, pode levar a resultados não seguros.

Analisando as tensões numa fissura para um tirante de concreto (Figura 2.14), pode-se fazer uma comparação gráfica entre a utilização ou não do processo de verificação das fissuras. Observa-se que, quando a verificação não é efetuada (Figura 2.14b), a tensão de tração ultrapassa a tensão de escoamento no aço. Já quando se faz a verificação da fissura (Figura 2.14c), pode-se visualizar que a tensão atinge o valor $\rho_s f_y$ antes mesmo de se atingir o nível de deformação ε_y .

Figura 2.14 – Procedimento de verificação da fissura para um tirante de concreto



Fonte: adaptado de Bentz (2000).

Para este caso unidimensional, ilustrado na Figura 2.14a, o processo de verificação da fissura consiste em limitar a tensão σ_1 , de modo que não seja ultrapassada a reserva de

capacidade da armadura (BENTZ, 2000):

$$\sigma_1 \leq \rho_s (\sigma_{sxc} - \sigma_{sx}) \quad (2.55)$$

No caso de um elemento de membrana, a verificação da fissura envolve a análise das tensões nas armaduras nas duas direções, além da inclusão de uma possível tensão de cisalhamento na interface. Assume-se que o concreto não seja capaz de transmitir tensões de tração através da fissura (BENTZ, 2000).

Para o equilíbrio das tensões na fissura, considera-se a condição ilustrada na Figura 2.12b, desconsiderando a parcela de compressão local no concreto ($\sigma_{ci} = 0$). Nota-se que existirá mais de uma solução para o equilíbrio na fissura, já que existem três incógnitas (σ_{sxc} , σ_{syc} e τ_{ci}) e apenas duas equações de equilíbrio ($\Sigma F_x = \Sigma F_y = 0$).

Segundo Bentz (2000), o procedimento padrão da TMCC assume que o mecanismo de resistência das armaduras é mais rígido que o mecanismo de cisalhamento na fissura, o que faz com que este último seja reduzido. Como consequência dessa consideração, toda a capacidade resistente da armadura, na direção mais fraca, será utilizada antes de ser necessária uma tensão de cisalhamento para o equilíbrio, sendo que somente se visualizará este efeito na resposta global caso seja necessária uma limitação da tensão σ_{c1} .

O processo de verificação da fissura abrange as verificações a seguir, descritas por Bentz (2000):

- (a) cálculo da tensão principal de tração no concreto (σ_{1a}) de acordo com o modelo constitutivo (Equação (2.45)), da máxima tensão de cisalhamento na fissura (Equação (2.51)) e das tensões médias nas armaduras (σ_{sx} e σ_{sy});
- (b) cálculo da reserva de capacidade das armaduras para as direções x e y (σ_{1cx} e σ_{1cy}):

$$\sigma_{1cx} = \rho_{sx} (f_{yx} - \sigma_{sx}) \quad \sigma_{1cy} = \rho_{sy} (f_{yy} - \sigma_{sy}) \quad (2.56)$$

- (c) cálculo da tensão de escoamento biaxial (σ_{1b}), sem existência de cisalhamento na fissura:

$$\sigma_{1b} = \sigma_{1cx} \sin^2 \theta_{c1} + \sigma_{1cy} \cos^2 \theta_{c1} \quad (2.57)$$

- (d) cálculo da tensão de cisalhamento (τ_{ci2}) que provoca o escoamento biaxial das armaduras:

$$\tau_{ci2} = |\sigma_{1cx} - \sigma_{1cy}| \sin \theta_{c1} \cos \theta_{c1} \quad (2.58)$$

- (e) cálculo da máxima tensão de tração para equilíbrio nas direções x (σ_{1c}) e y (σ_{1d}):

$$\sigma_{1c} = \sigma_{1cx} + \min(\tau_{cimax}, \tau_{ci2}) \tan \theta_{c1} \quad (2.59a)$$

$$\sigma_{1d} = \sigma_{1cy} + \min(\tau_{cimax}, \tau_{ci2}) \cotg \theta_{c1} \quad (2.59b)$$

- (f) adoção do menor valor para a tensão principal de tração:

$$\sigma_1 = \min(\sigma_{1a}, \sigma_{1b}, \sigma_{1c}, \sigma_{1d}) \quad (2.60)$$

A intensidade real de τ_{ci} está condicionada às tensões nas armaduras (sua direção dominante e se está escoando ou não). O cálculo de τ_{ci} pode ser feito a partir das equações definidas na Tabela 2.1.

Tabela 2.1 – Tensão de cisalhamento na fissura condicionada às tensões nas armaduras

Condição	Significado	Equação
$\sigma_{1cx} = \sigma_{1cy} = 0$	Escoamento biaxial	$\tau_{ci} = 0$
$\sigma_{1cx} > \sigma_{1cy}$ e $\sigma_{1cy} < \sigma_1$	Direção y dominante com escoamento da armadura	$\tau_{ci} = (\sigma_1 - \sigma_{1cy}) \operatorname{tg} \theta_{c1}$
$\sigma_{1cx} > \sigma_{1cy}$ e $\sigma_{1cy} > \sigma_1$	Direção y dominante sem escoamento da armadura	$\tau_{ci} = 0$
$\sigma_{1cx} < \sigma_{1cy}$ e $\sigma_{1cx} < \sigma_1$	Direção x dominante com escoamento da armadura	$\tau_{ci} = (\sigma_{1cx} - \sigma_1) \operatorname{cotg} \theta_{c1}$
$\sigma_{1cx} < \sigma_{1cy}$ e $\sigma_{1cx} > \sigma_1$	Direção x dominante sem escoamento da armadura	$\tau_{ci} = 0$

Fonte: adaptado de Bentz (2000).

Por fim, as tensões locais das armaduras na fissura podem ser obtidas por:

$$\sigma_{sxcr} = \frac{\sigma_1 + \tau_{ci} \operatorname{tg} \theta_{c1}}{\rho_{sx}} + \sigma_{sx} \quad \sigma_{syxr} = \frac{\sigma_1 - \tau_{ci} \operatorname{cotg} \theta_{c1}}{\rho_{sy}} + \sigma_{sy} \quad (2.61)$$

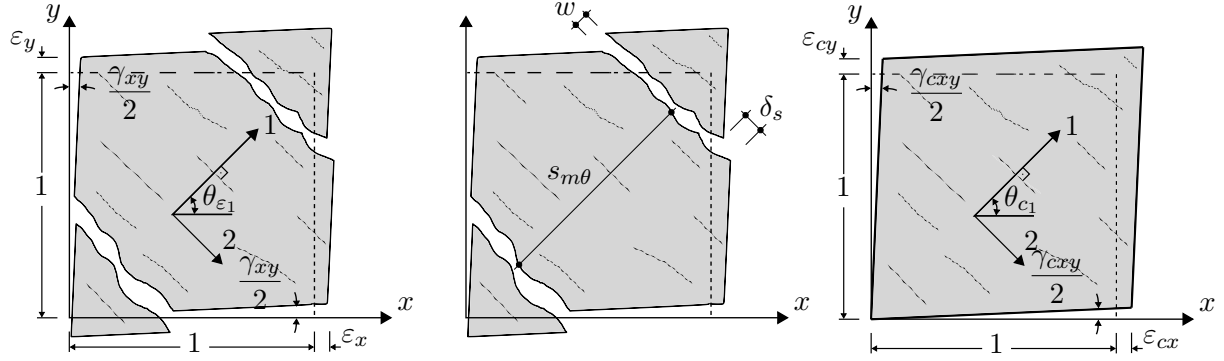
2.8 Modelo de Campo de Tensão Perturbado (MCTP)

Embora a formulação da TMCC tenha gerado resultados consistentes desde a sua origem, a resistência do elemento ao cisalhamento pode ser subestimada em casos com grande taxa de armadura ou alta compressão biaxial, e superestimada em casos com pouca armadura ou elementos reforçados uniaxialmente (VECCHIO, 2000).

Para tentar contornar estas deficiências, o MCTP (*Disturbed Stress Field Model*, originalmente) foi proposto por Vecchio (2000) como uma melhoria da TMCC. O modelo considera explicitamente as deformações de deslizamento das fissuras, o que resulta em deformações médias no elemento maiores que as deformações no concreto. O modelo constitutivo do concreto é atualizado, conforme visto na subseção 2.8.1, com uma nova formulação para o efeito de abrandamento à compressão do concreto e para o comportamento à tração pós-fissuração, que agora é afetado pela quantidade de armadura existente no elemento. Além disso, o MCTP também contempla o conceito de energia de fraturamento através de um modelo de abrandamento à tração.

Considera-se um elemento submetido a um estado de deformações médias (Figura 2.15a). Este estado, aliado à fissuração do concreto, ocasionará um deslocamento de corpo rígido ao longo da interface da fissura (VECCHIO, 2000), conforme a Figura 2.15b.

Figura 2.15 – Deformações no elemento de membrana de acordo com o MCTP



(a) Deformações médias

(b) Deslizamento da fissura

(c) Deformações no concreto

Fonte: adaptado de Vecchio (2000).

As deformações reais no concreto (Figura 2.15c), serão dadas pela diferença entre as deformações médias $\{\varepsilon\}$ e as deformações por deslizamento da fissura $\{\varepsilon_{sl}\}$:

$$\{\varepsilon_c\} = \{\varepsilon\} - \{\varepsilon_{sl}\} \quad (2.62)$$

com $\{\varepsilon_{sl}\}$ determinado na Seção 2.8.4.

Dessa forma, as deformações principais (ε_{c1} e ε_{c2}) no concreto e sua orientação (θ_{c1}) devem ser calculadas aplicando as equações (2.8) e (2.9) ao vetor $\{\varepsilon_c\}$. Assim como, para o vetor de deformações médias $\{\varepsilon\}$, pode-se calcular o estado principal da mesma forma. Pelo fato de as deformações serem diferentes, suas orientações também serão diferentes ($\theta_{c1} \neq \theta_\varepsilon$).

As deformações médias nas armaduras são consideradas equivalentes às deformações totais:

$$\varepsilon_{sx} = \varepsilon_x \quad \varepsilon_{sy} = \varepsilon_y \quad (2.63)$$

O estado de tensão $\{\sigma\}$, relacionado a um estado de deformação $\{\varepsilon\}$, de acordo com Vecchio (2001), é calculado por:

$$\{\sigma\} = [D] \{\varepsilon\} - \{\sigma_0\} \quad (2.64)$$

sendo $\{\sigma_0\}$ o vetor de pseudo-pré-tensão, que considera o efeito de deslocamentos elásticos e plásticos no concreto e nas armaduras. Considerando as deformações de deslizamento da fissura, $\{\sigma_0\}$ será dado por:

$$\{\sigma_0\} = [D_c] \{\varepsilon_{sl}\} \quad (2.65)$$

É possível, ainda, pela formulação do MCTP, a consideração de efeitos de variação térmica, retração do concreto, histórico de carregamento e fadiga (VECCHIO, 2001). Nos modelos analisados neste trabalho, estes efeitos não serão considerados.

A análise pelo MCTP é feita de maneira similar à TMCC, excentuando-se a inclusão do cálculo das deformações por deslizamento da fissura e de o procedimento de verificação das tensões na fissura ser simplificado. Caso o deslizamento da fissura não seja considerado, além da consideração de um fator $C_s = 1,0$ para o concreto (subseção 2.8.1), recomenda-se utilizar o procedimento completo de verificação, abordado em 2.7.4.

O processo se inicia pela obtenção das deformações médias no elemento e as deformações médias no concreto (2.62). Calculam-se, então, as deformações principais (2.7) e tensões principais no concreto, pela formulação apresentada em 2.8.1, sendo que a tensão principal de tração deve ser limitada, conforme descrito em 2.8.2.

A seguir, devem ser calculadas as tensões locais nas armaduras e a tensão de cisalhamento na fissura, por meio do processo definido em 2.8.3. Tendo a intensidade da tensão na fissura, calculam-se as deformações por deslizamento da fissura por meio das duas abordagens definidas em 2.8.4.

Por fim, obtêm-se as matrizes de rigidez secante do concreto (2.18), das armaduras (2.19) e a matriz do elemento (2.15); e se calculam as tensões finais no elemento (2.64). Verifica-se, então, a convergência de acordo com o estado de tensão conhecido.

A utilização do MCTP em conjunto ao MEF é feita da mesma forma que para a TMCC, apenas considerando que haverá uma parcela de deslocamento devida ao deslizamento das fissuras e, conseqüentemente, existirá um vetor de pseudo-forças nodais resultante de $\{\sigma_0\}$.

A seguir serão definidas as equações do MCTP para as relações tensão-deformação do concreto, cálculo e limitação das tensões na fissura e obtenção das deformações de deslizamento da fissura.

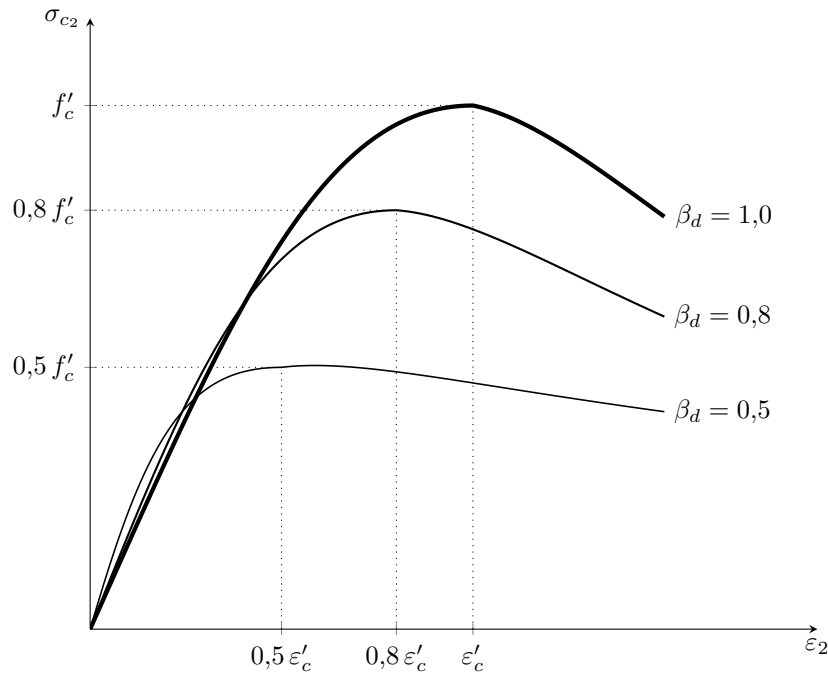
2.8.1 Modelos constitutivos

Nesta seção é apresentado o modelo constitutivo do concreto considerado pelo MCTP, de acordo com Vecchio (2000). Em relação ao aço, utiliza-se o modelo bilinear, definido em 2.7.2.

2.8.1.1 Concreto comprimido

O comportamento do concreto à compressão para o MCTP pode ser visualizado na Figura 2.16. Observa-se que, com o decréscimo de β_d , o concreto apresenta um efeito pós-pico de redução menos acentuada.

Figura 2.16 – Relação tensão-deformação para o concreto na compressão de acordo com o MCTP



Fonte: Adaptado de Vecchio (2000).

A relação utilizada para compressão é dada por:

$$\sigma_{c2} = f_p \frac{n (\varepsilon_{c2}/\varepsilon_p)}{n - 1 + (\varepsilon_{c2}/\varepsilon_p)^{nk}} \quad (2.66)$$

sendo:

$$n = 0,8 - \frac{f_p}{17} \quad k = \begin{cases} 1,0, & \text{se } \varepsilon_p \leq \varepsilon_{c2}, \\ 0,67 - f_p/62, & \text{se } \varepsilon_p > \varepsilon_{c2}. \end{cases}$$

A tensão de pico (f_p) e a respectiva deformação (ε_p), devidas ao efeito de abrandamento do concreto, são dadas por:

$$f_p = -\beta_d f'_c \quad \varepsilon_p = -\beta_d \varepsilon'_c \quad (2.67)$$

com:

$$\beta_d = \frac{1}{1 + C_s C_d} \leq 1,0$$

O coeficiente C_d leva em conta as deformações de compressão (ε_{c2}) e de tração transversal (ε_{c1}) existentes:

$$C_d = 0,35 \left(-\frac{\varepsilon_{c1}}{\varepsilon_{c2}} - 0,28 \right)^{0,8} \quad (2.68)$$

Segundo Vecchio (2000), o fator C_s leva em conta o deslizamento das fissuras. Se considerado, o efeito de abrandamento do concreto deve ser reduzido para compensar

maiores deformações médias no elemento de membrana. Pelas análises conduzidas por Vecchio (2000), a consideração de $C_s = 0,55$ mostrou resultados satisfatórios. Caso o efeito de deslizamento das fissuras não seja explicitamente considerado, pode-se utilizar $C_s = 1,0$.

2.8.1.2 Concreto tracionado

Para o concreto tracionado, o MCTP também considera uma relação linear até a fissuração, dada pelo primeiro caso em (2.45). Após a fissuração, o modelo considera uma formulação mista entre dois modelos: de abrandamento e de enrijecimento do concreto à tração.

De acordo com Vecchio (2000), a tensão devida ao efeito de abrandamento (σ_{c1}^a) está associada a mecanismos da Mecânica da Fratura, que são significantes em casos em que existe pouca ou nenhuma armadura distribuída. Esta tensão é dada por:

$$\sigma_{c1}^a = f_{cr} \left(1 - \frac{\varepsilon_{c1} - \varepsilon_{cr}}{\varepsilon_{ts} - \varepsilon_{cr}} \right), \quad \text{para } \varepsilon_{c1} > \varepsilon_{cr} \quad (2.69)$$

sendo ε_{ts} a deformação última à tração, calculada por:

$$\varepsilon_{ts} = 2 \frac{G_f}{f_{cr} L_r}$$

sendo: G_f : energia de fraturamento do concreto, podendo ser tomada como 75 N/m;
 L_r : comprimento característico.

Segundo Wong, Vecchio e Tamm (2013), o comprimento característico pode ser tomado como metade do espaçamento médio entre fissuras ($L_r = 0,5 s_m$), com s_m obtido em (2.53).

A tensão devida ao efeito de enrijecimento à tração (σ_{c1}^b) segue o mesmo conceito do TMCC, a formulação proposta por Vecchio (2000) é:

$$\sigma_{c1}^b = \frac{\sigma_{cr}}{1 + \sqrt{c_t \varepsilon_{c1}}} \quad (2.70)$$

O coeficiente c_t é uma modificação do modelo proposto por Bentz (2000), que mostrou que efeito está relacionado a taxa (ρ_s) a ao diâmetro (ϕ_s) das armaduras dispostas no elemento estrutural:

$$c_t = 2,2 m \quad (2.71)$$

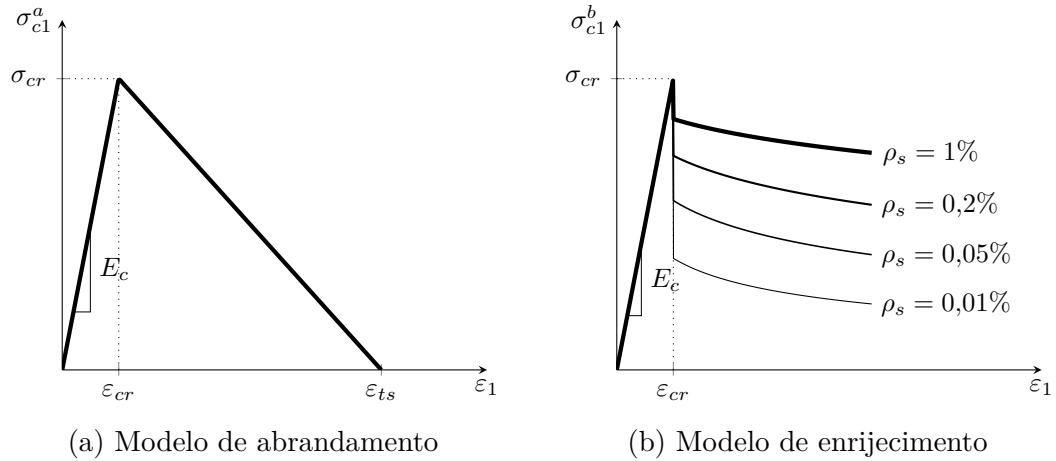
com:

$$\frac{1}{m} = \sum_{i=1}^n \frac{4 \rho_{si}}{\phi_{si}} |\cos(\theta_{ni})|$$

sendo θ_{ni} o ângulo que a armadura faz com o ângulo de inclinação (θ_{c1}) da tensão principal σ_{c1} .

Os modelos considerados para o cálculo da tensão de tração no concreto podem ser visualizados na Figura 2.17. Comparando um elemento com armaduras iguais nas duas direções, é possível observar a influência da taxa de armadura no comportamento à tração após a fissuração (Figura 2.17b), quanto menor a armadura no elemento, menor será a sua tensão principal de tração para o mesmo nível de deformação.

Figura 2.17 – Relações tensão-deformação para o concreto à tração de acordo com o MCTP



Fonte: Adaptado de Vecchio (2000).

Calculadas as tensões relacionadas aos dois modelos, a tensão utilizada deve ser a maior entre elas:

$$\sigma_{c1} = \max(\sigma_{c1}^a, \sigma_{c1}^b) \quad (2.72)$$

2.8.2 Limitação da tensão principal de tração

Tanto as condições de equilíbrio, definidas na seção 2.2, quanto as relações de equilíbrio na fissura (subseção 2.7.3) são válidas para o MCTP. O processo de verificação da fissura é simplificado, já que as deformações de deslizamento são consideradas. É necessário apenas limitar a tensão principal de tração no concreto (σ_{c1}) de modo que as armaduras sejam capazes de transmiti-la através da fissura (VECCHIO, 2000). Para armaduras dispostas nas direções x e y , tem-se:

$$\sigma_{c1} \leq \rho_{sx} (f_{yx} - \sigma_{sx}) \cos^2 \theta_{c1} + \rho_{sy} (f_{yy} - \sigma_{sy}) \sin^2 \theta_{c1} \quad (2.73)$$

2.8.3 Tensões locais nas fissuras

Numa fissura, conforme visualizado na Figura 2.11, as tensões e deformações locais nas armaduras aumentam para compensar a redução da tensão principal de tração no concreto (VECCHIO, 2000). Na direção desta tensão, haverá um incremento local de

deformação ($\Delta\varepsilon_{1cr}$) para satisfazer o equilíbrio. A deformação total nas armaduras, na região fissurada, será:

$$\varepsilon_{scrx} = \varepsilon_{sx} + \Delta\varepsilon_{1cr} \cos^2 \theta_{c1} \quad \varepsilon_{scry} = \varepsilon_{sy} + \Delta\varepsilon_{1cr} \sin^2 \theta_{c1} \quad (2.74)$$

A partir destas deformações, é possível calcular as tensões locais nas armaduras (σ_{scrx} e σ_{scry}) pela relação tensão-deformação (2.46), as quais devem satisfazer a condição de equilíbrio na fissura:

$$\rho_{sx} (\sigma_{scrx} - \sigma_{sx}) \cos^2 \theta_{c1} + \rho_{sy} (\sigma_{scry} - \sigma_{sy}) \sin^2 \theta_{c1} = \sigma_{c1} \quad (2.75)$$

Segundo Vecchio (2000), o equilíbrio também exigirá uma tensão de cisalhamento (τ_{ci}), dada por:

$$\tau_{ci} = [\rho_{sx} (\sigma_{scrx} - \sigma_{sx}) + \rho_{sy} (\sigma_{scry} - \sigma_{sy})] \cos \theta_{c1} \sin \theta_{c1} \quad (2.76)$$

A obtenção destas tensões pode ser feita por meio de um processo iterativo, estimando o valor de $\Delta\varepsilon_{1cr}$ e verificando se a condição de equilíbrio (2.75) é atendida. Atingida a convergência, calcula-se τ_{ci} , que será necessário para a obtenção do deslizamento da fissura.

2.8.4 Deslizamento da fissura

Conforme visualizado na Figura 2.15b, o cisalhamento no elemento de membrana resultará num deslizamento (δ_s) relativo nas interfaces da fissura. O MCTP adota a formulação de Walraven (1980) para a obtenção de δ_s :

$$\delta_s = \frac{\tau_{ci}}{1,8 w^{-0,8} + (0,234 w^{-0,707} - 0,2) f'_c} \quad (2.77)$$

sendo w a abertura média de fissuras e f'_c a tensão de compressão de pico do concreto.

A deformação por cisalhamento média devida ao deslizamento (γ_{sla}) é então definida por:

$$\gamma_{sla} = \frac{\delta_s}{s_{m\theta}} \quad (2.78)$$

com $s_{m\theta}$ calculado em (2.53).

Segundo Vecchio (2000), a implementação desta formulação pode gerar dois problemas:

- (a) a inclusão de um deslizamento inicial pode ser numericamente problemática, apesar deste existir antes da fissura se abrir;
- (b) em elementos sem armadura, o equilíbrio na fissura (2.76) resultará numa tensão de cisalhamento nula e, conseqüentemente, não existirá deslizamento na interface.

Para contornar estes problemas, Vecchio (2000) propõe a utilização de uma abordagem suplementar, denominada Defasagem de Rotação (*"Rotation Lag"*), que relaciona as mudanças nas direções das tensões principais no concreto com as direções principais aparentes. Essa rotação dos planos tem início com a abertura da primeira fissura e permanece aproximadamente constante até uma das componentes de armadura começar a escoar, podendo ter valores de 5° a 10°, dependendo das taxas de armadura.

A rotação do plano principal de deformação aparente ($\Delta\theta_\varepsilon$) é dada pela diferença entre a inclinação do estado atual (θ_ε) e a inclinação obtida na início da fissuração (θ_{ic}):

$$\Delta\theta_\varepsilon = \theta_\varepsilon - \theta_{ic} \quad (2.79)$$

A rotação do plano principal de tensão no concreto ($\Delta\theta_c$) é relacionada a $\Delta\theta_\varepsilon$ por:

$$\Delta\theta_c = \begin{cases} \Delta\theta_\varepsilon - \theta_l, & \text{se } |\Delta\theta_\varepsilon| > \theta_l, \\ \Delta\theta_\varepsilon & \text{se } |\Delta\theta_\varepsilon| \leq \theta_l. \end{cases} \quad (2.80)$$

sendo θ_l igual a 5° para elementos reforçados biaxialmente, 7,5° para elementos reforçados uniaxialmente, e 10° para elementos sem armadura.

Assim, a inclinação das tensões principais no concreto será:

$$\theta_c = \theta_{ic} + \Delta\theta_c \quad (2.81)$$

A deformação por cisalhamento, para esta abordagem, é então calculada por:

$$\gamma_{slb} = \gamma_{xy} \cos 2\theta_c + (\varepsilon_y - \varepsilon_x) \sin 2\theta_c \quad (2.82)$$

Por esta abordagem, o problema é ligado ao fato de, em estágios avançados de carregamento, a defasagem de rotação tende a ficar aproximadamente constante (VECCHIO, 2000). De modo a eliminar as deficiências de cada uma destas considerações, a deformação de deslizamento da fissura é tomada como o máximo valor entre as duas obtidas:

$$\gamma_{sl} = \max(\gamma_{sla}, \gamma_{slb}) \quad (2.83)$$

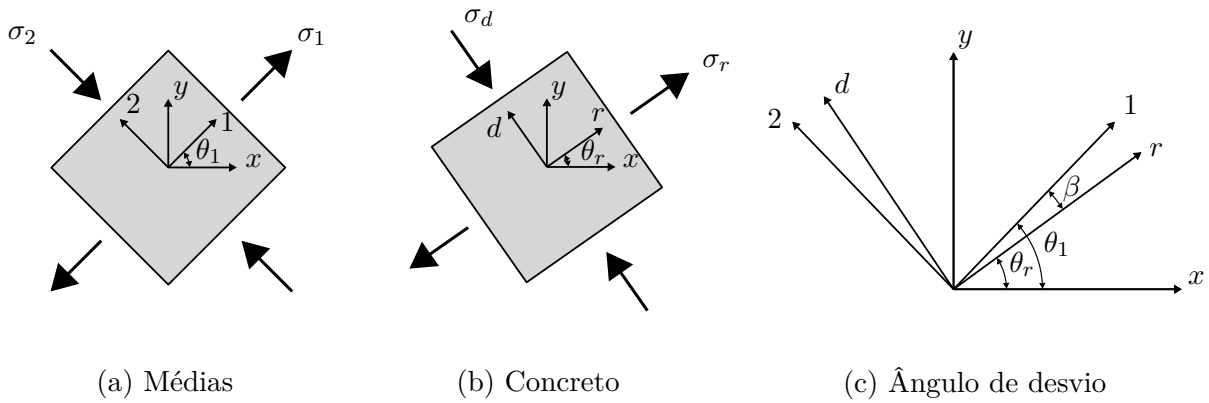
Utilizando esta formulação mista, nos estágios iniciais de carregamento o deslizamento será governado pela defasagem de rotação e, nos estágios finais, o cálculo explícito resultará num valor crítico para esta deformação. As deformações transformadas para o sistema de referência serão:

$$\{\varepsilon_{sl}\} = \begin{Bmatrix} \varepsilon_{slx} \\ \varepsilon_{sly} \\ \gamma_{slxy} \end{Bmatrix} = \begin{Bmatrix} -0.5 \gamma_{sl} \sin(2\theta_c) \\ 0.5 \gamma_{sl} \sin(2\theta_c) \\ \gamma_{sl} \cos(2\theta_c) \end{Bmatrix} \quad (2.84)$$

2.9 Modelo de Membrana Atenuado (MMA)

O MMA (HSU; ZHU, 2002) foi desenvolvida com base na formulação do MTA, incorporando a formulação de Zhu e Hsu (2002) para a consideração do efeito de Poisson no concreto fissurado. Conforme a Figura 2.18, o MMA considera o ângulo de desvio β , entre o plano principal das tensões médias no elemento (Figura 2.18a) e o plano das tensões principais no concreto (Figura 2.18b).

Figura 2.18 – Tensões principais no elemento de membrana conforme o MMA



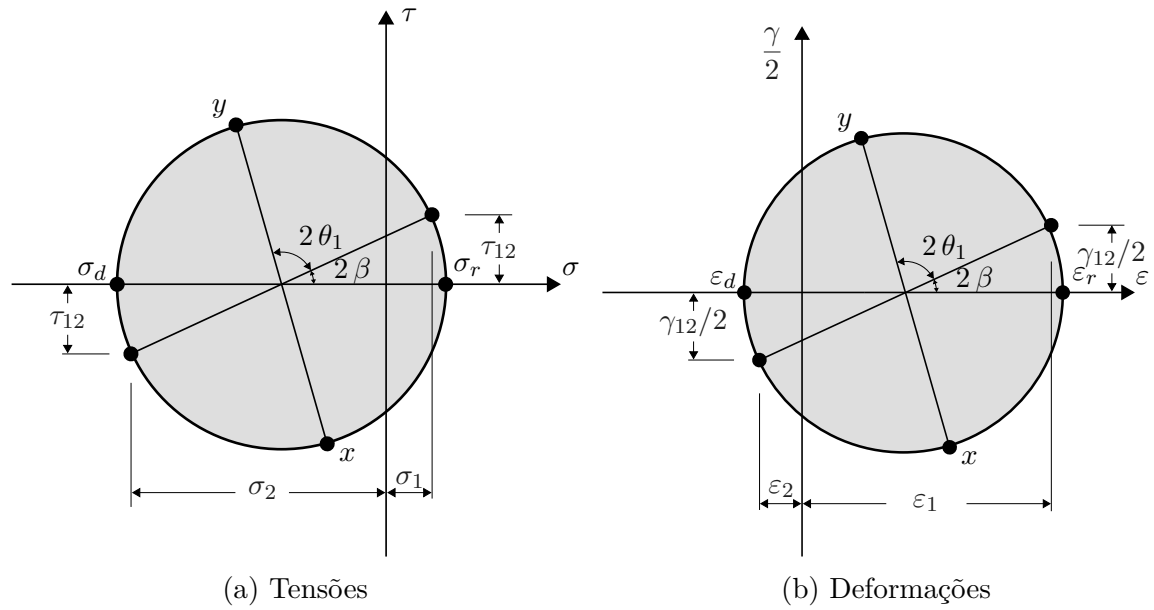
Fonte: adaptado de Zhu, Hsu e Lee (2001).

Havendo essa diferença de orientação, é visível que, nas direções principais médias, existirão no concreto a deformação γ_{12} e a tensão τ_{12} , as quais representam a contribuição do concreto na transmissão de esforços através das fissuras. Os planos principais visualizados na Figura 2.18 podem ser relacionados de acordo com o Círculo de Mohr representado na Figura 2.19.

As condições de equilíbrio e compatibilidade seguem a formulação apresentada em 2.2 e 2.3. As tensões no concreto são calculadas para os planos de tensão média (Figura 2.18a), sendo que a tensão de cisalhamento τ_{12} é obtida pela consideração de um módulo de cisalhamento (ZHU; HSU; LEE, 2001). O processo de solução pelo MMA é similar ao descrito em 2.7, sendo que das deformações médias devem ser removidas as parcelas devidas ao efeito de Poisson para o cálculo do estado de tensão. Além disso, é recomendado que se limite a tensão principal de tração (Equação 2.73), de modo que esta seja seguramente transmitida pelas armaduras nas regiões fissuradas.

A seguir serão apresentadas as formulações do MMA para a determinação dos estados de tensão e deformação, incorporando o efeito de Poisson, e dos modelos constitutivos utilizados.

Figura 2.19 – Círculo de Mohr para tensões e deformações no elemento de membrana conforme o MMA



Fonte: adaptado de Zhu, Hsu e Lee (2001).

2.9.1 Consideração do Efeito de Poisson

A influência do efeito de Poisson no comportamento de elementos de concreto armado submetidos a cisalhamento foi estudada por Zhu e Hsu (2002), a partir de dados experimentais de painéis. A seguir, serão apresentadas as equações para obtenção das deformações e dos coeficientes para a consideração deste efeito.

2.9.1.1 Concreto

Considerando o estado principal de deformações na matriz de concreto do elemento de membrana, as deformações finais, calculadas pelo Efeito de Poisson, serão:

$$\varepsilon_1 = \frac{\sigma_{c1}}{\bar{E}_{c1}} - \nu_{12} \frac{\sigma_{c2}}{\bar{E}_{c2}} \quad \varepsilon_2 = \frac{\sigma_{c2}}{\bar{E}_{c2}} - \nu_{21} \frac{\sigma_{c1}}{\bar{E}_{c1}} \quad (2.85)$$

sendo: $\varepsilon_1, \varepsilon_2$: deformações principais médias, considerando o Efeito de Poisson pelas taxas de Zhu e Hsu (2002);

$\bar{E}_{c1}, \bar{E}_{c2}$: módulos secantes do concreto, desconsiderando o Efeito de Poisson;

Introduzindo $\bar{\varepsilon}_1 = \sigma_{c1}/\bar{E}_{c1}$ e $\bar{\varepsilon}_2 = \sigma_{c2}/\bar{E}_{c2}$ em (2.85):

$$\varepsilon_1 = \bar{\varepsilon}_1 - \nu_{12} \bar{\varepsilon}_2 \quad \varepsilon_2 = \bar{\varepsilon}_2 - \nu_{21} \bar{\varepsilon}_1 \quad (2.86)$$

onde $\bar{\varepsilon}_1$ e $\bar{\varepsilon}_2$ são as deformações principais médias, desconsiderando o Efeito de Poisson.

Assim, de acordo com Hsu e Zhu (2002), subentende-se que as tensões principais no concreto σ_{c1} e σ_{c2} são calculadas considerando as deformações $\bar{\varepsilon}_1$ e $\bar{\varepsilon}_2$, respectivamente.

Pode-se também escrever as deformações $\bar{\varepsilon}$ em função de ε :

$$\bar{\varepsilon}_1 = \frac{1}{1 - \nu_{12} \nu_{21}} \varepsilon_1 + \frac{\nu_{12}}{1 - \nu_{12} \nu_{21}} \varepsilon_2 \quad \bar{\varepsilon}_2 = \frac{\nu_{21}}{1 - \nu_{12} \nu_{21}} \varepsilon_1 + \frac{1}{1 - \nu_{12} \nu_{21}} \varepsilon_2$$

A matriz de rigidez do concreto (2.16), pode ser reescrita considerando as novas deformações:

$$[D_c]' = \begin{bmatrix} \frac{\bar{E}_{c1}}{1 - \nu_{12} \nu_{21}} & \frac{\nu_{12} \bar{E}_{c1}}{1 - \nu_{12} \nu_{21}} & 0 \\ \frac{\nu_{21} \bar{E}_{c2}}{1 - \nu_{12} \nu_{21}} & \frac{\bar{E}_{c2}}{1 - \nu_{12} \nu_{21}} & 0 \\ 0 & 0 & \bar{G}_c \end{bmatrix} \quad (2.87)$$

2.9.1.2 Armaduras

Expandindo a equação de transformação (2.10) para as deformações nas armaduras e substituindo as relações de (2.86), são determinadas as deformações nas armaduras:

$$\varepsilon_x = \bar{\varepsilon}_1 \cos^2 \theta_1 + \bar{\varepsilon}_2 \sin^2 \theta_1 - \gamma_{12} \sin \theta_1 \cos \theta_1 - \nu_{12} \bar{\varepsilon}_2 \cos^2 \theta_1 - \nu_{21} \bar{\varepsilon}_1 \sin^2 \theta_1 \quad (2.88a)$$

$$\varepsilon_y = \bar{\varepsilon}_1 \sin^2 \theta_1 + \bar{\varepsilon}_2 \cos^2 \theta_1 - \gamma_{12} \sin \theta_1 \cos \theta_1 - \nu_{12} \bar{\varepsilon}_2 \sin^2 \theta_1 - \nu_{21} \bar{\varepsilon}_1 \cos^2 \theta_1 \quad (2.88b)$$

Sabendo que os três primeiros termos de cada equação se referem às deformações $\bar{\varepsilon}_1$ e $\bar{\varepsilon}_2$ transformadas para o plano xy , tem-se as deformação totais:

$$\varepsilon_x = \bar{\varepsilon}_x - \nu_{12} \bar{\varepsilon}_2 \cos^2 \theta_1 - \nu_{21} \bar{\varepsilon}_1 \sin^2 \theta_1 \quad (2.89a)$$

$$\varepsilon_y = \bar{\varepsilon}_y - \nu_{12} \bar{\varepsilon}_2 \sin^2 \theta_1 - \nu_{21} \bar{\varepsilon}_1 \cos^2 \theta_1 \quad (2.89b)$$

Assim como no caso do concreto, as tensões nas armaduras devem ser calculadas desconsiderando o efeito de Poisson; isto é, calculadas a partir das deformações $\bar{\varepsilon}_x$ e $\bar{\varepsilon}_y$.

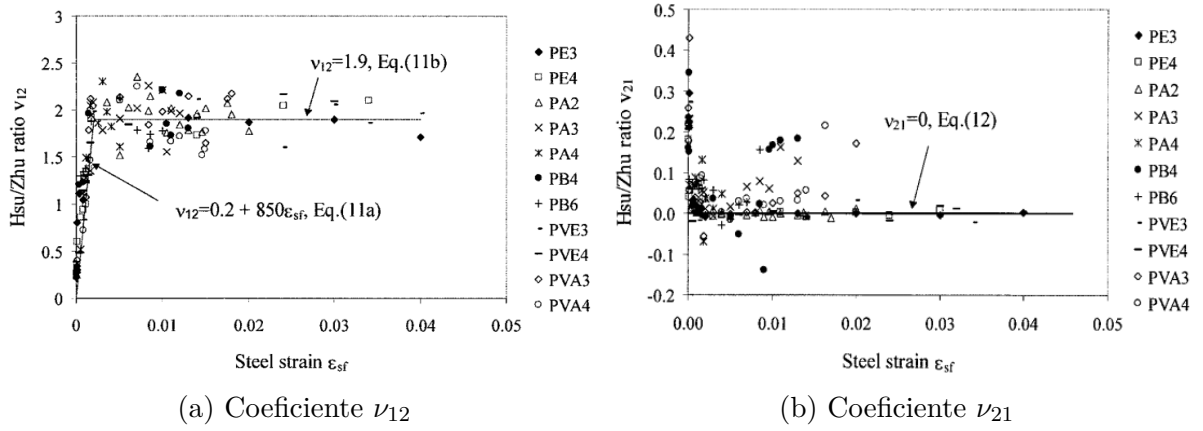
2.9.1.3 Cálculo dos coeficientes

A formulação proposta por Zhu e Hsu (2002) para os coeficientes de Poisson em elementos de membrana foi determinada após a análise experimental de doze painéis de concreto armado. Os resultados obtidos pelos autores podem ser visualizados na Figura 2.20.

O coeficiente ν_{12} (Figura 2.20a), antes da fissuração, mostrou-se dentro do valor esperado de aproximadamente 0,25. Após o concreto fissurar, este valor aumenta consideravelmente até chegar a um patamar próximo de 1,90. A correlação obtida por Zhu e Hsu (2002) para este fator é expressa por:

$$\nu_{12} = \begin{cases} 0,2 + 850 \varepsilon_{sf}, & \text{se } \varepsilon_{sf} \leq \varepsilon_y \\ 1,9, & \text{se } \varepsilon_{sf} > \varepsilon_y \end{cases} \quad (2.90)$$

Figura 2.20 – Dados experimentais dos coeficientes de Poisson em painéis de concreto armado



Fonte: Zhu e Hsu (2002).

sendo ε_{sf} a deformação da armadura na direção que escoou primeiro.

Quanto ao fator ν_{21} (Figura 2.20b), observou-se que, até a fissuração, este se mantém próximo do valor esperado de 0,25. Após a fissuração, o valor reduz para algo em torno de 0,05 e cai para zero com o escoamento da armadura. Zhu e Hsu (2002) propõem que tal coeficiente seja definido por:

$$\nu_{21} = \begin{cases} 0,2, & \text{se } \bar{\varepsilon}_1 \leq \varepsilon_{cr} \\ 0, & \text{se } \bar{\varepsilon}_1 > \varepsilon_{cr} \end{cases} \quad (2.91)$$

2.9.2 Modelos constitutivos

2.9.2.1 Concreto comprimido

O modelo constitutivo para o concreto comprimido é representado pelo traço destacado na Figura 2.21a, sendo que a tensão e a deformação de pico no concreto são reduzidos por um coeficiente de atenuação ζ . A formulação é distinta para as situações de pré e pós-pico:

$$\sigma_{c2} = \begin{cases} \zeta f'_c \left[2 \frac{\bar{\varepsilon}_2}{\zeta \varepsilon'_c} - \left(\frac{\bar{\varepsilon}_2}{\zeta \varepsilon'_c} \right)^2 \right], & \text{se } \frac{\bar{\varepsilon}_2}{\zeta \varepsilon'_c} \leq 1 \\ \zeta f'_c \left[1 - \left(\frac{\bar{\varepsilon}_2/\zeta \varepsilon'_c - 1}{4/\zeta - 1} \right)^2 \right], & \text{se } \frac{\bar{\varepsilon}_2}{\zeta \varepsilon'_c} > 1 \end{cases} \quad (2.92)$$

O coeficiente ζ é calculado a partir de três funções: f_1 , dependente da deformação de tração $\bar{\varepsilon}_1$; f_2 , da resistência à compressão do concreto; e f_3 , do ângulo de desvio β . Matematicamente:

$$\zeta = f_1(\bar{\varepsilon}_1) f_2(f'_c) f_3(\beta) \quad (2.93)$$

sendo:

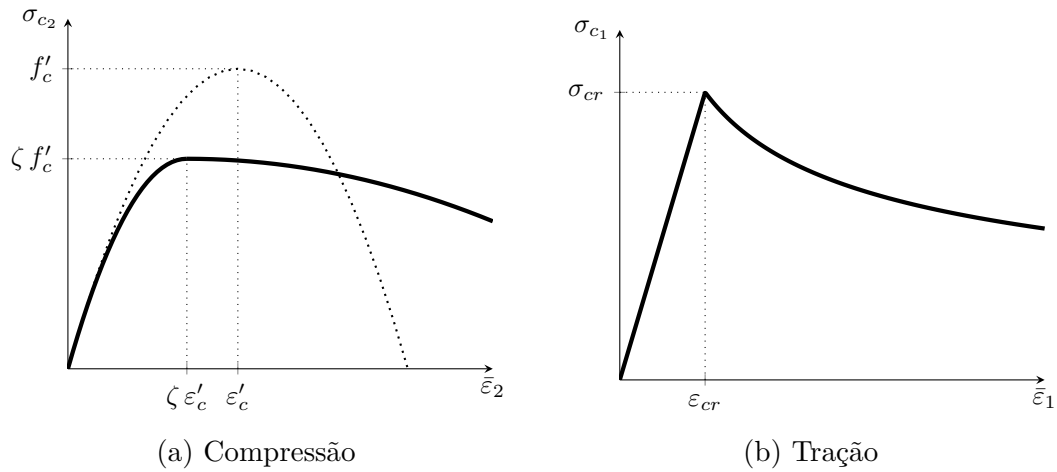
$$f_1(\bar{\varepsilon}_1) = \frac{1}{\sqrt{1 + 400\bar{\varepsilon}_1}} \quad f_2(f'_c) = \frac{5,8}{\sqrt{f'_c}} \leq 0,9 \quad f_3(\beta) = 1 - \frac{|\beta|}{24^\circ}$$

com f'_c em MPa.

O ângulo de desvio é dado pela diferença entre o plano das deformações principais médias no elemento e no concreto:

$$\beta = \frac{1}{2} \arctg \left(\frac{\gamma_{12}}{\varepsilon_1 - \varepsilon_2} \right) \quad (2.94)$$

Figura 2.21 – Relação tensão-deformação para o concreto de acordo com o MMA



Fonte: Adaptado de Hsu e Mo (2010).

2.9.2.2 Concreto tracionado

Para o caso de tração (Figura 2.21b), utiliza-se o modelo linear (2.45) até a fissuração ($\bar{\varepsilon}_1 \leq \varepsilon_{cr}$) e, após, a tensão é dada por:

$$\sigma_{c1} = \sigma_{cr} \left(\frac{\varepsilon_{cr}}{\bar{\varepsilon}_1} \right)^{0,4} \quad (2.95)$$

2.9.2.3 Cisalhamento

As relações tensão-deformação do MMA para cisalhamento no concreto foram deduzidas por Zhu, Hsu e Lee (2001), considerando que as direções das tensões e deformações principais são coincidentes. Observando os círculos de Mohr da Figura 2.19, pode-se escrever:

$$\tau_{21} = \frac{\sigma_1 - \sigma_2}{2} \operatorname{tg}(2\beta) \quad \gamma_{21} = (\varepsilon_1 - \varepsilon_2) \operatorname{tg}(2\beta) \quad (2.96)$$

Assim, a tensão τ_{21} pode ser relacionada à deformação γ_{21} através do módulo de cisalhamento G_{21} :

$$G_{21} = \frac{\tau_{21}}{\gamma_{21}} = \frac{\sigma_1 - \sigma_2}{2(\varepsilon_1 - \varepsilon_2)} \quad (2.97)$$

Escrevendo de uma forma mais simplificada:

$$\tau_{21} = \frac{\sigma_1 - \sigma_2}{2(\varepsilon_1 - \varepsilon_2)} \gamma_{21} \quad (2.98)$$

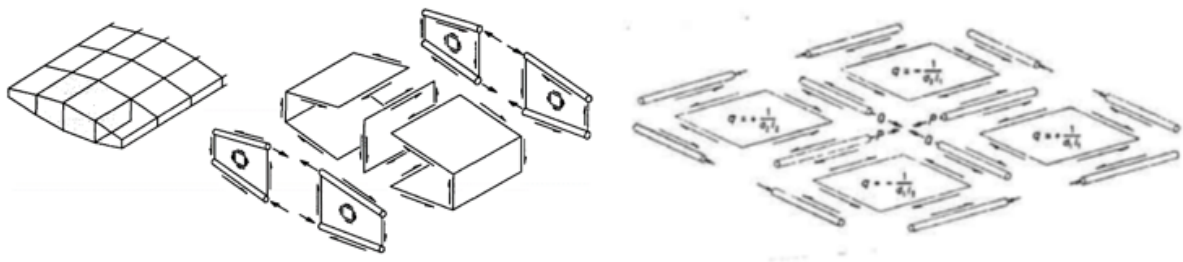
3 O Método Biela-Painel (MBP)

3.1 Histórico

Segundo Souza (2011), o Método Biela-Painel (MBP) ou *Stringer-Panel Method*, tem suas origens no Método das Forças, o qual foi sendo gradualmente substituído pelo Método dos Deslocamentos, pela maior simplicidade de programação deste último.

De acordo com Simone (1998), as primeiras aplicações do MBP se deram na engenharia mecânica, por volta da década de 1930, sendo amplamente utilizado a partir da década de 1950. Uma das mais importantes publicações desse período foi o livro de Argyris e Kelsey (1960), onde, na época, o modelo era chamado de “boom-panel system” e foi utilizado para modelagem de asas e fuselagem de aeronaves. A Figura 3.1 apresenta um exemplo deste sistema.

Figura 3.1 – Modelo biela-painel para uma asa de aeronave



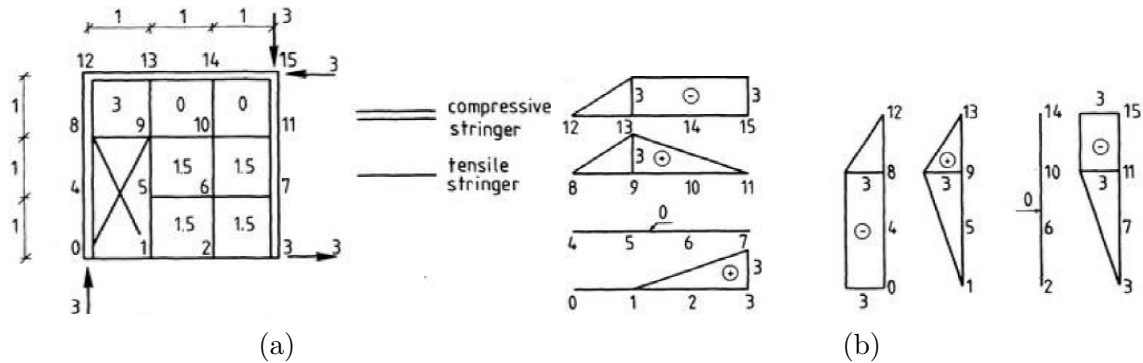
Fonte: Argyris e Kelsey (1960).

Ainda segundo Simone (1998), as primeiras aplicações na engenharia civil devem-se a Lundgren (1949 apud SIMONE, 1998), que aplicou o método ao dimensionamento de estruturas cilíndricas do tipo casca. Um pouco mais tarde, o modelo foi utilizado em estruturas do tipo parede, a partir das publicações de Nielsen (1971 apud SIMONE, 1998) e Kaern (1979), desenvolvidas na Universidade da Dinamarca; à época, o método era denominado simplesmente por *Stringer Method*. A Figura 3.2 apresenta um exemplo de aplicação para uma estrutura do tipo parede com abertura.

Rabbat e Collins (1978 apud SIMONE, 1998) apresentaram um modelo tridimensional para análise de vigas genericamente solicitadas. Esse modelo, denominado *Variable Angle Space Truss Model*, apresenta uma viga como uma combinação de quatro bielas, constituídas pelas armaduras longitudinais e, entre elas, quatro painéis constituídos de concreto. Nesse modelo, apresentado na Figura 3.3, “a divisão das funções estáticas é

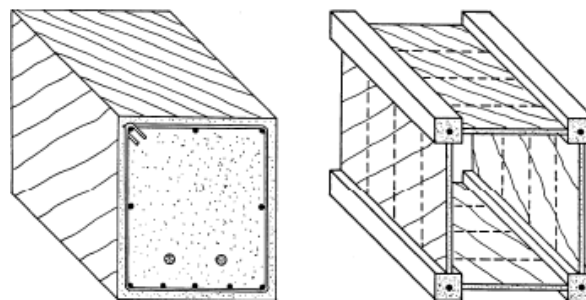
análoga aquela do MBP (momento e força normal suportados pelas bielas e cisalhamento e torção pelos painéis)” (SIMONE, 1998, p. 96).

Figura 3.2 – *Stringer Method* aplicado a uma parede com orifício: (a) modelo idealizado e (b) esforços nas bielas



Fonte: Kaern (1979).

Figura 3.3 – Idealização de uma viga fissurada em concreto armado

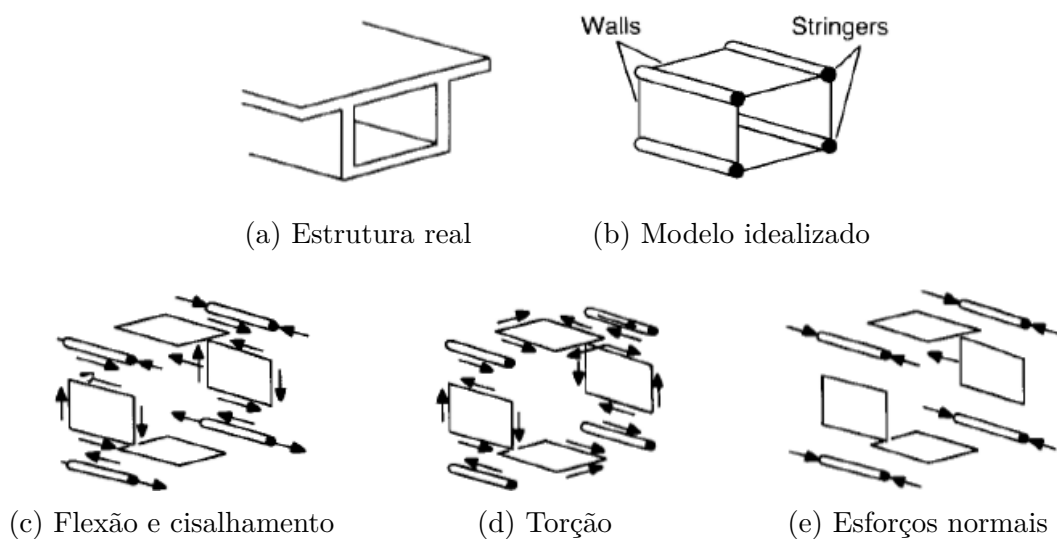


Fonte: Simone (1998).

O CEB-FIP Model Code 1990 (1993) recomenda que, na falta de um processo mais preciso, utilize-se um método chamado *Stringer and Wall* para estruturas formadas por paredes finas e submetidas a esforços normais, cortantes ou momentos fletores e torções. Uma exemplificação deste método, bem como a divisão da estrutura em bielas e painéis, são mostrados na Figura 3.4.

Na década de 1990, o desenvolvimento do método atingiu seu auge, principalmente através das pesquisas conduzidas no Politécnico de Milão e na Universidade Tecnológica de Delft. Dentre os principais trabalhos desse período pode-se citar as publicações de Blaauwendraad e Hoogenboom (1997), Hoogenboom (1993, 1998) e Simone (1998), as quais focaram no desenvolvimento computacional do método.

Damkilde e Krenk (1997) desenvolveram um programa, denominado *LimitS*, pelo qual era possível calcular o carregamento limite para dadas armaduras em modelos biela-painel e, através disso, otimizar o dimensionamento.

Figura 3.4 – Modelo *Stringer and Wall*

Fonte: CEB-FIP Model Code 1990 (1993).

O equacionamento matricial do MBP é apresentado nos trabalhos de Hoogenboom (1993) e Simone (1998). Esses autores também apresentam o processo de dimensionamento das armaduras dos modelos, a partir dos esforços nos elementos. O MBP também foi utilizado em programações não-lineares, sendo que Hoogenboom (1998) e Wang e Hoogenboom (2004) obtiveram bons resultados, comparando as capacidades de carga de simulações de estruturas analisadas anteriormente por ensaios experimentais. O equacionamento não-linear serviu de base para a criação do programa *SPanCAD* (BLAAUWENDRAAD; HOOGENBOOM, 1997; HOOGENBOOM, 1998), uma ferramenta capaz de fazer análises elásticas e não-lineares de modelos biela-painel.

O *SPanCAD* foi desenvolvido até o final da década de 1990, sendo que, para executar o programa, é necessário utilizar o programa *AutoCAD* (v. Release 14, 1997), pois se trata de um plugin para este último. Devido ao fato de esta ser uma versão obsoleta do *AutoCAD*, não é possível executá-la nas versões mais recentes do *Microsoft Windows*.

Após o final do século XX, o desenvolvimento do método cessou, existindo somente algumas aplicações do na literatura. Tarquini e Sgambi (2003) aplicaram o *SPanCAD* para o dimensionamento de uma viga convencional e um consolo curto, obtendo maiores taxas de armaduras distribuídas, em relação aos métodos convencionais.

Hauksdóttir (2007) aplicou o MBP ao dimensionamento de uma grande parede de cisalhamento (*shear wall*) submetida à atividade sísmica e, por meio da comparação com resultados obtidos por uma análise a partir do MEF, obteve valores próximos relacionados à quantidade de armaduras e aberturas de fissuras em serviço. Refer (2012) desenvolveu o equacionamento do MBP na plataforma MATLAB, criando um programa capaz de

otimizar o projeto de paredes de concreto armado no quesito econômico. Além disso, esse mesmo autor apresentou um roteiro manual de dimensionamento de estruturas pelo MBP.

No Brasil, o método é pouco divulgado, sendo possível dar destaque somente aos trabalhos de Silva (2004), Souza (2004, 2011, 2012), Mello (2015) e Mello e Souza (2016). Silva (2004) aplicou o MBP, na ocasião denominado por “Método dos Painéis Enrijecidos”, por meio da plataforma MATLAB, tanto para análises lineares quanto para não-lineares, obtendo taxas de armaduras relativamente menores, quando comparadas ao dimensionamento pelo MB.

Já Souza (2004), comparando o MBP e o MB, obteve uma armadura menor no tirante inferior da viga-parede analisada, porém, o primeiro método levou a uma taxa maior de estribos. Pesquisas posteriores também obtiveram esse mesmo resultado para as armaduras distribuídas (MELLO, 2015; MELLO; SOUZA, 2016), porém com armadura equivalente para o tirante principal.

Souza (2011, 2012), focou no desenvolvimento manual do MBP, que foi posteriormente abordado por Mello (2015) e Mello e Souza (2016). Foi mostrado que o modelo pode ser facilmente aplicado manualmente, sendo que, em estruturas simples, a aplicação de equações da estática é suficiente para obter os esforços solicitantes.

3.2 Descrição Geral do MBP

De acordo com Blaauwendraad (1994), o MBP é um modelo intermediário entre o MEF e o MB, sendo que: “[...] a armadura resultante consiste em uma ou mais faixas concentradas situadas próximas das bordas e uma malha distribuída na alma ou em grande parte dela, normalmente aplicada em duas direções ortogonais”. A diferença básica entre estes dois métodos é que, enquanto o MEF é aplicado para analisar uma estrutura a partir de uma malha com a discretização que gere melhor convergência, o MBP foi concebido para aplicar a malha mais grosseira a uma determinada geometria (BLAAUWENDRAAD; HOOGENBOOM, 2002).

O método tem como base o Teorema do Limite Inferior da Teoria da Plasticidade, assim, este se baseia no princípio de estabelecer um fluxo de tensões que seja estaticamente admissível e que esteja dentro da superfície de escoamento do material (NIELSEN; HOANG, 2011). Além disso, de acordo com Hauksdóttir (2007), o MBP pode ser utilizado para qualquer material em que esta teoria for aplicável.

De acordo com Souza (2004), o método introduz armaduras principais nas direções das cargas e apoios, e também armaduras secundárias, em forma de malha, no interior dos painéis. Seu funcionamento é parecido com o do MB, sendo que os elementos devem ser dispostos de modo que se obtenha um caminho realista para os esforços.

O modelo consiste na divisão da estrutura bidimensional de concreto em dois elementos distintos: as bielas e os painéis. As bielas têm o objetivo de transmitir as forças normais, sejam elas de tração ou compressão, podendo estar dispostas tanto no sentido horizontal, quanto no vertical (SOUZA, 2004), ou até mesmo em eixos inclinados, no caso de a estrutura possuir seção transversal variável. Uma biela trabalha em regime axial, sujeita a duas forças normais nas extremidades e forças tangenciais distribuídas ao longo do seu eixo (SIMONE, 1998). Normalmente elas devem existir no contorno do elemento estrutural ou de aberturas existentes ao longo do vão, assim como nas linhas das reações de apoio e carregamentos externos.

Quanto à aplicação das bielas, Souza (2004, p. 69) destaca:

...se uma biela está sendo comprimida a tensão no concreto deve ser verificada e, caso ultrapasse a tensão efetiva do material, deve-se prever armaduras para o confinamento da biela, visando aumentar a resistência à compressão da mesma. [...] por outro lado, se uma biela estiver sendo tracionada, despreza-se a resistência do concreto à tração e determina-se a quantidade de armaduras necessárias para combater a força normal atuante na biela.

Os painéis são elementos quadrilaterais que se encontram sempre entre 4 bielas, de modo a absorver os esforços cortantes. Eles trabalham em regime de membrana de cisalhamento puro, de intensidade uniforme (SIMONE, 1998) e, para haver equilíbrio entre os elementos, as forças de cisalhamento dos painéis devem estar equilibradas com as forças axiais das bielas adjacentes. Nos casos mais comuns, os painéis têm geometria retangular, porém, no caso de estruturas com seção variável, eles podem ter formato trapezoidal.

A Figura 3.5 apresenta a discretização de uma viga de acordo com o MBP, assim como as condições de equilíbrio entre os elementos (Figura 3.5a). Na Figura 3.5b, é possível visualizar a configuração típica destes dois tipos de elementos e as armaduras resultantes.

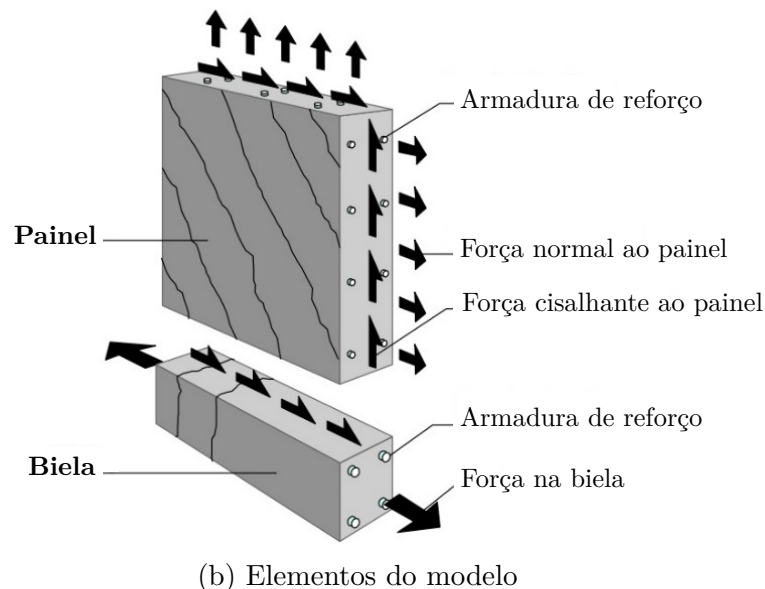
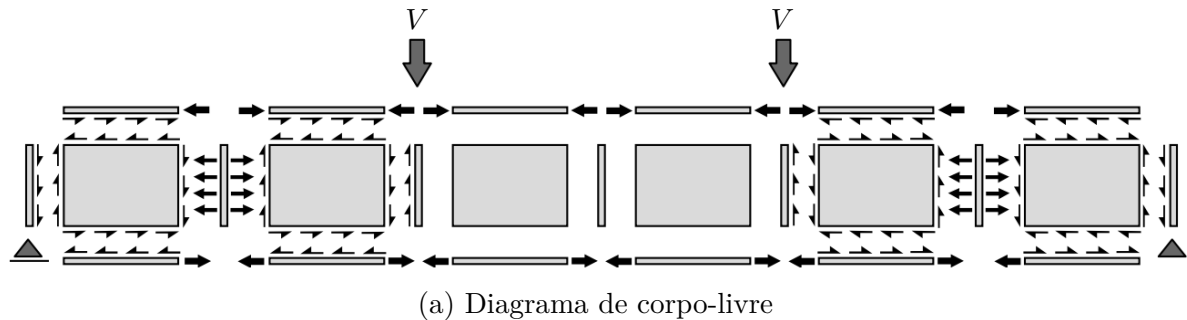
Segundo Blaauwendraad e Hoogenboom (1997) existe apenas uma força cortante (v) num painel, a qual possui o mesmo valor por unidade de comprimento em qualquer posição do painel. Essa força ainda trabalha na interface entre os painéis e as bielas e, de acordo com as configurações de equilíbrio, a força normal da biela pode aumentar ou diminuir linearmente, assim como mostrado na Figura 3.6.

A Figura 3.7 mostra um modelo de uma viga-parede que foi tratada por Hoogenboom (1993) utilizando o MBP, bem como os esforços obtidos na análise linear dessa estrutura.

De acordo com Hoogenboom (1998), existem duas versões do modelo, uma linear e outra não-linear. Na primeira, o comportamento dos materiais, tanto para as bielas, quanto para os painéis, é elástico-linear e, para os painéis, considera-se a hipótese de que estes absorvam somente forças cisalhantes. Desse modo, obtém-se um maior braço de alavanca para o momento na seção transversal, visto que as forças normais serão inteiramente

absorvidas pelas bielas. No modelo não-linear também se admite que os painéis absorvam esforços normais, além de cisalhantes, o que o torna mais próximo da situação real.

Figura 3.5 – Modelo Biela-Painel para uma viga biapoada



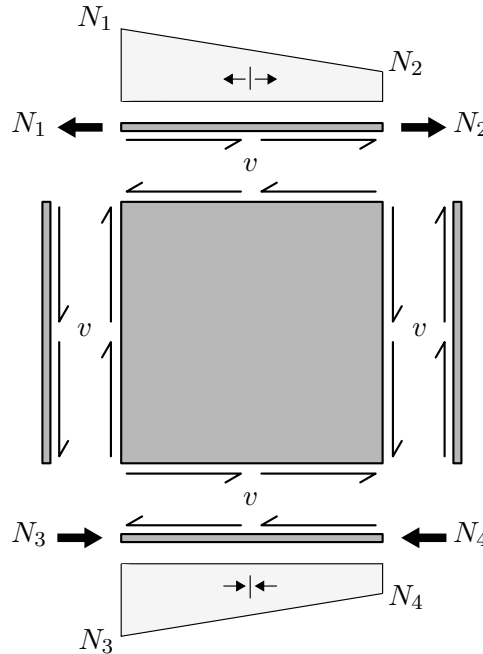
Fonte: adaptado de Blaauwendraad e Hoogenboom (1997).

Para as vigas-parede analisadas anteriormente (MELLO, 2015), o dimensionamento por ambos os métodos (MB e MBP) resultou na mesma quantidade de armadura nos tirantes. A diferença básica encontrada entre os dois métodos foi que, pelo MBP, é possível dimensionar uma armadura distribuída em malha por meio da força cisalhante em cada painel e, pelo MB, normalmente se distribui somente uma armadura mínima para “costurar” as escoras de concreto, sujeitas a tensões transversais de tração.

Segundo Silva (2004), o processo de aplicação do MBP é simples e intuitivo. Primeiramente, define-se o modelo biela-painel da estrutura e faz-se uma análise linear deste, na qual se admite o comportamento elástico-linear tanto para o concreto quanto para o aço. Com os resultados da primeira análise, é feito o dimensionamento das armaduras necessárias a resistir aos esforços obtidos e, também, as tensões de compressão no concreto são verificadas. Após isso, é feita uma análise não-linear, admitindo-se o comportamento

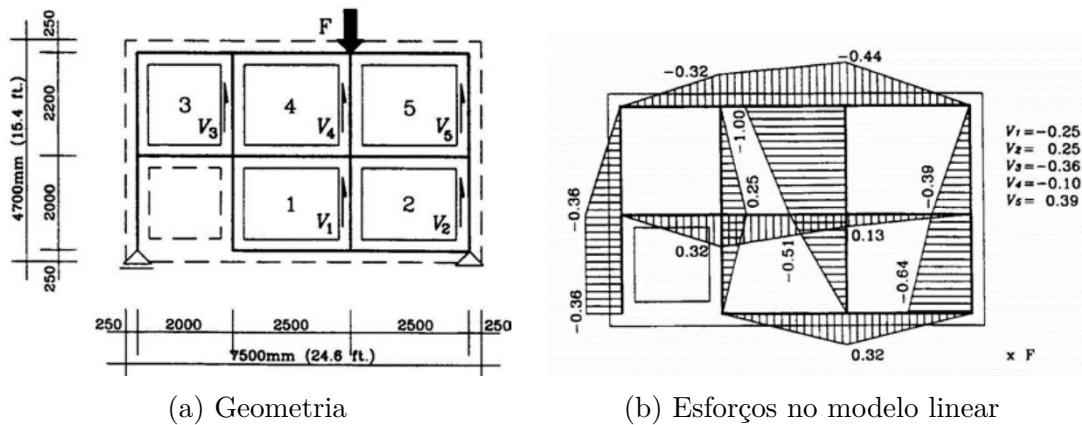
conjunto dos materiais e a perda de rigidez devido aos esforços, obtendo, assim, esforços mais realistas e novas armaduras.

Figura 3.6 – Comportamento linear das forças normais nas bielas



Fonte: adaptado de Hoogenboom (1993).

Figura 3.7 – Viga-parede com furo na alma investigada pelo MBP

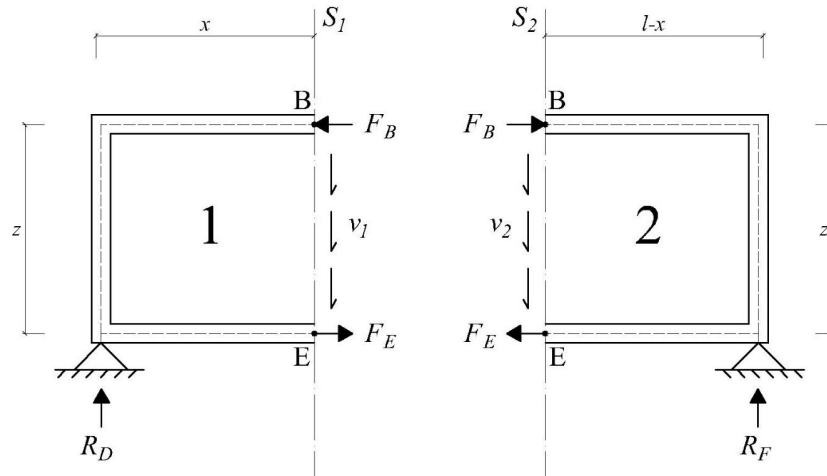


Fonte: Hoogenboom (1993).

A vantagem de utilização do MBP é relacionada à sua facilidade de aplicação manual (SOUZA, 2012), trazendo resultados mais rápidos que o MB, de modo que não é necessária tanta experiência para a definição do caminho das cargas. Nos modelos mais simples, como o caso da viga-parede ilustrada na Figura 3.8, a determinação dos esforços nos elementos pode ser feita rapidamente por meio de uma simples aplicação das equações da estática. Uma descrição completa do procedimento manual já foi abordada anteriormente (MELLO, 2015).

Na Figura 3.8 é possível verificar que existe uma altura efetiva (h_e) do painel, isto se deve ao fato de que a tensão de cisalhamento será distribuída na distância compreendida entre as linhas centrais das bielas adjacentes. Em outras palavras, o equilíbrio entre um painel e as bielas que se encontram em seu entorno se dá em toda a extensão da biela, que, neste caso, é simbolizada por h_e .

Figura 3.8 – Esforços atuantes num modelo de viga-parede

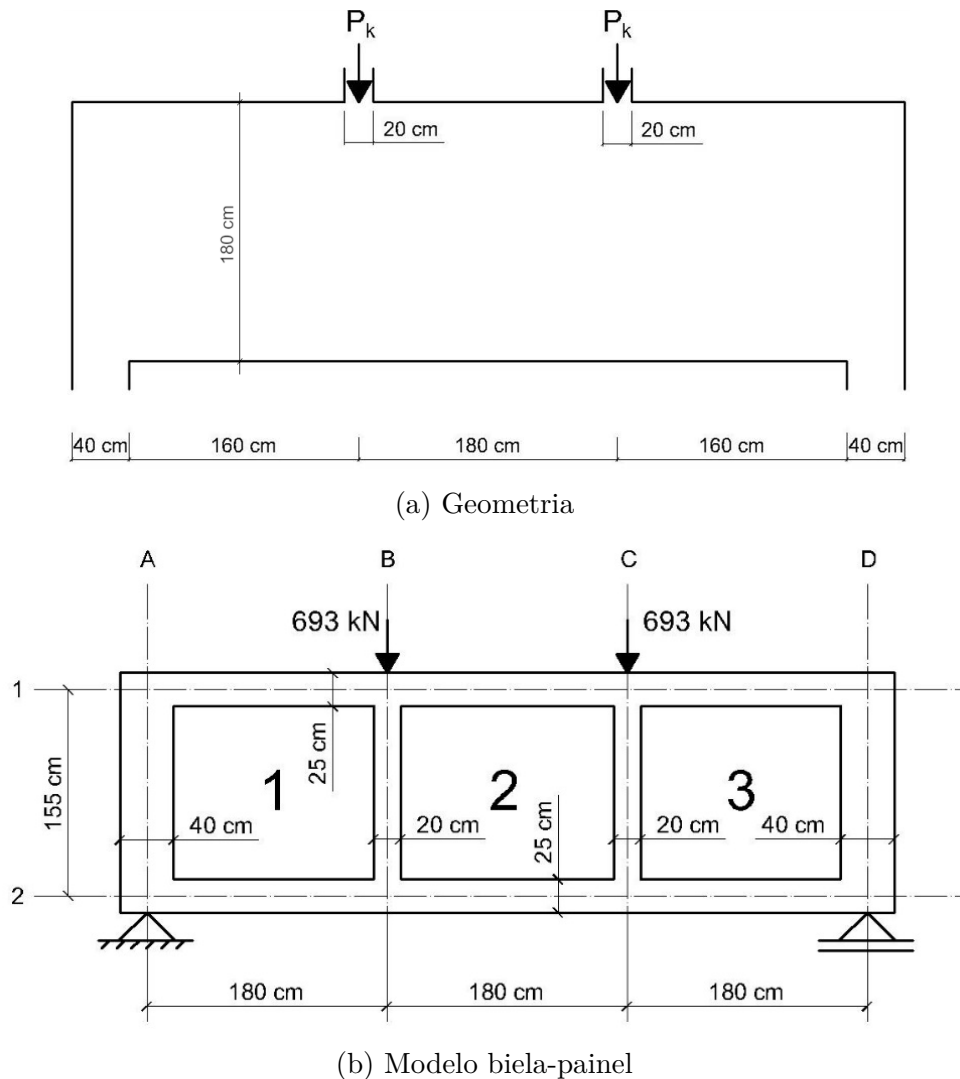


Fonte: Mello (2015).

Tendo em mãos a geometria do elemento estrutural, o processo de definição do modelo se inicia pelo cálculo da altura das bielas principais. Por exemplo, no caso de uma viga-parede, a ABNT NBR 6118:2014 recomenda que a armadura de flexão seja distribuída numa altura de 15% da altura total da viga, portanto, uma estimativa inicial da altura das bielas inferior e superior seria $0,15 h$. Para outros elementos estruturais, pode-se utilizar as recomendações normativas para modelagem pelo MB.

Dando continuidade à definição do modelo, obtém-se a geometria das bielas secundárias e, então, são definidos os painéis nos vazios localizados entre as bielas. A Figura 3.9 apresenta um exemplo de modelo biela-painel para uma viga-parede.

Figura 3.9 – Exemplo de modelagem de uma viga-parede pelo MBP



Fonte: Mello (2015).

3.3 Modelagem utilizando o MEF

O Método dos Elementos Finitos (MEF) é, sem dúvida, a ferramenta mais usual e completa para análises lineares e não-lineares de elementos estruturais sujeitos aos mais diversos tipos de carregamento. O princípio básico do método é substituir um problema complexo por um mais simples, sendo que uma estrutura é idealizada a partir da combinação de vários elementos interconectados, denominados elementos finitos, os quais darão uma solução aproximada para o problema real (RAO, 2011).

Uma solução pelo MEF gera uma série de equações algébricas que devem ser resolvidas (BATHE, 1996) e, devido à ineficácia da resolução destas equações manualmente, o método só se tornou uma alternativa prática após o surgimento dos computadores.

O procedimento de solução de uma estrutura pelo MEF é definido, de acordo com Rao (2011), pelas seguintes etapas:

- (a) discretização: divisão da estrutura em elementos discretos, sendo que seu número, tipo, tamanho e disposição devem ser definidos;
- (b) seleção de um modelo de interpolação ou deslocamento: assume-se uma solução para cada elemento, de modo a obter uma solução geral aproximada. Esta solução, que normalmente é polinomial, deve ser simples do ponto de vista computacional, mas satisfazer certos critérios de convergência;
- (c) obtenção das matrizes de rigidez dos elementos e vetores de carregamento, a partir de um princípio variacional adequado, uma abordagem residual ponderada (como o método de Galerkin) ou condições de equilíbrio;
- (d) reunião das equações dos elementos para obtenção das equações de equilíbrio globais: as matrizes de rigidez de cada elemento e vetores de carga são reunidos numa única matriz, que descreverá o comportamento da estrutura;
- (e) solução dos deslocamentos nodais desconhecidos: as equações de equilíbrio globais são modificadas para considerar as condições de contorno do problema. Em problemas lineares, isso é feito facilmente, porém, em problemas não-lineares, a solução deve ser obtida a partir de uma sequência de passos, sendo que em cada passo envolve a modificação da matriz de rigidez e do vetor de carregamento;
- (f) cálculo das tensões e deformações do elemento: a partir dos deslocamentos nodais, as tensões e deformações são calculadas utilizando as equações da Mecânica dos Sólidos e das Estruturas.

A solução do sistema global de equações pelo MEF consiste em:

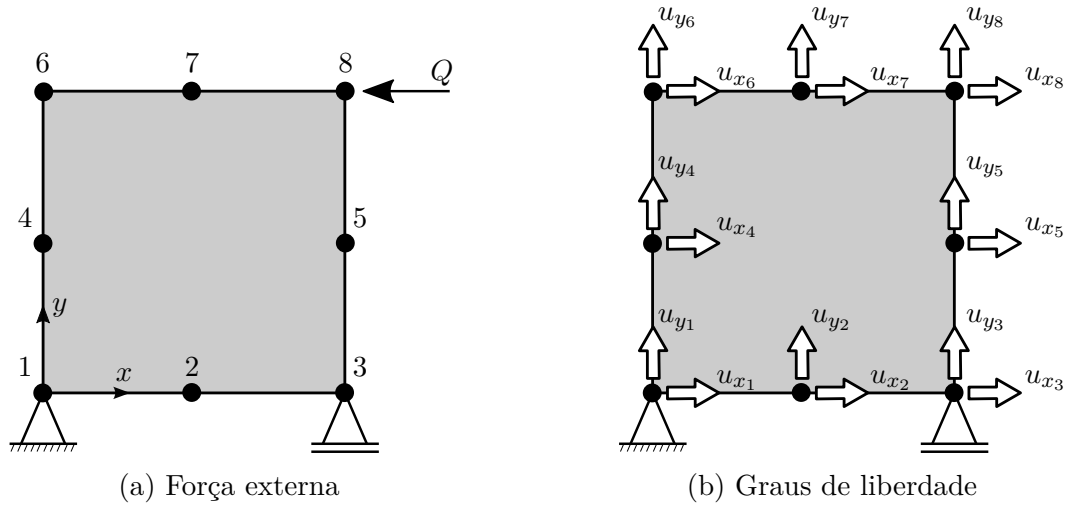
$$F = K U \quad (3.1)$$

sendo: F: vetor de forças nodais externas;
 U: vetor de deslocamentos nodais;
 K: matriz de rigidez global do sistema.

A formulação baseada no MEF para bielas e painéis foi apresentada por Hoogenboom (1998) e adaptada para os modelos utilizados neste trabalho, sendo descrita nos apêndices A e B.

Como exemplo de obtenção das variáveis para um modelo biela-painel, toma-se aquele ilustrado na Figura 3.10. Trata-se de um modelo simples composto por um painel e quatro bielas no seu contorno. Existe uma força, na direção x , aplicada no nó 8, enquanto existem restrições para o deslocamento em y nos nós 1 e 3 e, em x , no nó 1.

Figura 3.10 – Exemplo de modelo biela-painel



3.3.1 Vetor de forças

O vetor de forças terá tamanho de duas vezes o número de nós do sistema (forças em x e y). Para o modelo considerado (Figura 3.10a), o vetor terá todos os elementos, exceto o correspondente à força em x no nó 8, iguais a zero:

$$F_{[16 \times 1]} = \begin{Bmatrix} f_{x1} \\ f_{y1} \\ f_{x2} \\ f_{y2} \\ f_{x3} \\ f_{y3} \\ \vdots \\ f_{x8} \\ f_{y8} \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ -Q \\ 0 \end{Bmatrix} \quad (3.2)$$

sendo que o sinal negativo de Q indica direção contrária ao sentido do eixo x .

3.3.2 Vetor de deslocamentos

O vetor de deslocamentos, de mesmo tamanho que o vetor F , será a solução do sistema. Pelas condições de contorno do modelo (Figura 3.10b), sabe-se que:

$$u_{x1} = u_{y1} = u_{y3} = 0$$

O vetor U será, portanto:

$$U_{[16 \times 1]} = \begin{Bmatrix} u_{x1} \\ u_{y1} \\ u_{x2} \\ u_{y2} \\ u_{x3} \\ u_{y3} \\ \vdots \\ u_{x8} \\ u_{y8} \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ u_{x2} \\ u_{y2} \\ u_{x3} \\ 0 \\ \vdots \\ u_{x8} \\ u_{y8} \end{Bmatrix} \quad (3.3)$$

3.3.3 Matriz de rigidez global

A matriz de rigidez global, quadrada e de ordem de duas vezes o número de nós, será composta pela soma das matrizes de rigidez local de cada elemento, sendo cada componente da matriz acoplado na sua correspondente posição nodal. Por exemplo, o painel confere rigidez aos nós 2, 4, 5 e 7, a biela inferior, aos nós 1, 2 e 3, e assim sucessivamente.

Para o modelo considerado (Figura 3.10), a matriz de rigidez será:

$$K_{[16 \times 16]} = \begin{bmatrix} k_{11} & k_{12} & k_{13} & k_{14} & \dots & k_{115} & k_{116} \\ k_{21} & k_{22} & k_{23} & k_{24} & \dots & k_{215} & k_{216} \\ \vdots & \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ k_{151} & k_{152} & k_{153} & k_{154} & \dots & k_{1515} & k_{1516} \\ k_{161} & k_{162} & k_{163} & k_{164} & \dots & k_{1615} & k_{1616} \end{bmatrix} \quad (3.4)$$

Usando como exemplo a biela vertical da esquerda, que tem como graus de liberdade os nós 1, 4 e 6, e a mesma numeração nodal adotada para o sistema global, sua matriz de rigidez local, utilizando a formulação apresentada no Apêndice A, será:

$$K_b = \begin{bmatrix} k_{11}^b & k_{12}^b & k_{17}^b & k_{18}^b & k_{111}^b & k_{112}^b \\ k_{21}^b & k_{22}^b & k_{27}^b & k_{28}^b & k_{211}^b & k_{212}^b \\ k_{71}^b & k_{72}^b & k_{77}^b & k_{78}^b & k_{711}^b & k_{712}^b \\ k_{81}^b & k_{82}^b & k_{87}^b & k_{88}^b & k_{811}^b & k_{812}^b \\ k_{111}^b & k_{112}^b & k_{117}^b & k_{118}^b & k_{1111}^b & k_{1112}^b \\ k_{121}^b & k_{122}^b & k_{127}^b & k_{128}^b & k_{1211}^b & k_{1212}^b \end{bmatrix}, \quad \text{com: } U_b = \begin{Bmatrix} u_{x1} \\ u_{y1} \\ u_{x4} \\ u_{y4} \\ u_{x6} \\ u_{y6} \end{Bmatrix} \quad (3.5)$$

Do mesmo modo, para o painel, que tem como graus de liberdade os nós 2, 5, 7 e 4 (seguindo a ordem apresentada na Apêndice B), terá como matriz de rigidez:

$$K_p = \begin{bmatrix} k_{33}^p & k_{34}^p & k_{39}^p & k_{310}^p & k_{313}^p & k_{314}^p & k_{37}^p & k_{38}^p \\ k_{43}^p & k_{44}^p & k_{49}^p & k_{410}^p & k_{413}^p & k_{414}^p & k_{47}^p & k_{48}^p \\ k_{93}^p & k_{94}^p & k_{99}^p & k_{910}^p & k_{913}^p & k_{914}^p & k_{97}^p & k_{98}^p \\ k_{103}^p & k_{104}^p & k_{109}^p & k_{1010}^p & k_{1013}^p & k_{1014}^p & k_{107}^p & k_{108}^p \\ k_{133}^p & k_{134}^p & k_{139}^p & k_{1310}^p & k_{1313}^p & k_{1314}^p & k_{137}^p & k_{138}^p \\ k_{143}^p & k_{144}^p & k_{149}^p & k_{1410}^p & k_{1413}^p & k_{1414}^p & k_{147}^p & k_{148}^p \\ k_{73}^p & k_{74}^p & k_{79}^p & k_{710}^p & k_{713}^p & k_{714}^p & k_{77}^p & k_{78}^p \\ k_{83}^p & k_{84}^p & k_{89}^p & k_{810}^p & k_{813}^p & k_{814}^p & k_{87}^p & k_{88}^p \end{bmatrix}, U_p = \begin{Bmatrix} u_{x2} \\ u_{y2} \\ u_{x5} \\ u_{y5} \\ u_{x7} \\ u_{y7} \\ u_{x4} \\ u_{y4} \end{Bmatrix} \quad (3.6)$$

Observa-se que em nas equações (3.5) e (3.6) existem os elementos k_{77} , k_{78} , k_{87} e k_{88} , devido à biela e ao painel terem o nó 4 como grau de liberdade. Os elementos de cada matriz serão então somados para a posição na matriz global, ou seja:

$$\begin{aligned} k_{77} &= k_{77}^b + k_{77}^p & k_{78} &= k_{78}^b + k_{78}^p \\ k_{87} &= k_{87}^b + k_{87}^p & k_{88} &= k_{88}^b + k_{88}^p \end{aligned}$$

Resumidamente, chamando de i o número do nó em questão, os elementos que tenham este nó como grau de liberdade terão suas matrizes de rigidez somadas nas linhas $2i - 1$ e $2i$, e também nas colunas $2i - 1$ e $2i$, da matriz global do sistema.

3.3.4 Condições de contorno

De acordo com Rao (2011), as condições de contorno podem ser incluídas no sistema por meio de uma simplificação da matriz de rigidez global e do vetor de deslocamentos. Este passo é essencial para a solução do problema, já que aqui são retirados os movimentos de corpo rígido.

Considerando que se conheça o deslocamento U_j , podendo este ser nulo (no caso de existência de apoio), substitui-se todos os elementos da linha j e coluna j da matriz K por zero, exceto o elemento da diagonal, que será unitário:

$$K_{ij} = K_{ji} = 0 \quad (3.7a)$$

$$K_{jj} = 1 \quad (3.7b)$$

O valor de U_j é inserido no vetor de deslocamentos, na linha j :

$$F_j = U_j \quad (3.7c)$$

Dessa forma, o sistema é “forçado” a resultar em:

$$U_j = U_j \quad (3.7d)$$

Para o exemplo em questão (Figura 3.10), a simplificação pode ser aplicada para as linhas 1 ($u_{x1} = 0$), 2 ($u_{y1} = 0$) e 6 ($u_{y3} = 0$).

No caso do modelo linear, esta simplificação também deve ser aplicada às linhas que correspondem aos graus de liberdade perpendiculares às faces dos painéis, visto que estes (e também as bielas adjacentes) fornecem apenas rigidez na direção paralela às faces. Caso isso não seja feito, o sistema não terá solução, devido às linhas e colunas correspondentes a tal deslocamento serem formadas por zeros.

No caso considerado, estes graus de liberdade são u_{y2} , u_{x4} , u_{x5} e u_{y7} . Portanto, a simplificação resultará em:

$$k_{44} = k_{77} = k_{99} = k_{1414} = 1$$

3.3.5 Solução do sistema

Após a montagem da matriz de rigidez e com o vetor F conhecido, procura-se calcular o vetor U :

$$U = K^{-1} F \quad (3.8)$$

Com a obtenção do vetor de deslocamentos, é feito o caminho inverso, a partir dele se calculam os esforços locais de interesse em cada elemento do modelo a partir das equações definidas nos apêndices A e B.

3.3.6 Solução de equações não-lineares

Para o modelo não-linear, a solução do sistema não se torna tão trivial, pois o vetor de forças será formado por funções não-lineares dos deslocamentos (KIM, 2015):

$$F = P(U) = \begin{Bmatrix} p_1(U) \\ p_2(U) \\ p_3(U) \\ \vdots \\ p_n(U) \end{Bmatrix} \quad (3.9)$$

Segundo Kim (2015), todos os métodos de solução para estas funções não-lineares são iterativos. O processo se inicia com uma estimativa U_0 , um incremento ΔU é obtido pela solução de um sistema de equações lineares e a solução se atualiza repetidamente até um critério de convergência ser satisfeito. Dentre os métodos disponíveis para este tipo de solução pode se dar destaque ao Método de Newton-Raphson e o Método de Newton-Raphson Modificado.

3.3.6.1 Método de Newton-Raphson

Considerando que a solução aproximada para a i -ésima iteração seja U^i , a solução para a próxima iteração é aproximada por (KIM, 2015):

$$P(U^{i+1}) \approx P(U^i) + K_T^i(U^i) \Delta U^i = F \quad (3.10)$$

sendo $K_T^i(U^i)$ a matriz de rigidez tangente e ΔU^i o incremento de solução.

Rearranjando (3.10), obtém-se o sistema de equações linearizadas:

$$K_T^i \Delta U^i = F - P(U^i) \quad (3.11)$$

A matriz K_T^i não será constante, mas sim uma função de U^i que é obtida a partir das funções de deslocamento para cada elemento. Resolvendo (3.11), tem-se o incremento de deslocamentos e a nova aproximação será obtida por:

$$U^{i+1} = U^i + \Delta U^i \quad (3.12)$$

Como o processo não resultará na solução exata do sistema, haverá forças residuais (R), que serão obtidas pela diferença entre o vetor de forças conhecido e a solução obtida:

$$R^i = P(U^i) - F \quad (3.13)$$

As forças residuais serão, então, redistribuídas aos elementos, respeitando seus limites de resistência, por meio de um incremento de deformações, obtido manipulando (3.11):

$$K_T^i \Delta U^i = -R^i \quad (3.14)$$

O processo é finalizado quando a convergência tende a zero. O critério de convergência pode ser expresso pela seguinte forma normalizada:

$$conv = \frac{\sum_{j=1}^n (R_j^i)^2}{1 + \sum_{j=1}^n (F_j)^2} \leq tol \quad (3.15)$$

sendo tol a tolerância especificada (0,01, por exemplo).

A constante 1 é adicionada ao denominador para evitar divisão por zero. Se as forças residuais forem inferiores a uma tolerância especificada, a solução U^{i+1} é aceita e o processo é finalizado. Caso contrário, a solução é atualizada por (3.12) até estas forças se tornarem pequenas o suficiente (KIM, 2015).

Alternativamente, um critério de convergência baseado no incremento de deslocamentos pode ser utilizado. Nesse caso, considera-se atingida a convergência quando o

incremento da i -ésima iteração for muito pequeno em comparação ao incremento inicial (KIM, 2015), isto é:

$$conv = \frac{\sum_{j=1}^n (\Delta U_j^i)^2}{1 + \sum_{j=1}^n (\Delta U_j^0)^2} \leq tol \quad (3.16)$$

Para cada iteração, a matriz de rigidez tangente, ou matriz jacobiana, deve ser atualizada de acordo com a equação apresentada por Reddy (2014):

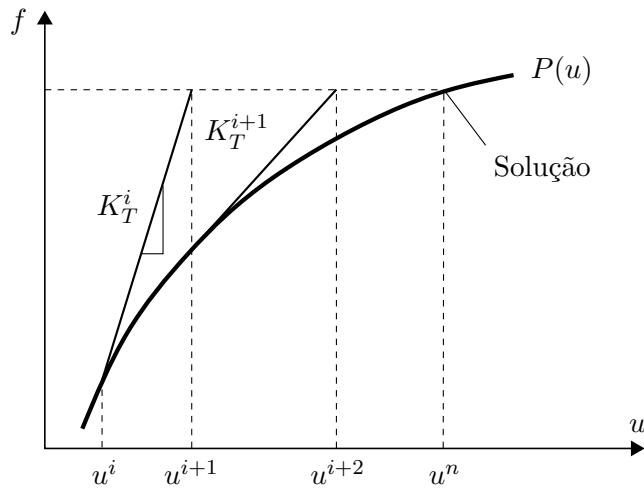
$$(K_T)_{ij} = \frac{\partial R_i}{\partial U_j} = K_{ij} + \sum_{k=1}^n \frac{\partial K_{ik}}{\partial U_j} U_k \quad (3.17)$$

sendo n o número de graus de liberdade do elemento.

Esta atualização deve ser primeiramente efetuada para cada elemento, a partir da matriz de rigidez e deslocamentos em seu sistema de coordenadas local. A matriz global é então montada de acordo com o processo descrito em 3.3.3.

A Figura 3.11 apresenta graficamente o processo de solução pelo Método de Newton-Raphson.

Figura 3.11 – Método de Newton-Raphson



Fonte: adaptado de Kim (2015).

3.3.6.2 Método de Newton-Raphson Modificado

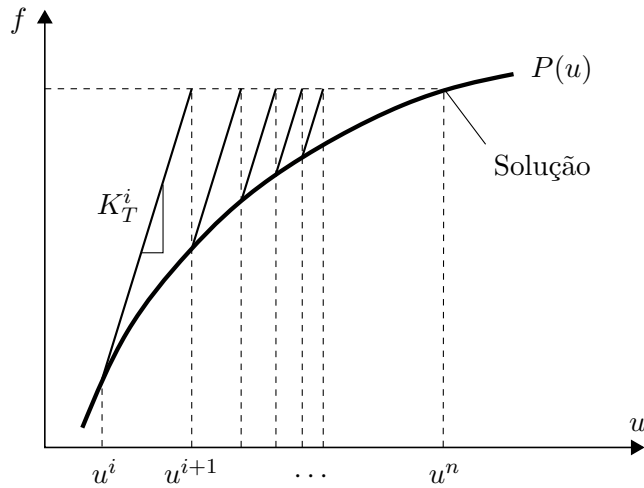
O método modificado se difere do Método de Newton-Raphson original por não haver a atualização da matriz de rigidez tangente em todas as iterações. A matriz inicial é utilizada até se conseguir a convergência do sistema de equações. Dessa forma, a solução requer um número maior de iterações, porém, o custo computacional se torna menor devido à cada iteração ser resolvida com maior agilidade. Esse método é também um pouco mais estável computacionalmente que o original (KIM, 2015).

Esta solução é útil em análises de resposta da estrutura, nas quais se procura a resposta da estrutura conforme o carregamento aumenta até atingir um valor limite especí-

ficado. A solução é obtida da mesma forma que para o método convencional, excetuando-se a atualização da matriz de rigidez tangente.

A Figura 3.12 apresenta graficamente o processo de solução pelo Método de Newton-Raphson Modificado.

Figura 3.12 – Método de Newton-Raphson Modificado



Fonte: adaptado de Kim (2015).

3.3.6.3 Método Secante Incremental

O Método Secante Incremental surge para suprimir duas tarefas que os métodos anteriores utilizam: a montagem da matriz de rigidez e a solução do sistema, os quais demandam bastante custo computacional. A ideia desse método é aproximar a matriz de rigidez através da atualização progressiva a matriz tangente pela direção secante entre duas soluções consecutivas. Além disso, pode-se reduzir ainda mais a demanda computacional usando diretamente a inversa da matriz tangente (KIM, 2015). Esta solução pode ser visualizada graficamente na Figura 3.13.

A matriz secante pode ser definida pelo método das diferenças finitas, por:

$$K_s^i = \frac{P(U^i) - P(U^{i-1})}{U^i - U^{i-1}} \quad (3.18)$$

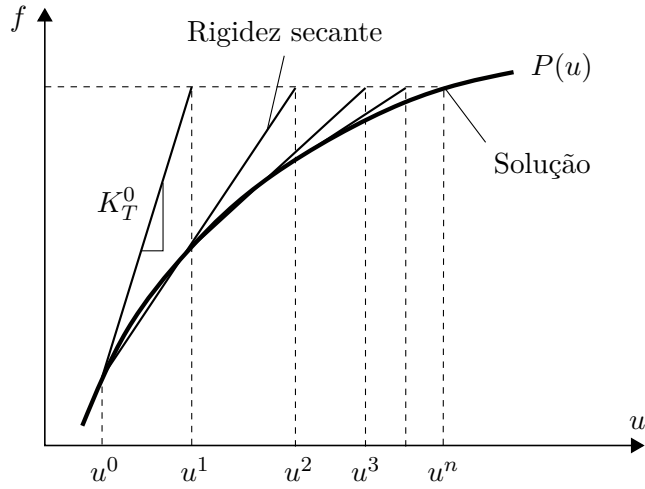
Quando U^{i-1} se aproxima de U^i , a matriz secante será aproximadamente igual à matriz tangente obtida pelo Método de Newton-Raphson. O incremento de solução (ΔU^i) será dado pela solução do sistema:

$$K_s^i \Delta U^i = -R^i \quad (3.19)$$

com:

$$R^i = P(U^i) - F$$

Figura 3.13 – Método Secante Incremental



Fonte: adaptado de Kim (2015).

sendo K_s^i a matriz de rigidez secante e R^i o vetor de força residual da i -ésima iteração.

A atualização da matriz secante pode ser feita pelo Método de Broyden, fazendo:

$$K_s^i = K_s^{i-1} + \frac{\Delta R - K_s^{i-1} \Delta U}{\|\Delta U\|^2} \Delta U^T \quad (3.20)$$

sendo:

$$\Delta R = R(U^i) - R(U^{i-1}) \quad \Delta U = U^i - U^{i-1}$$

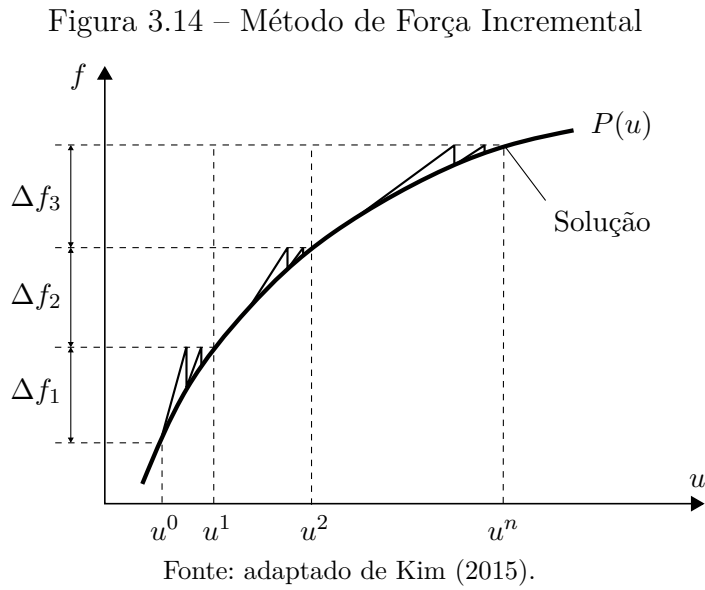
Os deslocamentos são então atualizados por aplicação de (3.12) e (3.14) e o processo iterativo se repete até que o vetor residual satisfaça o critério de convergência, dado em (3.15).

3.3.6.4 Método de Força Incremental

Em problemas não-lineares, a relação entre forças e deslocamentos não seguem uma proporcionalidade como em sistemas lineares. Como a estimativa inicial de deslocamentos parte de zero, os métodos de solução convergem rapidamente quando a força aplicada é pequena. À medida que as forças aumentam, a convergência se torna mais difícil (KIM, 2015).

A solução para esse problema é dividir o carregamento em vários incrementos, sendo que, em cada um deles, os deslocamentos são calculados pelos processos de solução vistos anteriormente. Após encontrar a solução de um incremento, esta será utilizada como estimativa inicial para o próximo, sucessivamente até atingir o carregamento total. Este processo é útil para determinar a resposta da estrutura para vários níveis de carregamento (KIM, 2015).

A Figura 3.14 apresenta graficamente a solução de um problema pelo Método de Força Incremental.



3.3.6.5 Método do Comprimento de Arco

Em problemas não-lineares, quando a estrutura atinge seu comportamento de pico, a tendência é de aumento de deformações, enquanto o esforço segue constante ou, até mesmo, diminui. Nesta etapa, quando se utiliza o Método de Força Incremental com acréscimos constantes, as forças residuais tendem a aumentar e a convergência não é atingida.

Uma forma de contornar este problema é utilizar o Método do Comprimento de Arco, no qual os deslocamentos e o fator de carga λ são desconhecidos e os pontos de equilíbrio são obtidos pela interseção de uma reta de comprimento Δs , tangente ao diagrama de força $\times$ deformação (REDDY, 2014). Este método tem grande aplicabilidade em problemas de geometria não-linear.

A formulação deduzida a seguir foi proposta por Crisfield (1981) e descrita por Reddy (2014), sendo que a representação gráfica deste método pode ser visualizada na Figura 3.15.

O vetor de força externa F num determinado passo de carga será dependente do fator de carga λ :

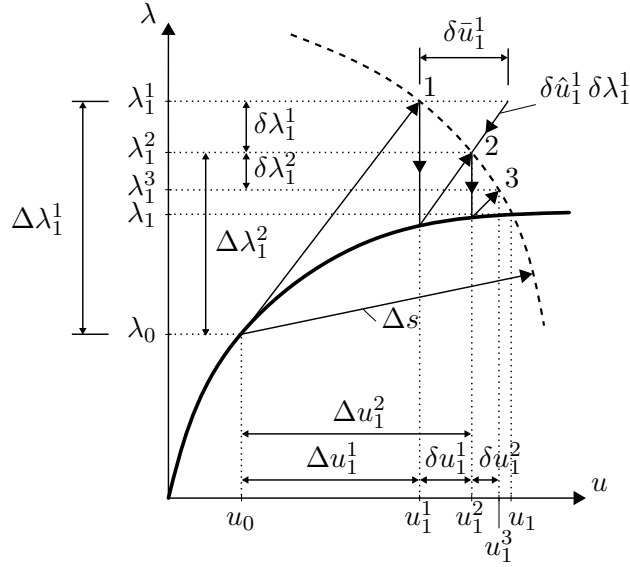
$$F = \lambda \bar{F} \quad (3.21)$$

sendo $\bar{F}$ o carregamento total aplicado.

O vetor residual será então calculado por:

$$R^i = P(U^i) - \lambda \bar{F} = K_T U - \lambda \bar{F} \quad (3.22)$$

Figura 3.15 – Método do Comprimento de Arco



Fonte: adaptado de Reddy (2014).

Num enésimo passo de carga, para um pequeno aumento de carga $\delta\lambda_n^i$, o aumento de deslocamento δU_n^i pode ser obtido por:

$$\delta U_n^i = -K_T^{-1} R^{i-1} + \delta\lambda_n^i K_T^{-1} \bar{F} \quad (3.23a)$$

ou:

$$\delta U_n^i = \delta \bar{U}_n^i + \delta\lambda_n^i \delta \hat{U}_n^i \quad (3.23b)$$

com:

$$\delta \bar{U}_n^i = -K_T^{-1} R^{i-1} \quad \delta \hat{U}_n^i = K_T^{-1} \bar{F}$$

A cada iteração, são atualizados o incremento de deslocamento no passo de carga ΔU_n e o deslocamento total U_n :

$$\Delta U_n^i = \Delta U_n^{i-1} + \delta U_n^i \quad (3.24a)$$

$$U_n = U_{n-1} + \Delta U_n^i \quad (3.24b)$$

No início primeiro passo de carga, assume-se um incremento inicial $\delta\lambda_1^0$; o comprimento de arco Δs_0 é, então, obtido por:

$$\Delta s_0 = \delta\lambda_1^0 \sqrt{\delta \hat{U}_1^T \delta \hat{U}_1} \quad (3.25)$$

Para os passos subsequentes, o comprimento de arco Δs_n será:

$$\Delta s_n = \Delta s_{n-1} \frac{I_d}{I_{n-1}} \quad (3.26)$$

sendo I_d o número desejado de iterações dentro do passo de carga; I_{n-1} o número requerido de iterações no passo anterior; e:

$$\Delta s_{n-1} = \sqrt{\delta \hat{\mathbf{U}}_{n-1}^T \delta \hat{\mathbf{U}}_{n-1}} \quad (3.27)$$

Substituindo (3.24a) em (3.27) e utilizando a definição de $\delta \mathbf{U}_n^i$ (3.23b), escreve-se a seguinte equação do segundo grau para determinação do incremento $\delta \lambda_n^i$:

$$a_1 \left(\delta \lambda_n^i \right)^2 + 2 a_2 \delta \lambda_n^i + a_3 = 0 \quad (3.28)$$

com:

$$\begin{aligned} a_1 &= \left(\delta \hat{\mathbf{U}}_n^i \right)^T \delta \hat{\mathbf{U}}_n^i \\ a_2 &= \left(\Delta \mathbf{U}_n^{i-1} + \delta \bar{\mathbf{U}}_n^i \right)^T \delta \hat{\mathbf{U}}_n^i \\ a_3 &= \left(\Delta \mathbf{U}_n^{i-1} + \delta \bar{\mathbf{U}}_n^i \right)^T \left(\Delta \mathbf{U}_n^{i-1} + \delta \bar{\mathbf{U}}_n^i \right) - \Delta s_n^2 \end{aligned}$$

Esta equação resultará em duas soluções: $\delta \lambda_{n_1}^i$ e $\delta \lambda_{n_2}^i$, as quais serão usadas para calcular os incrementos correspondentes: $\Delta \mathbf{U}_{n_1}^i$ e $\Delta \mathbf{U}_{n_2}^i$. Será selecionada a solução que resultar num valor positivo para o produto $\Delta \mathbf{U}_{n_1}^{i-1} \Delta \mathbf{U}_{n_1}^i$, de modo que a solução continue no mesmo caminho da iteração anterior. No caso de as duas soluções resultarem em valores positivos, utiliza-se a de valor mais próximo à divisão $(-a_3/a_2)$.

O procedimento completo de solução pelo Método do Comprimento de Arco pode ser encontrado no trabalho de Reddy (2014).

4 Desenvolvimento de uma biblioteca para análise de elementos de membrana

A biblioteca de classes de código aberto *Reinforced Concrete Membrane Library* (*RCMembrane*) foi desenvolvida em linguagem *C#* para *Microsoft .NET Framework* (2020), sendo que seu código-fonte está disponível no Apêndice C ou no repositório do autor no *Github* (MELLO, 2020b). Operações matriciais e conversões de unidades são possíveis pela utilização das bibliotecas *Math.NET Numerics* (RUEGG; CUDA; GAEL, 2020) e *Units.NET* (LARSEN, 2013), respectivamente.

A biblioteca auxiliar *Material*, também desenvolvida no decorrer desta pesquisa, é utilizada para cálculo dos parâmetros e das relações tensão-deformação dos materiais, assim como para obtenção das matrizes de rigidez e tensões no concreto e nas armaduras. O código-fonte desta biblioteca também é descrito no Apêndice C ou no repositório do autor (MELLO, 2020a). Esta biblioteca implementa os seguintes objetos:

- (a) *Parameters*: implementação dos parâmetros do concreto conforme a ABNT NBR 6118:2014 e o *fib* Model Code 2010 (2013), parâmetros padrão (*Default*), ou customizados. Exceto para este último caso, são necessários apenas a resistência à compressão do concreto, o tipo e o diâmetro máximo do agregado. As propriedades auxiliares são calculadas automaticamente segundo o modelo utilizado. O modelo padrão é constituído por:

$$E_c = 4732,98 \sqrt{f'_c} \quad f'_t = 0,65 \sqrt[3]{f'_c} \quad \varepsilon'_c = 0,002 \quad \varepsilon_{cu} = 0,0035 \quad (4.1)$$

onde E_c é o módulo de elasticidade do concreto, em MPa, obtido conforme o ACI 318-19 (2019); f'_t é a resistência à tração do concreto, em MPa, calculado pela proposta de Vecchio (2000); ε'_c e ε_{cu} são as deformações plástica e última do concreto; e f'_c é a resistência à compressão do concreto, em MPa;

- (b) *Constitutive*: modelos constitutivos do concreto (TMCC e MCTP);
- (c) *BiaxialConcrete*: criado a partir dos parâmetros e do modelo constitutivo do concreto. Implementa as relações biaxiais do concreto, vistas no Capítulo 2, como cálculo de tensões e da matriz de rigidez do concreto;
- (d) *Steel*: criado a partir da tensão de escoamento e do módulo de elasticidade. Opera as relações tensão-deformação do aço, a partir dos modelos constitutivos elastoplástico bilinear ou trilinear;
- (e) *WebReinforcementDirection*: trata-se de uma direção da armadura distribuída, criada a partir de um objeto *Steel*, do diâmetro e espaçamento das barras de aço, da largura da seção transversal e do ângulo de inclinação em relação ao eixo horizontal;

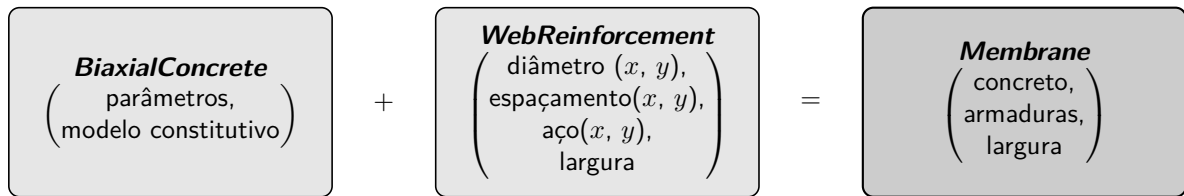
- (f) *WebReinforcement*: trata-se da armadura distribuída, criado a partir de dois objetos *WebReinforcementDirection* de direções perpendiculares. Opera o cálculo das tensões e da matriz de rigidez das armaduras;

O objeto *Membrane* é criado a partir dos objetos *BiaxialConcrete* e *WebReinforcement*, e da largura da seção transversal:

```
var membrane = Membrane.From([Concrete Parameters], [Web Reinforcement],
                               [Width], [Constitutive Model])
```

A Figura 4.1 representa ilustrativamente este processo de criação.

Figura 4.1 – Criação do objeto *Membrane* pela biblioteca *RCMembrane*



4.1 Processo de análise

Assim como mostrado por Vecchio (1989, 1990, 2001), a formulação de elementos de membrana pode ser utilizada em conjunto ao MEF. A Figura 4.2 ilustra como o procedimento padrão do MEF pode ser utilizado com a *RCMembrane*. As deformações em cada elemento são calculadas a partir dos deslocamentos nodais pelas funções de deslocamento, assim, a rotina *Membrane.Calculate* processa as tensões e a rigidez do elemento a partir destas deformações. Por fim, as forças internas são calculadas a partir das tensões e das funções de forma do elemento. O processo se repete até uma convergência ser atingida.

A rotina *Membrane.Calculate* é baseada no algoritmo proposto por Souza (2013, 2016) e é executada na sequência ilustrada na Figura 4.3. O vetor de deformações é tomado como entrada de dados, desse modo, as tensões no concreto e nas armaduras são calculados pela biblioteca *Material*. Etapas distintas são efetuadas para análises pela TMCC ou MMA e pelo MCTP. Por fim, são obtidos a matriz de rigidez do elemento e o vetor de tensões.

A biblioteca também implementa a classe *MembraneSolver*, na qual a rotina *Solve* pode ser usada para análise de elementos de membrana isolados, submetidos a um estado de tensões conhecido. O processo de solução é baseado no Método de Newton-Raphson

e envolve dividir o carregamento num número especificado de passos de carga, sendo o padrão 100.

Figura 4.2 – Fluxograma de análise pelo MEF utilizando a biblioteca *RCMembrane*

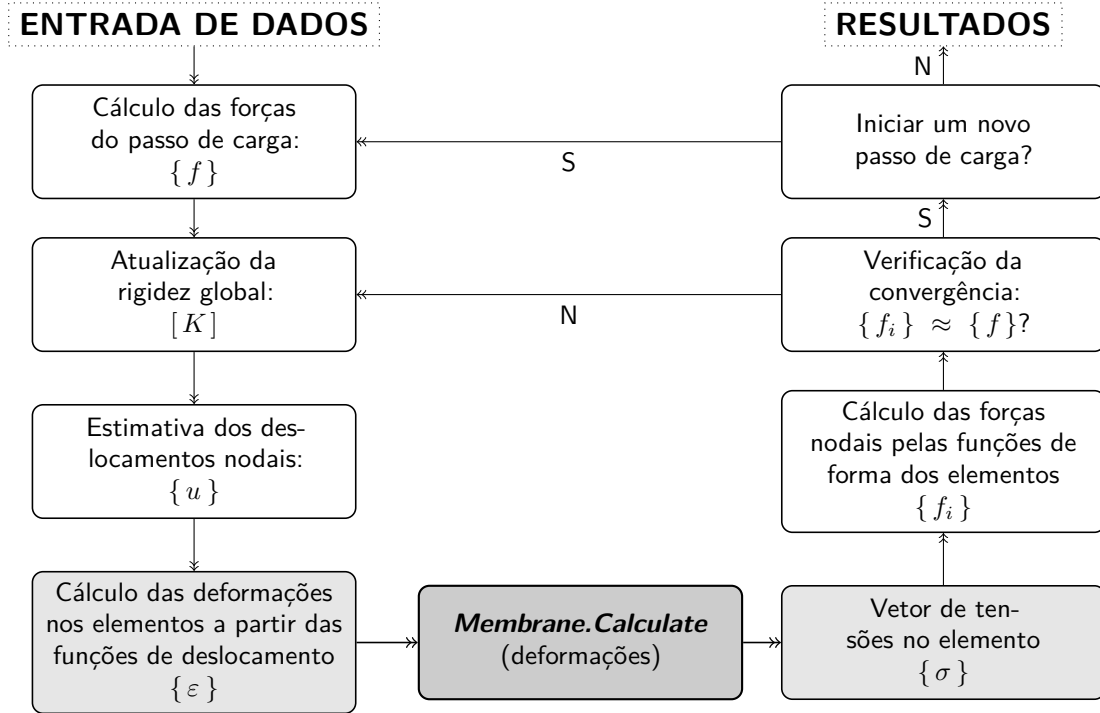
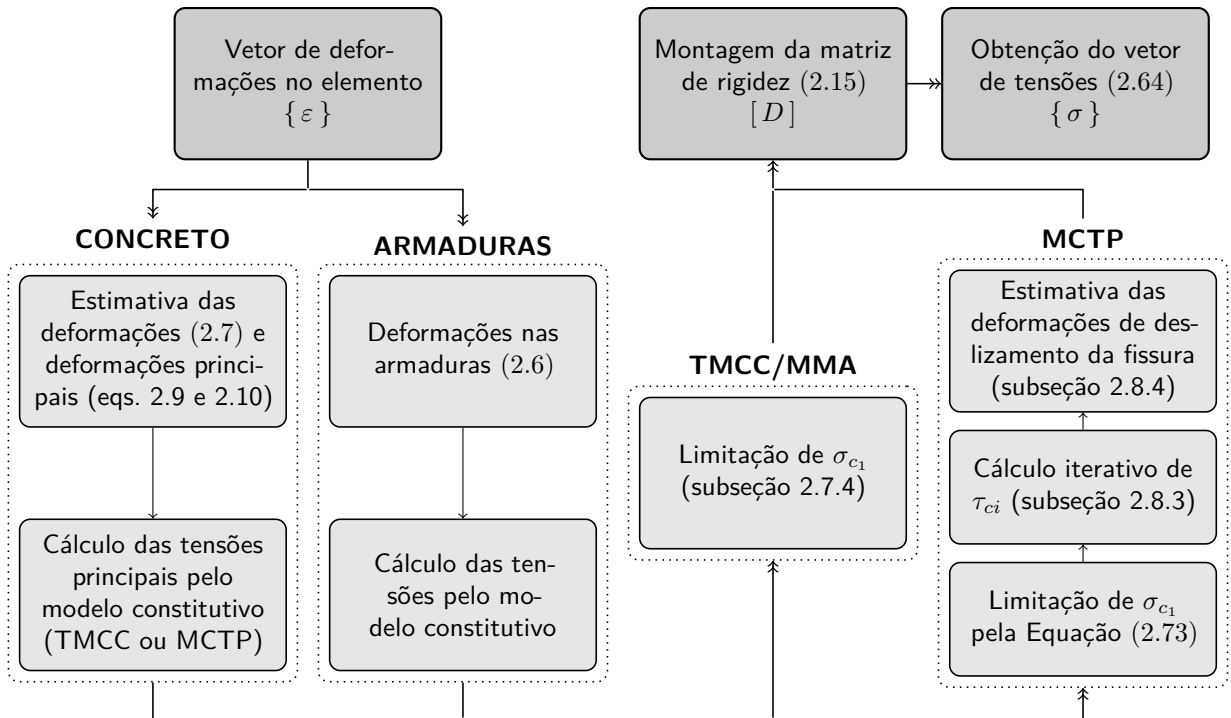


Figura 4.3 – Fluxograma da rotina *Membrane.Calculate*



Cada passo de carga é processado iterativamente até as tensões aplicadas e calculadas atingirem uma certa tolerância (0,001 por padrão), ou atingir um número máximo de iterações (10000 por padrão). Caso este último caso ocorra antes de se atingir o número máximo de passos de carga, a análise termina imediatamente, sendo provável que o elemento tenha atingido a ruína. O processo *Solve* pode ser iniciado fazendo:

```
var solver = new MembraneSolver([Membrane Element], [Nonlinear Solution Method])
solver.Solve([Applied Stresses], [Simulate until failure?])
solver.OutputResults(out [File Path], [Location to write file])
```

O método *OutputResults* salvará os resultados num arquivo de dados separados por vírgula (*CSV*).

4.2 Proposta de atualização do fator C_s para o MCTP

No processo de validação dos modelos, foi observado que, em alguns painéis submetidos a tensões de tração, a solução pelo MCTP era interrompida logo após a fissuração do concreto, o que não acontecia na análise pela TMCC. Concluiu-se que esse comportamento estava relacionado às grandes deformações por deslizamento das fissuras que ocorriam nesses casos. Como proposta deste trabalho, para compensar estas deformações, sugere-se a seguinte relação para o coeficiente C_s , de modo a minimizar a redução da resistência do concreto por abrandamento:

$$C_s = \begin{cases} 0,15 & \text{se } \frac{\max(\sigma_x, \sigma_y)}{|\tau_{xy}|} \geq 0,5; \\ 0,55 & \text{para os demais casos.} \end{cases} \quad (4.2)$$

O valor de 0,15 foi escolhido após várias tentativas de solução do conjunto de painéis que será apresentado na seção 4.3, sendo o que proporcionou uma melhor aproximação aos resultados experimentais desses casos. Quando não existem tensões de tração aplicadas, ou estas são pequenas em comparação à tensão cisalhante, o valor padrão de 0,55, proposto por Vecchio (2000), é usado.

4.3 Validação

Para validar o algoritmo da *RCMembrane*, foram analisados 72 painéis com variadas resistências do concreto e configurações de armadura encontrados na literatura. Estes elementos foram analisados por meio da rotina *Solve*, apresentada na seção anterior, e seus resultados de tensões de fissuração e últimas foram comparados aos resultados experimentais.

Considerou-se a tolerância padrão de 0,001, sendo que este valor se mostrou mais estável para os três modelos constitutivos implementados, após a análise de vários painéis com diversas resistências do concreto e diferentes configurações de armadura. Para o cálculo dos parâmetros do concreto, foi utilizado o modelo padrão (4.1).

Os resultados das análises podem ser graficamente visualizados nos diagramas de dispersão ilustrados na Figura 4.4, que relacionam os resultados numéricos (eixo vertical) obtidos por cada modelo com os respectivos resultados experimentais (eixo horizontal). Outra forma de visualizar o conjunto de resposta das análises é através dos diagramas de caixa ilustrados na Figura 4.5. Em ambas as figuras é possível concluir que os modelos resultaram em aproximadamente a mesma tensão de fissuração (τ_{cr}), o que é explicado pela utilização da mesma formulação para a resistência à tração do concreto. Quanto à tensão última (τ_u), o MMA apresentou respostas de dispersão mais concentrada em torno de 1.

Figura 4.4 – Diagrama de dispersão dos resultados obtidos

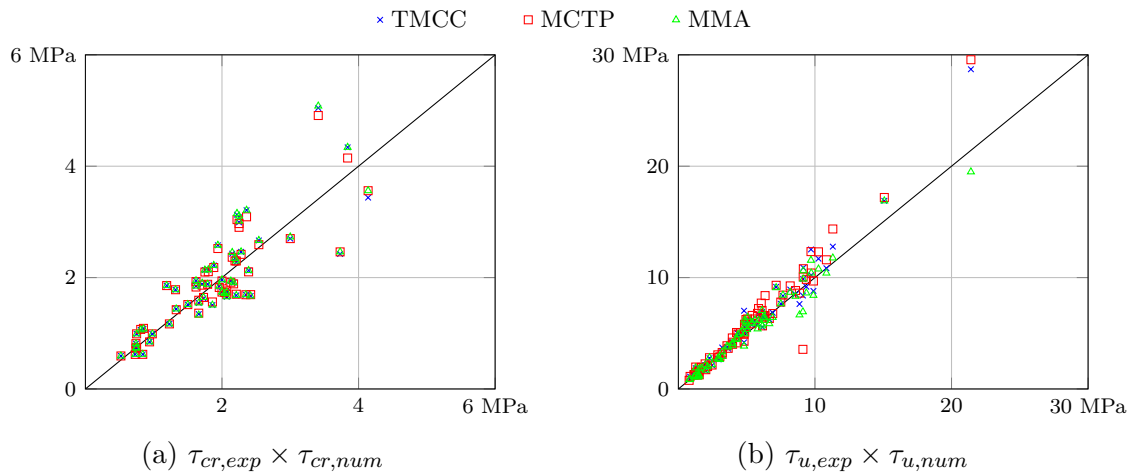
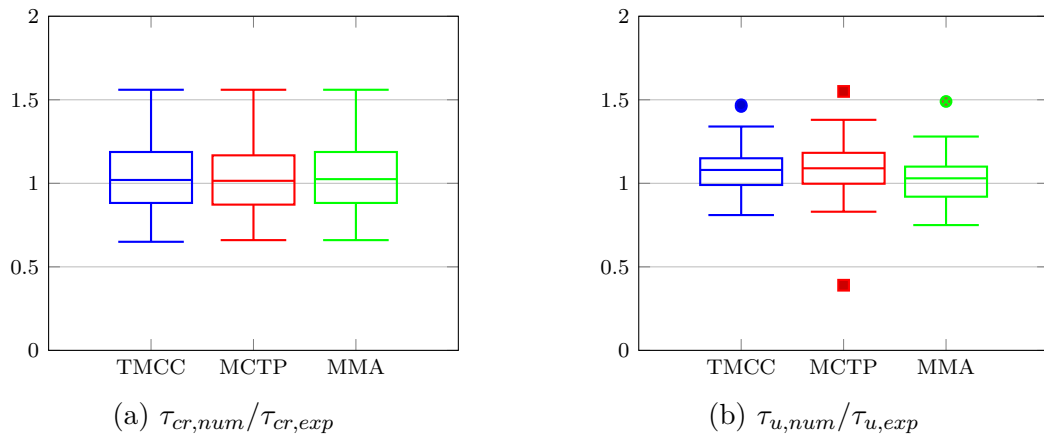


Figura 4.5 – Diagramas de caixa dos resultados obtidos



Diagramas tensão-deformação de alguns dos painéis analisados são ilustrados na Figura 4.6. Comparando aos dados experimentais, representados por pontos pretos nos gráficos, pode-se concluir que as duas abordagens utilizadas nas análises apresentaram boa aproximação, especialmente para os painéis PHS8 e PHS10.

Embora a formulação possa prever a capacidade de carga dos painéis com boa exatidão, em alguns casos o equilíbrio não é atingido após o escoamento das armaduras. Este fato explica a discrepância entre os valores numéricos e experimentais das deformações últimas para os painéis PV10, A3, VA3 e VB2.

A Tabela 4.1 apresenta os resultados obtidos para 44 painéis submetidos a cisalhamento puro. Nesse caso, os três métodos analisados apresentaram resultados muito próximos. Comparando os valores médios, o MCTP apresentou um resultado um pouco menos exato para a tensão última. Por outro lado, este mesmo modelo apresentou uma variação menor para as tensões de fissuração.

A Tabela 4.2 resume os resultados obtidos para 28 painéis submetidos a solicitações normais e cisalhantes combinadas. Para estes elementos, o MCTP se aproximou melhor dos resultados experimentais para a tensão última, sendo que nesta solução se utilizou a relação (4.2) para corrigir o fator C_s . Quanto às tensões de fissuração, os três métodos apresentaram aproximadamente os mesmos resultados.

Analisando os resultados de todos os 72 painéis, nas últimas duas linhas da Tabela 4.2, todos os métodos previram praticamente a mesma tensão de fissuração, dado que a mesma estimativa para a resistência do concreto à tração foi utilizada. Quanto às tensões de ruptura, o MMA resultou num valor médio muito próximo de 1 e, apesar de não apresentar a menor variação, este modelo foi selecionado como padrão na *RCMembrane*.

Além de obter boas respostas na análise de painéis, a *RCMembrane* também apresentou bons resultados quando aplicada em outros tipos de estruturas, em conjunto com o *SPMTool*, conforme será exposto na seção 5.3.

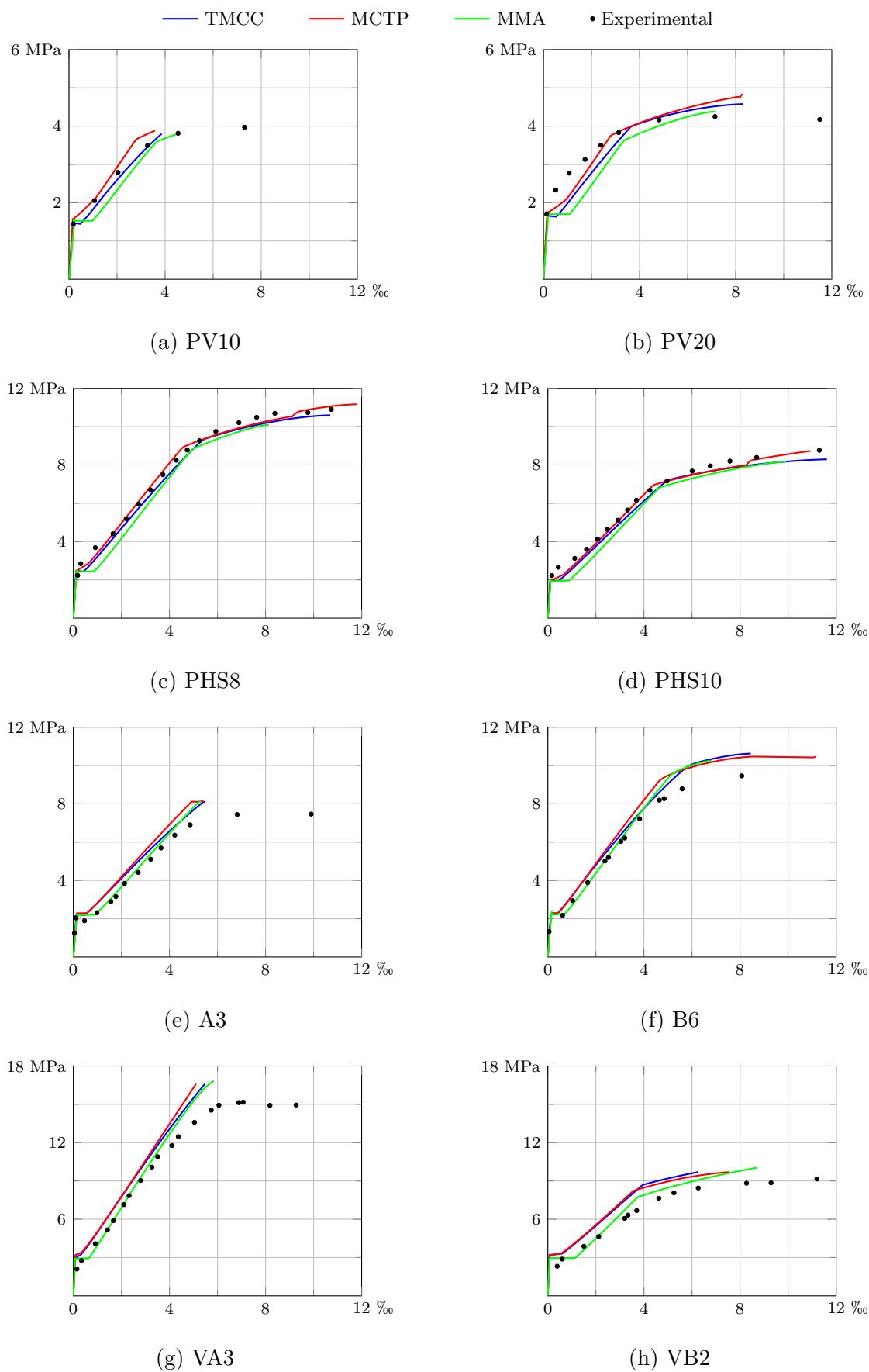
Figura 4.6 – Diagrama de deformação $\times$ tensão de cisalhamento para alguns painéis

Tabela 4.1 – Resultados obtidos em painéis carregados por cisalhamento puro

Ref.	ID	Exp. (MPa)		τ_{cr}/τ_{cr-exp}			τ_u/τ_{u-exp}		
		τ_{cr}	τ_u	TMCC	MCTP	MMA	TMCC	MCTP	MMA
Vecchio e Collins (1982)	PV4	1,79	2,89	1,05	1,05	1,05	0,93	1,06	0,95
	PV6	2,00	4,55	0,98	0,97	0,98	1,09	1,11	1,11
	PV10	1,86	3,97	0,81	0,84	0,81	0,99	1,01	0,99
	PV11	1,66	3,56	0,93	0,96	0,93	1,06	1,09	1,08
	PV13	1,73	2,01	0,94	0,95	0,95	0,81	0,86	0,84
	PV16	2,07	2,14	0,84	0,85	0,84	0,92	0,94	0,94
	PV19	2,07	3,95	0,80	0,82	0,80	1,07	1,16	1,04
	PV20	2,21	4,26	0,76	0,77	0,76	1,10	1,19	1,07
	PV21	2,35	5,03	0,72	0,72	0,72	1,11	1,21	1,06
	PV22	2,42	6,07	0,69	0,70	0,69	1,05	1,27	0,96
	PV26	2,00	5,41	0,87	0,87	0,87	1,15	1,22	1,10
	PV27	2,04	6,35	0,84	0,85	0,84	1,05	1,32	0,97
Bhide e Collins (1989)	PB11	1,19	1,27	1,56	1,56	1,56	1,47	1,55	1,49
	PB12	1,32	1,53	1,35	1,35	1,35	1,16	1,24	1,18
	PB15	1,80	1,96	1,19	1,17	1,19	1,09	1,15	1,10
	PB18	1,62	1,70	1,14	1,13	1,14	1,08	1,14	1,11
Vecchio e Chan (1990)	PC1	1,96	4,95	0,94	0,93	0,94	1,24	1,26	1,22
	PC1A	2,17	5,61	0,88	0,87	0,88	1,09	1,12	1,10
Marti e Meyboom (1992)	PP1	1,71	4,95	1,10	1,09	1,11	1,10	1,17	1,13

Tabela 4.1 (continuação)

Ref.	ID	Exp. (MPa)		τ_{cr}/τ_{cr-exp}			τ_u/τ_{u-exp}		
		τ_{cr}	τ_u	TMCC	MCTP	MMA	TMCC	MCTP	MMA
Vecchio, Collins e Aspiotis (1994)	PHS1	2,54	2,95	1,05	1,02	1,05	0,90	0,96	0,91
	PHS2	1,94	6,66	1,33	1,30	1,33	0,94	0,95	0,88
	PHS3	2,28	8,19	1,08	1,06	1,08	1,09	1,13	1,06
	PHS8	2,15	10,84	1,13	1,10	1,14	1,00	1,07	0,96
	PA1	2,19	6,34	1,07	1,05	1,07	1,01	1,04	1,01
	PA2	1,88	6,22	1,18	1,16	1,18	1,03	1,05	1,03
Pang e Hsu (1995)	A1	-	2,28	-	-	-	1,21	1,23	1,23
	A2	-	5,37	-	-	-	1,07	1,09	1,09
	A3	-	7,66	-	-	-	1,09	1,10	1,10
	A4	-	11,31	-	-	-	1,13	1,27	1,04
	B1	-	3,96	-	-	-	1,04	1,06	1,04
	B2	-	6,13	-	-	-	1,13	1,15	1,15
	B3	-	4,36	-	-	-	1,14	1,13	1,12
	B4	-	5,07	-	-	-	1,21	1,24	1,27
	B5	-	7,16	-	-	-	1,28	1,30	1,27
	B6	-	9,15	-	-	-	1,19	1,18	1,16

Tabela 4.1 (continuação)

Ref.	ID	Exp. (MPa)		$\tau_{cr} / \tau_{cr-exp}$			τ_u / τ_{u-exp}		
		τ_{cr}	τ_u	TMCC	MCTP	MMA	TMCC	MCTP	MMA
Zhang e Hsu (1998)	VA1	-	6,16	-	-	-	0,90	0,92	0,92
	VA2	-	9,73	-	-	-	1,05	1,07	1,07
	VA3	-	15,08	-	-	-	1,12	1,14	1,12
	VA4	-	21,42	-	-	-	1,34	1,38	0,91
	VB1	-	7,50	-	-	-	1,02	1,04	1,02
	VB2	-	9,14	-	-	-	1,08	1,10	1,08
	VB3	-	9,71	-	-	-	1,29	1,27	1,19
	VB4	-	4,86	-	-	-	1,05	1,05	1,02
Xie, Bentz e Collins (2011)	PL5	1,75	3,21	1,22	1,20	1,22	0,95	1,02	0,91
	Média			1,02	1,01	1,02	1,09	1,14	1,07
	Coeficiente de variação (%)			20,5	19,5	20,4	11,2	11,4	11,3

Tabela 4.2 – Resultados obtidos para painéis submetidos a solicitações combinadas

Ref.	ID	Proporção			Exp. (MPa)		τ_{cr}/τ_{cr-exp}			τ_u/τ_{u-exp}		
		σ_x	σ_y	τ_{xy}	τ_{cr}	τ_u	TMCC	MCTP	MMA	TMCC	MCTP	MMA
Vecchio e Collins (1982)	PV23	-0,39	: -0,39	: 1	3,73	8,87	0,65	0,66	0,66	0,86	0,97	0,75
	PV25	-0,69	: -0,69	: 1	4,14	9,12	0,83	0,86	0,86	0,92	0,39	0,76
	PV28	0,32	: 0,32	: 1	1,66	5,80	0,81	0,82	0,80	1,01	1,17	0,93
	PV29	-0,29	: -0,29	: 1	2,21	5,87	1,03	1,04	1,04	1,14	1,23	1,05
Bhide e Collins (1989)	PB4	1	: 0	: 1	0,81	1,16	1,31	1,32	1,31	1,02	1,10	0,94
	PB6	1	: 0	: 1	0,85	1,15	1,27	1,28	1,27	1,04	1,12	0,97
	PB7	1,90	: 0	: 1	0,74	0,86	1,10	1,10	1,10	1,29	1,31	1,01
	PB8	3	: 0	: 1	0,52	0,79	1,14	1,14	1,14	1,23	0,98	1,06
	PB16	2	: 0	: 1	0,98	1,45	1,01	1,01	1,01	1,22	1,06	0,82
	PB19	1	: 0	: 1	1,23	1,28	0,95	0,95	0,95	1,18	1,17	0,94
	PB20	2	: 0	: 1	0,94	1,42	0,90	0,90	0,90	1,08	0,99	0,75
	PB21	3,10	: 0	: 1	0,73	1,42	0,85	0,85	0,85	1,01	0,89	0,82
	PB28	3,10	: 0	: 1	0,84	1,53	0,74	0,74	0,74	0,94	0,83	0,76
	PB29	2	: 0	: 1	0,75	1,49	1,32	1,32	1,32	1,19	1,03	0,80
PB30	3	: 0	: 1	0,74	1,48	0,99	0,99	0,99	1,12	1	0,88	
Vecchio e Chan (1990)	PC4	-0,39	: -0,39	: 1	3	4,84	0,90	0,90	0,91	1,15	1,20	1,28
	PC7	0,32	: 0,32	: 1	1,50	3,65	1,01	1,01	1,01	0,99	1	1,03

Tabela 4.2 (continuação)

5 O programa *SPMTool*

O programa computacional *Stringer-Panel Modelling Tool* (*SPMTool*) foi desenvolvido com o intuito de se tornar uma alternativa simples para o dimensionamento e verificação de elementos isolados de concreto armado, de forma que não seja necessária a configuração de parâmetros avançados ou criação de modelos complexos como nos programas comerciais de elementos finitos. O programa não considera efeitos de aderência entre aço e concreto, efeito de pino das armaduras, fadiga e deformações devido retração e fluência do concreto.

O *SPMTool* funciona como um complemento do *AutoCAD*, a partir da versão 2020 (2019). Dessa forma, os conceitos de modelagem estrutural pelo MBP são aplicados a partir de uma ferramenta familiar aos profissionais da engenharia. Ainda existe a possibilidade de aplicação do programa na verificação de modelos de escoras e tirantes em serviço, bastando, para isso, incluir apenas as bielas no modelo.

As principais melhorias do *SPMTool*, em comparação ao *SPanCAD*, são: capacidade de dividir elementos; usar diferentes tipos de aço em cada elemento; escolher o modelo constitutivo do concreto entre TMCC, MCTP e MMA; modificar os parâmetros de análise não-linear; e visualizar resultados de elementos isolados.

O software consiste numa biblioteca para *Microsoft.NET Framework* (v. 4.8, 2020) e foi desenvolvido em linguagem *C#* através do ambiente de desenvolvimento *Microsoft Visual Studio Community 2019* (2019). O código-fonte é aberto e disponibilizado no repositório do autor no *GitHub* (2021c) sob licença não comercial *Creative Commons*. Isso permite que outros pesquisadores colaborem com o desenvolvimento do programa, porém, sem a possibilidade de aplicação em meios comerciais.

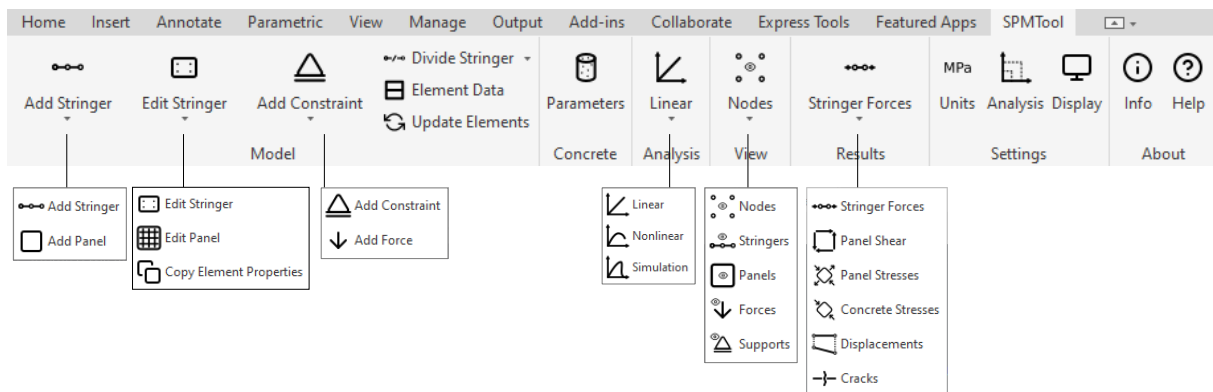
As seguintes bibliotecas foram utilizadas no desenvolvimento do programa:

- (a) *Autodesk ObjectARX Managed .NET API* (AUTODESK, 2020): biblioteca de classes que implementa os objetos utilizados pelo *AutoCAD*;
- (b) *Math.NET.Numerics* (RUEGG; CUDA; GAEL, 2020): implementação de matrizes, vetores e de solução de sistemas lineares e não-lineares;
- (c) *Units.NET* (LARSEN, 2013): biblioteca de unidades utilizada para conversões;
- (d) *LiveCharts* (RODRIGUEZ, 2017): utilizada para apresentação dos gráficos em análises não-lineares;
- (e) *Material*: biblioteca própria de modelos constitutivos do concreto e das armaduras, utilizada para o cálculo das tensões nesses materiais, a partir das deformações. O código-fonte é encontrado em C.1 ou no repositório do autor (MELLO, 2020a);

- (f) *Reinforced Concrete Membrane (RCMembrane)*: implementação própria de elementos de membrana em concreto armado, apresentada no Capítulo 4. Utilizada para calcular as tensões nos pontos de integração dos painéis. O código-fonte está disponível em C.2 ou no repositório do autor (MELLO, 2020b);
- (g) *FEMAnalysis*: biblioteca genérica de elementos finitos e métodos de solução de equações lineares e não-lineares, utilizada para resolver o sistema global de equações. Implementa os métodos apresentados na seção 3.3, sendo o código-fonte apresentado em C.3 ou no repositório do autor (MELLO, 2021a);
- (h) *SPMElements*: implementação dos elementos finitos do MBP, usada para se obter as matrizes de rigidez e calcular as forças nodais a partir dos deslocamentos. Utiliza a formulação apresentada nos Apêndices A e B, e tem seu código-fonte descrito em C.4 ou no repositório do autor (MELLO, 2021b).

Para utilizar o programa, basta executar o comando *NETLOAD* no *AutoCAD* e carregar um arquivo de diretrizes (*SPMTool.dll*). Dessa forma, é criada automaticamente uma aba na interface do *AutoCAD*, conforme a Figura 5.1.

Figura 5.1 – Interface do *SPMTool* como um complemento do *AutoCAD*



A interface é formada por blocos, seguindo o modelo do *AutoCAD*, que são divididos em *Model*, *Concrete*, *Analysis*, *View*, *Results*, *Settings* e *About*. Cada um destes blocos apresenta botões que executam os comandos existentes no *SPMTool*, sendo que botões que executam ações semelhantes, como adicionar uma biela ou um painel, são agrupados numa única posição e acessados por clique na seta existente abaixo do botão.

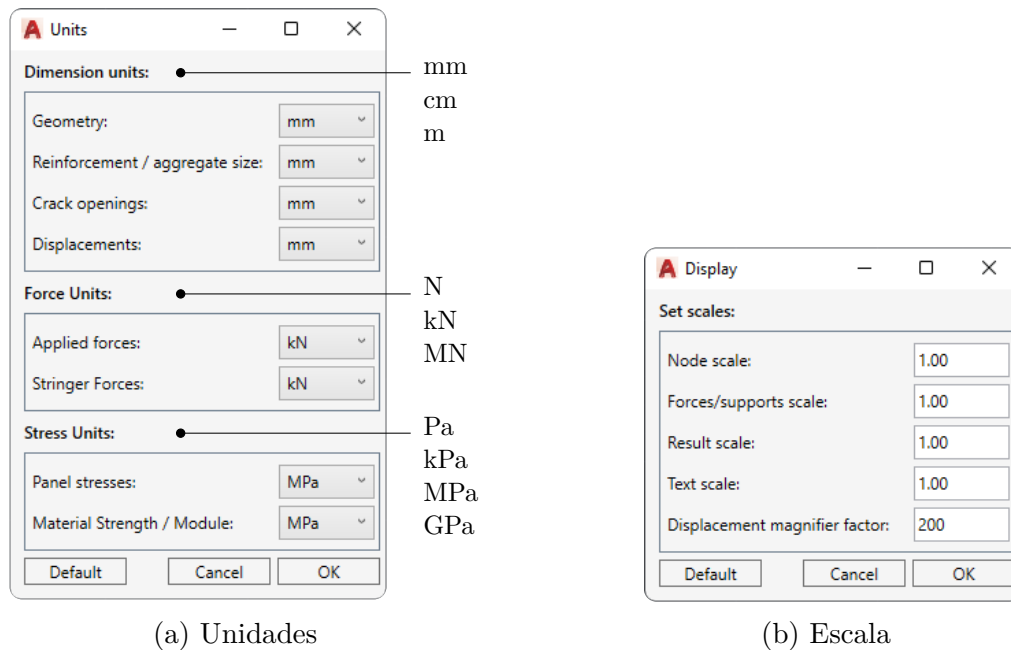
5.1 Pré-processamento

Ao carregar o programa, caso seja iniciado um novo arquivo, automaticamente a janela de configuração de unidades (Figura 5.2a) será mostrada. As unidades padrão são: milímetros para dimensões, kilonewtons para forças, e megapascal para tensões; podendo

ser aplicadas clicando no botão *Default*. A configuração de unidades também pode ser feita pelo botão *Units*, na aba principal do *SPMTool*.

Além disso, é possível configurar a escala dos elementos gráficos do modelo na janela de escalas (Figura 5.2b), acessada pelo botão *Display*. Nessa janela também é possível definir um fator de multiplicação dos deslocamentos para a representação do modelo deslocado.

Figura 5.2 – Janelas de configuração



5.1.1 Definição do modelo

A definição do modelo biela-painel no *SPMTool* deve se iniciar sempre pelas bielas, sendo que os nós serão adicionados automaticamente nessa etapa. Após isso, é possível adicionar os painéis, suportes e forças externas.

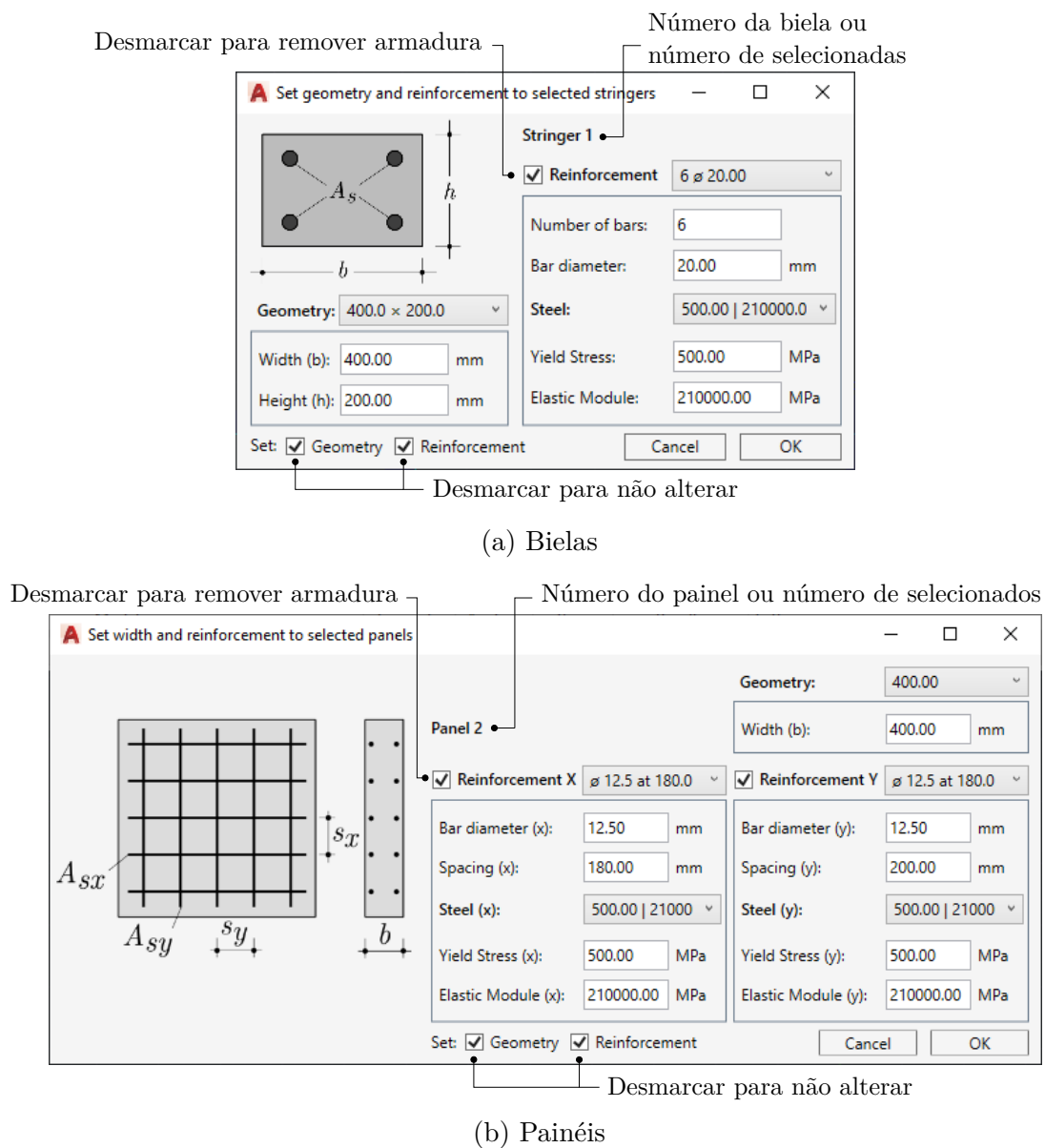
Todos os comandos de edição do modelo estão dispostos no bloco *Model*, sendo descritos a seguir:

- (a) *Add stringer*: adiciona uma biela conectando dois pontos. Após a adição, os nós extremos (representados em cor vermelha) e interno (representado em azul) são adicionados automaticamente;
- (b) *Add panel*: este botão é agrupado com o comando anterior e adiciona um painel selecionando quatro vértices (nós extremos das bielas). A seleção pode englobar também as bielas, pois o programa só irá escolher os nós necessários;
- (c) *Edit stringers*: abre a janela de edição das bielas (Figura 5.3a), onde é possível editar a geometria (largura e altura) e a armadura, se existente, para uma seleção

de bielas. Caso o usuário queira que uma destas propriedades permaneça inalterada, basta desmarcar a caixa de seleção correspondente, na parte inferior esquerda da janela, dessa forma se aplica somente o item que estiver marcado para todas as bielas selecionadas;

- (d) *Edit panels*: agrupado com *Edit stringers*, abre a janela de edição dos painéis (Figura 5.3b) para definição da largura e da armadura em malha, para uma seleção de painéis. Assim como para as bielas, pode-se aplicar somente uma das propriedades selecionando as caixas de seleção na parte inferior da janela;

Figura 5.3 – Janela de edição de geometria e armadura dos elementos

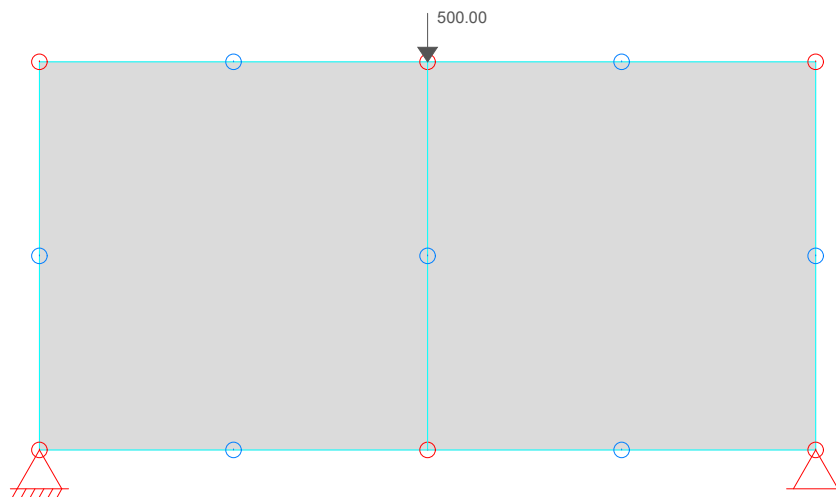


- (e) *Copy element properties*: copia as propriedades de um elemento (geometria, armadura ou ambas) para um conjunto de elementos do mesmo tipo;

- (f) *Constraints*: aplica restrições de deslocamento para um conjunto de nós externos (em vermelho). Após a seleção, é feita a escolha se os nós são livres ou possuem restrição na horizontal (x), vertical (y), ou em ambas as direções;
- (g) *Forces*: agrupado com *Constraints*, aplica forças num conjunto de nós externos (em vermelho). O usuário define a intensidade da força nas direções x e y ;
- (h) *Divide stringer*: divide uma biela em duas ou mais novas bielas;
- (i) *Divide panel*: agrupado com o comando anterior, divide um painel retangular especificando o número de linhas e colunas para divisão;
- (j) *Element data*: mostra na linha de comando as propriedades de cada elemento. É possível visualizar os dados de nós, bielas, painéis, suportes e forças. Também é possível visualizar estes dados mantendo o ponteiro do mouse no elemento;
- (k) *Update elements*: atualiza as numerações de cada elemento e escreve a quantidade de nós, bielas e painéis na linha de comando. Este comando não precisa ser executado pelo usuário, a menos que este deseje conferir o número de elementos, pois as numerações são automaticamente atualizadas quando há mudanças no modelo.

A Figura 5.4 ilustra um exemplo de um modelo biela-painel após a definição da geometria e inserção de forças e suportes no *SPMTool*. Observa-se a existência de dois tipos de nós, os de extremidade, em vermelho, que representam os graus de liberdade extremos das bielas, e os internos, que representam o grau de liberdade médio das bielas e também de cada face dos painéis. Portanto, para haver apenas esforços axiais nas bielas, forças e suportes podem ser adicionadas apenas aos nós de extremidade.

Figura 5.4 – Exemplo de modelo biela-painel para uma viga-parede, criado no *SPMTool*



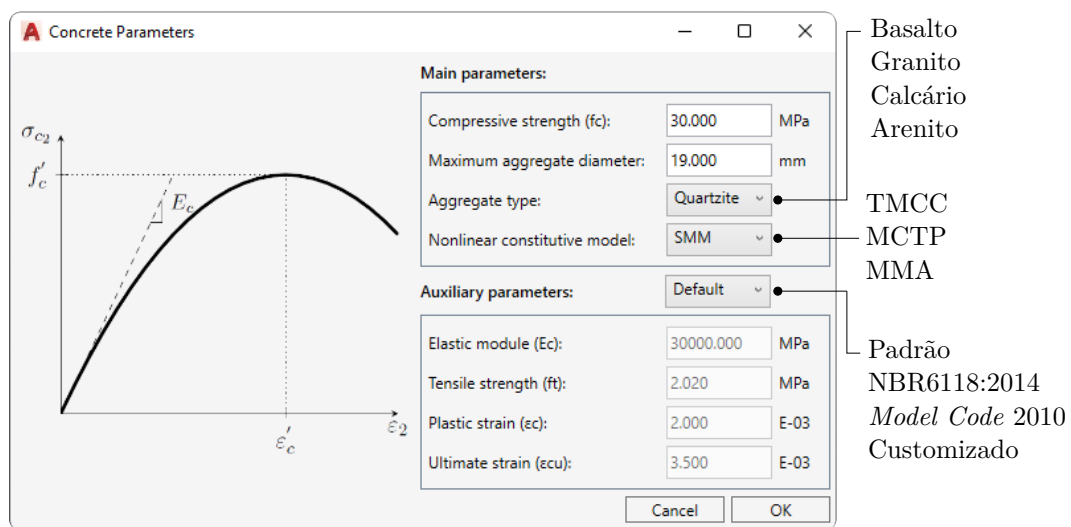
Ao utilizar os comandos *Edit stringers* e *Edit panels*, as várias geometrias e armaduras diferentes serão salvas na base de dados do arquivo, assim, o usuário pode utilizá-las em outros elementos selecionando-as nas caixas de seleção correspondentes. É possível ainda ligar e desligar a visualização dos vários tipos de elementos, forças e suportes por meio dos botões do bloco *View*.

Em todos os comandos que envolvem seleção de itens específicos é possível selecionar outros itens, pois o programa filtra os elementos de acordo sistema de camadas (*layers*) do *AutoCAD*. Por exemplo, para o comando *Stringer geometry*, pode-se selecionar também painéis e nós, porém, a geometria será somente aplicada aos elementos na camada *Stringer*.

5.1.2 Definição dos materiais

A definição do concreto é feita por meio do botão *Parameters*, que abre a janela de edição dos parâmetros do concreto (Figura 5.5), onde é possível definir os parâmetros principais, como resistência à compressão, tipo e diâmetro máximo do agregado e modelo constitutivo não-linear (TMCC ou MCTP). Os parâmetros secundários, como módulo de elasticidade e resistência à tração, podem ser calculados automaticamente selecionando o modelo padrão (*Default* - conforme (4.1)), um modelo normativo (Model Code 2010 e ABNT NBR 6118:2014) ou podem ser customizados selecionando a opção *Custom*.

Figura 5.5 – Janela de configuração dos parâmetros do concreto



A definição dos parâmetros do aço é feita na janela de edição das bielas e painéis (Figura 5.3). Dessa forma, é possível utilizar materiais de propriedades distintas para cada elemento. Após aplicar o tipo de aço num elemento, este ficará salvo na base de dados do arquivo para ser utilizado posteriormente, bastando selecioná-lo na caixa de seleção *Steel*.

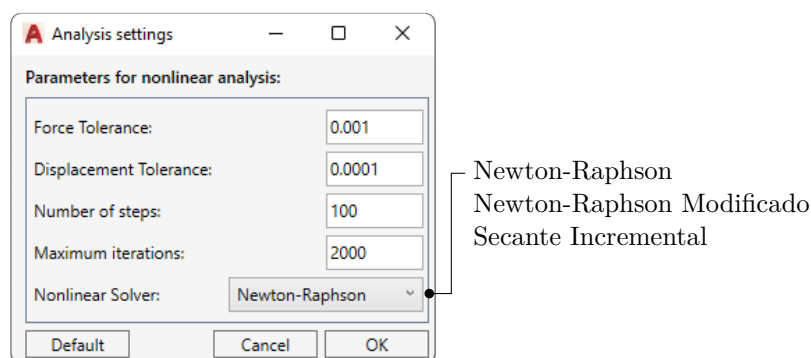
Após a definição da geometria e armaduras dos elementos, e os parâmetros do concreto, o modelo estará pronto para a análise, que pode ser feita pelos comandos do bloco *Analysis*. As armaduras apenas são necessárias no caso de análise não-linear, devendo elas serem dimensionadas a partir dos resultados da análise elástica.

5.1.3 Parâmetros de análise não-linear

A análise não-linear é executada por aplicação dos métodos descritos em 3.3.6 para solução do sistema de equações. Caso o usuário tenha conhecimento neste tipo de análise, é possível alterar os parâmetros de solução, como tolerância de convergência para forças residuais e incrementos de deslocamento, número de passos de carga e número limite de iterações.

Ao clicar no botão *Analysis*, no bloco *Settings*, será apresentada a janela de configuração dos parâmetros de análise (Figura 5.6). É importante salientar que os valores exibidos na Figura 5.6 representam os padrões adotados pelo programa. Esses valores foram definidos por apresentarem resultados alinhados a experimentos, como será visto em seção 5.3.

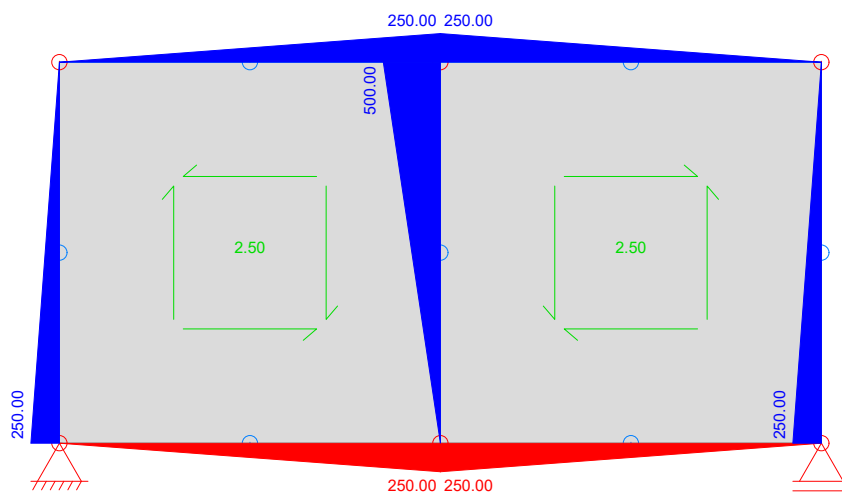
Figura 5.6 – Janela de configuração dos parâmetros de análise não-linear



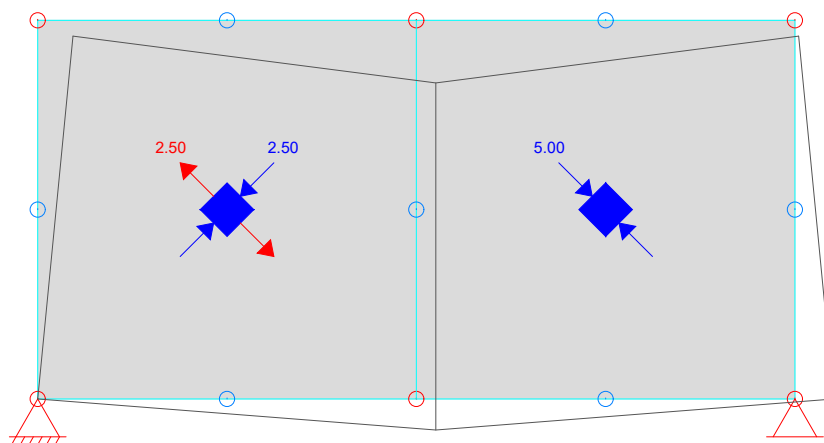
5.2 Pós-processamento

Após o processamento dos dados, o programa mostrará os resultados nos elementos, que podem ser visualizadas pelos botões agrupados no bloco *Results*. Para as bielas, será mostrada a variação do esforço axial e suas intensidades em suas extremidades. Para os painéis, serão mostradas as tensões de cisalhamento, as tensões principais médias e as tensões principais no concreto. Os esforços e tensões de compressão são indicados em azul e, os de tração, em vermelho.

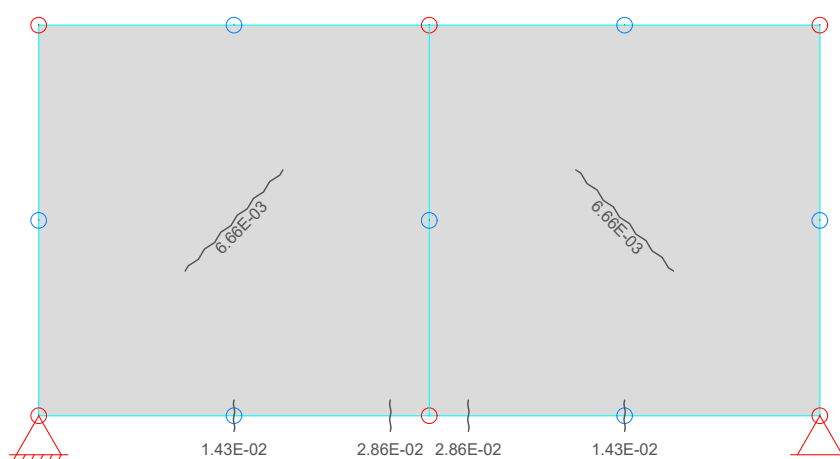
A Figura 5.7 apresenta os resultados de uma análise linear do modelo ilustrado na Figura 5.4. A Figura 5.7a apresenta as forças nas bielas e as tensões de cisalhamento nos painéis. Na Figura 5.7b podem ser vistas as tensões principais nos painéis, sendo que no painel à esquerda são mostradas as tensões médias e, à direita, as tensões no concreto, calculadas pelas equações do MPT. Além disso, nesta última também se observa o modelo deslocado, com o desenho destes deslocamentos majorado de acordo com o fator configurado anteriormente na janela de escalas.

Figura 5.7 – Exemplo de resultados do *SPMTool*

(a) Forças nas bielas e tensões cisalhantes nos painéis



(b) Tensões principais e modelo deformado



(c) Aberturas de fissuras

Os deslocamentos nodais podem ser visualizados selecionando um nó pelo comando *Element data*, sendo apresentados para cada direção. O deslocamento horizontal (u_x) será positivo se direcionado à direita e o vertical (u_y), positivo se direcionado para cima. Por exemplo, ao selecionar o nó inferior central do exemplo, o seguinte texto será mostrado na linha de comando:

```
Node 3
Position: (1000.00 mm, 0.00 mm)
DoFIndex: 4 - 5
Displacements:
ux = 0.1 mm
uy = -0.35 mm
```

O dimensionamento das armaduras e a verificação dos limites de tensão no concreto não são feitos automaticamente pelo programa. Tais procedimentos devem ser executados pelo projetista, a partir das equações que serão apresentadas no Capítulo 6, utilizando os esforços de projeto em cada elemento.

Após a inserção das armaduras no modelo, é possível efetuar uma análise não-linear, que é iniciada pelos comandos *NonlinearAnalysis* ou *Simulation*. Em ambos os comandos, o programa solicitará a seleção de um nó e uma direção para monitoramento do deslocamento e também de elementos para os quais se deseje salvar os resultados da análise. Ao se executar um desses comandos, uma janela se abrirá para plotar o gráfico de comportamento, exemplificado na Figura 5.8.

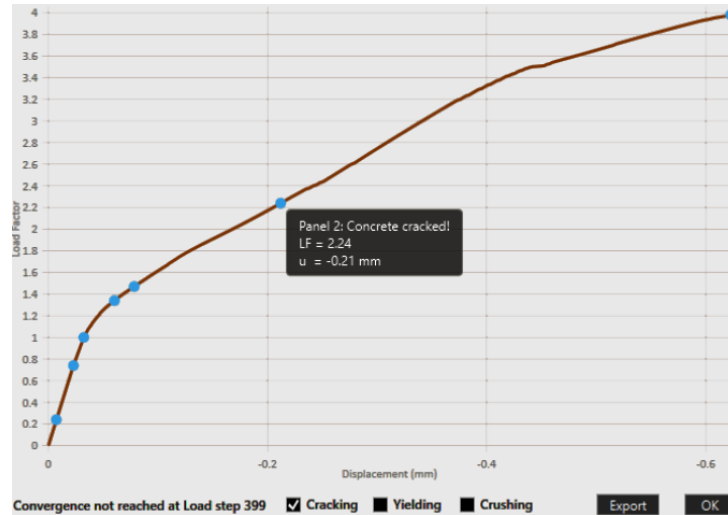
Com o comando *NonlinearAnalysis*, a análise é efetuada com incrementos constantes de carga, até se atingir o carregamento especificado no modelo. Esse caso é útil para aprimorar o dimensionamento ou verificar o comportamento do modelo submetido aos carregamentos de serviço.

O comando *Simulation* executa a análise pelo Método do Comprimento de Arco, o qual apresenta uma melhor resposta em problemas de plasticidade. Dessa forma, a análise é executada sem limitação de carregamento, terminando no incremento de carga onde não se consiga convergência ou se exceda o número limite de iterações.

Ainda é possível observar cada ponto plotado no gráfico colocando o ponteiro do mouse no ponto desejado. Quando ocorre a fissuração, escoamento ou esmagamento de um elemento, um ponto é plotado no gráfico, sendo possível ativar a visualização destes pontos nas caixas de seleção *Cracking*, *Yielding* e *Crushing*. Além disso, pode-se exportar os dados deste gráfico clicando no botão *Export*, na parte inferior da janela mostrada na Figura 5.8. Os fatores de carga e os deslocamentos correspondentes serão salvos num arquivo de valores separados por vírgula (*CSV*), no mesmo local de salvamento do arquivo

do *AutoCAD*. Se o usuário selecionou outros elementos para monitorar, também serão salvos arquivos separados para cada um deles.

Figura 5.8 – Exemplo de gráfico Fator de carga $\times$ deslocamento plotado pelo *SPMTool*



Após ser efetuada a análise não-linear, o *SPMTool* também apresenta as aberturas médias de fissuras nos elementos, conforme ilustrado na Figura 5.7c. Essas aberturas são calculadas a partir da Equação 2.52 e exibidas na unidade especificada pelo usuário.

Com os resultados não-lineares, ainda é possível otimizar manualmente as armaduras do modelo, retirando dos elementos que eventualmente tenham uma certa folga e adicionando dos elementos mais solicitados. Esse passo, obviamente, deve ser tratado cuidadosamente pelo engenheiro estrutural e verificado por meio de análises sucessivas.

5.3 Validação dos modelos não-lineares

Os resultados não-lineares do *SPMTool* foram validados por comparação a resultados experimentais de vigas-parede (BIRRCHEER et al., 2009; ISMAIL; GUADAGNINI; PILAKOUTAS, 2016), vigas com abertura (ASHOUR, 2000; JASIM; ALLAWI; OUKAILI, 2019) e consolos de concreto armado (CHAKRABARTI; FARAH; KASHOU, 1989; FOSTER; POWELL; SELIM, 1996), que romperam por cisalhamento nos experimentos. É importante salientar que nenhum dos elementos analisados foi dimensionado pelos autores por aplicação do MBP.

Os modelos foram idealizados com a menor discretização possível, seguindo o conceito do MBP. A Figura 5.12 apresenta os modelos genéricos utilizados nas análises, sendo importante observar que os consolos (Figuras 5.13c e 5.13d) foram analisados de cabeça para baixo, seguindo os modelos experimentais. No caso das vigas de Ashour (2000)

(Figura 5.14e), por serem contínuas e simétricas, a simetria foi considerada por meio de um apoio horizontal na extremidade direita da biela superior.

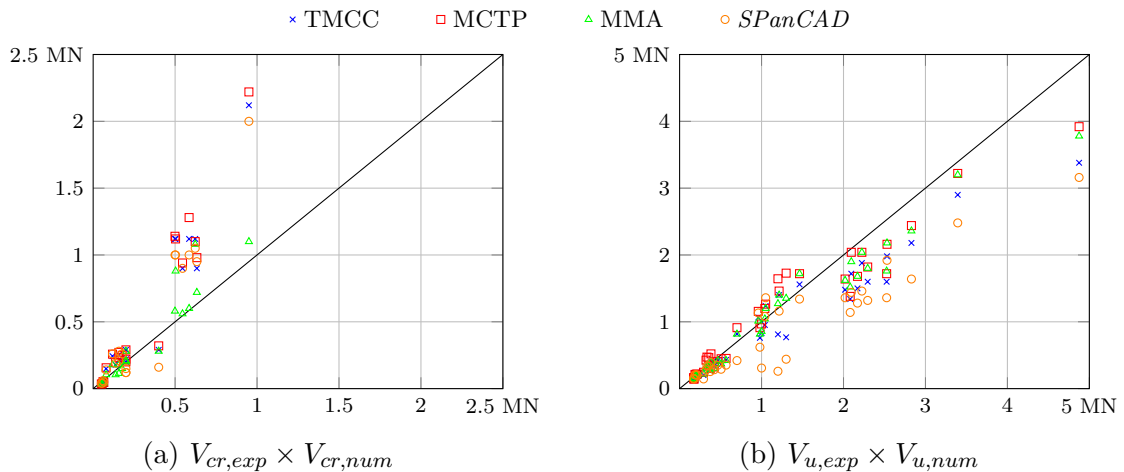
Todas as dimensões utilizadas nos elementos são apresentadas, em mm, na Tabela 5.2, sendo que L representa o comprimento da viga; n , o número de painéis considerados nas regiões detalhadas; b_w , a espessura do elemento; d_e , a profundidade efetiva, ou seja, a distância vertical entre os eixos das bielas superior e inferior; a , a distância entre a força aplicada e o apoio mais próximo; e g , a altura secundária dos consolos não-retangulares ou a medida da abertura nas vigas de Jasim, Allawi e Oukaili (2019).

Também se observa na Tabela 5.2 que são adotadas 5 geometrias distintas para as bielas em cada modelo (S_1 a S_5), representadas em cores diferentes na Figura 5.12. Além disso, a Tabela 5.2 apresenta as armaduras de cada elemento, onde A_s e A'_s são as armaduras das bielas inferior e superior, respectivamente; e A_{sx} e A_{sy} são as armaduras distribuídas nas direções x (longitudinal) e y (transversal), respectivamente. Outras informações, como as propriedades dos materiais utilizados, podem ser encontrados nas fontes citadas.

Cada elemento foi analisado a partir dos três modelos constitutivos implementados (TMCC, MCTP e MMA), sendo que, quando os parâmetros auxiliares do concreto não eram informados pelos autores, foram utilizados os parâmetros padrão do programa (Equação 4.1). De modo a obter melhores conclusões, os mesmos modelos foram analisados no *SPanCAD*.

A Figura 5.9 apresenta os diagramas de dispersão das cargas de fissuração (V_{cr}) e de ruína (V_u) para os modelos analisados, sendo que o eixo horizontal representa o resultado experimental e o eixo vertical, o resultado numérico. Observando a dispersão dos carregamentos de ruína (Figura 5.9b), é possível concluir que o *SPMTool* obteve estimativas mais consistentes que o *SPanCAD*.

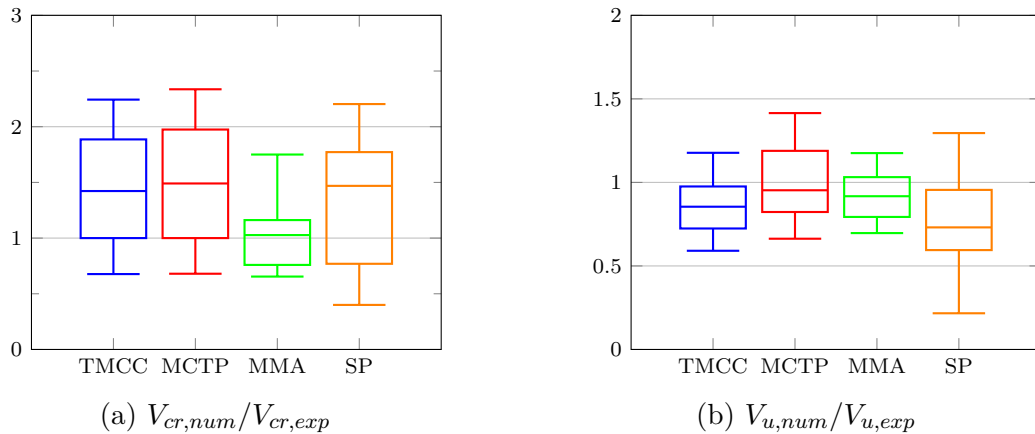
Figura 5.9 – Diagramas de dispersão para os modelos analisados



Nota: as análises pelos modelos TMCC, MCTP e MMA foram realizadas pelo *SPMTool*.

A Figura 5.10 mostra os diagramas de caixa que resumem todos os resultados obtidos nas análises. Em ambos, se visualiza que o MMA apresentou as melhores amplitudes de resposta entre os modelos do *SPMTool*. Além disso, também é possível observar a maior distribuição de resultados do *SPanCAD*.

Figura 5.10 – Diagramas de caixa para os modelos analisados



Nota: SP representa o *SPanCAD*.

Os resultados foram resumidos na Tabela 5.1, na qual são apresentadas as cargas experimentais de fissuração diagonal (V_{cr}), quando disponíveis, e de ruína (V_u), e as relações obtidas entre os valores numéricos e experimentais. Em todas as simulações ocorreu ruptura por escoamento da armadura distribuída, seguido por esmagamento da diagonal comprimida. Nos casos de vigas com abertura, esse rompimento se dava nos painéis adjacentes (acima ou abaixo) à abertura.

Nos casos de vigas-parede, o MMA apresentou melhores previsões para a carga de fissuração. Isso pode ser explicado pelo fato deste modelo considerar o efeito de Poisson na avaliação das deformações. Quanto às cargas de ruína, os três métodos resultaram em valores muito próximos e, em todos os casos, apresentaram melhores estimativas que o *SPanCAD*. Conforme se observa na Figura 5.9b, os modelos numéricos tendem a fornecer estimativas conservadoras para o carregamento último.

A Figura 5.11 apresenta os diagramas de carga (V) $\times$ deslocamento (u) para as vigas de Ismail, Guadagnini e Pilakoutas (2016). Na maior parte dos casos, as maiores divergências entre as soluções numéricas e o experimento ocorrem no comportamento pré-fissuração. Uma vantagem que pode ser visualizada para o MMA é que este apresenta uma transição mais suave entre as regiões não-fissurada e fissurada. Nos outros modelos é possível observar um salto, que representa a fissuração da biela inferior.

Quanto aos resultados de vigas com abertura, os três modelos do *SPMTool* não apresentaram valores muito distantes. Exceto para a carga de fissuração, o *SPanCAD*

também apresentou boas previsões, já que se notou que este programa apresentava melhores resultados em modelos com maior discretização.

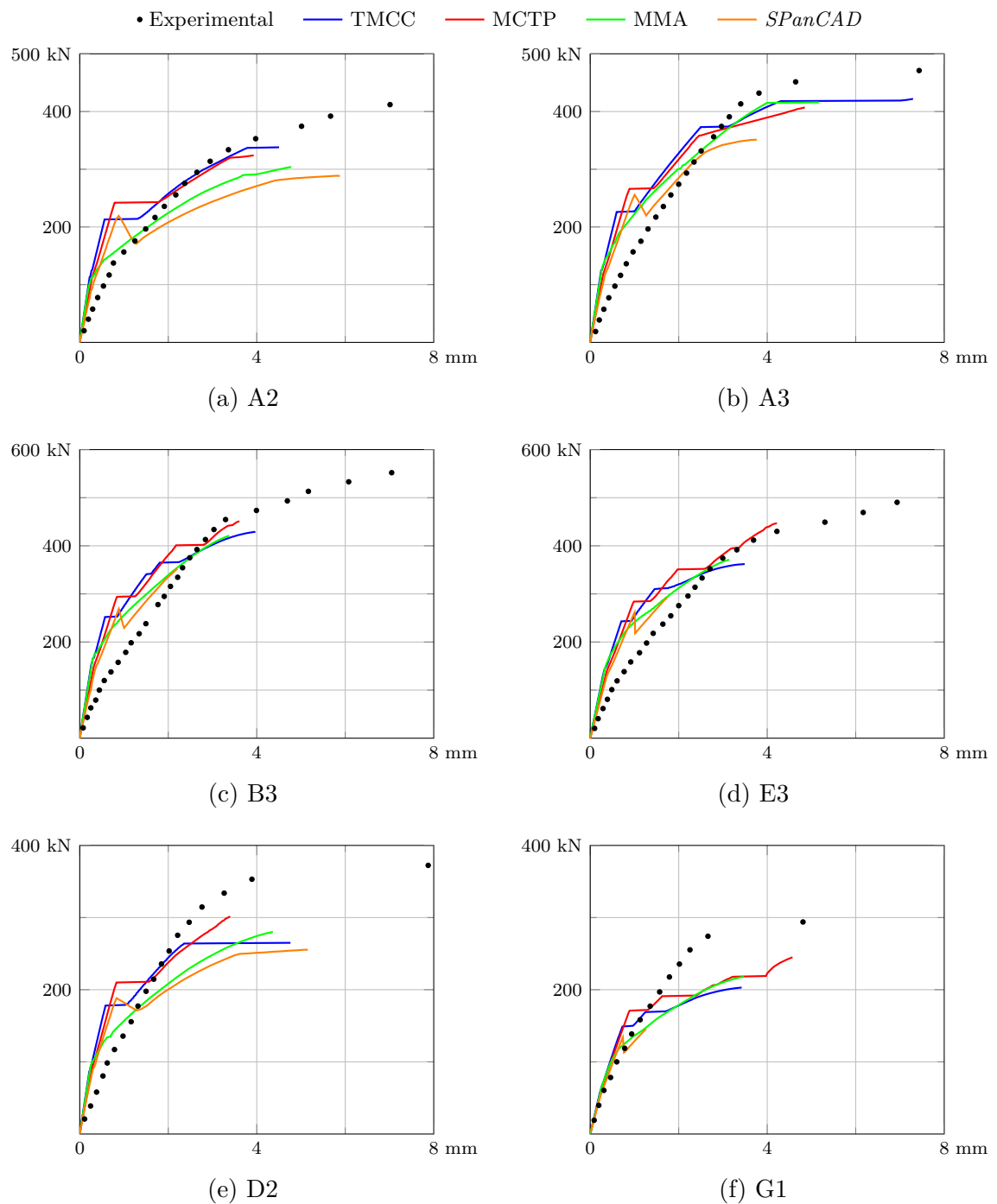
Nos casos de consolos, o MMA resultou no melhor valor médio e também na menor variação. O *SPanCAD* não apresentou bons resultados nesses casos, especialmente nos consolos de geometria não-retangular (série S de Foster, Powell e Selim (1996)), nos quais a ruptura ocorria muito antes do que se conseguiu nas análises experimentais.

No geral, o MMA conseguiu prever com melhor precisão a carga de fissuração e também a carga de ruína com a menor variação. Apesar de o MCTP resultar num valor muito próximo de 1 para esta última consideração, este modelo superestimou em até 40% a capacidade de carga nos consolos. Por este motivo, o MMA foi escolhido como o modelo padrão para as análises não-lineares no *SPMTool*, sendo que a utilização dos outros dois modelos continua disponível para escolha pelo usuário, como mostrado na Figura 5.5.

Em comparação ao *SPanCAD*, em média, o *SPMTool* conseguiu prever com maior precisão e menor variação os carregamentos analisados. O que mostra que, além de ser compatível com versões recentes do *AutoCAD*, também se obteve uma melhora nos resultados.

Por fim, foi possível verificar a qualidade dos resultados, já que as estruturas analisadas não foram dimensionadas pelo MBP pelos autores. Isso mostra que o programa tem aplicação em qualquer tipo de estrutura, tanto para análise estrutural, quanto para simulações.

Figura 5.11 – Gráficos força $\times$ deslocamento para as vigas de Ismail, Guadagnini e Pilakoutas (2016)



Nota: as análises pelos modelos TMCC, MCTP e MMA foram realizadas pelo *SPMTool*.

Tabela 5.1 – Resultados não-lineares do SPMTool

Vigas-parede													
Ref.	ID	$V_{cr,exp}$		$V_{u,exp}$	$V_{cr}/V_{cr,exp}$				$V_u/V_{u,exp}$				
		(kN)	(kN)		TMCC	MCTP	MMA	SP	TMCC	MCTP	MMA	SP	
Birrcher et al. (2009)	I-03-2	632,7	2530,9	1,42	1,55	1,14	1,50	0,78	0,85	0,86	0,76		
	I-02-2	545,2	2019,4	1,65	1,72	1,03	1,65	0,73	0,81	0,80	0,67		
	II-03-CCC2021	622,7	2224,0	1,80	1,77	1,73	1,69	0,85	0,92	0,92	0,66		
	II-02-CCC1021	585,3	1463,4	1,91	2,19	1,03	1,71	1,07	1,18	1,18	0,92		
	II-03-CCT1021	-	2828,9	-	-	-	-	0,77	0,86	0,83	0,58		
	II-02-CCT0521	-	2526,5	-	-	-	-	0,63	0,68	0,70	0,54		
	III-1.85-02	499,2	2170,6	2,24	2,28	1,16	2,00	0,69	0,77	0,77	0,59		
	III-1.85-025	-	2295,2	-	-	-	-	0,70	0,79	0,78	0,58		
	III-1.85-01	-	1214,3	-	-	-	-	1,15	1,20	1,15	0,96		
	III-1.85-03b	502,8	2095,0	2,23	2,23	1,75	1,99	0,82	0,97	0,91	0,69		
	III-1.85-02b	-	2081,7	-	-	-	-	0,64	0,66	0,73	0,55		
	IV-2175-1.85-02	950,2	3393,8	2,23	2,34	1,16	2,10	0,85	0,95	0,94	0,73		
	M-03-2-CCC2436	-	4875,0	-	-	-	-	0,69	0,80	0,78	0,65		
	Ismail, Guadagnini e Pilakoutas (2016)	A2	157,0	422,0	1,36	1,49	0,73	1,34	0,80	0,77	0,72	0,67	
A3		177,0	466,0	1,28	1,41	0,85	1,47	0,91	0,87	0,89	0,75		
B3		157,0	567,0	1,61	1,73	1,13	1,78	0,76	0,80	0,74	0,62		
D2		137,0	373,0	1,31	1,42	0,76	1,39	0,71	0,81	0,75	0,68		
E3		118,0	506,0	2,06	2,19	1,52	2,20	0,72	0,88	0,73	0,57		
G1		79,0	292,0	1,89	1,97	1,32	1,77	0,70	0,84	0,75	0,50		
Coeficiente de variação (%)		Média		1,77	1,87	1,18	1,74	0,79	0,86	0,84	0,67		
				19,53	17,83	26,99	15,23	16,70	15,58	15,76	17,52		

Tabela 5.1 (continuação)

Vigas com abertura											
Ref.	ID	$V_{cr,exp}$		$V_{u,exp}$		$V_{cr}/V_{cr,exp}$				$V_u/V_{u,exp}$	
		(kN)	(kN)	(kN)	(kN)	TMCC	MCTP	MMA	SP	TMCC	MCTP
Ashour e Rishi (2000)	A-E-S	400,00	920,00	0,73	0,80	0,70	0,40	0,76	0,83	0,68	0,78
	A-I-S	200,00	600,00	1,00	1,00	1,00	0,60	1,17	1,08	1,03	1,17
	B-E-S	200,00	883,00	1,45	1,45	1,40	0,75	0,67	0,70	0,61	0,66
	B-I-S	200,00	698,00	1,10	1,10	1,05	0,60	0,87	0,85	0,86	0,83
Jasim, Allawi e Oukaili (2019)	DP-C-200	60,00	190,00	1,00	0,87	0,83	1,00	1,11	1,12	1,03	1,08
	DP-E-200	65,00	210,00	0,68	0,71	0,68	0,77	0,94	1,03	0,92	0,97
	DP-C-230	55,00	175,00	0,69	0,69	0,65	0,73	0,87	0,82	0,82	0,83
	DP-E-230	50,00	170,00	0,68	0,68	0,68	0,80	0,93	0,95	0,92	0,89
Média				0,92	0,91	0,87	0,71	0,91	0,92	0,86	0,90
Coeficiente de variação (%)				28,20	27,18	28,02	23,38	16,81	14,81	16,47	17,17

Tabela 5.1 (continuação)

Consolos										
Ref.	ID	$V_{cr,exp}$		$V_{cr}/V_{cr,exp}$				$V_u/V_{u,exp}$		
		(kN)	$V_{u,exp}$ (kN)	TMCC	MCTP	MMA	SP	TMCC	MCTP	MMA
Chakrabarti, Farahi e Kashou (1989)	TA-1	-	322,5	-	-	-	-	0,93	1,32	0,95
	TA-2	-	333,6	-	-	-	-	1,07	1,41	1,04
	TA-3	-	356,0	-	-	-	-	1,01	1,30	0,98
	TA-4	-	382,5	-	-	-	-	1,06	1,36	1,04
Foster, Powell e Selim (1996)	SA2	-	1300,0	-	-	-	-	0,59	1,33	1,04
	SB2	-	1200,0	-	-	-	-	0,68	1,37	1,06
	SC1-3	-	700,0	-	-	-	-	1,17	1,30	1,17
	SC2-1	-	980,0	-	-	-	-	0,78	0,93	0,83
	SD1	-	1000,0	-	-	-	-	0,83	0,97	0,86
	PC2	-	1040,0	-	-	-	-	0,92	1,15	1,00
	PD2	-	960,0	-	-	-	-	1,11	1,20	1,05
	PF2	-	1050,0	-	-	-	-	1,18	1,20	1,14
	Média		-	-	-	-	-	0,94	1,24	1,01
	Coeficiente de variação (%)		-	-	-	-	-	19,46	11,97	9,41

Nota: as cargas de fissuração dos consolos não foram informadas pelos autores.

Resumo Geral (39 elementos)										
$V_{cr}/V_{cr,exp}$								$V_u/V_{u,exp}$		
		TMCC	MCTP	MMA	SP	TMCC	MCTP	MMA	SP	
Média		1,42	1,47	1,06	1,33	0,85	0,98	0,88	0,74	
Coeficiente de variação (%)		34,25	30,98	25,59	41,85	19,30	22,18	15,45	32,47	

Nota: as análises pelo TMCC, MCTP e MMA foram executadas no *SPMTool*, e SP se refere aos resultados do *SPanCAD*.

Figura 5.12 – Geometria dos modelos analisados no SPMTool

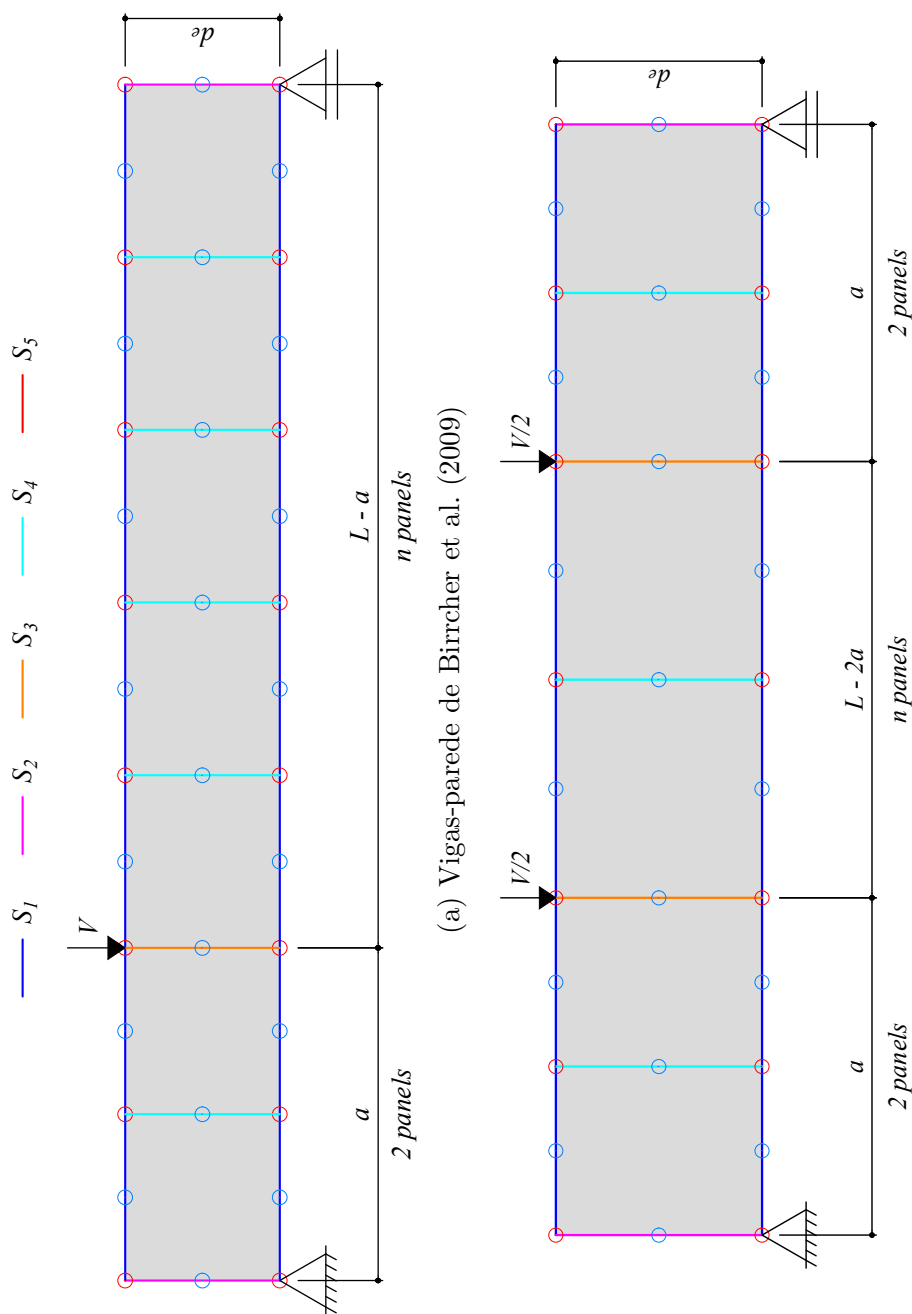
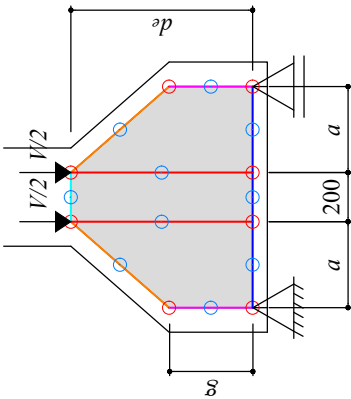


Tabela 5.2 – Geometria e armaduras dos modelos biela-painel analisados no SPMTool

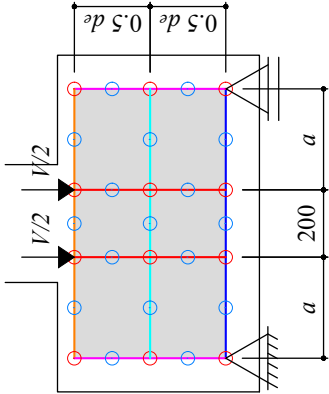
ID	Geometria (mm)						Altura da biela (mm)						Armaduras		
	<i>L</i>	<i>n</i>	<i>b_w</i>	<i>d_e</i>	<i>a</i>	<i>g</i>	<i>S₁</i>	<i>S₂</i>	<i>S₃</i>	<i>S₄</i>	<i>S₅</i>	<i>A_s</i>	<i>A'_s</i>	<i>A_{sx}</i>	<i>A_{sy}</i>
I-03-2	6483,4	5	533,4	838,2	1799,3	-	279,4	406,4	508,0	100,0	-	42 ϕ 19,1	6 ϕ 35,8	ϕ 12,7 c/ 146,1	ϕ 12,7 c/ 165,1
I-02-2	.	.	.	.	.	-	.	.	.	.	-	.	.	ϕ 12,7 c/ 241,3	ϕ 12,7 c/ 241,3
II-03-CCC2021	.	.	.	894,1	1804,0	-	172,7	254,0	.	.	-	12 ϕ 35,8	.	ϕ 15,9 c/ 167,6	ϕ 15,9 c/ 241,3
II-02-CCC1021	.	.	.	.	.	-	.	.	254,0	.	-	.	.	ϕ 12,7 c/ 256,5	ϕ 15,9 c/ 381
II-03-CCT1021	.	.	.	.	.	-	.	.	914,4	.	-	.	.	ϕ 15,9 c/ 167,6	ϕ 15,9 c/ 241,3
II-02-CCT0521	.	.	.	.	.	-	.	127,0	508,0	.	-	.	.	ϕ 12,7 c/ 256,5	ϕ 15,9 c/ 381
(a) III-1.85-02	.	.	.	.	.	-	.	406,4	.	.	-	.	.	.	ϕ 15,9 c/ 368,3
III-1.85-025	.	.	.	.	.	-	.	.	.	.	-	.	.	ϕ 9,5 c/ 193	ϕ 15,9 c/ 304,8
III-1.85-01	.	.	.	.	.	-	.	.	.	.	-	.	.	.	ϕ 12,7 c/ 457,2
III-1.85-03b	.	.	.	.	.	-	.	.	.	.	-	.	.	ϕ 15,9 c/ 256,5	ϕ 12,7 c/ 152,4
III-1.85-02b	.	.	.	.	.	-	.	.	.	.	-	.	.	ϕ 12,7 c/ 256,5	ϕ 12,7 c/ 241,3
IV-2175-1.85-02	.	4	.	1595,1	3237,6	-	309,9	.	406,4	.	-	22 ϕ 35,8	12 ϕ 35,8	.	.
M-03-2-CCC2436	.	5	914,4	812,8	1879,6	-	406,4	.	609,6	.	-	27 ϕ 35,8	4 ϕ 35,8	ϕ 15,9 c/ 165,1	ϕ 22,2 c/ 279,4
A2	1400,0	1	100,0	260,0	550,0	-	140,0	100,0	100,0	50,0	-	6 ϕ 16	2 ϕ 12	ϕ 6 c/ 260	ϕ 8 c/ 180
A3	.	.	.	.	.	-	.	.	.	.	-	.	.	.	ϕ 8 c/ 80
B3	.	2	.	.	425,0	-	.	.	.	.	-	.	.	.	ϕ 8 c/ 75
D2	.	1	.	.	550,0	-	.	.	.	.	-	.	.	.	ϕ 8 c/ 180
E3	.	2	.	.	425,0	-	.	.	.	.	-	.	.	.	ϕ 8 c/ 75
G1	.	1	.	.	550,0	-	.	.	.	.	-	.	.	.	ϕ 8 c/ 180

Notas: “.” indica que o valor se repete da linha acima e “-” indica que o valor é nulo. Diâmetro e espaçamento das armaduras em mm.

Figura 5.12 (continuação)



(c) Consolos série S de Foster et al. (1996)

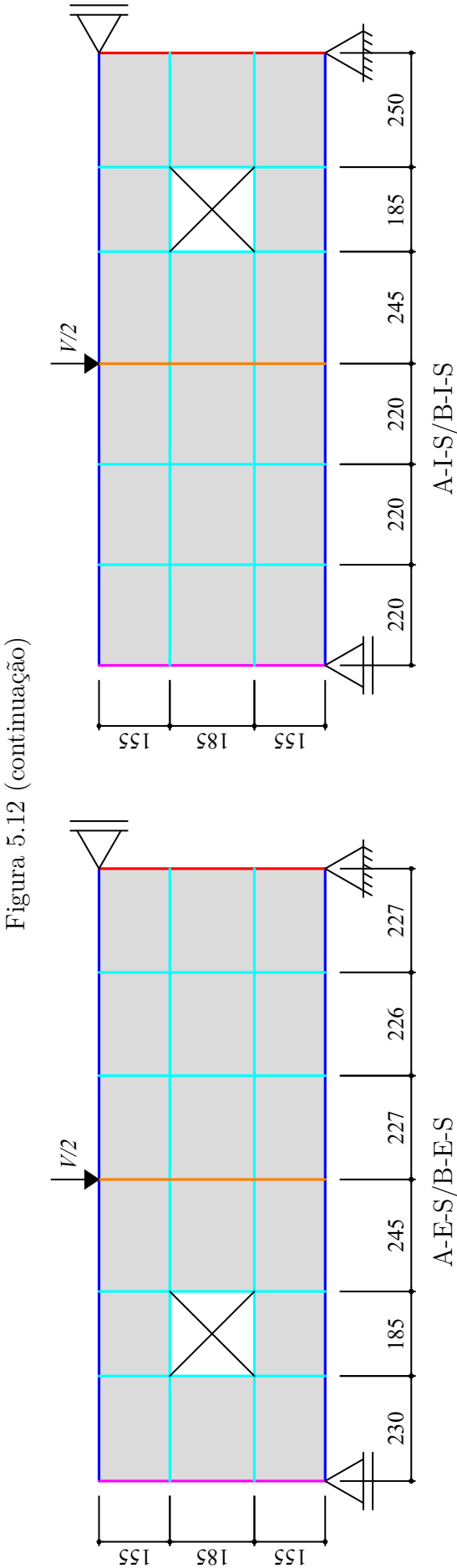


(d) Consolos série P de Foster et al. (1996) e Chakrabarti et al. (1989)

Tabela 5.2 (continuação)

Geometria (mm)							Altura da biela (mm)					Armaduras			
ID	L	n	b _w	d _e	a	g	S ₁	S ₂	S ₃	S ₄	S ₅	A _s	A' _s	A _{sx}	A _{sy}
SA2	-	-	150,0	740,0	250,0	340	120	300	210	100	200	6 ϕ20	-	ϕ6 c/ 56,9	-
SB2	-	-	.	.	.	290	.	.	.	.	.	4 ϕ20	-	.	-
(c) SC1-3	-	-	125,0	600,0	300,0	.	200	250	160	.	.	6 ϕ12	-	ϕ6 c/ 46,2	-
SC2-1	-	-	.	.	.	.	.	.	.	.	.	6 ϕ20	-	.	-
SD1	-	-	.	.	.	.	.	.	.	.	.	.	-	.	-
PC2	-	-	150,0	450,0	300,0	-	200	150	100	50	200	6 ϕ12	-	ϕ10 c/ 40,9	-
PD2	-	-	.	.	.	-	.	200	.	.	.	3 ϕ28	-	.	-
PF2	-	-	.	.	150,0	-	.	100	.	.	.	6 ϕ12	-	.	-
(d) TA-1	-	-	152,0	183,34	76,0	-	22,7	76,2	22,7	22,7	76,2	3 ϕ9,5	-	ϕ9 c/ 51,5	-
TA-2	-	-	.	207,74	.	-	23,4	.	23,4	23,4	.	.	-	ϕ9 c/ 57,8	-
TA-3	-	-	.	183,34	.	-	22,7	.	22,7	22,7	.	.	-	ϕ9 c/ 51,5	-
TA-4	-	-	.	207,74	.	-	23,4	.	23,4	23,4	.	.	-	ϕ9 c/ 57,8	-

Notas: “.” indica que o valor se repete da linha acima e “-” indica que o valor é nulo. Diâmetro e espaçamento das armaduras em mm.



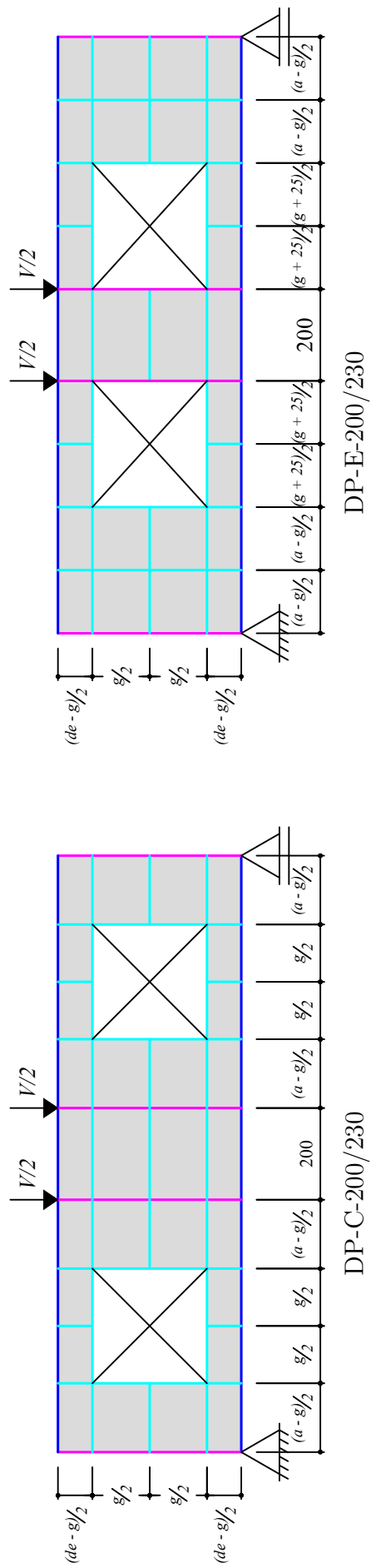
(e) Vigas com abertura de Ashour e Rishi (2000)

Tabela 5.2 (continuação)

ID	Geometria (mm)						Altura da biela (mm)					Armaduras			
	L	n	b _w	d _e	a	g	S ₁	S ₂	S ₃	S ₄	S ₅	A _s	A' _s	A _{sx}	A _{sy}
A-E-S	1340	-	120	495	660	-	130	120	200	60	160	4 ϕ 12	6 ϕ 11,4	ϕ 8 c/ 100	ϕ 8 c/ 100
A-I-S	.	-	.	.	.	-	.	.	.	.	.	.	.	.	.
B-E-S	.	-	.	.	.	-	.	.	.	.	.	.	.	ϕ 8 c/ 200	.
B-I-S	.	-	.	.	.	-	.	.	.	.	.	.	.	.	.

Notas: “.” indica que o valor se repete da linha acima e “_” indica que o valor é nulo. Diâmetro e espaçamento das armaduras em mm.

Figura 5.12 (continuação)



(f) Vigas com abertura de Jasim, Allawi e Oukaili (2019)

Tabela 5.2 (continuação)

ID	Geometria (mm)					Altura da biela (mm)							Armaduras		
	L	n	b_w	d_e	a	g	S_1	S_2	S_3	S_4	S_5	A_s	A'_s	A_{sx}	A_{sy}
DP-C-200	1300	-	150	400	550	250	100	100	-	50	-	3 $\phi 16$	2 $\phi 6$	$\phi 6$ c/ 86	$\phi 6$ c/ 90
DP-E-200	.	-	.	.	.	.	.	.	-	.	-	.	.	.	.
DP-C-230	.	-	.	.	.	280	.	.	-	.	-	.	.	.	.
DP-E-230	.	-	.	.	.	.	.	.	-	.	-	.	.	.	.

Notas: “.” indica que o valor se repete da linha acima e “-” indica que o valor é nulo. Diâmetro e espaçamento das armaduras em mm.

6 Aplicação do MBP ao dimensionamento de estruturas de concreto

Este capítulo apresenta a formulação utilizada na prática de dimensionamento de modelos biela-painel. Para isso, foram adaptadas algumas recomendações da ABNT NBR 6118:2014 e do Model Code 2010 para modelos de escoras e tirantes, e aplicadas algumas relações da TMCC simplificada (BENTZ; VECCHIO; COLLINS, 2006).

Serão abordados os procedimentos de dimensionamento das armaduras e verificação das tensões no concreto para as bielas e os painéis, a partir de resultados de análises lineares em ELU. Além disso, serão idealizados exemplos práticos de elementos especiais de concreto, sendo estes dimensionados com a aplicação do *SPMTool*.

6.1 Limitação da tensão de compressão no concreto

Para o dimensionamento de estruturas sujeitas ao estado biaxial de tensões, como é o caso dos Modelos de Escoras e Tirantes ou Modelos Biela-Painel, a resistência à compressão do concreto será afetada pela existência de tensões transversais de tração, efeito esse conhecido como abrandamento.

A ABNT NBR 6118:2014 considera três parâmetros de resistência para a verificação de Modelos de Escoras e Tirantes, que podem ser também aplicados ao dimensionamento pelo MBP. Estes fatores dependem da existência de tirantes transversais:

- (a) Bielas prismáticas (compressão uniaxial):

$$f_{cd1} = 0,85 \alpha_{V2} f_{cd} \quad (6.1a)$$

- (b) Bielas atravessadas por mais de um tirante:

$$f_{cd2} = 0,60 \alpha_{V2} f_{cd} \quad (6.1b)$$

- (c) Bielas atravessadas por um único tirante:

$$f_{cd3} = 0,72 \alpha_{V2} f_{cd} \quad (6.1c)$$

sendo:

$$f_{cd} = \frac{f_{ck}}{\gamma_c} \quad \alpha_{V2} = 1 - \frac{f_{ck}}{250}$$

Nas equações acima, f_{cd} representa a resistência de cálculo do concreto; f_{ck} , a resistência característica, dada em MPa; e γ_c o fator de ponderação de resistência do concreto, sendo igual a 1,4 em combinações normais em ELU.

Já o *fib* Model Code 2010 (2013) considera três situações distintas, nas quais a resistência reduzida do concreto é calculada por:

$$f'_{cd} = k_c f_{cd} \quad (6.2)$$

sendo:

$$k_c = \begin{cases} 1,00 \eta_{f_c}, & \text{no caso de compressão uniaxial ou biaxial;} \\ 0,75 \eta_{f_c}, & \text{para armadura tracionada perpendicular à compressão;} \\ 0,55 \eta_{f_c}, & \text{para armadura tracionada oblíqua à compressão.} \end{cases}$$

O coeficiente η_{f_c} é dado por:

$$\eta_{f_c} = \left(\frac{30}{f_{ck}} \right)^{1/3} \leq 1,0 \quad (6.3)$$

com f_{ck} em MPa.

6.2 Consideração da contribuição do concreto ao cisalhamento

A formulação clássica de dimensionamento das armaduras dos painéis (HOOGEN-BOOM, 1998; SIMONE, 1998) aplica as condições do MPT (seção 2.6), que despreza a contribuição do concreto ao cisalhamento e considera que as tensões médias nas armaduras sejam iguais às suas resistências de cálculo (condição de escoamento das armaduras).

Em estudos anteriores, observou-se que esta formulação resultava em maiores taxas de armaduras distribuídas, em relação ao MB (MELLO, 2015; MELLO; SOUZA, 2016), já que é sabido que as tensões nas armaduras serão críticas na áreas fissuradas e reduzidas nas regiões entre fissuras.

De modo a reduzir as taxas calculadas, este trabalho propõe a consideração da contribuição do concreto à resistência ao cisalhamento nos painéis, que se dá principalmente através do engrenamento dos agregados nas regiões fissuradas e por uma pequena resistência à tração nas áreas entre fissuras.

Considerando as equações de equilíbrio na fissura (subseção 2.7.3), desprezadas as tensões normais (σ_{ci}), é possível escrever para um elemento de membrana submetido ao estado de cisalhamento puro (Figura 6.1):

$$\tau_{xy} = -\tau_{ci} + \rho_{sx} \sigma_{sxcr} \cotg(\theta_{c1}) \quad \tau_{xy} = \tau_{ci} + \rho_{sy} \sigma_{sy cr} \tg(\theta_{c1}) \quad (6.4)$$

Bentz, Vecchio e Collins (2006) desenvolveram uma formulação simplificada da TMCC, na qual a tensão de cisalhamento (τ_{ci}) transmitida por meio do engrenamento

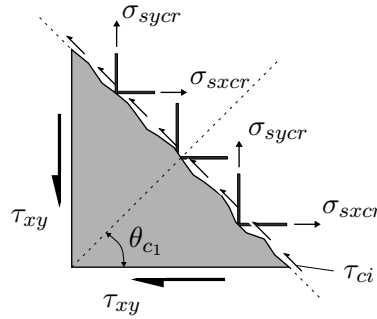
dos agregados é estimada a partir de um coeficiente β . Este coeficiente foi obtido a partir das relações constitutivas do concreto à tração, desprezando a deformação transversal no elemento ($\varepsilon_y = 0$), por esta ser muito inferior à ε_x em elementos lineares. A tensão τ_{ci} é determinada por:

$$\tau_{ci} = \beta \sqrt{f'_c} \quad (6.5)$$

com:

$$\beta = \frac{0.4}{1 + 1500 \varepsilon_x} \frac{1300}{1000 + s_{xe}} \quad s_{xe} = k_s s_x \quad k_s = \frac{35}{\phi_a + 16} \quad (6.6)$$

Figura 6.1 – Equilíbrio na fissura para um elemento de membrana submetido ao estado de cisalhamento puro



Na Equação 6.6, ε_x é a deformação média na direção x ; s_x é a distância média entre fissuras, expressa em mm; e ϕ_a é o diâmetro máximo do agregado no concreto, em mm.

Bentz, Vecchio e Collins (2006) também fornecem uma equação para o cálculo do ângulo da tensão principal de compressão (θ_{c2}). Adaptando para o cálculo de θ_{c1} , tem-se:

$$\theta_{c1} = 90^\circ - (29^\circ + 7000 \varepsilon_x) \left(0,88 + \frac{s_{xe}}{2500} \right) \geq 25^\circ \quad (6.7)$$

Na Equação 6.7, os termos representados já foram convertidos para que se obtenha o resultado em graus.

A deformação ε_x pode ser estimada de acordo com o Model Code 2010, quando não haver força normal na seção transversal, por:

$$\varepsilon_x = \frac{1}{2 E_s A_s} \left(\frac{M_d}{z} + V_d \right) \quad (6.8)$$

sendo E_s e A_s o módulo de elasticidade e a área de aço da armadura principal de tração na seção transversal; M_d e V_d , o momento fletor e a força cortante na seção transversal onde M_d é máximo; e z a distância entre as resultantes de tração e compressão na seção transversal (distância entre os centros de gravidade das bielas inferior e superior). Num modelo simples, a relação M_d/z será equivalente à força de tração na armadura A_s .

Substituindo a relação de Kaklauskas, Ramanauskas e Ng (2019) para a distância s_x (2.54), reescreve-se s_{xe} :

$$s_{xe} = k_s \left(21 + 0,155 \frac{\phi_{sx}}{\rho_{sx}} \right) \quad (6.9)$$

sendo ϕ_{sx} e ρ_{sx} o diâmetro das barras (em mm) e a taxa de armadura na direção x .

Observa-se que o coeficiente β depende da taxa de armadura na direção x . Assim, reescreve-se as equações de equilíbrio:

$$\tau_{xy} = -\beta \sqrt{f'_c} + \rho_{sx} \sigma_{sacr} \cotg(\theta_{c1}) \quad \tau_{xy} = \beta \sqrt{f'_c} + \rho_{sy} \sigma_{sydr} \tg(\theta_{c1}) \quad (6.10)$$

Para casos onde exista apenas armadura transversal ($\rho_{sx} = 0$), o Model Code 2010 fornece uma estimativa conservadora para β , por meio do coeficiente k_v . Para o nível II de aproximação, tem-se:

$$k_v = \frac{0.4}{1 + 1500 \varepsilon_x} \frac{1300}{1000 + k_{dg} z} \quad k_{dg} = \frac{32}{\phi_a + 16} \geq 0,75 \quad (6.11)$$

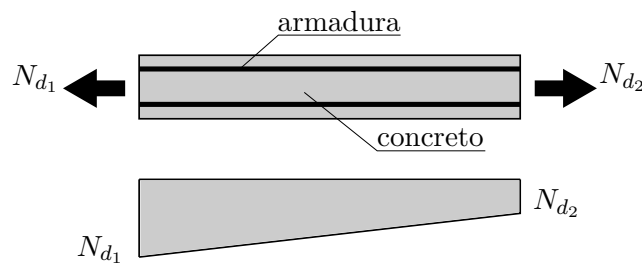
sendo z a distância entre as resultantes de tração e compressão na seção transversal, em mm.

Estas equações podem ser utilizadas para a obtenção das armaduras dos painéis, conforme será abordado em 6.4.

6.3 Dimensionamento das bielas

Considera-se uma biela (Figura 6.2) sujeita a duas forças normais (N_{d1} e N_{d2}), já consideradas com seus valores de cálculo. Conforme visto na seção 3.2, o esforço normal na biela terá variação linear devido ao equilíbrio com o painel adjacente.

Figura 6.2 – Esforços atuantes numa biela



Fonte: adaptado de Hoogenboom (1998).

6.3.1 Força de tração

Para um esforço de tração, desconsiderando a parcela resistente do concreto devido à fissuração, a área de aço (A_s) necessária a resistir ao esforço normal de cálculo (N_d) é calculada por:

$$A_s = \frac{N_d}{f_{yd}} \quad (6.12)$$

sendo f_{yd} a tensão de escoamento de cálculo do aço.

Esta armadura deve ser dimensionada para o valor máximo do esforço normal na biela e, segundo Nielsen e Hoang (2011), deve estendida por todo o sistema de uma biela, até as extremidades do elemento estrutural, sem ser reduzida ou cortada. Além disso, essa armadura deve possuir os espaçamentos mínimos entre as barras de acordo com a normativa vigente e, no caso dos tirantes inferiores de vigas-parede, deve estar disposta em várias camadas, cobrindo grande parte da biela tracionada.

6.3.2 Força de compressão

No caso de força normal de compressão, deve ser verificada a tensão no concreto (σ_c), limitada a um valor reduzido da resistência à compressão (f'_{cd}):

$$\sigma_c = \frac{|N_d|}{A_c} \leq f'_{cd} \quad (6.13)$$

sendo A_c a área da seção de concreto da biela e N_d a máxima força normal de compressão em valor absoluto.

A tensão limite f'_{cd} deve levar em conta a existência ou não de tensões transversais de tração. Sugere-se que sejam considerados os seguintes casos:

- (a) Bielas de borda (que sejam adjacentes a um único painel): caso de compressão uniaxial, sem existência de tração transversal;
- (b) Bielas internas (que se localizam entre dois painéis) ou atravessadas por um tirante: caso de compressão com tração transversal.

A critério de exemplo, caso se usem os parâmetros da ABNT NBR 6118:2014, o caso (a) será limitado a f_{cd1} e o caso (b) a f_{cd2} , calculados em (6.1). Caso se utilizem os parâmetros do Model Code 2010 ((6.2)), o parâmetro k_c será igual a $1,0\eta_{fc}$ para o caso (a) e $0,75\eta_{fc}$ para o caso (b).

Se a tensão de compressão ultrapassar o valor definido para f'_{cd} , uma opção seria aumentar a largura da seção transversal, o que reduziria a tensão atuante. Se isto não for possível, Simone (1998) recomenda que se disponibilize uma armadura de compressão, de modo que o concreto seja confinado e obtenha uma maior resistência a compressão. Essa armadura é dada por duas relações, uma longitudinal ($A_{sc,l}$) e uma transversal ($A_{sc,t}$) em forma de estribos, calculadas de acordo com as seguintes expressões:

$$A_{sc,l} = \frac{|N_d| - A_c f'_{cd}}{f_{yd} - f'_{cd}} \quad A_{sc,t} = \frac{1}{4} \rho_{conf} \phi_c s \quad (6.14)$$

com:

$$\rho_{conf} = \frac{f'_{cd}}{2 f_{yd}} \left(\frac{A_c}{A_{conf}} - 1 \right)$$

sendo: A_{conf} : área da seção de concreto confinada pela armadura de confinamento;

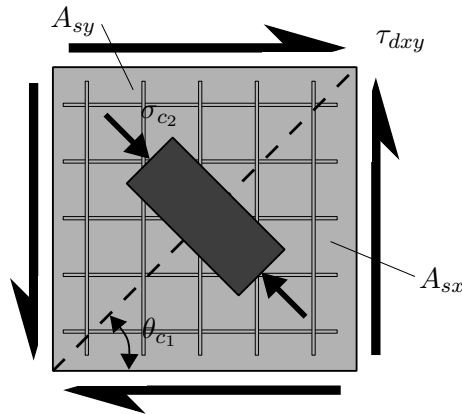
- ρ_{conf} : taxa geométrica de armadura transversal de confinamento;
 ϕ_c : diâmetro da armadura transversal de confinamento;
 s : espaçamento da armadura transversal de confinamento.

Caso exista uma força de tração em uma das extremidades da biela e uma força de compressão na outra, as armaduras deverão ser dimensionadas para a força de tração e as tensões no concreto verificadas para a força de compressão.

6.4 Dimensionamento dos painéis

Os painéis podem ser tratados como um elemento de membrana sujeito ao estado de cisalhamento puro, conforme ilustrado na Figura 6.3. Dessa forma, este trabalho propõe um processo iterativo de dimensionamento, considerando as equações simplificadas da TMCC, descritas na seção 6.2.

Figura 6.3 – Painel de concreto armado submetido ao estado de cisalhamento puro



Considerando que o painel esteja submetido a uma tensão de cisalhamento (τ_{dxy}), já em seu valor de cálculo, será necessária a existência de armaduras distribuídas longitudinalmente (A_{sx}) e transversalmente (A_{sy}). De modo a proporcionar uma ruptura dúctil, as armaduras devem escoar antes do esmagamento do concreto.

Na situação de dimensionamento em ELU, deseja-se que as armaduras estejam escoando nas regiões fissuradas. Portanto, pode-se escrever o equilíbrio na situação de cálculo:

$$\tau_{dxy} = -\beta \bar{\tau}_{cd} + \rho_{sx} f_{ydx} \cotg(\theta_{c1}) \quad \tau_{dxy} = \beta \bar{\tau}_{cd} + \rho_{sy} f_{ydy} \tg(\theta_{c1}) \quad (6.15)$$

sendo f_{ydx} e f_{ydy} os valores de cálculo das tensões de escoamento das armaduras nas direções x e y , respectivamente; e:

$$\bar{\tau}_{cd} = \frac{\sqrt{f_{ck}}}{\gamma_c} \leq \frac{8}{\gamma_c} \text{ MPa} \quad (6.16)$$

A limitação de $\sqrt{f_{ck}}$ a 8 MPa é recomendada pelo Model Code 2010, devido à alta variabilidade da resistência ao cisalhamento de concretos de alta resistência (SIGRIST et al., 2013).

A situação mais econômica se trata de armaduras iguais nas duas direções ($\rho_{sx} = \rho_{sy} = \rho_s$ e $f_{ydx} = f_{ydy} = f_{yd}$). Nesse caso, pode-se facilmente obter a taxa ρ_s por meio de um processo iterativo, fazendo:

- (a) Estima-se a deformação normal ε_x (6.8), para a seção de momento máximo no modelo. Esta equação fornece uma aproximação conservadora para esta deformação, que será considerada constante para todos os painéis.
- (b) Determina-se a taxa inicial ρ_{s0} por meio do MPT:

$$\rho_{s0} = \frac{\tau_{dxy}}{f_{yd}} \quad (6.17a)$$

- (c) Com a taxa ρ_{si} , calculam-se o distanciamento s_{xe} (6.9), o coeficiente β (6.6), a tensão τ_{ci} (6.5) e o ângulo θ_{c1} (6.7);
- (d) Se obtém uma nova taxa ρ_{si} por meio da equação de equilíbrio em y :

$$\rho_{si} = \frac{\tau_{dxy} - \beta \bar{\tau}_{cd}}{f_{yd}} \cotg(\theta_{c1}) \quad (6.17b)$$

- (e) Compara-se ρ_{si} calculado com a estimativa anterior ρ_{si-1} , com um erro tolerável de 0,001%:

$$|\rho_{si} - \rho_{si-1}| \leq 0,001\% \quad (6.17c)$$

- (f) Se esta condição for satisfeita, adota-se a taxa ρ_{si} para as duas direções ($\rho_{sx} = \rho_{sy} = \rho_{si}$). Caso contrário, retorna-se ao passo (c) até satisfazer (6.17c).

Na prática de dimensionamento de um modelo biela-painel, pode-se dimensionar as armaduras do painel crítico por meio deste processo iterativo e adotar esta taxa para a direção x em todos os painéis do modelo. Dessa forma, calculam-se os parâmetros definidos no item (c), não sendo necessário iterar, já que a taxa ρ_{sx} já foi obtida anteriormente. A taxa ρ_{sy} é, finalmente, calculada a partir de (6.17b).

A verificação da tensão de compressão média no concreto pode ser feita a partir do MPT, desprezando a existência de tensões de tração entre as fissuras. Esta tensão é dada por:

$$\sigma_{c2} = -\tau_{dxy} (\tg \theta_{c1} + \cotg \theta_{c1}) \quad (6.18)$$

A tensão σ_{c2} deve ser limitada ao valor reduzido da resistência à compressão de cálculo do concreto (f'_{cd}), devido ao efeito de abrandamento causado pelas tensões transversais de tração:

$$|\sigma_{c2}| \leq f'_{cd} \quad (6.19)$$

Como exemplo, pelos parâmetros da ABNT NBR 6118:2014, a tensão deve ser limitada a f_{cd2} , calculada em (6.1b). Já pelos parâmetros do Model Code 2010 (6.2), o

parâmetro k_c será igual a $0,55 \eta_{fc}$, por existir armaduras tracionadas na direção oblíqua à tensão de compressão no concreto (σ_{c2}).

Caso a tensão exceda o valor de f'_{cd} , deve-se readequar o modelo por meio do aumento da largura da seção transversal, de modo que se satisfaça a condição estabelecida em (6.19).

6.5 Exemplos de aplicação

Esta seção apresenta exemplos de aplicação do *SPMTool* no dimensionamento de uma viga-parede contínua, uma viga-parede com aberturas e um consolo curto. Além disso, são efetuadas análises não-lineares com as armaduras calculadas a partir dos resultados lineares do programa, no próprio *SPMTool* e também no software comercial *ATENA 2D* (v. 5.7.0, 2019). Nesta último, foram propostos dois modelos para cada exemplo, um com maior e outro com menor discretização.

É importante destacar que as análises efetuadas no *ATENA* sem a modificação de parâmetros avançados e análise de convergência. Apenas foram feitas as configurações da malha, que serão expostas em cada exemplo, e dos materiais, de acordo com o descrito na seção 6.5.1. Essa proposta visa comparar as respostas dos dois programas, considerando a utilização por um usuário iniciante.

Para isso, serão considerados, em cada exemplo: as cargas em ELU (F_d), para o dimensionamento dos elementos; e as cargas em Estado Limite de Serviço (ELS) ($F_{d,ser}$), para verificações de deslocamentos e aberturas de fissuras. Para o primeiro caso, considerando as ações diretas, a carga F_d , segundo a ABNT NBR 6118:2014, é obtida por:

$$F_d = \gamma_g F_{gk} + \gamma_q \left(F_{q1k} + \sum \psi_{0j} F_{qjk} \right) \quad (6.20)$$

sendo:

- F_{gk} : ações permanentes diretas;
- F_{qk} : ações variáveis diretas, onde F_{q1k} representa a ação principal e F_{qjk} , as ações secundárias;
- γ_g, γ_q : coeficientes de ponderação das ações permanentes e variáveis, respectivamente. Obtidos na Tabela 11.1 da ABNT NBR 6118:2014, sendo utilizado, para ambos, o valor de 1,4 para as ações desfavoráveis em combinações normais;
- ψ_0 : coeficiente de ponderação que considera a baixa probabilidade da atuação simultânea das ações variáveis, obtidos na Tabela 11.2 da ABNT NBR 6118:2014.

Pela consideração da ABNT NBR 6118:2014 e da NBR 8681:2003, quando se trata de elementos especiais de concreto armado, o carregamento de cálculo deve ser adicionalmente

majorado através de um coeficiente γ_n . Nos exemplos considerados neste trabalho, será adotado o valor $\gamma_n = 1,10$. Portanto, o esforço final de cálculo será determinado por:

$$F_d = \gamma_n \left[\gamma_g F_{gk} + \gamma_q \left(F_{q1k} + \sum \psi_{0j} F_{qjk} \right) \right] \quad (6.21)$$

As verificações do Estado Limite de Deformações Excessivas (ELS-DEF) serão feitas a partir da Combinação Quase-Permanente (CQP) ($F_{d,CQP}$) e do Estado Limite de Abertura de Fissuras (ELS-W), a partir da Combinação Frequente (CF) ($F_{d,CF}$). De acordo com a ABNT NBR 6118:2014, estas combinações são definidas por:

$$F_{d,CQP} = \sum F_{gik} + \sum \psi_{2j} F_{qjk} \quad (6.22a)$$

$$F_{d,CF} = \sum F_{gik} + \psi_1 F_{q1k} + \sum \psi_{2j} F_{qjk} \quad (6.22b)$$

sendo os coeficientes ψ_1 e ψ_2 obtidos de acordo com a Tabela 11.2 da ABNT NBR 6118:2014.

É importante destacar que, por não ser o intuito deste trabalho, a aderência entre as barras de aço e concreto não foi verificada nos exemplos apresentados neste capítulo. Na prática, essa etapa é imprescindível e deve ser executada de acordo com a recomendação da norma vigente.

6.5.1 Propriedades dos materiais

Os exemplos desenvolvidos foram dimensionados considerando os materiais concreto classe C30 e aço CA-50, caracterizados na Tabela 6.1. As resistências de cálculo do concreto (f_{cd}) e do aço (f_{yd}) foram calculadas de acordo com a ABNT NBR 6118:2014, considerando os fatores de ponderação de 1,4 e 1,15, respectivamente.

Tabela 6.1 – Propriedades dos materiais utilizados nos dimensionamentos

Concreto C30		Aço CA-50	
Resistência característica (f_{ck})	30,00 MPa	Resistência característica (f_{yk})	500,00 MPa
Resistência de cálculo (f_{cd})	21,43 MPa	Resistência de cálculo (f_{yd})	434,78 MPa
Resistência reduzida f_{cd1}	16,03 MPa	Módulo de Elasticidade (E_s)	210,00 GPa
Resistência reduzida f_{cd2}	11,31 MPa	Deformação plástica (ε_y)	2,07 ‰
Resistência reduzida f_{cd3}	13,58 MPa	Deformação última (ε_{su})	10,00 ‰
Resistência característica à tração (f'_t)	2,02 MPa		
Módulo de Elasticidade (E_c)	25,92 GPa		
Deformação plástica (ε'_c)	2,00 ‰		
Deformação última (ε_{cu})	3,50 ‰		

As resistências reduzidas do concreto (f_{cd1} , f_{cd2} e f_{cd3}) também foram obtidas de acordo com a norma brasileira (Equação 6.1). Para os demais parâmetros do concreto, utilizaram-se as equações do modelo padrão do *SPMTool*, descritas em (4.1).

No *ATENA 2D*, utilizou-se o material *SBETA* para caracterizar o concreto, com as propriedades já descritas na Tabela 6.1. A resistência de corpo-de-prova cúbico (f_{cu}), necessária para a definição do material, foi obtido por: $f_{cu} = f_{ck}/0,85$. O modelo de abrandamento à tração foi definido como exponencial e o de fissuração foi definido como fixo (*fixed angle crack model*)

Os parâmetros secundários do concreto são obtidos automaticamente pela formulação do *ATENA* (CERVENKA; JENDELE; CERVENKA, 2021), exceto para f'_t e E_c , para os quais se utilizaram os valores já descritos na Tabela 6.1. O aço foi definido como bilinear, seguindo o modelo já utilizado pelo *SPMTool*. É importante frisar que todas as análises não-lineares consideraram as resistências características dos materiais.

6.5.2 Viga-parede contínua

A viga-parede contínua ilustrada na Figura 6.4a possui largura de 40 cm e recebe os carregamentos de dois pilares, distribuindo-os a outros três pilares num pavimento de garagem de um edifício de cinco pavimentos. As cargas permanente (P_g) e variável de utilização (P_q) foram estimadas em 850 kN e 150 kN, respectivamente.

Assim, calculam-se as combinações em ELU e ELS, seguindo as equações (6.21) e (6.22):

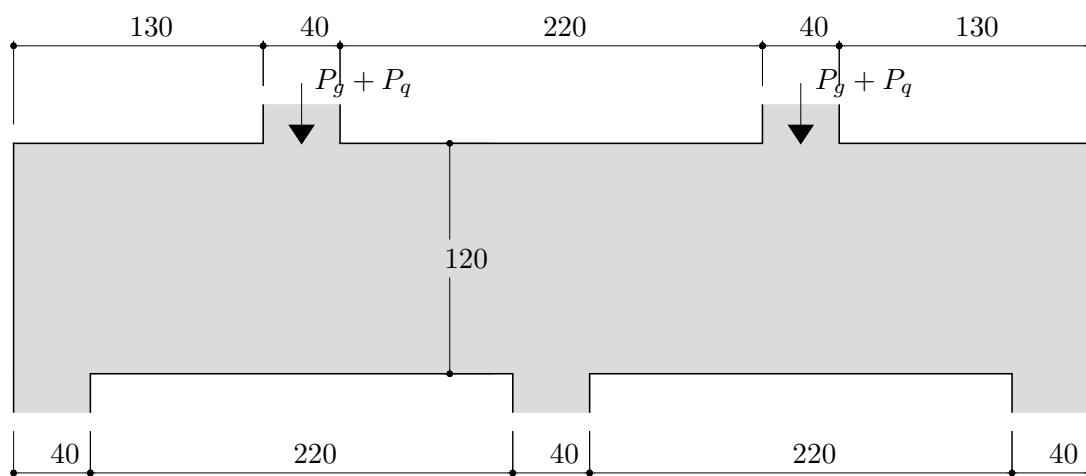
$$P_d = 1540 \text{ kN} \quad P_{d,CQP} = 895 \text{ kN} \quad P_{d,CF} = 910 \text{ kN} \quad (6.23)$$

O modelo biela-painel idealizado para a viga-parede (Figura 6.4b) é formado por 7 bielas e 2 painéis. Considerou-se simplificada a simetria da estrutura por meio da restrição dos deslocamentos horizontais na linha do pilar central. As bielas são representadas em cores diferentes, de acordo com suas seções transversais descritas na legenda posicionada abaixo do modelo.

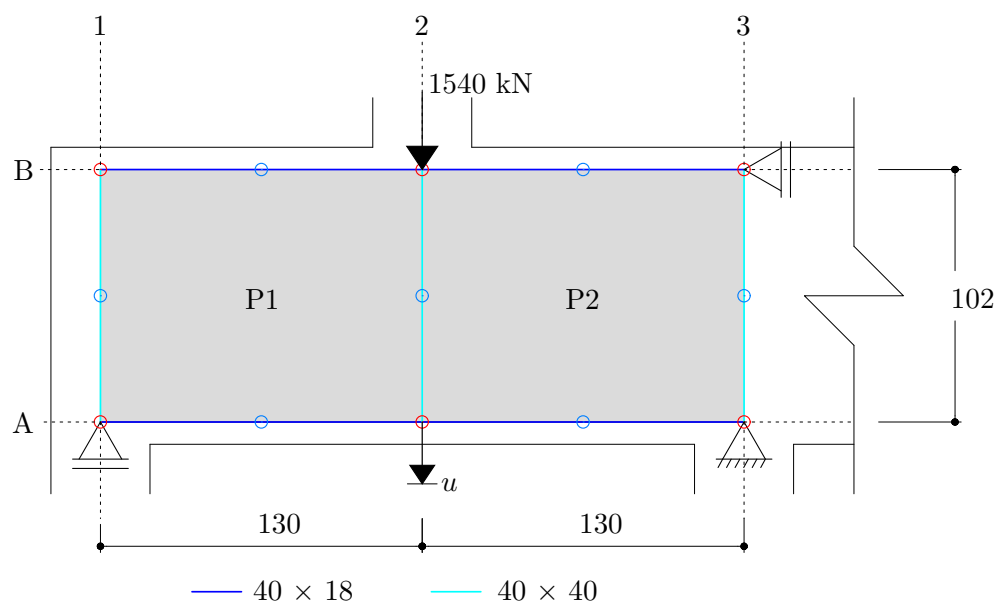
A nomenclatura adotada para as bielas é formada por três caracteres (XY-Z), onde X representa a linha de direção do elemento, e Y e Z, as linhas perpendiculares a X, onde a biela se inicia e termina, respectivamente. Por exemplo, B2-3 se trata da biela horizontal superior (linha B), que se inicia na linha 2 e termina na linha 3; 2A-B é o elemento vertical central (linha 2), com início na linha A e término na linha B.

A altura das bielas horizontais foi estimada como 15% da altura total da viga, seguindo a recomendação da ABNT NBR 6118:2014 para a distribuição das armaduras principais nesse tipo de estrutura. Já as bielas verticais seguem as dimensões dos pilares de carga e de apoio. A partir de uma análise linear em ELU no *SPMTool*, foram determinadas as forças nas bielas e as tensões de cisalhamento nos painéis (Figura 6.4c), sendo que os diagramas em azul representam forças de compressão e, em vermelho, tração.

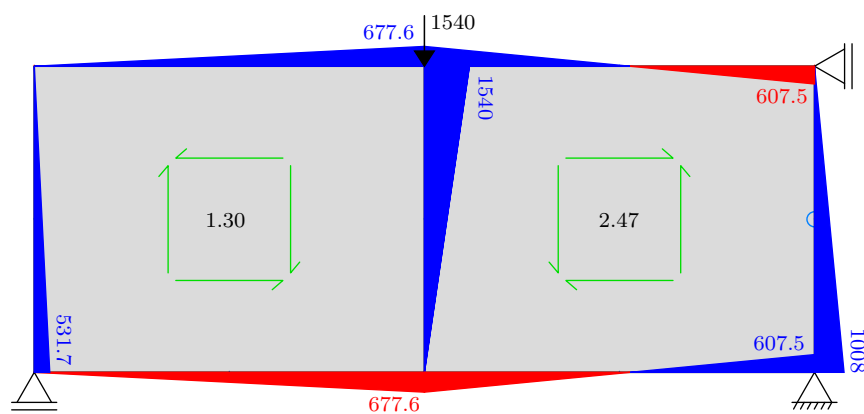
Figura 6.4 – Viga-parede contínua



(a) Geometria



(b) Modelo biela-painel



(c) Esforços nos elementos

Nota: dimensões em cm, forças nas bielas em kN e tensões nos painéis em MPa.

Considerando as equações apresentadas na seção 6.3 e a resistência de cálculo das armaduras (Tabela 6.1), a Tabela 6.2 mostra o dimensionamento das armaduras das bielas, a partir dos esforços obtidos na Figura 6.4c. A tabela apresenta as posições e numerações das bielas e as armaduras calculada ($A_{s,calc}$), obtida a partir de (6.12), e adotada ($A_{s,adot}$), a partir dos diâmetros comerciais disponíveis. Tratando-se de armaduras de flexão, foram respeitadas as armaduras mínimas exigidas pela norma brasileira, sendo $A_{s,min}$ igual a 0,15% da área de concreto da seção transversal:

$$A_{s,min} = 0,15\% b_w h = 7,2 \text{ cm}^2 \quad (6.24)$$

Tabela 6.2 – Viga-parede contínua: dimensionamento das armaduras das bielas

Posição	Biela	F_d (kN)	$A_{s,calc}$ (cm ²)	$A_{s,adot}$ (cm ²)
Inferior	A1-3	677,60	15,58	17,18 (14 ϕ 12,5 mm)
Superior (central)	B2-3	607,50	13,97	14,73 (12 ϕ 12,5 mm)

O dimensionamento dos painéis se inicia pelo painel crítico (P2), utilizando o processo iterativo definido na seção 6.4. Inicialmente, faz-se uma estimativa da deformação ε_x neste painel, considerando a seção de máximo momento fletor (máxima tração no tirante inferior). Utilizando (6.8), com unidades em kN e cm, obtém-se:

$$\varepsilon_x = \frac{1}{2 \cdot 21000 \cdot 15,58} (677,6 + 1008) = 2,34(10^{-3}) \quad (6.25)$$

Tendo calculado o valor de ε_x , procedem-se às iterações para encontrar a taxa de armadura. A Tabela 6.3 apresenta o resumo das iterações realizadas, sendo que, ao final, obtém-se uma taxa 11% menor que a definida inicialmente de acordo com o MPT. Esta taxa deve ser utilizada para as duas direções de armadura neste painel.

Tabela 6.3 – Viga-parede contínua: processo iterativo de dimensionamento do painel P2

i	$\rho_{s_{i-1}}$	s_{xe} (mm)	β	τ_{ci} (MPa)	θ_{c1}	ρ_{s_i}	Erro
1	0,568%	293,84	0,09	0,35	44,76°	0,492%	-
2	0,492%	336,10	0,09	0,34	43,99°	0,508%	0,016%
3	0,508%	326,19	0,09	0,34	44,17°	0,504%	0,004%
4	0,504%	328,47	0,09	0,34	44,13°	0,505%	0,001%

A Tabela 6.4 mostra resumidamente as taxas de armadura calculadas ($\rho_{s,calc}$) e adotadas ($\rho_{s,adot}$) para os painéis do modelo. A armadura calculada na Tabela 6.3 foi

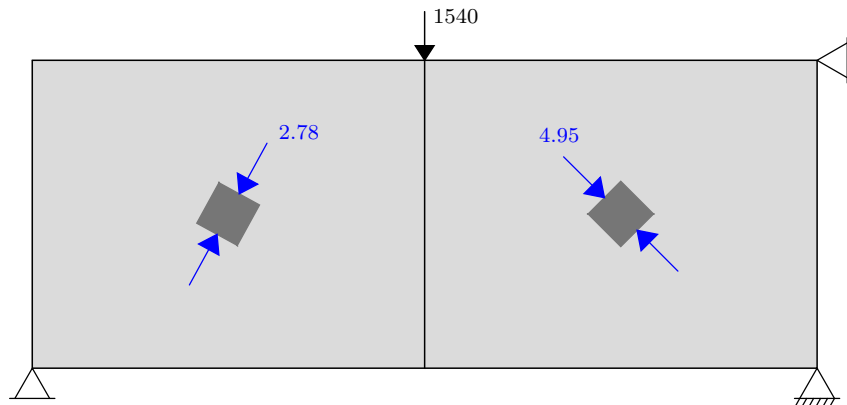
adotada como constante na direção x em todo o modelo; com ela são obtidas as taxas na direção y aplicando as relações dispostas em (6.17). Em todos os casos, foram obedecidas as taxas mínimas recomendadas pela norma brasileira para vigas-parede, sendo a horizontal determinada pela armadura de pele (0,20%) e a transversal, por 0,15%.

Tabela 6.4 – Viga-parede contínua: dimensionamento das armaduras dos painéis

Dir.	Painel	ρ_{sx}	τ_{dxy} (MPa)	θ_{c1}	$\rho_{s,calc}$	$\rho_{s,adot}$
x	-	-	2,47	44,14°	0,50%	0,51% ($\phi 12,5$ mm c/ 12 cm)
y	P2	0,51%	2,47	44,21°	0,50%	0,56% ($\phi 10$ mm c/ 7 cm)
y	P1	0,51%	1,30	44,21°	0,23%	0,25% ($\phi 10$ mm c/ 16 cm)

As armaduras calculadas são então inseridas *SPMTool*, em cada elemento. Uma segunda análise linear é efetuada para verificar as tensões no concreto, já que, de acordo com a Equação 6.18, as tensões nos painéis dependerão das quantidades de armadura dispostas em cada direção. A Figura 6.5 mostra as tensões atuantes no concreto, para os painéis.

Figura 6.5 – Viga-parede contínua: tensões no concreto



Nota: tensões em MPa.

Finalmente, efetuam-se as verificações das tensões no concreto, que devem ser limitadas aos valores reduzidos de resistência (f_{cd1} , f_{cd2} e f_{cd3}). Para a bielas horizontais, o ponto crítico é o nó de aplicação da carga e, já que só existe compressão neste local, a tensão deve ser limitada a f_{cd1} . A biela vertical mais solicitada também se encontra no mesmo ponto e, como ela é atravessada por uma malha de armaduras, deve ter como limitação o valor de f_{cd2} . Já para os painéis, a tensão crítica é encontrada no painel P2, devendo também ser limitada a f_{cd2} pelo mesmo motivo já descrito.

A Tabela 6.5 apresenta resumidamente as verificações do concreto para os elementos do modelo, sendo que, para as bielas, a tensão $\sigma_{cd,max}$ é obtida pela relação entre a força

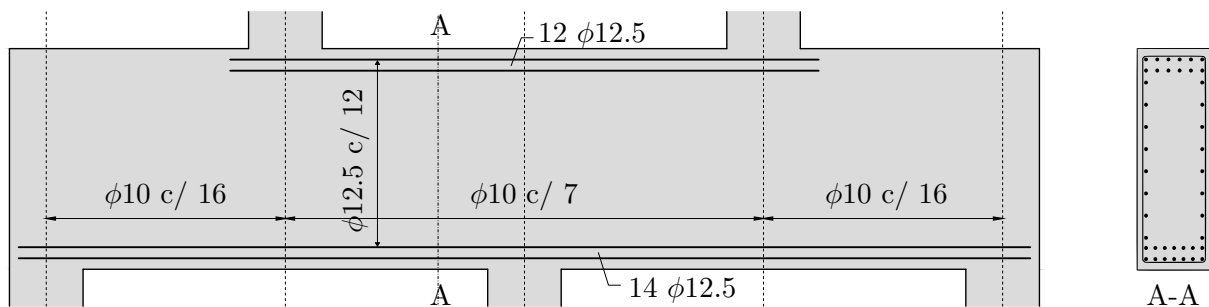
atuante e a área da seção transversal e, para os painéis, utilizam-se as tensões ilustradas na Figura 6.5.

Tabela 6.5 – Viga-parede contínua: verificação das tensões no concreto

Elemento	Crítico	$\sigma_{cd,max}$ (MPa)	σ_{lim} (MPa)	Verificação
Bielas horizontais (40×18)	B1-2	9,41	16,03 (f_{cd1})	OK
Bielas verticais (40×40)	2A-B	9,63	11,31 (f_{cd2})	OK
Painéis	P2	4,95	11,31 (f_{cd2})	OK

Efetuada as verificações, faz-se o detalhamento da viga-parede (Figura 6.6), onde as armaduras distribuídas são representadas simplificadaamente por cotas perpendiculares à direção das barras. Estas devem estar presentes nas duas faces laterais da viga-parede e serem, obrigatoriamente, compostas por estribos.

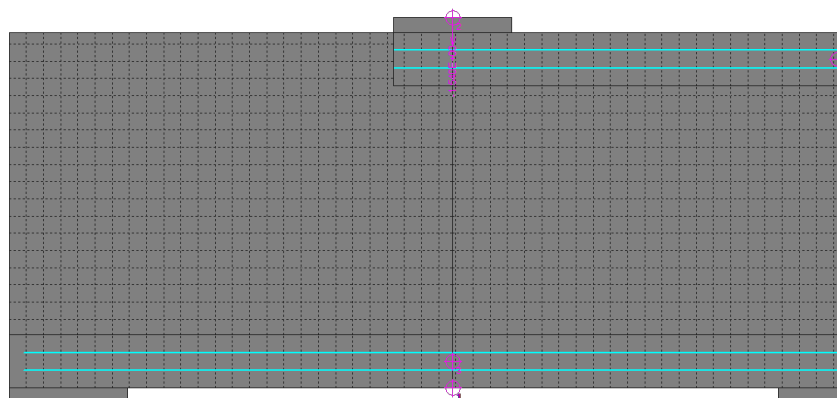
Figura 6.6 – Viga-parede contínua: detalhamento das armaduras



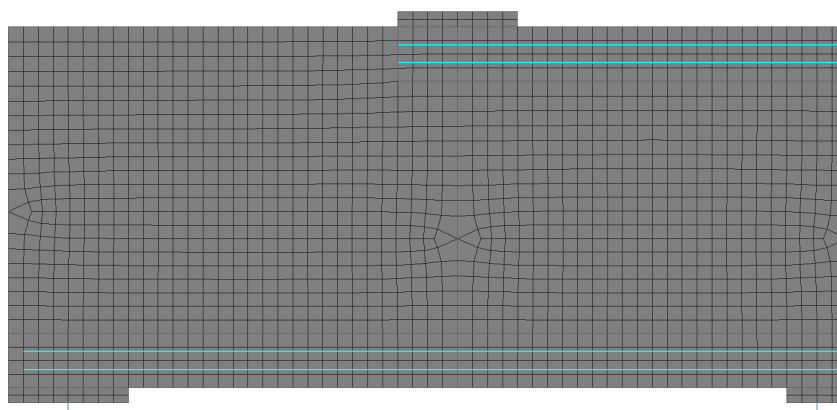
Nota: diâmetros das barras em mm e espaçamentos em cm.

A partir do detalhamento, foram desenvolvidos dois modelos no *ATENA 2D*. A Figura 6.7a mostra a geometria utilizada, onde as armaduras das bielas foram definidas como localizadas e, dos painéis, do tipo distribuído (*smeared reinforcement*). De modo a evitar a sobreposição das barras, foi necessário definir macro-regiões distintas nas regiões dos tirantes para se considerar apenas a armadura distribuída na direção vertical. Nos apoios e nos pontos de aplicação de carga, foram utilizadas placas de aço para distribuir uniformemente os esforços para o concreto.

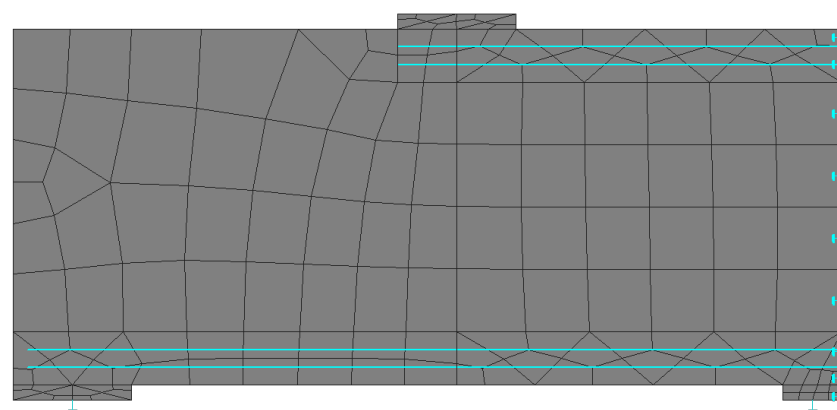
O modelo AT-A (Figura 6.7b) apresenta maior discretização, sendo o tamanho máximo de 5 cm para os elementos finitos. Já o modelo AT-B (Figura 6.7c), de menor discretização, foi formado por elementos com dimensões de até 20 cm. Ambos os modelos são formados por elementos quadrangulares e, para considerar a simetria da viga, possuem apoio na direção horizontal, na linha central da estrutura.

Figura 6.7 – Viga-parede contínua: modelos desenvolvidos no *ATENA 2D*

(a) Geometria



(b) AT-A

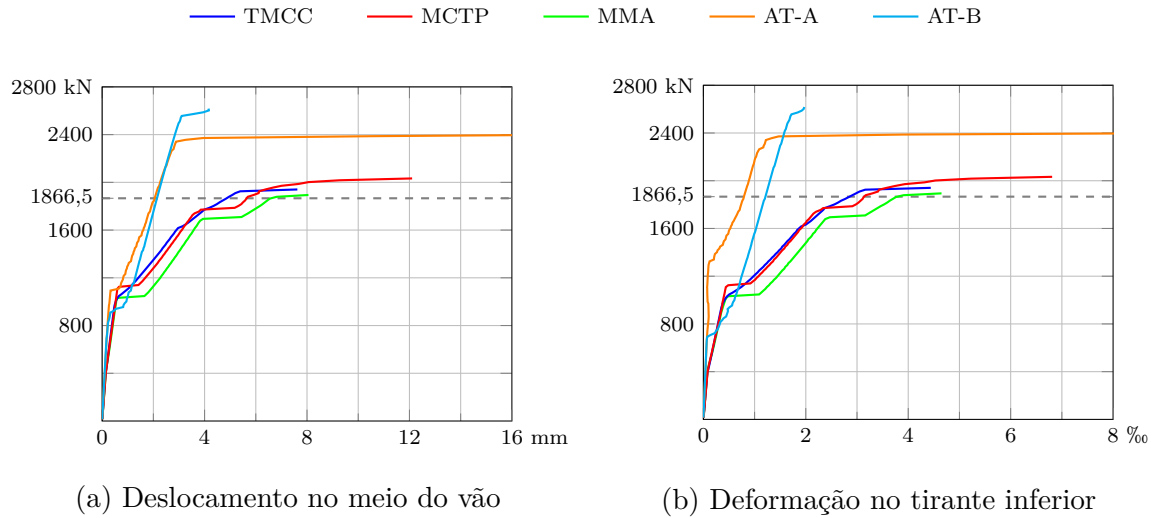


(c) AT-B

As análises não-lineares foram então conduzidas no *SPMTool* e no *ATENA 2D*. Em ambos, utilizaram-se os parâmetros padrão de solução não-linear, pelo procedimento de Newton-Raphson. A Figura 6.8 apresenta os diagramas de comportamento para todas as análises efetuadas, sendo comparados os deslocamentos e a deformação no tirante inferior, no ponto médio do vão. O valor de 1866,5 kN, representado pela linha tracejada, simboliza o carregamento limite da viga-parede, calculado pelo modelo biela-painel analítico (Figura 6.4b). Tal limite foi obtido considerando as resistências características dos materiais

e se trata da carga que causaria o escoamento do tirante inferior. Logo, é possível observar que todos os modelos superaram o carregamento de ruptura estimado, sendo que os modelos do *ATENA* apresentaram valores maiores que os do *SPMTool*.

Figura 6.8 – Viga-parede contínua: diagramas de comportamento



As curvas do *SPMTool*, conforme já observado em 5.3, possuem uma certa proximidade. Comparando com os diagramas do *ATENA*, conclui-se que os modelos do primeiro programa tendem a se deformar mais, especialmente após a fissuração. Esse comportamento já era esperado, visto que, pelo princípio do MBP, as bielas que carregam a maior parte das tensões normais, o que faz com que os níveis de tensões e, consequentemente, deformações nessas regiões se tornem mais elevados.

A Tabela 6.6 apresenta as comparações entre os modelos do *SPMTool* e do *ATENA*. São analisados os deslocamentos e as deformações para o carregamento de serviço $P_{d,CQP}$, e as aberturas máximas de fissuras, para $P_{d,CF}$. Esses mesmos quesitos são analisados para o carregamento último (P_d). Além disso, são comparados os níveis de carga que causam a fissuração de flexão e diagonal, a carga última e o tipo de ruptura. Para cada modelo (MMA, TMCC e MCTP), são apresentadas as relações entre os valores obtidos por eles e os modelos AT-A e AT-B, simbolizados apenas por A e B. As colunas finais mostram os valores numéricos encontrados por AT-A e AT-B.

Analisando os deslocamentos e as deformações comparadas, pode-se observar que os modelos do *SPMTool* se aproximam melhor dos resultados de AT-B, já que esse também apresenta maiores deformações que AT-A. Por outro lado, este último consegue captar o comportamento pós-escoamento do tirante principal, que acontece após o esmagamento do concreto no ponto de aplicação da carga. O modelo AT-B atinge a ruptura do concreto dessa mesma posição e não consegue convergência após isso.

Tabela 6.6 – Viga-parede contínua: comparação entre os modelos não-lineares

Valor comparado		<i>SPMTool</i>			<i>ATENA</i>	
		MMA	TMCC	MCTP	AT-A	AT-B
ELS	Deslocamento no meio do vão (mm)	0,473	0,435	0,458	0,239	0,310
	Deformação no tirante inferior (‰)	0,347	0,330	0,339	0,162	0,455
	Deformação no tirante superior (‰)	0,588	0,545	0,552	0,098	0,420
	Máxima fissura de flexão (mm)	0,036	0,034	0,035	0,025	0,099
	Máxima fissura diagonal (mm)	0,000	0,000	0,000	0,000	0,129
ELU	Deslocamento no meio do vão (mm)	3,385	2,684	2,906	1,448	1,735
	Deformação no tirante inferior (‰)	2,116	1,736	1,787	0,890	1,068
	Deformação no tirante superior (‰)	0,538	0,669	0,569	0,740	1,116
	Máxima fissura de flexão (mm)	0,212	0,174	0,179	0,108	0,344
	Máxima fissura diagonal (mm)	0,597	0,456	0,503	0,244	0,792
Carga de fissuração de flexão (kN)		400,4	400,4	385,0	600,6	646,8
Carga de fissuração diagonal (kN)		893,2	939,4	954,8	1062,6	954,8
Carga última (kN)		1894,2	1940,4	2032,8	2417,8	2571,8
Tipo de ruptura		A-C	A-C	A-C	B-A	B

Notas: A: escoamento do tirante inferior;

B: esmagamento do concreto na região de aplicação da carga;

C: esmagamento do concreto na diagonal (P2).

Deslocamentos e aberturas de fissuras em mm e cargas em kN.

Comparando as aberturas de fissuras em ELS, obteve-se uma maior aproximação com AT-A, tendo em vista os valores de fissuração de flexão e da inexistência de fissuras diagonais, observadas tanto no *SPMTool* quanto em AT-A. Para o carregamento em ELU, as aberturas encontradas se aproximam da média entre AT-A e AT-B. Em todos os casos, as aberturas foram menores que as obtidas em AT-B, o que mostra que o *SPMTool* apresentou dimensões mais condizentes com o modelo de maior discretização. Por possuir elementos maiores, o AT-B acaba apresentando maiores aberturas, com início da fissuração diagonal antes de AT-A.

Em relação aos carregamentos de fissuração, foram obtidas melhores aproximações para o caso diagonal. Nos modelos do *SPMTool*, as fissuras de flexão aparecem muito antes que do *ATENA*, o que não acontece para a outra comparação. Quanto às cargas de ruptura, as análises do primeiro programa chegaram a aproximadamente 80% das efetuadas no segundo.

Os tipos de ruptura também se diferem, enquanto no *SPMTool* ocorre o escoamento do tirante principal seguido por esmagamento do concreto na diagonal que leva a carga

para o pilar central (painel P2), no *ATENA* acontece o esmagamento do concreto no ponto de aplicação da força. No modelo AT-A, também se sucede o escoamento das armaduras principais.

6.5.3 Viga-parede com aberturas

A geometria da viga-parede com aberturas considerada nesta seção está ilustrada na Figura 6.9a. O elemento possui seção transversal de 40 cm de largura e duas aberturas de 60×60 cm, localizadas entre o ponto de aplicação de carga e os apoios. A viga está sujeita aos carregamentos permanente ($P_g = 382,5$ kN) e variável de utilização ($P_q = 67,5$ kN), oriundos dos três pavimentos superiores de um edifício. As combinações de ações são dadas por:

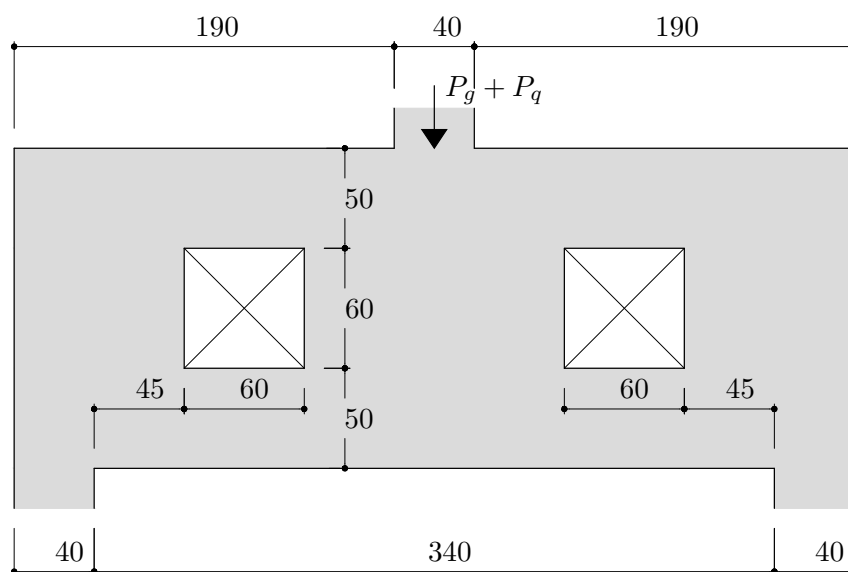
$$P_d = 693 \text{ kN} \quad P_{d,CQP} = 402,75 \text{ kN} \quad P_{d,CF} = 409,5 \text{ kN} \quad (6.26)$$

O modelo biela-painel proposto (Figura 6.9b) possui 45 bielas e 16 painéis. Uma maior discretização se fez necessária neste modelo devido à existência das aberturas, que fazem com que sejam necessárias linhas de bielas adicionais para contorná-las. A nomenclatura das bielas segue o padrão descrito na seção anterior e as geometrias são descritas na legenda posicionada sob o modelo.

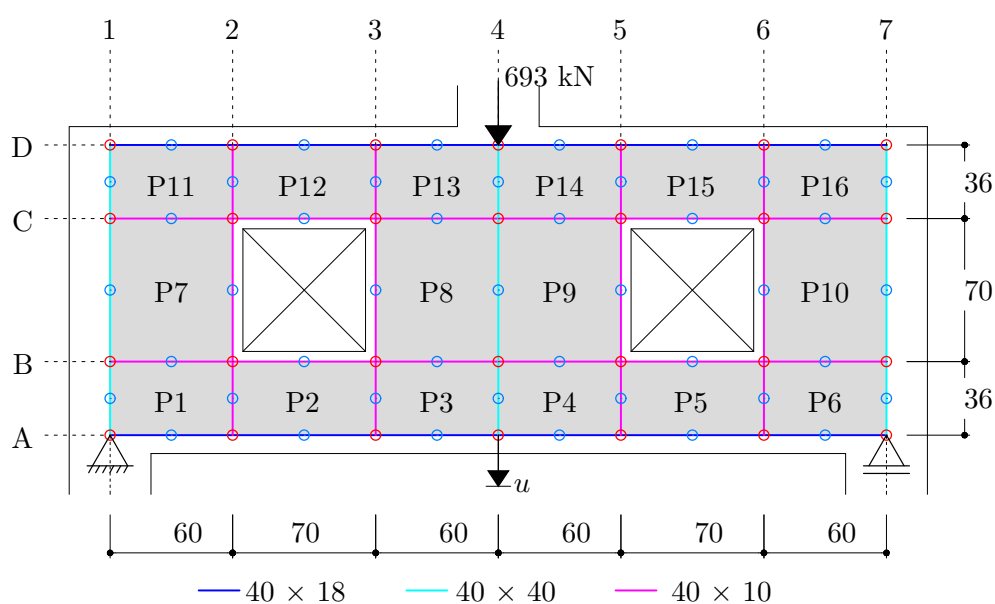
Quanto às alturas das bielas, foi seguida a mesma recomendação da norma brasileira, utilizada para a viga-parede contínua, para as bielas inferior e superior (15% da altura total da seção transversal). As bielas de carga e dos apoios seguem as dimensões dos pilares ligados a elas e, para as bielas internas, utilizou-se a altura de 10 cm, para possibilitar a utilização de 2 ou 3 camadas de armadura.

Fazendo uma análise elástica para a carga última (P_d) no *SPMTool*, obtiveram-se os esforços ilustrados na Figura 6.9c, que foram utilizados para o dimensionamento das armaduras do modelo. Tratando-se de uma viga-parede, as armaduras mínimas são calculadas da mesma forma já exposta em 6.5.2, sendo a área longitudinal de no mínimo $9,6 \text{ cm}^2$. Além da armadura principal inferior, faz-se necessária a disposição de barras localizadas no entorno das aberturas, devido à existência de esforços de tração nas bielas nessas posições. Para as bielas verticais, as armaduras resultantes devem ser compostas por estribos. A Tabela 6.7 apresenta o dimensionamento das bielas do modelo.

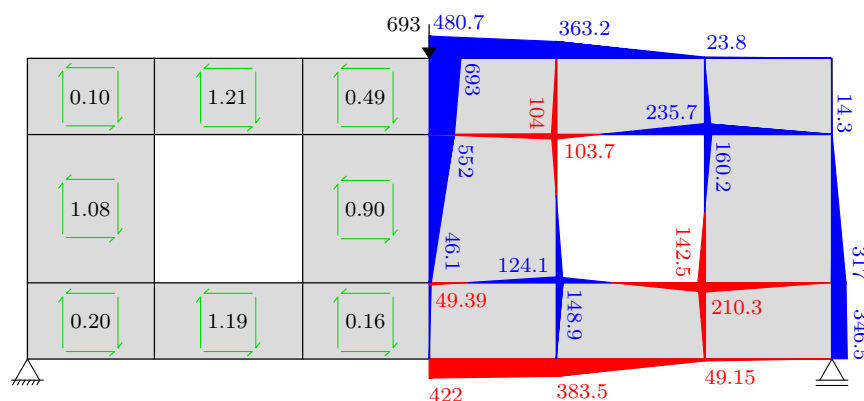
Figura 6.9 – Viga-parede com aberturas



(a) Geometria



(b) Modelo biela-painel



(c) Esforços nos elementos

Nota: dimensões em cm, forças nas bielas em kN e tensões nos painéis em MPa.

Tabela 6.7 – Viga-parede com aberturas: dimensionamento das armaduras das bielas

Posição	Biela	F_d (kN)	$A_{s,calc}$ (cm ²)	$A_{s,adot}$ (cm ²)
Inferior	A1-7	422,00	9,71	9,82 (8 ϕ 12,5 mm)
Abaixo da abertura	B1-7	210,30	4,84	4,91 (4 ϕ 12,5 mm)
Acima da abertura	C2-6	103,70	2,39	2,45 (2 ϕ 12,5 mm)
Lateral da abertura	2A-D/6A-D	142,50	3,28	4,71 (3 estribos ϕ 10 mm)
Lateral da abertura	3A-D/5A-D	104,00	2,39	3,14 (2 estribos ϕ 10 mm)

Quanto aos painéis, utilizou-se a máxima tensão cisalhante (1,21 MPa), existente no painel P2, para o cálculo das armaduras distribuídas. A Tabela 6.8 mostra as iterações realizadas para a obtenção da taxa, sendo que, ao final, conseguiu-se uma redução de 14% em relação à estimativa inicial pelo MPT. A deformação ε_x foi estimada por:

$$\varepsilon_x = \frac{1}{2 \cdot 21000 \cdot 9,82} \left(\frac{346,5 \cdot 190}{142} + 346,5 \right) = 1,96(10^{-3}) \quad (6.27)$$

Tabela 6.8 – Viga-parede com aberturas: processo iterativo de dimensionamento do painel P2

i	$\rho_{s_{i-1}}$	s_{xe} (mm)	β	τ_{ci} (MPa)	θ_{c_1}	p_{s_i}	Erro
1	0,278%	577,95	0,08	0,33	42,49°	0,222%	-
2	0,222%	719,90	0,08	0,30	40,07°	0,249%	0,027%
3	0,249%	643,64	0,08	0,31	41,37°	0,234%	0,015%
4	0,234%	683,09	0,08	0,31	40,70°	0,242%	0,008%
5	0,242%	662,28	0,08	0,31	41,05°	0,238%	0,004%
6	0,238%	673,15	0,08	0,31	40,87°	0,240%	0,002%
7	0,240%	667,44	0,08	0,31	40,96°	0,239%	0,001%

Adotando o resultado de 0,24% como constante por toda a altura da viga, é possível calcular os estribos verticais para os demais painéis. A Tabela 6.9 resume o dimensionamento das armaduras dos painéis, sendo possível observar que as regiões localizadas abaixo e acima das aberturas terão uma maior concentração de estribos transversais.

Com os resultados de dimensionamento, executa-se novamente uma análise linear no *SPMTool*, para se obter as tensões no concreto dos painéis (Figura 6.10). Observa-se que os elementos mais carregados são aqueles que estão nas adjacências da abertura.

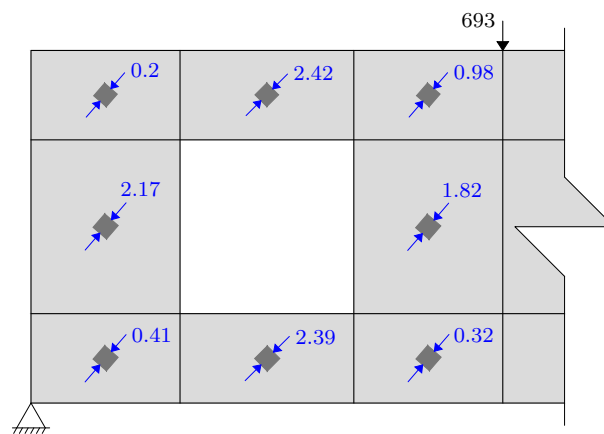
Considerando os elementos mais solicitados, é realizada a verificação das tensões no concreto, resumida na Tabela 6.10. As bielas superior e vertical central tem carga crítica no ponto de aplicação da força (trechos D4-5 e 4C-D), enquanto a biela interna mais

solicitada se localiza acima da abertura (C5-6). Conforme observado na Figura 6.10, a máxima tensão de compressão existe no painel P12.

Tabela 6.9 – Viga-parede com aberturas: dimensionamento das armaduras dos painéis

Dir.	Painel	ρ_{sx}	τ_{dxy} (MPa)	θ_{c1}	$\rho_{s,calc}$	$\rho_{s,adot}$
x	-	-	1,21	40,91°	0,24%	0,25% ($\phi 10$ mm c/ 16 cm)
y	P2/12	0,25%	1,21	40,91°	0,24%	0,25% ($\phi 10$ mm c/ 16 cm)
y	P1/3/7/8/11/13	0,25%	1,08	41,22°	0,20%	0,20% ($\phi 10$ mm c/ 20 cm)

Figura 6.10 – Viga-parede com aberturas: tensões no concreto



Nota: tensões em MPa.

Tabela 6.10 – Viga-parede com aberturas: verificação das tensões no concreto

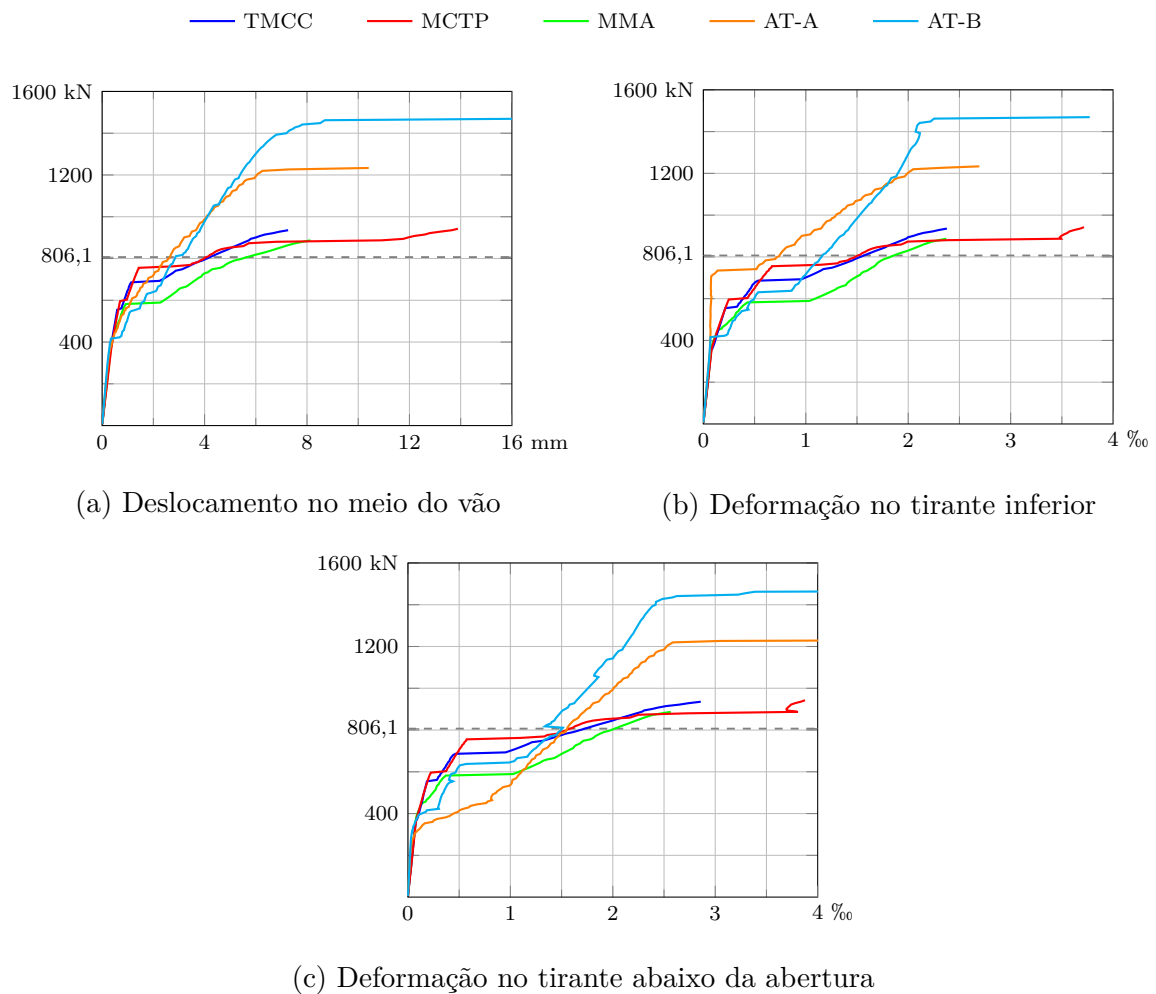
Elemento	Crítico	$\sigma_{cd,max}$ (MPa)	σ_{lim} (MPa)	Verificação
Bielas superiores (40×18)	D4-5	6,68	16,03 (f_{cd1})	OK
Bielas internas (40×10)	C5-6	5,89	11,31 (f_{cd2})	OK
Bielas de carga/apoio (40×40)	4C-D	4,33	11,31 (f_{cd2})	OK
Painéis	P12	2,42	11,31 (f_{cd2})	OK

Com as verificações satisfeitas, apresenta-se o detalhamento das armaduras na Figura 6.11. Observa-se que o elemento necessita de reforço adicional no entorno das aberturas, sendo este formado por barras longitudinais horizontais e por estribos transversais adicionais. Além disso, também foi necessária a existência de uma malha mais concentrada nas regiões localizadas acima e abaixo das aberturas.

Utilizando o mesmo conceito já apresentado na seção anterior, a geometria dos modelos do *ATENA 2D* (Figura 6.12a) foi formada por vários macro-elementos, para evitar a sobreposição das armaduras distribuídas com as localizadas. Para considerar a simetria, a linha central foi fixada na direção horizontal, mantendo livre o deslocamento vertical e aplicando apenas metade do carregamento total. Com isso, dois modelos foram desenvolvidos: AT-A (Figura 6.12b), com elementos de dimensões de até 5 cm; e AT-B (Figura 6.12c), com elementos de até 20 cm.

A Figura 6.13 apresenta os diagramas de comportamento obtidos a partir das análises efetuadas, sendo que a linha tracejada representa o valor de 806,1 kN, calculada pelo modelo analítico como a força que causaria o escoamento do tirante inferior. Observa-se que todas as análises superaram esse nível de carregamento e, assim como para o exemplo anterior, os modelos do *ATENA* resultaram em maiores capacidades de carga.

Figura 6.13 – Viga-parede com aberturas: diagramas de comportamento



São comparados os deslocamentos no ponto médio do vão (Figura 6.13a) e as deformações no tirante principal (Figura 6.13b) e no tirante abaixo da abertura (Figura 6.13c), no ponto à esquerda dessa área. Assim como já observado no exemplo anterior, os modelos

do *SPMTool* apresentam maiores deformações para os mesmos níveis de carregamento do *ATENA*, especialmente após a fissuração. Ao contrário do exemplo anterior, as curvas do primeiro programa têm um maior distanciamento entre si, devido a influência da abertura no campo de tensões da estrutura.

A Tabela 6.11 apresenta as comparações dos resultados dos modelos não-lineares. Para os níveis de serviço, os deslocamentos encontrados no *SPMTool* se assemelham com aqueles do modelo AT-A. As deformações e aberturas de fissuras, por outro lado, tiveram um maior distanciamento em relação ao *ATENA*, havendo também maior aproximação à AT-A, especialmente quanto à fissuração de flexão.

Tabela 6.11 – Viga-parede com aberturas: comparação entre os modelos não-lineares

Valor comparado		<i>SPMTool</i>			<i>ATENA</i>	
		MMA	TMCC	MCTP	AT-A	AT-B
ELS	Deslocamento no meio do vão (mm)	0,450	0,433	0,460	0,494	0,777
	Deformação no tirante inferior (‰)	0,137	0,142	0,134	0,540	0,638
	Deformação no tirante superior (‰)	0,132	0,129	0,123	0,553	0,726
	Máxima fissura de flexão (mm)	0,023	0,024	0,022	0,077	0,431
	Máxima fissura diagonal (mm)	-	-	-	0,099	0,160
ELU	Deslocamento no meio do vão (mm)	3,599	2,243	1,223	1,766	2,356
	Deformação no tirante inferior (‰)	1,453	0,959	0,564	0,831	1,165
	Deformação no tirante superior (‰)	1,528	0,954	0,484	1,276	1,367
	Máxima fissura de flexão (mm)	0,234	0,155	0,091	0,131	0,770
	Máxima fissura diagonal (mm)	1,060	0,549	0,231	0,190	0,349
Carga de fissuração de flexão (kN)		360,4	360,4	346,5	304,9	284,1
Carga de fissuração na abertura (kN)		374,2	367,3	360,4	180,2	201,0
Carga de fissuração diagonal (kN)		457,4	561,3	561,3	429,7	422,7
Carga última (kN)		887,0	935,6	942,5	1233,5	1469,2
Tipo de ruptura		C-B-D-F	B-E-C-G	B-A-C-D	D-B-A	B-D-A

Notas: A: escoamento do tirante inferior;

B: escoamento do tirante abaixo da abertura;

C: escoamento do tirante acima da abertura;

D: escoamento das armaduras distribuídas abaixo da abertura;

E: escoamento das armaduras distribuídas acima da abertura;

F: esmagamento do concreto na diagonal abaixo da abertura (P2);

G: esmagamento do concreto na diagonal acima da abertura (P12).

Deslocamentos e aberturas de fissuras em mm e cargas em kN.

Em ELU, as deformações obtidas no *SPMTool* se aproximaram mais do *ATENA*, porém, conforme já observado nos diagramas, os três modelos implementados apresen-

taram maiores divergências. Nessa situação, a análise pelo TMCC gerou resultados mais condizentes com os modelos do *ATENA*.

Comparando os carregamentos de fissuração, ao contrário da viga anterior, os modelos do *SPMTool* fissuram após os do *ATENA*. Por outro lado, a ruptura no primeiro programa ocorre antes. Os tipos de ruptura também variam entre as análises, normalmente acontecendo o escoamento das armaduras localizadas no entorno da abertura antes do tirante inferior.

6.5.4 Consolo curto

Considera-se um consolo curto de uma estrutura de concreto pré-moldada, de geometria ilustrada na Figura 6.14a, submetido às cargas verticais permanente ($P_g = 150$ kN) e variável ($P_q = 20$ kN). As forças de cálculo são dadas por:

$$P_d = 261,8 \text{ kN} \quad P_{d,CQP} = 156 \text{ kN} \quad P_{d,CF} = 158 \text{ kN} \quad (6.28a)$$

Segundo a ABNT NBR 9062:2017, para o dimensionamento deve ser considerada adicionalmente uma força horizontal H_d , quando não haver impedimento da movimentação horizontal. Considerando que o aparelho de apoio seja constituído por elastômero, esta força é estimada em 16% da força vertical. Portanto:

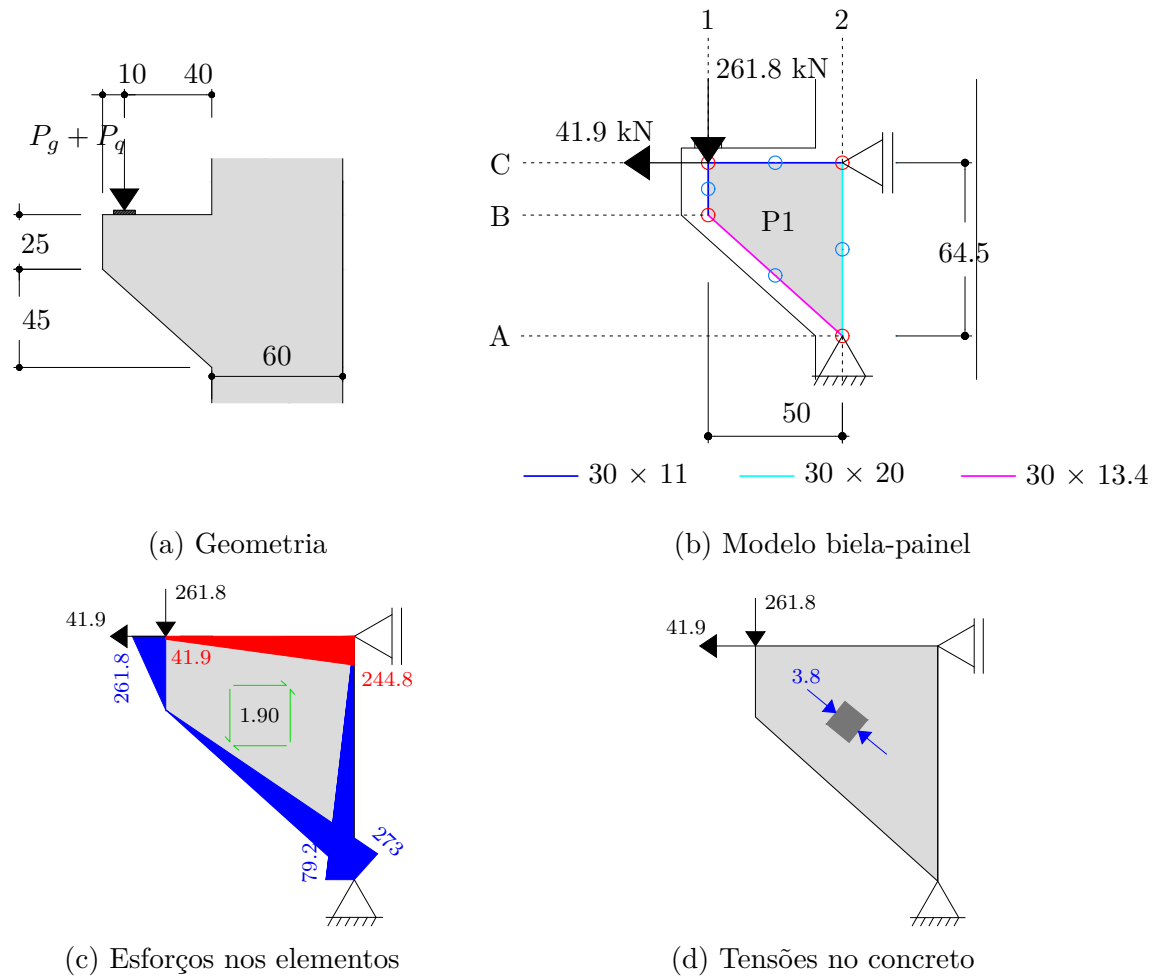
$$H_d = 41,9 \text{ kN} \quad H_{d,CQP} = 24,96 \text{ kN} \quad H_{d,CF} = 25,28 \text{ kN} \quad (6.28b)$$

Propõe-se um modelo biela-painel (Figura 6.14b) composto por 4 bielas e 1 painéis para o consolo, considerado perfeitamente engastado no pilar de apoio. Para o tirante superior, foi considerada a altura de 15% da altura total do consolo, medida na interface engastada no pilar. Para as bielas verticais no pilar, considerou-se seção de 30×20 cm e, para a biela diagonal, a altura foi obtida a partir da geometria do modelo.

Após uma análise linear no *SPMTool*, obtêm-se os esforços ilustrados na Figura 6.14c, que são utilizados para o dimensionamento das armaduras do modelo. As armaduras mínimas são determinadas de acordo com a ABNT NBR 6118:2014 e NBR 9062:2017, por: 0,15% da área da seção transversal engastada no pilar ($3,15 \text{ cm}^2$), para a armadura principal; 40% da área de aço do tirante, para a armadura de costura (distribuída horizontalmente); e com taxa de 0,15% para os estribos verticais. A Tabela 6.12 mostra o resumo de dimensionamento das bielas.

As armaduras distribuídas foram dimensionadas a partir da tensão cisalhante no painel (1,9 MPa). Iterativamente, calculou-se a taxa de 0,36%, conforme a Tabela 6.13. Pode-se observar que, nesse caso, foi possível uma redução de 17% da estimativa inicial de 0,43%. As taxas de armadura adotadas em cada direção do consolo são resumidas na Tabela 6.14.

Figura 6.14 – Consolo curto



Nota: dimensões em cm, forças nas bielas em kN e tensões nos painéis em MPa.

Tabela 6.12 – Consolo curto: dimensionamento das armaduras das bielas

Posição	Biela	F_d (kN)	$A_{s,calc}$ (cm ²)	$A_{s,adot}$ (cm ²)
Superior	C1-2	244,80	5,63	6,03 (3 ϕ 16 mm)

Considerando as armaduras apresentadas, foi executada novamente uma análise elástica no *SPMTool*, para a determinação das tensões no concreto dos painéis (Figura 6.14d). Com isso, as tensões são verificadas na Tabela 6.15, sendo estas limitadas ao valor de f_{cd2} pelo fato de todos os elementos serem atravessados por barras de aço tracionadas.

Finalmente, a Figura 6.15 apresenta o detalhamento das armaduras calculadas para o consolo considerado. É importante destacar que, tanto as armaduras distribuídas na vertical, quanto na horizontal, devem ser constituídas por estribos, sendo que a primeira deve ser disposta entre o ponto de aplicação de carga e a face do pilar de apoio e, a segunda, deve ser estendida e ancorada nas barras do tirante principal (dobradas na

vertical). Embora não presentes no detalhamento, será também necessária a disposição de barras construtivas para a efetiva amarração desses elementos.

Tabela 6.13 – Consolo curto: processo iterativo de dimensionamento dos painéis

i	$\rho_{s_{i-1}}$	s_{xe} (mm)	β	τ_{ci} (MPa)	θ_{c_1}	p_{s_i}	Erro
1	0,435%	377,57	0,10	0,38	46,25°	0,332%	-
2	0,332%	487,34	0,09	0,35	44,38°	0,361%	0,029%
3	0,361%	449,98	0,09	0,36	45,02°	0,351%	0,010%
4	0,351%	462,19	0,09	0,36	44,81°	0,355%	0,003%
5	0,355%	458,15	0,09	0,36	44,88°	0,353%	0,001%

Tabela 6.14 – Consolo curto: dimensionamento das armaduras dos painéis

Dir.	Painel	ρ_{sx}	τ_{dxy} (MPa)	θ_{c_1}	$\rho_{s,calc}$	$\rho_{s,adot}$
x	-	-	1,89	44,42°	0,36%	0,37% ($\phi 10$ mm c/ 14 cm)
y	-	0,37%	1,89	44,67°	0,36%	0,37% ($\phi 10$ mm c/ 14 cm)

Tabela 6.15 – Consolo curto: verificação das tensões no concreto

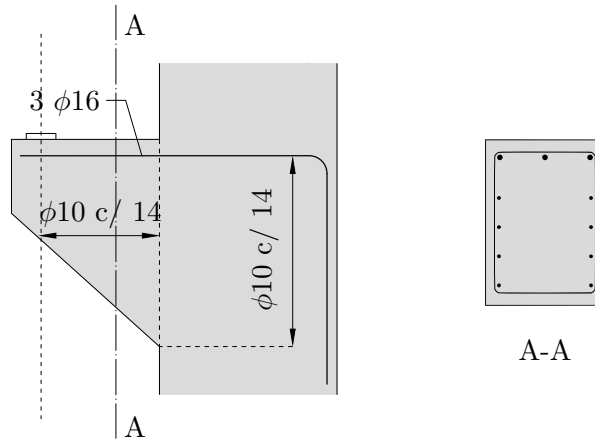
Elemento	Crítico	$\sigma_{cd,max}$ (MPa)	σ_{lim} (MPa)	Verificação
Biela de carga (30 × 10)	1B-C	8,73	11,31 (f_{cd_2})	OK
Biela vertical no apoio (30 × 20)	2A-C	7,90	11,31 (f_{cd_2})	OK
Biela diagonal (30 × 13,4)	B1-A2	6,79	11,31 (f_{cd_2})	OK
Painéis	P1	3,80	11,31 (f_{cd_2})	OK

A Figura 6.16 apresenta os modelos desenvolvidos no *ATENA 2D* a partir do detalhamento das armaduras. Foram definidos dois macro-elementos: um para representar o tirante superior e outro para o painel e as demais bielas; dessa forma, no primeiro se considerou apenas a armadura distribuída vertical e, no segundo, as duas direções calculadas. A condição de engastamento no pilar foi considerada por meio da restrição dos deslocamentos verticais e horizontais na linha de contato. Foram então geradas duas malhas para o consolo: AT-A (Figura 6.16b), com elementos de até 2,5 cm, e AT-B (Figura 6.16c), com elementos de até 10 cm.

As análises não-lineares foram então conduzidas no *SPMTool* e no *ATENA 2D*. A Figura 6.17 apresenta os diagramas de comportamento encontrados nas análises, onde são comparados os deslocamentos verticais no ponto de aplicação da carga e a deformação no tirante, analisada no ponto de engastamento entre o consolo e o pilar. A linha tracejada se

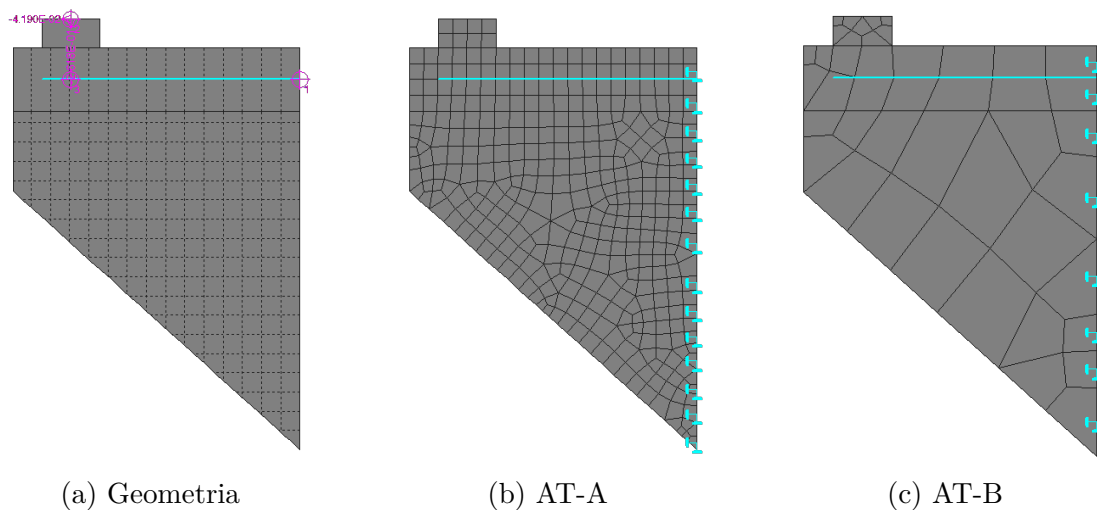
trata do nível de 322,5 kN, calculado analiticamente pelo modelo biela-painel (Figura 6.14b) como o carregamento para o qual ocorreria o escoamento do tirante superior.

Figura 6.15 – Consolo curto: detalhamento das armaduras



Nota: diâmetros das barras em mm e espaçamentos em cm.

Figura 6.16 – Consolo curto: modelos desenvolvidos no *ATENA 2D*



Para os dois diagramas, obtiveram-se curvas muito próximas entre os modelos do *SPMTool*. Comparando os deslocamentos, observa-se que as análises do *ATENA* resultaram em valores maiores após a fissuração diagonal, já que tal fato não ocorre nos modelos do primeiro programa. Já em relação às deformações, as divergências encontradas foram mais condizentes com os modelos anteriores, com as curvas do primeiro programa apresentando maiores deformações.

A Tabela 6.16 resume os resultados encontrados nas análises. Pode-se observar que, em serviço, os modelos do *SPMTool* trazem valores mais compatíveis com AT-B. Isso também é verdade para o ELU, exceto para os deslocamentos, que estão mais próximos de AT-A. Não foi possível comparar as aberturas de fissuras diagonais, pois elas não

acontecem nos primeiros, isto é, o tirante principal escoia antes da fissuração do painel e a análise não atinge a convergência.

Figura 6.17 – Consolo curto: diagramas de comportamento

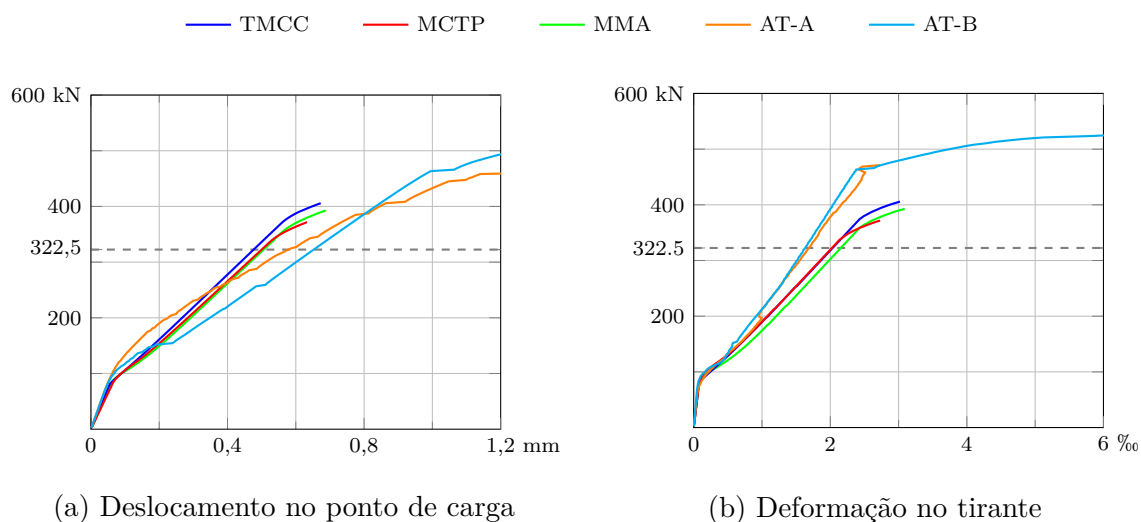


Tabela 6.16 – Consolo curto: comparação entre os modelos não-lineares

Valor comparado		<i>SPMTool</i>			<i>ATENA</i>	
		MMA	TMCC	MCTP	AT-A	AT-B
ELS	Deslocamento máximo (mm)	0,234	0,211	0,225	0,154	0,270
	Deformação no tirante principal (‰)	0,945	0,824	0,819	0,620	0,706
	Máxima fissura de flexão (mm)	0,149	0,130	0,130	0,062	0,132
	Máxima fissura diagonal (mm)	0,000	0,000	0,000	0,000	0,177
ELU	Deslocamento máximo (mm)	0,400	0,373	0,393	0,397	0,516
	Deformação no tirante principal (‰)	1,690	1,566	1,562	1,310	1,305
	Máxima fissura de flexão (mm)	0,261	0,241	0,241	0,136	0,229
	Máxima fissura diagonal (mm)	0,000	0,000	0,000	0,093	0,294
Carga de fissuração de flexão (kN)		81,2	81,2	81,2	52,4	65,5
Carga de fissuração diagonal (kN)		-	-	-	183,3	138,8
Carga de ruptura (kN)		392,7	405,8	371,8	466,0	539,3
Tipo de ruptura		A	A	A	B-A	A-B

Notas: A: escoamento do tirante principal;

B: escoamento da armadura distribuída na horizontal;

Deslocamentos e aberturas de fissuras em mm e cargas em kN.

Quanto à carga de fissuração de flexão, percebe-se que os modelos implementados são mais consistentes com os obtidos por AT-B. Porém, o mesmo não acontece com o carregamento de ruptura, já que esse modelo atinge um nível superior ao de AT-A. Quanto à forma que se dá a ruína, estabeleceu-se uma certa consistência, sendo o modo principal dado pelo escoamento do tirante. Para os modelos do *ATENA* também se observa o escoamento da armadura horizontal distribuída, acontecendo antes mesmo do tirante para o modelo AT-A.

6.5.5 Resumo das análises não-lineares

Em geral, a comparação entre os modelos do *SPMTool* e do *ATENA*, quanto aos deslocamentos e deformações, pôde-se notar que houve melhor aproximação nos estágios iniciais de carregamento. Os modelos constitutivos considerados pelo primeiro programa resultaram em curvas e estimativas de carga muito próximas, conforme visualizado nas figuras 6.8, 6.13 e 6.17. Além disso, em todas as análises foram superadas as estimativas analíticas de capacidade de carga dos elementos analisados, sendo que o *SPMTool* obteve previsões mais conservadoras e mais próximas às estimativas.

Devido ao fato de os programas considerarem elementos finitos e discretizações distintas, já era esperado que se obtivesse maior divergência nos resultados de deformações e deslocamentos. O *SPMTool* segue o conceito do MBP, em que as tensões normais são absorvidas, em sua maior parte, pelas bielas. Isso justifica a existência de maiores deformações, já que as tensões serão mais elevadas nessas posições.

Por outro lado, os resultados de carregamento obtidos pelo *SPMTool* são notáveis por se tratarem de modelos com discretizações muito inferiores aos utilizados no *ATENA*. Inclusive, isso pode ser percebido no custo computacional das análises; enquanto o processo todo pode levar mais de dez minutos, para os modelos AT-A, menos de 15 segundos bastaram no *SPMTool*. Esse fato, aliado à facilidade de aplicação do programa, justificam a sua utilização na prática da engenharia de estruturas.

É importante destacar que não se pode tratar os resultados do *ATENA* como corretos, visto que também se tratam de simulações e não representam a realidade. Tal afirmação apenas poderia ser feita com a utilização de modelos validados após análises de convergência para estruturas ensaiadas em experimentos.

O intuito deste trabalho é apenas demonstrar o potencial do *SPMTool* para aplicação em estruturas de concreto. Na prática, a inexperiência ou falta de conhecimento do usuário podem gerar resultados incoerentes em softwares avançados de elementos finitos, como o próprio *ATENA*.

7 Conclusão

Este trabalho procurou, como principal objetivo, desenvolver uma aplicação computacional moderna para análise de estruturas de concreto armado pelo MBP. Isso foi alcançado a partir do desenvolvimento do *SPMTool*, uma ferramenta simples capaz de efetuar análises elásticas e não-lineares de modelos biela-painel. De modo a avançar no conhecimento do tema, o MBP foi aplicado em conjunto com modelos de membrana mais atuais, como o MCTP e o MMA, além da já muito difundida TMCC.

Os modelos de membrana discutidos neste trabalho foram implementados na biblioteca *RCMembrane* e validados a partir de resultados experimentais de painéis de concreto armado, disponíveis na literatura. É importante destacar que, com o objetivo de otimizar os resultados do MCTP, uma nova relação foi proposta para minimizar o efeito do abrandamento do concreto em painéis submetidos a tensões de tração. Em geral, o algoritmo conseguiu prever satisfatoriamente os carregamentos de fissuração e de ruptura dos painéis, especialmente para o último caso, que apresentou menor variação de resultados.

A biblioteca *RCMembrane* é de código aberto e pode ser encontrada no repositório do autor (MELLO, 2020b), podendo, assim, ser utilizada em programas de elementos finitos desenvolvidos na plataforma *Microsoft.NET*, como rotina de cálculo das tensões nos pontos de integração dos elementos.

Conforme exposto no Capítulo 5, o *SPMTool* foi desenvolvido como um complemento para as versões mais recentes do *AutoCAD*, sendo uma ferramenta simples e de uso intuitivo que utiliza apenas parâmetros usuais dos materiais, como resistência e módulo de elasticidade, a qual é capaz de efetuar análises elásticas e não-lineares de modelos biela-painel. O software foi desenvolvido na plataforma *Microsoft.NET Framework* e seu código-fonte é aberto e hospedado na rede *Github* (MELLO, 2021c) sob licença não-comercial *Creative Commons*. Dessa forma, é possível que outros pesquisadores ao redor do mundo acessem e contribuam com o desenvolvimento.

Os modelos não-lineares do *SPMTool* aplicam a biblioteca *RCMembrane*, sendo as simulações validadas a partir de resultados experimentais de vigas-parede, vigas com abertura e consolos, obtidos anteriormente por outros pesquisadores. Em geral, os modelos superestimaram as cargas de fissuração desses elementos, exceto o MMA, que ultrapassou apenas 6%, em média, e obteve a menor variação.

Quanto às previsões de ruptura, também se conseguiram resultados consistentes com os dados experimentais; nesse caso, o MMA também apresentou menor variação e, por esse fato, foi escolhido como o modelo padrão no *SPMTool*. É importante destacar que o modelo constitutivo ainda pode ser escolhido pelo usuário, conforme exposto no Capítulo 5.

Em comparação ao seu antecessor *SPanCAD*, o *SPMTool*, além de ser compatível com versões recentes do *AutoCAD*, conseguiu prever com melhor precisão as cargas de fissuração e ruína das estruturas analisadas.

O Capítulo 6 apresentou as considerações de dimensionamento das armaduras e verificação das tensões no concreto em modelos biela-painel. Para os painéis, foi desenvolvido um processo iterativo que considera a contribuição do concreto na transmissão de tensões cisalhantes. A partir desse procedimento, foi possível reduzir as taxas de armadura em torno de 15%, em relação às calculadas pelo MPT.

O *SPMTool* foi aplicado ao dimensionamento de uma viga-parede contínua, uma viga-parede com aberturas e um consolo curto. Mostrou-se a simplicidade de aplicação do programa através das análises dos modelos desenvolvidos, além da facilidade do dimensionamento das armaduras e verificação do concreto, a partir da formulação apresentada.

Com as armaduras dimensionadas, os modelos foram analisados não-linearmente no próprio *SPMTool* e no software comercial *ATENA 2D*. De modo geral, observou-se que os modelos se deformaram mais no primeiro programa, apresentando deslocamentos e deformações nos tirantes superiores. Esse comportamento já era esperado, já que, pelo conceito do MBP, as tensões normais são absorvidas, em sua maior parte, pelas bielas. Isso faz com que as tensões e, conseqüentemente, as deformações sejam mais elevadas nessas regiões.

Por outro lado, pôde-se verificar que todos os modelos, em ambos os programas, ultrapassaram os carregamentos previstos analiticamente, sendo os resultados do *SPMTool* mais conservadores e próximos às estimativas. Os resultados de carregamentos de fissuração e de ruptura do *SPMTool*, mesmo com uma discretização muito inferior que do *ATENA 2D*, foram notáveis. Esse resultado se torna ainda mais relevante se forem comparados os custos computacionais de cada análise; enquanto o tempo de processamento pode facilmente passar de dez minutos nos modelos mais discretizados no *ATENA*, menos de 15 segundos bastam para completar o processo no *SPMTool*. É importante salientar que não se pode tratar as respostas do *ATENA* como corretas por também se tratarem de simulações, que dependem de múltiplos fatores, como a discretização da malha, os parâmetros adotados e a experiência do usuário.

Por se tratar de um complemento do *AutoCAD*, o *SPMTool* não exige adaptação adicional ao usuário, além de não necessitar da configuração de parâmetros avançados. Portanto, essa facilidade de aplicação, aliada à velocidade de execução das análises e à consistência das comparações com resultados experimentais, tornam o programa atrativo para utilização no dia-a-dia dos escritórios de cálculo estrutural, além da possibilidade de aplicação educacional em universidades.

7.1 Sugestões para trabalhos futuros

Sugere-se que sejam desenvolvidas em pesquisas futuras:

- aplicação dos *SPMTool* em outros tipos de estruturas de concreto armado, como blocos de fundação sobre duas estacas, dentes Gerber e, até mesmo, vigas convencionais;
- consideração de outras variáveis que influenciam na transferência de tensões de cisalhamento nos modelos de membrana, como o efeito escala e o efeito de pino das armaduras;
- desenvolvimento de rotinas de otimização das armaduras em modelos biela-painel;
- desenvolvimento de pesquisas experimentais de elementos dimensionados a partir das análises do *SPMTool*, para comparar com os resultados da solução não-linear do programa. Dessa forma, melhores conclusões podem ser obtidas a partir do posicionamento condizente das armaduras do modelo e do espécime de laboratório.

Referências

- AMERICAN CONCRETE INSTITUTE. *Building code requirements for structural concrete (ACI 318-19)*. Farmington Hills, 2019. 628 p.
- ARGYRIS, J. H.; KELSEY, S. Part I: General Theory. In: _____. *Energy theorems and structural analysis*. 1. ed. Londres: Butterworths, 1960. cap. 1-9, p. 2-59.
- ASHOUR, A. F. Shear capacity of reinforced concrete deep beams. *Journal of Structural Engineering*, v. 126, n. 9, p. 1045-1052, 2000. Disponível em: <[doi:10.1061/\(asce\)0733-9445\(2000\)126:9\(1045\)](https://doi.org/10.1061/(asce)0733-9445(2000)126:9(1045))>.
- ASHOUR, A. F.; RISHI, G. Tests of reinforced concrete continuous deep beams with web openings. *ACI Structural Journal*, v. 97, n. 3, 2000. Disponível em: <[doi:10.14359/4636](https://doi.org/10.14359/4636)>.
- ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. *NBR 8681: Ações e segurança nas estruturas - procedimento*. Rio de Janeiro, 2003. 15 p.
- ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. *NBR 6118: Projeto de estruturas de concreto - procedimento*. Rio de Janeiro, 2014. 256 p.
- ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. *NBR 9062: Projeto e execução de estruturas de concreto pré-moldado*. Rio de Janeiro, 2017. 86 p.
- ATENA 2D para Windows. v. 5.7.0. Praga: CERVENKA CONSULTING, 2019. Disponível em: <<https://www.cervenka.cz/products/atena/>>.
- AUTODESK. *Autodesk ObjectARX Managed .NET API*. 2020. Disponível em: <<https://www.autodesk.com/developer-network/platform-technologies/autocad>>.
- AUTODESK AutoCAD para Windows. Release 14. [S.l.]: AUTODESK, 1997.
- AUTODESK AutoCAD para Windows. v. 2020. AUTODESK, 2019. Disponível em: <<https://www.autodesk.com/products/autocad/>>.
- BATHE, K. J. *Finite Element Procedures*. 2. ed. Nova Jersey: Prentice Hall, 1996.
- BAUMANN, T. Zur Frage der Netzbewehrung von Flachentragwerken. *Der Bauingenieur*, v. 47, n. 10, p. 367-377, 1972.
- BENTZ, E. C. *Sectional Analysis of Reinforced Concrete Members*. Tese (Doutorado em Engenharia Civil) — University of Toronto, Toronto, 2000.
- BENTZ, E. C.; VECCHIO, F. J.; COLLINS, M. P. Simplified modified compression field theory for calculating shear strength of reinforced concrete elements. *ACI Structural Journal*, v. 103, n. 4, p. 614-624, 2006. Disponível em: <[doi:10.14359/16438](https://doi.org/10.14359/16438)>.
- BHIDE, S. B.; COLLINS, M. P. Influence of axial tension on the shear capacity of reinforced concrete members. *ACI Structural Journal*, v. 86, n. 5, p. 570-581, 1989. Disponível em: <[doi:10.14359/3013](https://doi.org/10.14359/3013)>.

- BIRRCHE, D. et al. *Strength and serviceability design of reinforced concrete deep beams*. Austin, 2009.
- BLAAUWENDRAAD, J. *Design of structural concrete with a Stringer-Panel-Model (SPM)*. Zurique: Instituto Federal de Tecnologia de Zurique, 1994.
- BLAAUWENDRAAD, J.; HOOGENBOOM, P. C. J. Discrete elements in structural concrete design. *Heron*, v. 42, n. 3, p. 159–168, 1997.
- BLAAUWENDRAAD, J.; HOOGENBOOM, P. C. J. Design Instrument SPanCAD for Shear Walls and D-Regions. In: FIB CONGRESS, 1., 2002, Osaka. Osaka: International Federation for Structural Concrete, 2002.
- CERVENKA, V.; JENDELE, L.; CERVENKA, J. *ATENA Program Documentation Part 1: Theory*. Praga, 2021. Disponível em: <https://www.cervenka.cz/assets/files/atena-pdf/ATENA_Theory.pdf>.
- CHAKRABARTI, P. R.; FARAH, D. J.; KASHOU, S. I. Reinforced and precompressed concrete corbels—an experimental study. *ACI Structural Journal*, v. 86, n. 4, 1989. Disponível em: <[doi:10.14359/2927](https://doi.org/10.14359/2927)>.
- CHANDRUPATLA, T. R.; BELEGUNDU, A. D. *Introduction to Finite Elements in Engineering Fourth Edition*. 4. ed. Harlow: Pearson Education, 2012. ISBN 978-0-273-76368-0.
- COLLINS, M. P.; MITCHELL, D. Shear and torsion design of prestressed and non-prestressed concrete beams. *PCI Journal*, v. 25, n. 5, p. 32–100, 1980. Disponível em: <[doi:10.15554/pci.09011980.32.100](https://doi.org/10.15554/pci.09011980.32.100)>.
- COLLINS, M. P.; VECCHIO, F. J.; MEHLHORN, G. An international competition to predict the response of reinforced concrete panels. *Canadian Journal of Civil Engineering*, v. 12, n. 3, p. 624–644, 1985. Disponível em: <[doi:10.1139/l85-070](https://doi.org/10.1139/l85-070)>.
- COMITE EURO-INTERNATIONAL DU BETON. *CEB-FIP Model Code 1990*. Lausanne, 1993. 461 p.
- CREATIVE COMMONS. *Licença de Atribuição-NãoComercial 4.0 Internacional (CC BY-NC 4.0)*. Disponível em: <<https://creativecommons.org/licenses/by-nc/4.0/>>.
- CRISFIELD, M. A fast incremental/iterative solution procedure that handles “snap-through”. *Computers & Structures*, v. 13, n. 1-3, p. 55–62, jun 1981.
- DAMKILDE, L.; KRENK, S. Limits - A system for limit state analysis and optimal material layout. *Computers & Structures*, v. 64, n. 1-4, p. 709–718, 1997. Disponível em: <[doi:10.1016/s0045-7949\(96\)00160-5](https://doi.org/10.1016/s0045-7949(96)00160-5)>.
- DANISH ENERGY AGENCY. *DK NA:2013 - National Annex to Eurocode 2*. Goteborg, 2014. 35 p.
- ELFGREN, L. *Reinforced concrete beam loaded in combined torsion, bending and shear: a study of the ultimate load-carrying capacity*. Göteborg, 1972. 232 p.
- FOSTER, S. J.; POWELL, R. E.; SELIM, H. S. Performance of high-strength concrete corbels. *ACI Structural Journal*, v. 93, n. 5, 1996. Disponível em: <[doi:10.14359/9714](https://doi.org/10.14359/9714)>.

- GUPTA, P. R. *Shear Design of Reinforced Concrete Members under Axial Compression*. Tese (Doutorado em Engenharia Civil) — Universidade de Toronto, Toronto, 1998.
- HAUKSDÓTTIR, B. *Analysis of a Reinforced Concrete Shear Wall*. Dissertação (Mestrado em Engenharia Civil) — Universidade Técnica da Dinamarca, Lyngby, 2007.
- HIBBELER, R. C. *Resistência dos Materiais*. 7. ed. São Paulo: Pearson Prentice Hall, 2010.
- HOGNESTAD, E. Ultimate strength of reinforced concrete in American design practice. In: SYMPOSIUM ON THE STRENGTH OF CONCRETE STRUCTURES, 1956, Londres. *Anais...* Londres: [s.n.], 1956.
- HOGNESTAD, E.; HANSON, N. W.; MCHENRY, D. Concrete Stress Distribution in Ultimate Strength Design. *Journal of the American Concrete Institute*, v. 52, n. 12, p. 455–480, 1955.
- HOOGENBOOM, P. C. J. *Het staaf-paneel-model*. Dissertação (Mestrado em Engenharia Civil) — Universidade Tecnológica de Delft, Delft, 1993.
- HOOGENBOOM, P. C. J. *Discrete elements and nonlinearity in design of structural concrete walls*. Tese (Doutorado em Engenharia Civil) — Universidade Tecnológica de Delft, Delft, 1998.
- HOOGENBOOM, P. C. J.; BLAAUWENDRAAD, J. Quadrilateral shear panel. *Engineering structures*, v. 22, p. 1690–1698, 2000. Disponível em: <doi:10.1016/s0141-0296(99)00061-9>.
- HSU, T. T. C. Softened truss model theory for shear and torsion. *ACI Structural Journal*, v. 85, n. 6, p. 624–635, 1988. Disponível em: <doi:10.14359/2740>.
- HSU, T. T. C. Stresses and Crack Angles in Concrete Membrane Elements. *Journal of Structural Engineering*, v. 124, n. 12, p. 1476–1484, 1998. Disponível em: <doi:10.1061/(asce)0733-9445(1998)124:12(1476)>.
- HSU, T. T. C. Unified Approach to Shear Analysis and Design. *Cement and Concrete Composites*, v. 20, p. 419–4335, 1998. Disponível em: <doi:10.1016/s0958-9465(98)00028-6>.
- HSU, T. T. C.; MO, Y. L. *Unified theory of concrete structures*. 1. ed. Chichester: Wiley, 2010.
- HSU, T. T. C.; ZHU, R. R. H. Softened membrane model for reinforced concrete elements in shear. *ACI Structural Journal*, American Concrete Institute, v. 99, n. 4, 2002. Disponível em: <doi:10.14359/12115>.
- INTERNATIONAL FEDERATION FOR STRUCTURAL CONCRETE. *fib Model Code for Concrete Structures 2010*. Lausanne, 2013. 436 p.
- ISMAIL, K. S.; GUADAGNINI, M.; PILAKOUTAS, K. Shear behavior of reinforced concrete deep beams. *ACI Structural Journal*, American Concrete Institute, v. 114, n. 1, jan 2016. Disponível em: <doi:10.14359/51689151>.

- JASIM, W. A.; ALLAWI, A. A.; OUKAILI, N. K. A. Effect of size and location of square web openings on the entire behavior of reinforced concrete deep beams. *Civil Engineering Journal*, v. 5, n. 1, p. 209, jan 2019. Disponível em: <[doi:10.28991/cej-2019-03091239](https://doi.org/10.28991/cej-2019-03091239)>.
- KAERN, J. C. *The stringer method applied to discs with holes*. Copenhagen, 1979. 7 p.
- KAKLAUSKAS, G.; RAMANAUSKAS, R.; NG, P.-L. Predicting crack spacing of reinforced concrete tension members using strain compliance approach with debonding. *Journal of Civil Engineering and Management*, v. 25, n. 5, p. 422–430, may 2019. Disponível em: <[doi:10.3846/jcem.2019.9871](https://doi.org/10.3846/jcem.2019.9871)>.
- KIM, N. *Introduction to Nonlinear Finite Element Analysis*. Nova Iorque: Springer, 2015.
- LAMPERT, P.; THÜRLIMANN, B. *Torsionsversuche an Stahlbetonbalken (Ensaio de torção em vigas de concreto armado)*. Zúrich, 1968. 103 p.
- LARSEN, A. G. *Repositório da biblioteca Units.NET*. 2013. Disponível em: <<https://www.github.com/angularsen/UnitsNet>>.
- LEONHARDT, F.; MÖNNIG, E. *Estructuras de hormigón armado*. 3. ed. Buenos Aires: El Ateneo, 1988. v. 1.
- LUNDGREN, H. *Cylindrical Shells*. Copenhagen: The Danish Technical Press, 1949.
- MARTI, P.; MEYBOOM, J. Response of prestressed concrete elements to in-plane shear forces. *ACI Structural Journal*, v. 89, n. 5, 1992. Disponível em: <[doi:10.14359/2956](https://doi.org/10.14359/2956)>.
- MELLO, A. F. A. *Análise e dimensionamento de vigas-parede em concreto armado utilizando o Método Biela-Painel*. Dissertação (Mestrado em Engenharia Civil) — Universidade Estadual de Maringá, Maringá, 2015.
- MELLO, A. F. A. *Repositório da biblioteca Material*. 2020. Disponível em: <<https://www.github.com/andrefmello91/Material>>.
- MELLO, A. F. A. *Repositório da biblioteca RCMembrane*. 2020. Disponível em: <<https://www.github.com/andrefmello91/Reinforced-Concrete-Membrane>>.
- MELLO, A. F. A. *Repositório da biblioteca FEMAnalysis*. 2021. Disponível em: <<https://github.com/andrefmello91/FEM-Analysis>>.
- MELLO, A. F. A. *Repositório da biblioteca SPMElements*. 2021. Disponível em: <<https://github.com/andrefmello91/SPMElements>>.
- MELLO, A. F. A. *Repositório do SPMTool*. 2021. Disponível em: <<https://www.github.com/andrefmello91/SPMTool>>.
- MELLO, A. F. A.; SOUZA, R. A. Analysis and Design of Reinforced Concrete Deep Beams by a Manual Approach of Stringer-Panel Method. *Latin American Journal of Solids and Structures*, v. 13, n. 6, p. 1126–1151, jun. 2016. ISSN 1679-7825. Disponível em: <[doi:10.1590/1679-78252623](https://doi.org/10.1590/1679-78252623)>.
- MICROSOFT .NET Cross-platform v. 4.5. MICROSOFT, 2020. Disponível em: <<https://www.dotnet.microsoft.com/>>.

- MICROSOFT Visual Studio Community 2019. MICROSOFT, 2019. Disponível em: <<https://www.visualstudio.microsoft.com/>>.
- MITCHELL, D.; COOK, W. D. *Design of disturbed regions*. Stuttgart, 1991. 6 p.
- NIELSEN, M. P. Om forskydningsarmering i jernbetonbjælker (Armadura de cisalhamento em vigas de concreto armado). *Bygningsstatistiske Meddelelser*, v. 38, n. 2, p. 33–58, 1967.
- NIELSEN, M. P. On the strength of reinforced concrete discs. *Acta Polytechnica Scandinavica, Civil Engineering and Building Construction Series*, v. 70, n. 1, 1971.
- NIELSEN, M. P.; HOANG, L. C. *Limit analysis and concrete plasticity*. 3. ed. Boca Raton: CRC Press, 2011.
- PANG, X.-B. D.; HSU, T. T. C. Behavior of reinforced concrete membrane elements in shear. *ACI Structural Journal*, American Concrete Institute, v. 92, n. 6, 1995. Disponível em: <[doi:10.14359/9661](https://doi.org/10.14359/9661)>.
- RABBAT, B. G.; COLLINS, M. P. A variable angle space truss model for structural concrete members subjected to complex loading. *ACI Special Publication*, v. 55, p. 547–587, 1978.
- RAO, S. S. *The Finite Element Method in Engineering*. 5. ed. Burlington: Elsevier, 2011.
- REDDY, J. N. *An Introduction to Nonlinear Finite Element Analysis*. 2. ed. Oxford: Oxford University Press, 2014.
- REFER, D. *Computer Based FE Analysis of Reinforced Concrete Walls by the Stringer Method*. Dissertação (Mestrado em Engenharia Civil) — Universidade de Aalborg, Aalborg, 2012.
- RODRIGUEZ, A. *LiveCharts for .NET*. 2017. Disponível em: <<https://www.lvcharts.net>>.
- RUEGG, C.; CUDA, M.; GAEL, J. V. *Biblioteca Math.NET Numerics*. Math.NET Project, 2020. Disponível em: <<https://numerics.mathdotnet.com/>>.
- SCHLAICH, J.; SCHÄFER, K.; JENNEWEIN, M. Toward a consistent design of structural concrete. *PCI journal*, v. 32, n. 3, p. 74–150, 1987. Disponível em: <[doi:10.15554/pci.05011987.74.150](https://doi.org/10.15554/pci.05011987.74.150)>.
- SIGRIST, V. et al. Background to the fib model code 2010 shear provisions - part i: beams and slabs. *Structural Concrete*, v. 14, n. 3, p. 195–203, sep 2013. Disponível em: <[doi:10.1002/suco.201200066](https://doi.org/10.1002/suco.201200066)>.
- SILVA, J. G. T. *Contribuição ao projeto de elementos estruturais de concreto armado com descontinuidades através do modelo de painéis enrijecidos*. Dissertação (Mestrado em Engenharia Civil) — Universidade Federal de Alagoas, Maceió, 2004.
- SIMONE, A. *Progetto di strutture in C. A. con un Modello a Pannelli e Correnti*. Tese (Doutorado em Engenharia Civil) — Politécnico de Milão, Milão, 1998.
- SOUZA, R. A. *Concreto estrutural: análise e dimensionamento de elementos com descontinuidades*. Tese (Doutorado em Engenharia Civil) — Universidade de São Paulo, São Paulo, 2004.

SOUZA, R. A. Análise de elementos de membrana em concreto estrutural utilizando o “Softened Truss Model”. *Revista Engenharia Civil - UM*, n. 38, p. 17–34, 2010.

SOUZA, R. A. Dimensionamento automático de elementos de membrana em concreto armado utilizando o modelo “Equilibrium Plasticity Truss”. In: CONGRESSO BRASILEIRO DO CONCRETO, 52., 2010, Fortaleza. *Anais...* Fortaleza: Instituto Brasileiro do Concreto, 2010.

SOUZA, R. A. *Relatório de estágios de pós-doutorado de curta duração realizados na École Polytechnique Fédérale de Lausanne (Suíça) e na Technology University of Delft (Holanda)*. Lausanne, 2011. 54 p.

SOUZA, R. A. Abordagem manual e computacional do “Stringer and Panel Method” para análise e dimensionamento de paredes em concreto estrutural. In: CONGRESSO BRASILEIRO DO CONCRETO, 54., 2012, Maceió. *Anais...* Maceió: Instituto Brasileiro do Concreto, 2012.

SOUZA, R. A. Desenvolvimento de uma ferramenta computacional para a análise de elementos de membrana em concreto estrutural. *Revista Internacional de Métodos Numéricos para Cálculo y Diseño en Ingeniería*, v. 29, n. 1, p. 38–51, jan 2013. Disponível em: <[doi:10.1016/j.rimni.2011.10.003](https://doi.org/10.1016/j.rimni.2011.10.003)>.

SOUZA, R. A. *Análise e dimensionamento de elementos de membrana em concreto estrutural*. Maringá: EDUEM, 2016.

TARQUINI, G.; SGAMBI, L. Stringer Panel Method: a discrete model to project structural reinforced concrete elements. In: INTERNATIONAL STRUCTURAL ENGINEERING CONSTRUCTION CONFERENCE, 2., 2003, Roma. *Anais...* Roma, 2003.

THÜRLIMANN, B. *Plastic analysis of reinforced concrete beams*. Copenhagen, 1979. 20 p.

VECCHIO, F. J. Nonlinear finite element analysis of reinforced concrete membranes. *ACI Structural Journal*, v. 86, n. 1, p. 26–35, 1989. Disponível em: <[doi:10.14359/2620](https://doi.org/10.14359/2620)>.

VECCHIO, F. J. Reinforced concrete membrane element formulations. *Journal of Structural Engineering*, v. 116, n. 3, p. 730–750, 1990. Disponível em: <[doi:10.1061/\(asce\)0733-9445\(1990\)116:3\(730\)](https://doi.org/10.1061/(asce)0733-9445(1990)116:3(730))>.

VECCHIO, F. J. Disturbed Stress Field Model for reinforced concrete: formulation. *Journal of Structural Engineering*, v. 126, n. 9, p. 1070–1077, 2000. Disponível em: <[doi:10.1061/\(asce\)0733-9445\(2000\)126:9\(1070\)](https://doi.org/10.1061/(asce)0733-9445(2000)126:9(1070))>.

VECCHIO, F. J. Disturbed stress field model for reinforced concrete: implementation. *Journal of Structural Engineering*, v. 127, n. 1, p. 12–20, 2001. Disponível em: <[doi:10.1061/\(asce\)0733-9445\(2001\)127:1\(12\)](https://doi.org/10.1061/(asce)0733-9445(2001)127:1(12))>.

VECCHIO, F. J.; CHAN, C. C. L. Reinforced concrete membrane elements with perforations. *Journal of Structural Engineering*, v. 116, n. 9, p. 2344–2360, 1990. Disponível em: <[doi:10.1061/\(asce\)0733-9445\(1990\)116:9\(2344\)](https://doi.org/10.1061/(asce)0733-9445(1990)116:9(2344))>.

VECCHIO, F. J.; COLLINS, M. P. *The response of reinforced concrete to in-plane shear and normal stresses*. Toronto, 1982.

- VECCHIO, F. J.; COLLINS, M. P. The modified compression-field theory for reinforced concrete elements subjected to shear. *ACI Structural Journal*, v. 83, n. 22, p. 219–231, 1986. Disponível em: [doi:10.14359/10416](https://doi.org/10.14359/10416).
- VECCHIO, F. J.; COLLINS, M. P. Predicting the response of reinforced concrete beams subjected to shear using modified compression field theory. *ACI Structural Journal*, v. 85, n. 3, p. 258–268, 1988. Disponível em: [doi:10.14359/2515](https://doi.org/10.14359/2515).
- VECCHIO, F. J.; COLLINS, M. P.; ASPIOTIS, J. High-strength concrete elements subjected to shear. *ACI Structural Journal*, v. 91, n. 4, p. 423–433, 1994. Disponível em: [doi:10.14359/4149](https://doi.org/10.14359/4149).
- VECCHIO, F. J.; LAI, D. Crack shear-slip in reinforced concrete elements. *Journal of Advanced Concrete Technology*, Japan Concrete Institute, v. 2, n. 3, p. 289–300, 2004. Disponível em: [doi:10.3151/jact.2.289](https://doi.org/10.3151/jact.2.289).
- VECCHIO, F. J. et al. Disturbed stress field model for reinforced concrete: validation. *Journal of Structural Engineering*, v. 127, n. 4, p. 350–358, 2001. Disponível em: [doi:10.1061/\(asce\)0733-9445\(2001\)127:4\(350\)](https://doi.org/10.1061/(asce)0733-9445(2001)127:4(350)).
- WALRAVEN, J. C. *Aggregate interlock: A theoretical and experimental analysis*. Tese (Doutorado em Engenharia Civil) — Universidade Técnica de Delft, Delft, 1980.
- WANG, Q.; HOOGENBOOM, P. C. J. Nonlinear analysis of reinforced concrete continuous deep beams using stringer-panel model. *Asian Journal of Civil Engineering*, v. 5, n. 1-2, p. 25–40, 2004.
- WONG, P. S.; VECCHIO, F. J.; TROMMELS, H. *VecTor2 & FormWorks User's Manual*. Toronto, 2013.
- XIE, L.; BENTZ, E. C.; COLLINS, M. P. Influence of axial stress on shear response of reinforced concrete elements. *ACI Structural Journal*, v. 108, n. 6, 2011. Disponível em: [doi:10.14359/51683373](https://doi.org/10.14359/51683373).
- ZHANG, L.; HSU, T. T. C. Behavior and analysis of 100 MPa concrete membrane elements. *Journal of Structural Engineering*, v. 124, n. 1, p. 24–34, 1998. Disponível em: [doi:10.1061/\(asce\)0733-9445\(1998\)124:1\(24\)](https://doi.org/10.1061/(asce)0733-9445(1998)124:1(24)).
- ZHU, R. R. H.; HSU, T. T. C. Poisson effect in reinforced concrete membrane elements. *ACI Structural Journal*, American Concrete Institute, v. 99, n. 5, 2002. Disponível em: [doi:10.14359/12303](https://doi.org/10.14359/12303).
- ZHU, R. R. H.; HSU, T. T. C.; LEE, J. Rational shear modulus for smeared-crack analysis of reinforced concrete. *ACI Structural Journal*, American Concrete Institute, v. 98, n. 4, 2001. Disponível em: [doi:10.14359/10287](https://doi.org/10.14359/10287).

APÊNDICE A – Formulação das Bielas

Este apêndice apresenta a formulação das bielas. São apresentadas as equações para obtenção dos esforços nodais e da matriz de rigidez de cada elemento. A Figura A.1 apresenta resumidamente o processo de solução das forças nodais a partir dos deslocamentos.

Figura A.1 – Relações entre os parâmetros da biela

$$\begin{array}{ccccc}
 & & & & [K] \\
 & \underbrace{\left\{ \begin{array}{c} f_{x_1} \\ f_{y_1} \\ f_{x_2} \\ f_{y_2} \\ f_{x_3} \\ f_{y_3} \end{array} \right\}}_{[T]^T} & \underbrace{\left\{ \begin{array}{c} f_1 \\ f_2 \\ f_3 \end{array} \right\}}_{[K_t]} & [D] & \underbrace{\left\{ \begin{array}{c} u_1 \\ u_2 \\ u_3 \end{array} \right\}}_{[T]} \\
 & & & & \underbrace{\left\{ \begin{array}{c} u_{x_1} \\ u_{y_1} \\ u_{x_2} \\ u_{y_2} \\ u_{x_3} \\ u_{y_3} \end{array} \right\}}_{[K]}
 \end{array}$$

Fonte: adaptado de Hoogenboom (1998).

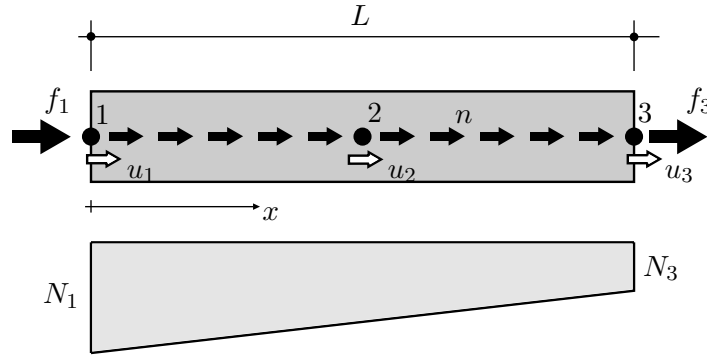
As definições dos parâmetros mostrados na Figura A.1 serão apresentadas nas seções seguintes. Todo o equacionamento aqui apresentado foi obtido de Hoogenboom (1998) e Chandrupatla e Belegundu (2012).

A.1 Geometria

As bielas são elementos lineares carregados por apenas esforços axiais, existindo uma força distribuída (n) ao longo de seu comprimento (L) e duas forças pontuais externas (f_1 e f_3) nas extremidades, como ilustrado na Figura A.2.

Estas três forças (n , f_1 e f_3) existentes na biela devem satisfazer uma equação de equilíbrio. Além dessas forças, o modelo considera três deslocamentos (u_1 , u_2 e u_3) na direção axial da biela. O esforço normal variará linearmente ao longo da biela, sendo igual a N_1 na extremidade esquerda e N_3 , na direita.

Figura A.2 – Biela carregada com força distribuída

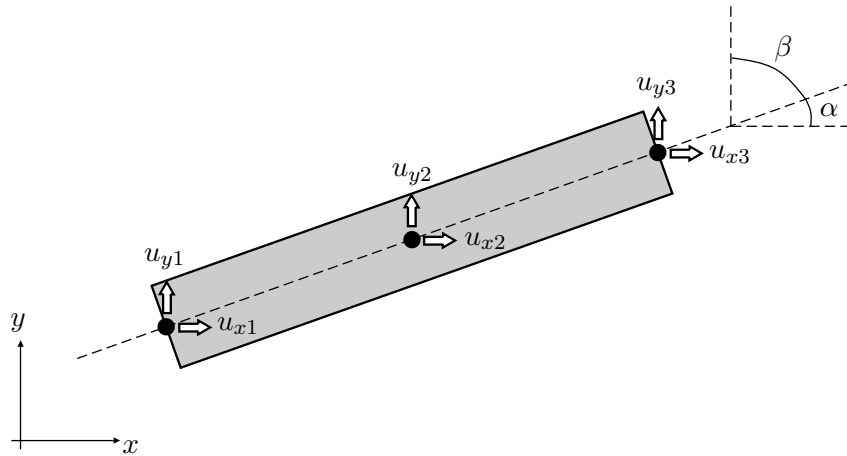


Fonte: adaptado de Hoogenboom (1998).

A.2 Sistema de referência local

O modelo de biela considera um sistema de referência local que inicia no nó 1 e termina no nó 3. Os deslocamentos e forças locais são relacionados ao sistema de coordenadas globais através da matriz de transformação T . Esta matriz tem como função transformar os deslocamentos e forças nodais locais (Figura A.2) num sistema global de coordenadas (Figura A.3).

Figura A.3 – Deslocamentos de uma biela num sistema global de coordenadas



Os cossenos diretores (l e m) do eixo axial da biela são dados por:

$$l = \cos \alpha \quad m = \cos \beta \quad (\text{A.1})$$

sendo α e β os ângulos formados entre o eixo local x da biela e os eixos globais x e y , respectivamente.

A contribuição da força f_i às forças nodais é:

$$f_{xi} = f_i l \quad f_{yi} = f_i m \quad (\text{A.2})$$

O deslocamento nodal local (u_i) pode ser expresso em função dos deslocamentos nodais globais por:

$$u_i = u_{xi} l + u_{yi} m \quad (\text{A.3})$$

A matriz de transformação de coordenadas $[T]$, é dada por:

$$[T] = \begin{bmatrix} l & m & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & l & m & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & l & m \end{bmatrix} \quad (\text{A.4})$$

sendo os zeros representados por pontos em todas as matrizes desta seção.

Os vetores de força global $\{f\}$ e local $\{f_l\}$ podem ser relacionados pela matriz de transformação por:

$$\{f_l\} = [T]^T \{f\} \quad (\text{A.5})$$

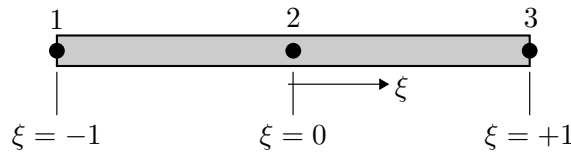
Já os vetores de deslocamento global $\{u\}$ e $\{u_l\}$ se relacionam por:

$$\{u\} = [T] \{u_l\} \quad (\text{A.6})$$

A.3 Funções de forma quadráticas

As bielas podem ser discretizadas como um elemento linear com funções de forma quadráticas. Baseando-se na formulação de Chandrupatla e Belegundu (2012), o sistema x de coordenadas é remapeado a um sistema adimensional ξ , como ilustrado na Figura A.4.

Figura A.4 – Elemento quadrático em coordenadas ξ



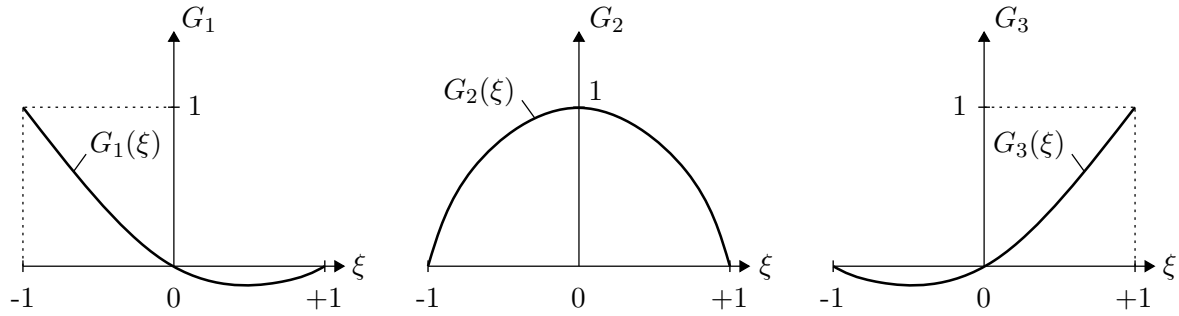
Fonte: Adaptado de Chandrupatla e Belegundu (2012).

A transformação entre os dois sistemas é feita por:

$$\xi = \frac{2(x - x_2)}{x_3 - x_1} \quad (\text{A.7})$$

As funções de forma quadráticas são ilustradas graficamente na Figura A.5 e são obtidas por:

$$G_1(\xi) = -\frac{1}{2}\xi(1 - \xi) \quad G_2(\xi) = (1 + \xi)(1 - \xi) \quad G_3(\xi) = \frac{1}{2}\xi(1 + \xi) \quad (\text{A.8})$$

Figura A.5 – Funções de forma G_1 , G_2 e G_3 

Fonte: Adaptado de Chandrupatla e Belegundu (2012).

O campo de deslocamentos do elemento é ilustrado na Figura A.6 e é expresso por:

$$u = G_1 q_1 + G_2 q_2 + G_3 q_3 \quad (\text{A.9a})$$

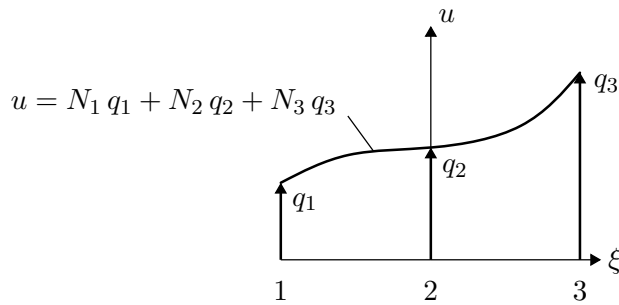
ou:

$$u = G q \quad (\text{A.9b})$$

sendo:

$$G = \{G_1 \quad G_2 \quad G_3\} \quad q = \{q_1 \quad q_2 \quad q_3\}^T$$

Figura A.6 – Interpolação quadrática do campo de deslocamentos



Fonte: Adaptado de Chandrupatla e Belegundu (2012).

A deformação pode ser encontrada fazendo:

$$\varepsilon = \frac{du}{dx} = \frac{du}{d\xi} \frac{d\xi}{dx} \quad (\text{A.10})$$

As derivadas parciais são encontradas a partir de (A.7) e (A.9):

$$\frac{du}{d\xi} = \left\{ \frac{dG_1}{d\xi} \quad \frac{dG_2}{d\xi} \quad \frac{dG_3}{d\xi} \right\} q \quad \frac{d\xi}{dx} = \frac{2}{x_3 - x_1} \quad (\text{A.11})$$

Portanto, já que se utiliza uma função quadrática para os deslocamentos, a deformação variará linearmente no elemento. O vetor de deformações será:

$$\varepsilon = B q \quad (\text{A.12})$$

sendo:

$$B = \frac{2}{x_3 - x_1} \left\{ -\frac{1-2\xi}{2} \quad -2\xi \quad \frac{1+2\xi}{2} \right\} \quad (\text{A.13})$$

A.4 Matriz de rigidez

Considerando a biela ilustrada na Figura A.2, no contexto de energia potencial complementar, os deslocamentos u_1 e u_3 são fixos e as forças f_1 e f_3 são reações de apoio. O deslocamento ao longo de toda a biela é fixo e a força n é a sua consequente reação de apoio. A biela é um elemento que possui somente rigidez na direção dos seus três graus de liberdade (u_1 , u_2 e u_3), não fornecendo, portanto, rigidez à flexão ou cisalhamento.

Assim, pode-se definir a energia potencial total na biela como:

$$\Pi = \int_0^L N(\varepsilon) \varepsilon dx - f_1 u_1 - f_2 u_2 - f_3 u_3 \quad (\text{A.14a})$$

sendo ε a deformação normal na biela, definida em (A.12), e $N(\varepsilon)$ a força normal devida a ε .

A partir de (A.7), sabendo que $dx = L/2 d\xi$, escreve-se a energia potencial total como:

$$\Pi = \frac{L}{2} \int_{-1}^1 N(\varepsilon) \varepsilon d\xi - f_1 u_1 - f_2 u_2 - f_3 u_3 \quad (\text{A.14b})$$

Utilizando o princípio da mínima energia potencial,

$$\frac{\partial \Pi}{\partial u_1} = \frac{\partial \Pi}{\partial u_2} = \frac{\partial \Pi}{\partial u_3} = 0 \quad (\text{A.15})$$

e considerando os deslocamentos conhecidos: $q_1 = u_1$, $q_2 = u_2$ e $q_3 = u_3$, encontram-se as forças nodais:

$$f_1 = \frac{1}{2} \int_{-1}^1 N(\varepsilon) (2\xi - 1) d\xi \quad (\text{A.16a})$$

$$f_2 = -2 \int_{-1}^1 N(\varepsilon) \xi d\xi \quad (\text{A.16b})$$

$$f_3 = \frac{1}{2} \int_{-1}^1 N(\varepsilon) (2\xi + 1) d\xi \quad (\text{A.16c})$$

A matriz de rigidez local da biela é então dada por:

$$[K_l] = \begin{bmatrix} K_{l11} & K_{l12} & K_{l13} \\ K_{l21} & K_{l22} & K_{l23} \\ K_{l31} & K_{l32} & K_{l33} \end{bmatrix} \quad (\text{A.17})$$

sendo que seus elementos são encontrados a partir das derivadas parciais das forças nodais:

$$\begin{aligned}
 K_{l_{11}} &= \frac{\partial f_1}{\partial u_1} = \frac{1}{2L} \int_{-1}^1 N'(\varepsilon) (2\xi - 1)^2 d\xi \\
 K_{l_{22}} &= \frac{\partial f_2}{\partial u_2} = \frac{8}{L} \int_{-1}^1 N'(\varepsilon) \xi^2 d\xi \\
 K_{l_{33}} &= \frac{\partial f_3}{\partial u_3} = \frac{1}{2L} \int_{-1}^1 N'(\varepsilon) (2\xi + 1)^2 d\xi \\
 K_{l_{12}} &= K_{l_{21}} = \frac{\partial f_1}{\partial u_2} = -\frac{2}{L} \int_{-1}^1 N'(\varepsilon) \xi (2\xi - 1) d\xi \\
 K_{l_{13}} &= K_{l_{31}} = \frac{\partial f_1}{\partial u_3} = \frac{1}{2L} \int_{-1}^1 N'(\varepsilon) (4\xi^2 - 1) d\xi \\
 K_{l_{23}} &= K_{l_{32}} = \frac{\partial f_2}{\partial u_3} = -\frac{2}{L} \int_{-1}^1 N'(\varepsilon) \xi (2\xi + 1) d\xi
 \end{aligned}$$

com:

$$N'(\varepsilon) = \frac{\partial N(\varepsilon)}{\partial \varepsilon}$$

Por fim, a solução das forças nodais será dada por:

$$\{f\} = [K] \{u\} \quad (\text{A.18})$$

sendo $\{f\}$ e $\{u\}$ os vetores de forças e deslocamentos nodais no sistema de referência global, e $[K]$ a matriz de rigidez, no sistema global, dada por:

$$[K] = [T]^T [K_l] [T] \quad (\text{A.19})$$

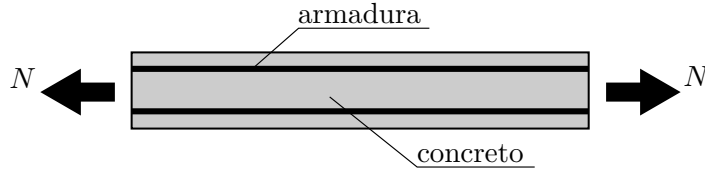
A.5 Determinação das forças nodais

As forças nodais na biela podem ser obtidas, para cada nó, considerando uma aproximação de uma biela submetida a um esforço normal constante (N) (Figura A.7), aplicada no centro de gravidade da seção transversal, que é composta por uma área de concreto (A_c) e uma área de aço (A_s).

Considerando aderência perfeita entre aço e concreto, ambos os materiais terão deformações equivalentes:

$$\{\varepsilon_c\} = \{\varepsilon_s\} = \{\varepsilon\}$$

Figura A.7 – Biela carregada com força normal constante



Fonte: adaptado de Hoogenboom (1998).

As deformações em cada nó podem ser obtidas aplicando as coordenadas nodais em (A.12). Assim, o vetor de deformação será:

$$\{ \varepsilon \} = \begin{Bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \end{Bmatrix} = \frac{1}{L} \begin{bmatrix} -3 & 4 & -1 \\ -1 & 0 & 1 \\ 1 & -4 & 3 \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \end{Bmatrix} \quad (\text{A.20})$$

O equilíbrio das forças internas se dá pela soma das contribuições do concreto (N_c) e do aço (N_s):

$$N(\varepsilon) = N_c(\varepsilon) + N_s(\varepsilon) = \sigma_c(\varepsilon) A_c + \sigma_s(\varepsilon) A_s \quad (\text{A.21})$$

sendo: σ_c, σ_s : tensão no concreto e no aço, respectivamente;

A_c, A_s : área da seção de concreto e de aço, respectivamente.

Observa-se que a força normal na seção ($N(\varepsilon)$) pode ser obtida a partir dos modelos constitutivos para os dois materiais. Pode-se, portanto, aplicar os modelos abordados no Capítulo 2 para calcular as tensões a partir das deformações em cada nó.

As forças nodais, por sua vez, podem ser obtidas a partir do equilíbrio na biela (Figura A.2). De modo a simplificar o cálculo introduz-se a força f_2 no nó intermediário como a resultante da força distribuída n :

$$f_2 = n L \quad (\text{A.22})$$

Analisando os nós de extremidade, pode-se escrever as forças f_1 e f_3 como:

$$f_1 = -N_1 \quad f_3 = N_3 \quad (\text{A.23})$$

Pelo equilíbrio global da biela, tem-se:

$$f_2 = -f_1 - f_3 = N_1 - N_3 \quad (\text{A.24})$$

Portanto, o vetor de forças nodais locais pode ser definido como:

$$\{ f_l \} = \begin{Bmatrix} f_1 \\ f_2 \\ f_3 \end{Bmatrix} = \begin{Bmatrix} -N_1 \\ N_1 - N_3 \\ N_3 \end{Bmatrix} \quad (\text{A.25})$$

com $N_1 = N(\varepsilon_1)$ e $N_3 = N(\varepsilon_3)$.

Observa-se que não é necessário calcular diretamente a força normal interna no nó intermediário, já que a força f_2 pode ser obtida diretamente pelo equilíbrio na biela.

A.6 Modelo linear

No modelo elástico-linear, considera-se uma seção íntegra de concreto ($A_s = 0$). Pela Lei de Hooke, a força normal na biela será:

$$N(\varepsilon) = E_c A \varepsilon \quad (\text{A.26})$$

sendo: E_c : módulo de elasticidade do concreto;
 A : área da seção transversal da biela.

Os elementos da matriz de rigidez local são calculados de acordo com (A.17), considerando que a derivada da força normal será constante e igual a:

$$N'(\varepsilon) = E_c A \quad (\text{A.27})$$

A matriz será então dada por:

$$[K_l] = \frac{E_c A}{3L} \begin{bmatrix} 7 & -8 & 1 \\ -8 & 16 & -8 \\ 1 & -8 & 7 \end{bmatrix} \quad (\text{A.28})$$

APÊNDICE B – Formulação dos Painéis

Este apêndice apresenta a formulação dos painéis. São apresentadas as equações para a obtenção de esforços e da matriz de rigidez dos elementos, sendo que os modelos linear e não-linear apresentam formulação distinta. Todo o equacionamento aqui apresentado foi obtido no trabalho de Hoogenboom (1998).

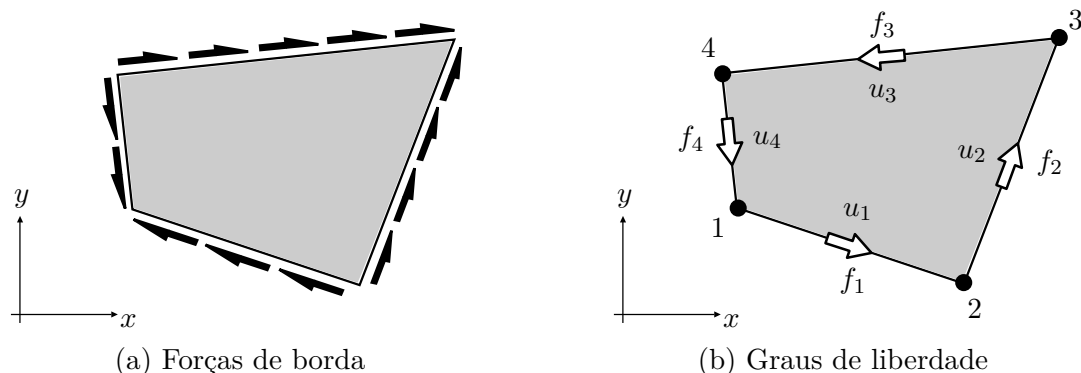
B.1 Modelo linear

Na formulação elástico-linear os painéis carregam apenas forças cisalhantes em seus lados. Dessa forma, os esforços normais são todos absorvidos pelas bielas, o que resulta numa armadura mais eficiente.

B.1.1 Relações

O painel é um elemento discreto quadrilateral que apresenta quatro forças de borda uniformemente distribuídas e quatro graus de liberdade em conjunto com estas forças, como ilustrado na Figura B.1a.

Figura B.1 – Pannel: modelo linear



Fonte: adaptado de Hoogenboom (1998).

É possível obter três relações de equilíbrio (equilíbrio de forças em x e y e equilíbrio de momentos), assim como três deslocamentos de corpo rígido independentes (duas translações e uma rotação). Portanto, existe apenas uma tensão generalizada (β) para descrever o campo de tensões e uma deformação generalizada (e) para descrever as deformações no painel.

A tensão generalizada (β) está associada à tensão de cisalhamento (τ) e a deformação generalizada (e) está associada à deformação por cisalhamento (γ), existentes no interior do painel. A relação tensão-deformação para um material elástico-linear é dada por:

$$\tau = G \gamma \quad (\text{B.1})$$

sendo G o módulo de elasticidade ao cisalhamento do material.

Para um material elástico linear, G pode ser obtido por (HIBBELER, 2010):

$$G = \frac{E}{2(1 + \nu)} \quad (\text{B.2})$$

sendo ν o coeficiente de Poisson do material. Para o concreto, este valor pode ser tomado como $\nu = 0,2$.

A relação de compatibilidade é dada por:

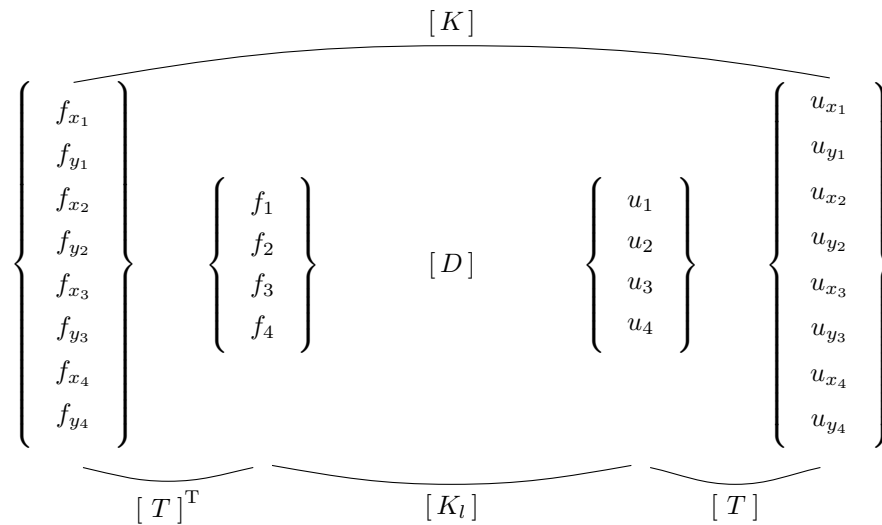
$$\{f\} = [K_l] \{u\} \quad (\text{B.3})$$

sendo $\{f\}$ o vetor de forças, $\{u\}$ o vetor de deslocamentos e $[K_l]$ a matriz de rigidez local do painel, dada por:

$$[K_l] = [B]^T [D] [B] \quad (\text{B.4})$$

A matriz $[B]$ relaciona as forças e os deslocamentos de borda com a tensão e a deformação generalizadas. A Figura B.2 apresenta as relações entre os parâmetros dos painéis.

Figura B.2 – Relações entre os parâmetros do painel

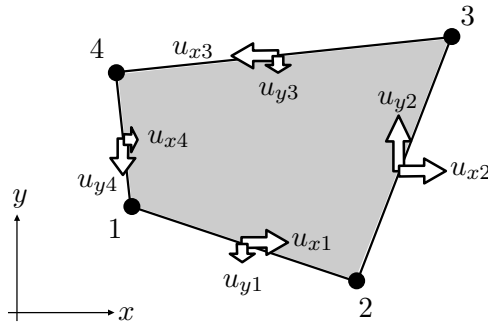


Fonte: adaptado de Hoogenboom (1998).

B.1.2 Transformação de coordenadas

A matriz de rigidez deduzida para os painéis relaciona as forças com os deslocamentos, que seguem a direção da face considerada no painel (Figura B.1b). Quando se consideram as coordenadas globais de um determinado modelo, é necessário obter as componentes destas forças e deslocamentos para os eixos globais do sistema (Figura B.3).

Figura B.3 – Painel: componentes de deslocamento no sistema de coordenadas global



As componentes da forças de borda f_i , em cada face do painel, serão:

$$f_{xi} = f_i m_i \quad f_{yi} = f_i n_i \quad (\text{B.5})$$

sendo m_i e n_i os cossenos diretores das faces (l_i) do painel:

$$m_i = \cos(x, l_i) \quad n_i = \cos(y, l_i)$$

sendo que a direção l_i sempre se inicia no nó i e termina no nó $i + 1$, fazendo o caminho no sentido anti-horário pelas faces do painel.

O deslocamento nodal local (u_i) pode ser expresso em função dos deslocamentos nodais globais por:

$$u_i = u_{xi} m_i + u_{yi} n_i \quad (\text{B.6})$$

Sendo o painel formado por quatro lados, a matriz de transformação de coordenadas $[T]$, será dada por:

$$[T] = \begin{bmatrix} m_1 & n_1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & m_2 & n_2 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & m_3 & n_3 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & m_4 & n_4 \end{bmatrix} \quad (\text{B.7})$$

sendo os zeros representados por pontos nas matrizes desta seção.

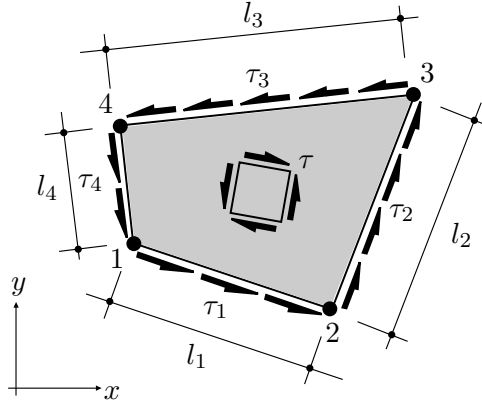
A matriz de rigidez do painel, referida aos eixos coordenados globais, será calculada por:

$$[K] = [T]^T [K_l] [T] \quad (\text{B.8})$$

B.1.3 Equilíbrio

O equilíbrio de forças é obtido considerando a Figura B.4. Em cada lado, existe uma tensão cisalhante (τ_i) com direção positiva seguindo a numeração dos vértices do painel, em sentido anti-horário.

Figura B.4 – Equilíbrio de forças no painel



Fonte: adaptado de Hoogenboom (1998).

Analisando o equilíbrio de forças em x e y , respectivamente, tem-se:

$$\tau_1 t l_1 \cos(x, l_1) + \tau_2 t l_2 \cos(x, l_2) + \tau_3 t l_3 \cos(x, l_3) + \tau_4 t l_4 \cos(x, l_4) = 0 \quad (\text{B.9a})$$

$$\tau_1 t l_1 \sin(x, l_1) + \tau_2 t l_2 \sin(x, l_2) + \tau_3 t l_3 \sin(x, l_3) + \tau_4 t l_4 \sin(x, l_4) = 0 \quad (\text{B.9b})$$

sendo: l_i : comprimento do lado i do painel;
 $\cos(x, l_i)$: cosseno do ângulo formado entre o lado i e o eixo x ;
 $\sin(x, l_i)$: seno do ângulo formado entre o lado i e o eixo x ;
 t : espessura do painel.

O equilíbrio de momentos, na origem do sistema cartesiano, resulta em:

$$\begin{aligned} & -\tau_1 t l_1 \cos(x, l_1) y_1 + \tau_1 t l_1 \sin(x, l_1) + \\ & -\tau_2 t l_2 \cos(x, l_2) y_2 + \tau_2 t l_2 \sin(x, l_2) + \\ & -\tau_3 t l_3 \cos(x, l_3) y_3 + \tau_3 t l_3 \sin(x, l_3) + \\ & -\tau_4 t l_4 \cos(x, l_4) y_4 + \tau_4 t l_4 \sin(x, l_4) = 0 \end{aligned} \quad (\text{B.10})$$

As relações trigonométricas utilizadas nas equações (B.9) e (B.10) podem ser expressas por:

$$\begin{aligned}
 \cos(x, l_1) &= \frac{x_2 - x_1}{l_1} & \text{sen}(x, l_1) &= \frac{y_2 - y_1}{l_1} \\
 \cos(x, l_2) &= \frac{x_3 - x_2}{l_2} & \text{sen}(x, l_2) &= \frac{y_3 - y_2}{l_2} \\
 \cos(x, l_3) &= \frac{x_4 - x_3}{l_3} & \text{sen}(x, l_3) &= \frac{y_4 - y_3}{l_3} \\
 \cos(x, l_4) &= \frac{x_1 - x_4}{l_4} & \text{sen}(x, l_4) &= \frac{y_1 - y_4}{l_4}
 \end{aligned} \tag{B.11}$$

As equações de equilíbrio podem ser reescritas por:

$$\tau_1 (x_2 - x_1) + \tau_2 (x_3 - x_2) + \tau_3 (x_4 - x_3) + \tau_4 (x_1 - x_4) = 0 \tag{B.12a}$$

$$\tau_1 (y_2 - y_1) + \tau_2 (y_3 - y_2) + \tau_3 (y_4 - y_3) + \tau_4 (y_1 - y_4) = 0 \tag{B.12b}$$

$$\tau_1 (x_1 y_2 - x_2 y_1) + \tau_2 (x_2 y_3 - x_3 y_2) + \tau_3 (x_3 y_4 - x_4 y_3) + \tau_4 (x_4 y_1 - x_1 y_4) = 0 \tag{B.12c}$$

Define-se, então, a tensão cisalhante média (τ) que atua no interior do painel, na mesma direção das faces. Esta tensão é dada por:

$$\tau = \frac{1}{4} (-\tau_1 + \tau_2 - \tau_3 + \tau_4) \tag{B.13}$$

sendo que os sinais negativos de τ_1 e τ_3 indicam que a tensão na face está no sentido contrário à tensão média, ilustrada na Figura B.4.

As equações (B.12) e (B.13) podem ser escritas da seguinte forma matricial:

$$\begin{bmatrix} c_1 & c_2 & c_3 & c_4 \\ s_1 & s_2 & s_3 & s_4 \\ r_1 & r_2 & r_3 & r_4 \\ -1 & 1 & -1 & 1 \end{bmatrix} \begin{Bmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \end{Bmatrix} = \begin{Bmatrix} \cdot \\ \cdot \\ \cdot \\ 4\tau \end{Bmatrix} \tag{B.14}$$

sendo que os elementos da primeira matriz são:

$$\begin{aligned}
 c_1 &= x_2 - x_1 & s_1 &= y_2 - y_1 & r_1 &= x_1 y_2 - x_2 y_1 \\
 c_2 &= x_3 - x_2 & s_2 &= y_3 - y_2 & r_2 &= x_2 y_3 - x_3 y_2 \\
 c_3 &= x_4 - x_3 & s_3 &= y_4 - y_3 & r_3 &= x_3 y_4 - x_4 y_3 \\
 c_4 &= x_1 - x_4 & s_4 &= y_1 - y_4 & r_4 &= x_4 y_1 - x_1 y_4
 \end{aligned} \tag{B.15}$$

O valor de τ_1 pode ser obtido pela aplicação da regra de Cramer:

$$\tau_1 = \frac{D_\tau}{D} \tag{B.16}$$

com:

$$D_\tau = \begin{vmatrix} \cdot & c_2 & c_3 & c_4 \\ \cdot & s_2 & s_3 & s_4 \\ \cdot & r_2 & r_3 & r_4 \\ 4\tau & 1 & -1 & 1 \end{vmatrix} \quad D = \begin{vmatrix} c_1 & c_2 & c_3 & c_4 \\ s_1 & s_2 & s_3 & s_4 \\ r_1 & r_2 & r_3 & r_4 \\ -1 & 1 & -1 & 1 \end{vmatrix}$$

Os determinantes, por sua vez, podem ser encontrados aplicando o Teorema de Laplace para a quarta linha de cada matriz:

$$D_\tau = -4\tau k_1 \quad D = k_1 + k_2 + k_3 + k_4 \quad (\text{B.17})$$

sendo k_i os valores dos elementos a_{ij} já multiplicados pelos seus respectivos cofatores A_{ij} :

$$\begin{aligned} k_1 &= \begin{vmatrix} c_2 & c_3 & c_4 \\ s_2 & s_3 & s_4 \\ r_2 & r_3 & r_4 \end{vmatrix} & k_2 &= \begin{vmatrix} c_1 & c_3 & c_4 \\ s_1 & s_3 & s_4 \\ r_1 & r_3 & r_4 \end{vmatrix} \\ k_3 &= \begin{vmatrix} c_1 & c_2 & c_4 \\ s_1 & s_2 & s_4 \\ r_1 & r_2 & r_4 \end{vmatrix} & k_4 &= \begin{vmatrix} c_1 & c_2 & c_3 \\ s_1 & s_2 & s_3 \\ r_1 & r_2 & r_3 \end{vmatrix} \end{aligned} \quad (\text{B.18})$$

Considerando a força total distribuída ($f_1 = l_1 \tau_1 t$) no lado 1 do painel, tem-se:

$$f_1 = \frac{-4 k_1 l_1 t \tau}{k_f} \quad (\text{B.19})$$

Do mesmo modo, pode-se obter as forças f_2 , f_3 e f_4 , atuantes nos demais lados do painel:

$$f_2 = \frac{4 k_2 l_2 t \tau}{k_f} \quad f_3 = \frac{-4 k_3 l_3 t \tau}{k_f} \quad f_4 = \frac{4 k_4 l_4 t \tau}{k_f} \quad (\text{B.20})$$

sendo:

$$k_f = k_1 + k_2 + k_3 + k_4$$

Matricialmente, tem-se:

$$\{f\} = [B]^T \beta \quad (\text{B.21})$$

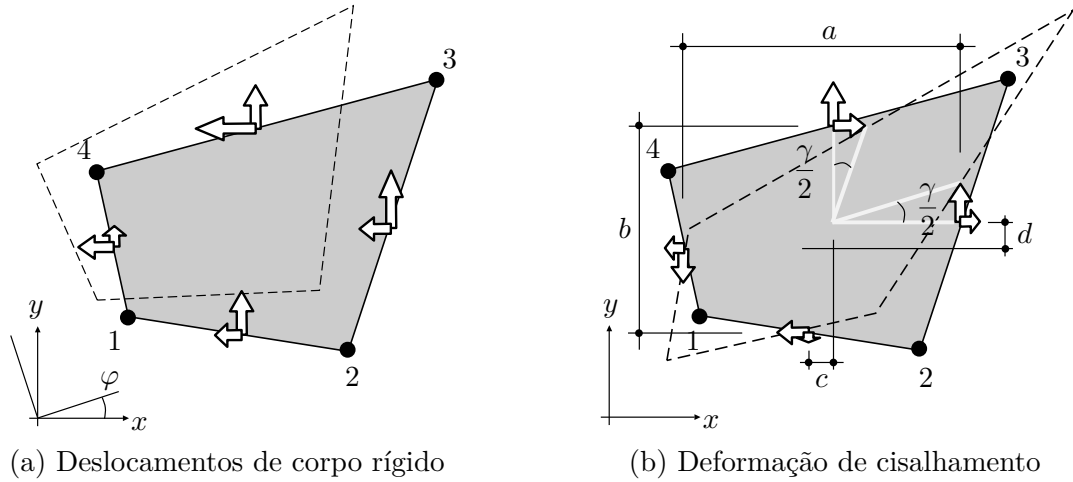
sendo a matriz $[B]^T$ e a tensão generalizada β dados por:

$$[B]^T = \begin{bmatrix} -k_1 l_1 \\ k_2 l_2 \\ -k_3 l_3 \\ k_4 l_4 \end{bmatrix} \quad \beta = \frac{4t}{k_f} \tau$$

B.1.4 Cinemática

O painel apresenta três movimentos de corpo rígido: uma translação v na direção x , uma translação w na direção y e uma rotação ϑ em torno da origem. Estes três movimentos são ilustrados na Figura B.5a.

Figura B.5 – Painel linear: deslocamentos



Fonte: adaptado de Hoogenboom (1998).

A translação v , na direção x , resulta no vetor de deslocamento $\{u_v\}$, assim como a translação w , em y , resulta no vetor de deslocamento $\{u_w\}$:

$$\{u_v\} = v \begin{Bmatrix} \cos(x, l_1) \\ \cos(x, l_2) \\ \cos(x, l_3) \\ \cos(x, l_4) \end{Bmatrix} \quad \{u_w\} = w \begin{Bmatrix} \sin(x, l_1) \\ \sin(x, l_2) \\ \sin(x, l_3) \\ \sin(x, l_4) \end{Bmatrix} \quad (\text{B.22})$$

A rotação ϑ , em torno da origem, resulta no vetor $\{u_\vartheta\}$:

$$\{u_\vartheta\} = \vartheta \begin{Bmatrix} \frac{x_2 + x_1}{2} \sin(x, l_1) - \frac{y_2 + y_1}{2} \cos(x, l_1) \\ \frac{x_3 + x_2}{2} \sin(x, l_2) - \frac{y_3 + y_2}{2} \cos(x, l_2) \\ \frac{x_4 + x_3}{2} \sin(x, l_3) - \frac{y_4 + y_3}{2} \cos(x, l_3) \\ \frac{x_1 + x_4}{2} \sin(x, l_4) - \frac{y_1 + y_4}{2} \cos(x, l_4) \end{Bmatrix} \quad (\text{B.23})$$

É importante salientar que estes deslocamentos são linearizados. Portanto, esta hipótese é válida somente para pequenas rotações, caso em que se enquadram as estruturas de concreto (HOOGENBOOM, 1998).

Utilizando as definições de cossenos e senos, em (B.11), e as variáveis c , s e r , definidas em (B.15), pode-se reescrever os deslocamentos $\{u_v\}$, $\{u_w\}$ e $\{u_\vartheta\}$ como:

$$\{u_v\} = v \begin{Bmatrix} \frac{c_1}{l_1} \\ \frac{c_2}{l_2} \\ \frac{c_3}{l_3} \\ \frac{c_4}{l_4} \end{Bmatrix} \quad \{u_w\} = w \begin{Bmatrix} \frac{s_1}{l_1} \\ \frac{s_2}{l_2} \\ \frac{s_3}{l_3} \\ \frac{s_4}{l_4} \end{Bmatrix} \quad \{u_\vartheta\} = \vartheta \begin{Bmatrix} \frac{r_1}{l_1} \\ \frac{r_2}{l_2} \\ \frac{r_3}{l_3} \\ \frac{r_4}{l_4} \end{Bmatrix} \quad (\text{B.24})$$

Considerando a deformação cisalhante homogênea no painel (γ) (Figura B.5b), têm-se como resultado o seguinte vetor de deslocamentos:

$$\{u_\gamma\} = \frac{\gamma}{2} \begin{Bmatrix} -\frac{b}{2} \cos(x, l_1) - \frac{c}{2} \sin(x, l_1) \\ \frac{a}{2} \sin(x, l_2) + \frac{d}{2} \cos(x, l_2) \\ \frac{b}{2} \cos(x, l_3) + \frac{c}{2} \sin(x, l_3) \\ -\frac{a}{2} \sin(x, l_4) - \frac{d}{2} \cos(x, l_4) \end{Bmatrix} \quad (\text{B.25})$$

sendo a , b , c e d as dimensões de referência do painel (Figura B.5b):

$$\begin{aligned} a &= \frac{x_2 + x_3}{2} - \frac{x_1 + x_4}{2} = \frac{c_1 - c_3}{2} & b &= \frac{y_3 + y_4}{2} - \frac{y_1 + y_2}{2} = \frac{s_2 - s_4}{2} \\ c &= \frac{x_3 + x_4}{2} - \frac{x_1 + x_2}{2} = \frac{c_2 - c_4}{2} & d &= \frac{y_2 + y_3}{2} - \frac{y_1 + y_4}{2} = \frac{s_1 - s_3}{2} \end{aligned} \quad (\text{B.26})$$

Introduzindo as variáveis:

$$\begin{aligned} t_1 &= -b c_1 - c s_1 & t_2 &= a s_2 + d c_2 \\ t_3 &= b c_3 + c s_3 & t_4 &= -a s_4 - d c_4 \end{aligned} \quad (\text{B.27})$$

O vetor $\{u_\gamma\}$ pode então ser simplificado por:

$$\{u_\gamma\} = \frac{\gamma}{4} \begin{Bmatrix} \frac{t_1}{l_1} \\ \frac{t_2}{l_2} \\ \frac{t_3}{l_3} \\ \frac{t_4}{l_4} \end{Bmatrix} \quad (\text{B.28})$$

A deformação γ será uma combinação dos deslocamentos dos lados do painel:

$$\gamma = u_1 p_1 + u_2 p_2 + u_3 p_3 + u_4 p_4 \quad (\text{B.29})$$

sendo p_i incógnitas do modo de deformação do painel.

Considerando os quatro vetores de deformação, definidos em (B.24) e (B.28), e a definição da deformação de cisalhamento (B.29), pode-se escrever, em forma matricial, o seguinte sistema de equações:

$$\begin{bmatrix} v \frac{c_1}{l_1} & v \frac{c_2}{l_2} & v \frac{c_3}{l_3} & v \frac{c_4}{l_4} \\ w \frac{s_1}{l_1} & w \frac{s_2}{l_2} & w \frac{s_3}{l_3} & w \frac{s_4}{l_4} \\ \vartheta \frac{r_1}{l_1} & \vartheta \frac{r_2}{l_2} & \vartheta \frac{r_3}{l_3} & \vartheta \frac{r_4}{l_4} \\ \frac{\gamma}{4} \frac{t_1}{l_1} & \frac{\gamma}{4} \frac{t_2}{l_2} & \frac{\gamma}{4} \frac{t_3}{l_3} & \frac{\gamma}{4} \frac{t_4}{l_4} \end{bmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \gamma \end{pmatrix} \quad (\text{B.30})$$

As três primeiras linhas da matriz se referem aos vetores dos deslocamentos de corpo rígido ($\{u_v\}$, $\{u_w\}$ e $\{u_\vartheta\}$), que não geram deformação no painel e, portanto, estão associados a zero no vetor da direita. A última linha corresponde à deformação γ , definida em (B.29). Este sistema pode ser escrito simplificadamente por:

$$\begin{bmatrix} c_1 & c_2 & c_3 & c_4 \\ s_1 & s_2 & s_3 & s_4 \\ r_1 & r_2 & r_3 & r_4 \\ t_1 & t_2 & t_3 & t_4 \end{bmatrix} \begin{pmatrix} \frac{p_1}{l_1} \\ \frac{p_2}{l_2} \\ \frac{p_3}{l_3} \\ \frac{p_4}{l_4} \end{pmatrix} = \begin{pmatrix} \cdot \\ \cdot \\ \cdot \\ 4 \end{pmatrix} \quad (\text{B.31})$$

O valor de p_1 pode ser obtido pela aplicação da regra de Cramer:

$$\frac{p_1}{l_1} = \frac{D_\gamma}{D} \quad (\text{B.32})$$

com:

$$D_\gamma = \begin{vmatrix} \cdot & c_2 & c_3 & c_4 \\ \cdot & s_2 & s_3 & s_4 \\ \cdot & r_2 & r_3 & r_4 \\ 4 & t_2 & t_3 & t_4 \end{vmatrix} \quad D = \begin{vmatrix} c_1 & c_2 & c_3 & c_4 \\ s_1 & s_2 & s_3 & s_4 \\ r_1 & r_2 & r_3 & r_4 \\ t_1 & t_2 & t_3 & t_4 \end{vmatrix}$$

Os determinantes, por sua vez, podem ser encontrados aplicando o Teorema de Laplace para a primeira coluna da matriz D_γ e para a quarta linha da matriz D :

$$D_\gamma = -4 k_1 \quad D = -t_1 k_1 + t_2 k_2 - t_3 k_3 + t_4 k_4 \quad (\text{B.33})$$

sendo k_i os determinantes definidos em (B.18).

Portanto, p_1 é calculado por:

$$p_1 = \frac{-4 k_1 l_1}{k_u} \quad (\text{B.34})$$

com:

$$k_u = D = -t_1 k_1 + t_2 k_2 - t_3 k_3 + t_4 k_4$$

Do mesmo modo, pode-se chegar aos valores de p_2 , p_3 e p_4 :

$$p_2 = \frac{4 k_2 l_2}{k_u} \quad p_3 = \frac{-4 k_3 l_3}{k_u} \quad p_4 = \frac{4 k_4 l_4}{k_u} \quad (\text{B.35})$$

Considerando a relação para a deformação γ (B.29) e a matriz $[B]$ (B.21), tem-se:

$$\gamma = \frac{4}{k_u} e \quad (\text{B.36})$$

com:

$$e = [B] \{u\} \quad [B] = \begin{bmatrix} -k_1 l_1 & k_2 l_2 & -k_3 l_3 & k_4 l_4 \end{bmatrix}$$

B.1.5 Matriz de rigidez

Considerando as relações de equilíbrio:

$$\{f\} = [B]^T \beta, \quad \beta = \frac{4t}{k_f} \tau,$$

as relações da cinemática:

$$\gamma = \frac{4}{k_u} e, \quad e = [B] \{u\},$$

a relação tensão-deformação:

$$\tau = G \gamma,$$

e sabendo que:

$$\{f\} = [K_l] \{u\}, \quad [K_l] = [B]^T D [B],$$

obtem-se o valor da relação constitutiva D por (HOOGENBOOM; BLAAUWENDRAAD, 2000):

$$D = \frac{\beta}{e} = \frac{16 G t}{k_f k_u} \quad (\text{B.37})$$

Isso resulta na matriz de rigidez local $[K_l]$, simétrica e de quarta ordem. Após a transformação, a partir de (B.8), tem-se a matriz global $[K]$, de ordem 8.

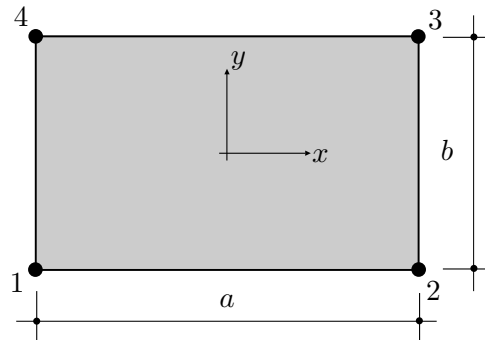
B.1.6 Painel retangular

Na prática, na maioria dos problemas das estruturas de concreto, os painéis têm forma retangular. Nesse caso, o modelo pode ser simplificado, de forma a exigir menor esforço computacional. A geometria do painel retangular é ilustrada na Figura B.6.

Observa-se que a geometria é dependente somente das duas dimensões (a e b). Considerando as definições das constantes c , s e r (B.15), pode-se escrever a relação de equilíbrio simplificada:

$$\begin{bmatrix} a & \cdot & -a & \cdot \\ \cdot & b & \cdot & -b \\ \frac{1}{2}ab & \frac{1}{2}ab & \frac{1}{2}ab & \frac{1}{2}ab \\ -1 & 1 & -1 & 1 \end{bmatrix} \begin{Bmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \end{Bmatrix} = \begin{Bmatrix} \cdot \\ \cdot \\ \cdot \\ 4\tau \end{Bmatrix} \quad (\text{B.38})$$

Figura B.6 – Geometria de um painel retangular



Fonte: adaptado de Hoogenboom (1998).

Os valores dos coeficientes k_i são simplificados para

$$\begin{aligned} k_1 &= \begin{vmatrix} \cdot & -a & \cdot \\ b & \cdot & -b \\ \frac{1}{2}ab & \frac{1}{2}ab & \frac{1}{2}ab \end{vmatrix} & k_2 &= \begin{vmatrix} a & -a & \cdot \\ \cdot & \cdot & -b \\ \frac{1}{2}ab & \frac{1}{2}ab & \frac{1}{2}ab \end{vmatrix} \\ k_3 &= \begin{vmatrix} a & \cdot & \cdot \\ \cdot & b & -b \\ \frac{1}{2}ab & \frac{1}{2}ab & \frac{1}{2}ab \end{vmatrix} & k_4 &= \begin{vmatrix} a & \cdot & -a \\ \cdot & b & \cdot \\ \frac{1}{2}ab & \frac{1}{2}ab & \frac{1}{2}ab \end{vmatrix} \end{aligned} \quad (\text{B.39})$$

Portanto:

$$k_1 = k_2 = k_3 = k_4 = a^2 b^2 \quad (\text{B.40})$$

Os valores dos coeficientes t_i (B.27) são:

$$t_1 = t_3 = -a b \quad t_2 = t_4 = a b \quad (\text{B.41})$$

Portanto,

$$k_f = 4 a^2 b^2, \quad k_u = 4 a^3 b^3, \quad D = \frac{G t}{a^5 b^5} \quad (\text{B.42})$$

A matriz $[B]$ será:

$$[B] = \begin{bmatrix} -a^3 b^2 & a^2 b^3 & -a^3 b^2 & a^2 b^3 \end{bmatrix} \quad (\text{B.43})$$

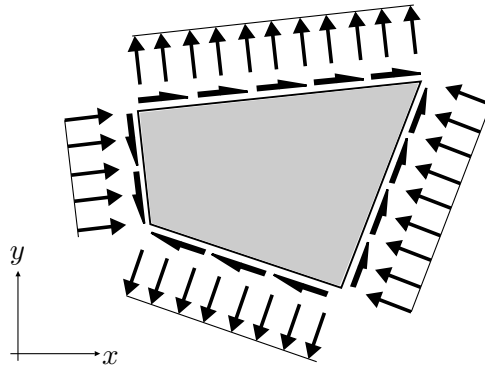
e a matriz de rigidez resulta em:

$$[K] = [B]^T D [B] = G t \begin{bmatrix} \frac{a}{b} & -1 & \frac{a}{b} & -1 \\ -1 & \frac{b}{a} & -1 & \frac{b}{a} \\ \frac{a}{b} & -1 & \frac{a}{b} & -1 \\ -1 & \frac{b}{a} & -1 & \frac{b}{a} \end{bmatrix} \quad (\text{B.44})$$

B.2 Modelo não-linear

Na modelagem não-linear do sistema, os painéis não apenas comportam esforços cisalhantes em suas faces, mas também esforços normais (Figura B.7). Durante a análise não-linear, as forças nos painéis são computadas a cada passo de carga e a cada iteração de equilíbrio.

Figura B.7 – Painel: modelo não-linear com esforços normais



Fonte: adaptado de Hoogenboom (1998).

De acordo com Hoogenboom (1998), os esforços normais foram incluídos no modelo por haver contradição dos resultados do programa quando comparados a resultados experimentais, caso fossem considerados apenas esforços cisalhantes. Além disso, a existência de

armadura distribuída contribui tanto para a rigidez ao cisalhamento quanto para a rigidez ao esforço normal. Assim, o painel tem a possibilidade de se deformar independentemente, carregando também tensões normais.

B.2.1 Relações

O painel deve ser capaz de transmitir uma tensão normal de uma de suas faces para as outras como uma tensão cisalhante. Assim, as tensões no painel serão no mínimo lineares em x e y (σ_{xx} e σ_{yy}). É utilizado o seguinte campo de tensões:

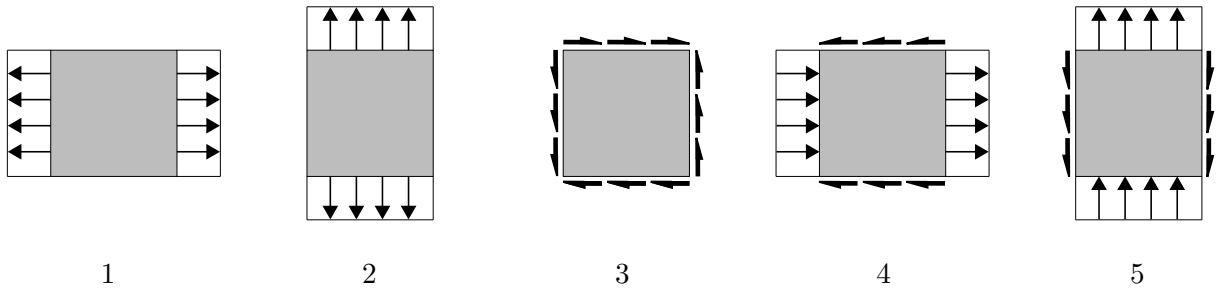
$$\begin{Bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{Bmatrix} = \begin{bmatrix} 1 & \cdot & \cdot & \frac{2x}{a} & \cdot \\ \cdot & 1 & \cdot & \cdot & \frac{2y}{b} \\ \cdot & \cdot & 1 & -\frac{2y}{a} & -\frac{2x}{b} \end{bmatrix} \begin{Bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \beta_4 \\ \beta_5 \end{Bmatrix} \quad (\text{B.45})$$

sendo a e b as dimensões do painel (Figura B.10a).

Vale salientar que, na matriz acima e também nas subsequentes, os zeros são substituídos por pontos para melhor visualização.

Cada uma das tensões generalizadas (β_1 a β_5) representa um dos modos de tensão no painel, apresentados na Figura B.8.

Figura B.8 – Modos independentes do campo de tensões do painel



Fonte: adaptado de Hoogenboom (1998).

Este campo de tensões satisfaz as condições de equilíbrio, dadas por:

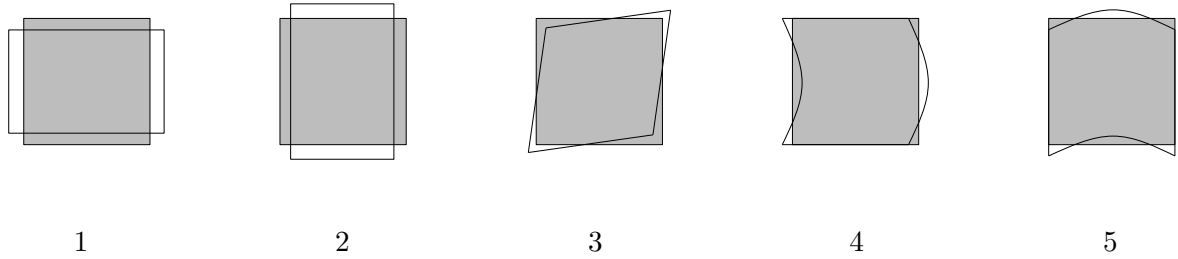
$$\frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{xy}}{\partial y} = 0 \quad \frac{\partial \sigma_{yy}}{\partial y} + \frac{\partial \sigma_{xy}}{\partial x} = 0 \quad (\text{B.46})$$

Um campo de deformações consistente com o campo de tensões é dado por:

$$\begin{Bmatrix} u_x \\ u_y \end{Bmatrix} = \begin{bmatrix} x & \cdot & y & \frac{x^2 - 2y^2}{a} & \cdot & 1 & \cdot & y \\ \cdot & y & x & \cdot & \frac{y^2 - 2x^2}{b} & \cdot & 1 & -x \end{bmatrix} \begin{Bmatrix} e_1 \\ e_2 \\ \vdots \\ e_8 \end{Bmatrix} \quad (\text{B.47})$$

sendo que as deformações generalizadas e_1 a e_5 representam os modos de deformação do painel (Figura B.9), e_6 e e_7 , os deslocamentos e e_8 a rotação de corpo rígido.

Figura B.9 – Modos de deformação do painel em concordância com os modos de tensão



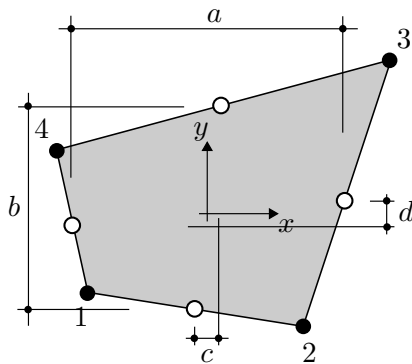
Fonte: adaptado de Hoogenboom (1998).

A Figura B.10a apresenta as dimensões do painel, que são calculadas a partir das coordenadas dos vértices 1 a 4. A Figura B.10b apresenta os nós do painel, que também são numerados de 1 a 4, e seus respectivos graus de liberdade (u_i).

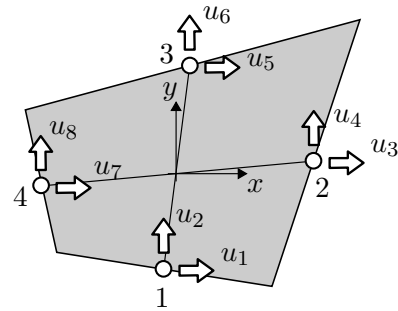
Considerando que o sistema de coordenadas tenha origem no ponto central do painel, as posições nodais serão:

- Nó 1: $(-c/2, -b/2)$;
- Nó 2: $(a/2, d/2)$;
- Nó 3: $(c/2, b/2)$;
- Nó 4: $(-a/2, -d/2)$;

Figura B.10 – Painel não-linear



(a) Vértices e dimensões



(b) Graus de liberdade

Fonte: adaptado de Hoogenboom (1998).

B.2.2 Deformações generalizadas

Substituindo as posições nodais no campo de deformações (B.47), tem-se a solução dos deslocamentos nodais:

$$\begin{Bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \end{Bmatrix} = \begin{bmatrix} -\frac{c}{2} & \cdot & -\frac{b}{2} & \frac{c^2 - 2b^2}{4a} & \cdot & 1 & \cdot & -\frac{b}{2} \\ \cdot & -\frac{b}{2} & -\frac{c}{2} & \cdot & \frac{b^2 - 2c^2}{4b} & \cdot & 1 & \frac{c}{2} \\ \frac{a}{2} & \cdot & \frac{d}{2} & \frac{a^2 - 2d^2}{4a} & \cdot & 1 & \cdot & \frac{d}{2} \\ \cdot & \frac{d}{2} & \frac{a}{2} & \cdot & \frac{d^2 - 2a^2}{4b} & \cdot & 1 & -\frac{a}{2} \\ \frac{c}{2} & \cdot & \frac{b}{2} & \frac{c^2 - 2b^2}{4a} & \cdot & 1 & \cdot & \frac{b}{2} \\ \frac{a}{2} & \cdot & \frac{d}{2} & \frac{a^2 - 2d^2}{4a} & \cdot & 1 & \cdot & \frac{d}{2} \\ \cdot & \frac{b}{2} & \frac{c}{2} & \cdot & \frac{b^2 - 2c^2}{4b} & \cdot & 1 & -\frac{c}{2} \\ -\frac{a}{2} & \cdot & -\frac{d}{2} & \frac{a^2 - 2d^2}{4a} & \cdot & 1 & \cdot & -\frac{d}{2} \\ \cdot & -\frac{d}{2} & -\frac{a}{2} & \cdot & \frac{d^2 - 2a^2}{4b} & \cdot & 1 & \frac{a}{2} \end{bmatrix} \begin{Bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \\ e_8 \end{Bmatrix} \quad (\text{B.48})$$

As deformações generalizadas (e_1 a e_5), de interesse, são obtidas pela inversão da matriz acima, removendo as linhas correspondentes a e_6 , e_7 e e_8 :

$$\begin{Bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \end{Bmatrix} = \begin{bmatrix} \frac{d}{t_1} & \cdot & \frac{b}{t_1} & \cdot & -\frac{d}{t_1} & \cdot & -\frac{b}{t_1} & \cdot \\ \cdot & -\frac{a}{t_1} & \cdot & -\frac{c}{t_1} & \cdot & \frac{a}{t_1} & \cdot & \frac{c}{t_1} \\ -\frac{a}{2t_1} & \frac{d}{2t_1} & -\frac{c}{2t_1} & \frac{b}{2t_1} & \frac{a}{2t_1} & -\frac{d}{2t_1} & \frac{c}{2t_1} & -\frac{b}{2t_1} \\ -\frac{a}{t_2} & \cdot & \frac{a}{t_2} & \cdot & -\frac{a}{t_2} & \cdot & \frac{a}{t_2} & \cdot \\ \cdot & \frac{b}{t_3} & \cdot & -\frac{b}{t_3} & \cdot & \frac{b}{t_3} & \cdot & -\frac{b}{t_3} \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \end{Bmatrix} \quad (\text{B.49})$$

ou:

$$\{e\} = [A] \{u\} \quad (\text{B.50})$$

com:

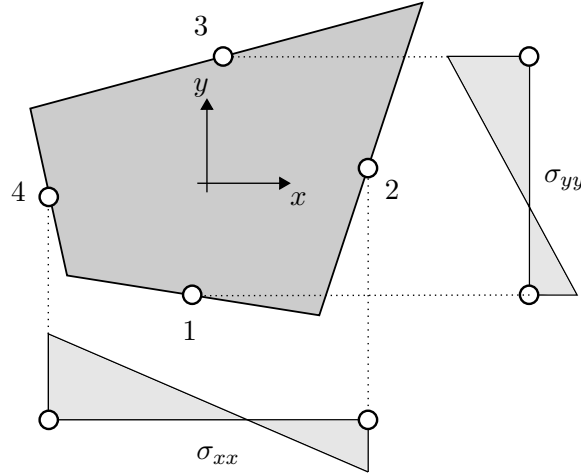
$$t_1 = ab - cd \quad t_2 = \frac{1}{2}(a^2 - c^2) + b^2 - d^2 \quad t_3 = \frac{1}{2}(b^2 - d^2) + a^2 - c^2$$

B.2.3 Deformações nos pontos de integração

O modelo do painel considera quatro pontos de integração, com posições coincidentes com os nós, isto é, no ponto médio de cada face. Esta posição foi escolhida por Hoogenboom

(1998) devido ao campo de tensão em x e y ser linear. Dessa forma, as tensões mínimas e máximas se localizarão nas bordas do painel, como ilustrado na Figura B.11.

Figura B.11 – Pontos de integração do painel e distribuição linear de tensões em x e y



Fonte: adaptado de Hoogenboom (1998).

As deformações são derivadas do campo de deslocamentos:

$$\varepsilon_{xx} = \frac{\partial u_x}{\partial x} \quad \varepsilon_{yy} = \frac{\partial u_y}{\partial y} \quad \gamma_{xy} = \frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x} \quad (\text{B.51})$$

sendo válidas apenas para pequenas deformações. Assim, este comportamento apenas é preciso para pequenas rotações, caso válido para as estruturas de concreto.

Utilizando a definição do campo de deslocamentos (B.47), obtém-se o campo de deformações do painel:

$$\begin{Bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ \gamma_{xy} \end{Bmatrix} = \begin{bmatrix} 1 & \cdot & \cdot & \frac{2x}{a} & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \frac{2y}{b} & \cdot & \cdot & \cdot \\ \cdot & \cdot & 2 & -\frac{4y}{a} & -\frac{4x}{b} & \cdot & \cdot & \cdot \end{bmatrix} \begin{Bmatrix} e_1 \\ e_2 \\ \vdots \\ e_8 \end{Bmatrix} \quad (\text{B.52a})$$

Uma relação reduzida será utilizada por apresentar melhores resultados em comparação com resultados experimentais (HOOGENBOOM, 1998):

$$\begin{Bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ \gamma_{xy} \end{Bmatrix} = \begin{bmatrix} 1 & \cdot & \cdot & \frac{2x}{a} & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \frac{2y}{b} & \cdot & \cdot & \cdot \\ \cdot & \cdot & 2 & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix} \begin{Bmatrix} e_1 \\ e_2 \\ \vdots \\ e_8 \end{Bmatrix} \quad (\text{B.52b})$$

Dessa forma, a deformação por cisalhamento no painel será constante e igual ao valor interno.

Substituindo as coordenadas dos pontos de integração em (B.52b), têm-se as deformações em cada ponto de integração:

$$\begin{Bmatrix} \varepsilon_{xx1} \\ \varepsilon_{yy1} \\ \gamma_{xy1} \\ \varepsilon_{xx2} \\ \varepsilon_{yy2} \\ \gamma_{xy2} \\ \varepsilon_{xx3} \\ \varepsilon_{yy3} \\ \gamma_{xy3} \\ \varepsilon_{xx4} \\ \varepsilon_{yy4} \\ \gamma_{xy4} \end{Bmatrix} = \begin{bmatrix} 1 & \cdot & \cdot & -\frac{c}{a} & \cdot \\ \cdot & 1 & \cdot & \cdot & -1 \\ \cdot & \cdot & 2 & \cdot & \cdot \\ 1 & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \frac{d}{b} \\ \cdot & \cdot & 2 & \cdot & \cdot \\ 1 & \cdot & \cdot & \frac{c}{a} & \cdot \\ \cdot & 1 & \cdot & \cdot & 1 \\ \cdot & \cdot & 2 & \cdot & \cdot \\ 1 & \cdot & \cdot & -1 & \cdot \\ \cdot & 1 & \cdot & \cdot & -\frac{d}{b} \\ \cdot & \cdot & 2 & \cdot & \cdot \end{bmatrix} \begin{Bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \end{Bmatrix} \quad (\text{B.53a})$$

ou:

$$\{\varepsilon\} = [B] \{e\} \quad (\text{B.53b})$$

B.2.4 Tensões nos pontos de integração

As deformações nos pontos de integração são convertidas para tensões utilizando a TMCC (VECCHIO; COLLINS, 1986). Primeiramente, são calculadas as deformações principais e, a partir das relações constitutivas, são calculadas as tensões principais no concreto. Estas são transformadas seguindo as direções do painel e adicionadas às tensões na armadura. Por fim, as tensões são verificadas e corrigidas de forma que estejam em equilíbrio nas fissuras.

Assim, as tensões médias no concreto (σ_{xyc} , σ_{yyc} e σ_{xyc}) e as tensões médias no aço (σ_{xcs} e σ_{yys}) são calculadas nos pontos de integração. Estas tensões são adicionadas de modo a obter a tensão média no painel:

$$\sigma_{xx} = \sigma_{xyc} + \sigma_{xcs} \quad \sigma_{yy} = \sigma_{yyc} + \sigma_{yys} \quad \sigma_{xy} = \sigma_{xyc} \quad (\text{B.54})$$

Devido ao comportamento não-linear, a matriz de rigidez tangente do painel não será simétrica. Como consequência, essa matriz não pode ser obtida analiticamente pois a TMCC encontra o equilíbrio por meio de iterações.

B.2.5 Tensões Generalizadas

Nesta seção, as doze componentes de tensão (três em cada ponto de integração) são reduzidas a um campo de tensões descrito pelas tensões generalizadas β_1 a β_5 . Substituindo $x = y = 0$ no campo de tensões do painel (B.45), obtêm-se as tensões no ponto médio do painel:

$$\sigma_{xxm} = \beta_1 \quad \sigma_{yy m} = \beta_2 \quad \sigma_{xym} = \beta_3 \quad (\text{B.55})$$

Substituindo as coordenadas dos pontos de integração do painel em (B.45), tem-se:

$$\begin{Bmatrix} \sigma_{xx1} \\ \sigma_{yy1} \\ \sigma_{xy1} \\ \sigma_{xx2} \\ \sigma_{yy2} \\ \sigma_{xy2} \\ \sigma_{xx3} \\ \sigma_{yy3} \\ \sigma_{xy3} \\ \sigma_{xx4} \\ \sigma_{yy4} \\ \sigma_{xy4} \end{Bmatrix} = \begin{bmatrix} 1 & \cdot & \cdot & -\frac{c}{a} & \cdot \\ \cdot & 1 & \cdot & \cdot & -1 \\ \cdot & \cdot & 1 & \frac{b}{a} & \frac{c}{b} \\ 1 & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \frac{d}{b} \\ \cdot & \cdot & 1 & -\frac{d}{a} & -\frac{a}{b} \\ 1 & \cdot & \cdot & \frac{c}{a} & \cdot \\ \cdot & 1 & \cdot & \cdot & 1 \\ \cdot & \cdot & 1 & -\frac{b}{a} & -\frac{c}{b} \\ 1 & \cdot & \cdot & -1 & \cdot \\ \cdot & 1 & \cdot & \cdot & -\frac{d}{b} \\ \cdot & \cdot & 1 & \frac{d}{a} & \frac{a}{b} \end{bmatrix} \begin{Bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \beta_4 \\ \beta_5 \end{Bmatrix} \quad (\text{B.56a})$$

que também pode ser escrita como:

$$\{ \sigma \} = [L] \{ \beta \} \quad (\text{B.56b})$$

Para a solução de $\{ \beta \}$ não é possível inverter $[L]$ por esta não ser uma matriz quadrada. A solução adotada por Hoogenboom (1998) foi utilizar o Método dos Mínimos Quadrados para encontrar uma melhor solução para as tensões nos pontos de integração:

$$\beta = ([L]^T [L])^{-1} [L]^T \{ \sigma \} \quad (\text{B.57})$$

Para o caso mais comum, de painel retangular ($c = d = 0$), a solução é simplificada para:

$$\begin{Bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ 2 \frac{a^2 + b^2}{a^2} \beta_4 \\ 2 \frac{a^2 + b^2}{a^2} \beta_5 \end{Bmatrix} = \frac{1}{4} \begin{bmatrix} 1 & \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & 1 \\ \cdot & \cdot & \frac{b}{a} & 1 & \cdot & \cdot & \cdot & \cdot & -\frac{b}{a} & -1 & \cdot & \cdot \\ \cdot & -1 & \cdot & \cdot & \cdot & -\frac{a}{b} & \cdot & 1 & \cdot & \cdot & \cdot & \frac{a}{b} \end{bmatrix} \begin{Bmatrix} \sigma_{xx1} \\ \sigma_{yy1} \\ \sigma_{xy1} \\ \vdots \\ \sigma_{xy4} \end{Bmatrix} \quad (\text{B.58})$$

Portanto, as tensões ponto médio do painel são as médias das tensões nos pontos de integração:

$$\sigma_{xxm} = \beta_1 = \frac{1}{4}(\sigma_{xx1} + \sigma_{xx2} + \sigma_{xx3} + \sigma_{xx4}) \quad (\text{B.59a})$$

$$\sigma_{yy m} = \beta_2 = \frac{1}{4}(\sigma_{yy1} + \sigma_{yy2} + \sigma_{yy3} + \sigma_{yy4}) \quad (\text{B.59b})$$

$$\sigma_{xym} = \beta_3 = \frac{1}{4}(\sigma_{xy1} + \sigma_{xy2} + \sigma_{xy3} + \sigma_{xy4}) \quad (\text{B.59c})$$

As tensões principais são encontradas fazendo:

$$\sigma_1 = s_1 + s_2 \quad \sigma_2 = s_1 - s_2 \quad \alpha = \frac{\pi}{2} - \arctg\left(\frac{\beta_1 - \sigma_2}{\beta_3}\right) \quad (\text{B.60})$$

sendo: σ_1 : tensão principal máxima;
 σ_2 : tensão principal mínima;
 α : ângulo entre a direção de σ_1 e o eixo x .

e:

$$s_1 = \frac{1}{2}(\beta_1 + \beta_2) \quad s_2 = \sqrt{\frac{1}{4}(\beta_1 - \beta_2)^2 + \beta_3^2}$$

B.2.6 Forças nodais

Pelas tensões generalizadas, pode-se calcular as forças nodais (f_i) por trabalho virtual. A Figura B.12b apresenta as forças nodais do painel, que são calculadas a partir das tensões nos pontos de integração. Nesta etapa, como o material constituinte do painel e das bielas adjacentes se sobrepõem, parte das tensões já estão inseridas nas bielas e não podem ser contabilizadas novamente para o painel. A Figura B.12a ilustra este comportamento, sendo as larguras das bielas representadas por c_i .

A Figura B.13 apresenta as relações de dimensões entre o painel e as bielas adjacentes. É possível observar que as relações serão exatas somente para o caso de painel retangular. Para painéis de outra geometria as relações apresentam uma aproximação satisfatória para as larguras das bielas (c_i) (HOOGENBOOM, 1998).

Já que o campo de tensões é linear em todas as direções, as forças nodais podem ser encontradas multiplicando as tensões nos pontos de integração pelo respectivo comprimento projetado da face:

$$\begin{aligned}
\frac{\tilde{f}_1}{t} &= -\sigma_{xx1}(y_1 - y_2) - \sigma_{xy1}(x_2 - x_1) \\
\frac{f_2}{t} &= -\sigma_{yy1c}(x_2 - x_1 - c_2 - c_4) - \sigma_{yy1s}(x_2 - x_1) - \sigma_{xy1}(y_1 - y_2) \\
\frac{f_3}{t} &= \sigma_{xx2c}(y_3 - y_2 - c_3 - c_1) + \sigma_{xx2s}(y_3 - y_2) - \sigma_{xy2}(x_3 - x_2) \\
\frac{\tilde{f}_4}{t} &= -\sigma_{yy2}(x_3 - x_2) + \sigma_{xy2}(y_3 - y_2) \\
\frac{\tilde{f}_5}{t} &= -\sigma_{xx3}(y_3 - y_4) + \sigma_{xy3}(x_3 - x_4) \\
\frac{f_6}{t} &= \sigma_{yy3c}(x_3 - x_4 - c_2 - c_4) + \sigma_{yy3s}(x_3 - x_4) - \sigma_{xy3}(y_3 - y_4) \\
\frac{f_7}{t} &= -\sigma_{xx4c}(y_4 - y_1 - c_1 - c_3) - \sigma_{xx4s}(y_4 - y_1) - \sigma_{xy4}(x_1 - x_4) \\
\frac{\tilde{f}_8}{t} &= -\sigma_{yy4}(x_1 - x_4) - \sigma_{xy4}(y_4 - y_1)
\end{aligned} \tag{B.61}$$

sendo que as forças destacadas com um til são aquelas relacionadas principalmente ao cisalhamento e necessitam de uma correção, pois não estão em equilíbrio.

Segundo Hoogenboom (1998), as forças nodais descritas acima não estão em equilíbrio por três motivos:

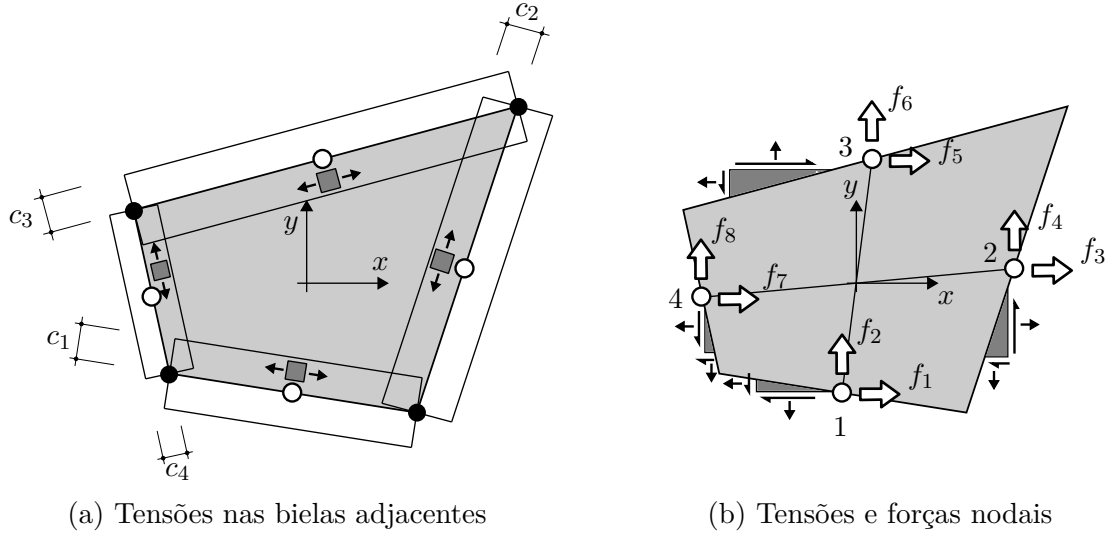
- (i) As forças nodais são calculadas a partir das tensões nos pontos de integração e não das tensões generalizadas;
- (ii) O campo de tensões linear nas faces também gera um momento nodal, que não é considerado;
- (iii) Uma parte das tensões não é considerada pois foi considerada nas bielas adjacentes.

De forma a minimizar o erro, a correção das forças f_1 , f_4 , f_5 e f_8 é feita pelo Método dos Mínimos Quadrados, através da seguinte função:

$$G' = (f_1 - \tilde{f}_1)^2 + (f_4 - \tilde{f}_4)^2 + (f_5 - \tilde{f}_5)^2 + (f_8 - \tilde{f}_8)^2 \tag{B.62}$$

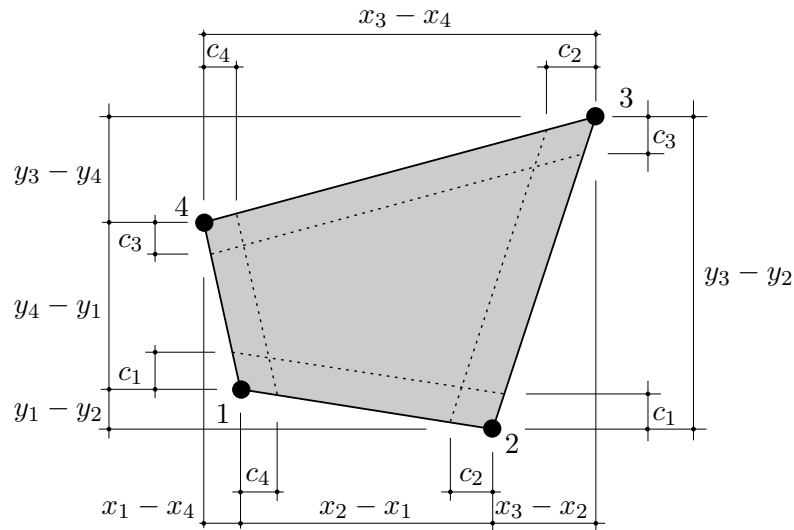
sendo esta sujeita às restrições dadas pelas equações de equilíbrio.

Figura B.12 – Pannel não-linear: tensões e forças nodais



Fonte: adaptado de Hoogenboom (1998).

Figura B.13 – Dimensões relativas entre o pannel e as bielas adjacentes



Fonte: adaptado de Hoogenboom (1998).

O equilíbrio do pannel, em x e y , é dado por:

$$\sum F_x = f_1 + f_3 + f_5 + f_7 = 0 \quad (\text{B.63a})$$

$$\sum F_y = f_2 + f_4 + f_6 + f_8 = 0 \quad (\text{B.63b})$$

O equilíbrio de momentos em torno da origem é dado por:

$$\sum M_O = f_1 \frac{b}{2} - f_2 \frac{c}{2} - f_3 \frac{d}{2} + f_4 \frac{a}{2} - f_5 \frac{b}{2} + f_6 \frac{c}{2} + f_7 \frac{d}{2} - f_8 \frac{a}{2} = 0 \quad (\text{B.63c})$$

A equações de equilíbrio (B.63) são adicionadas à função G' (B.62) utilizando multiplicadores de Lagrange:

$$\begin{aligned} G = & (f_1 - \tilde{f}_1)^2 + (f_4 - \tilde{f}_4)^2 + (f_5 - \tilde{f}_5)^2 + (f_8 - \tilde{f}_8)^2 + \\ & \lambda_1 (f_1 + f_3 + f_5 + f_7) + \lambda_2 (f_2 + f_4 + f_6 + f_8) + \\ & \lambda_3 [(f_1 - f_5)b + (f_6 - f_2)c + (f_7 - f_3)d + (f_4 - f_8)a] \end{aligned} \quad (\text{B.64})$$

Derivando a função G em relação às forças e aos multiplicadores de Lagrange, tem-se:

$$\begin{aligned} \frac{\partial G}{\partial f_1} = 2(f_1 - \tilde{f}_1) + \lambda_1 + \lambda_3 b = 0 & \quad \frac{\partial G}{\partial f_4} = 2(f_4 - \tilde{f}_4) + \lambda_2 + \lambda_3 a = 0 \\ \frac{\partial G}{\partial f_5} = 2(f_5 - \tilde{f}_5) + \lambda_1 - \lambda_3 b = 0 & \quad \frac{\partial G}{\partial f_8} = 2(f_8 - \tilde{f}_8) + \lambda_2 - \lambda_3 a = 0 \\ \frac{\partial G}{\partial \lambda_1} = f_1 + f_3 + f_5 + f_7 = 0 & \quad \frac{\partial G}{\partial \lambda_2} = f_2 + f_4 + f_6 + f_8 = 0 \\ \frac{\partial G}{\partial \lambda_3} = (f_1 - f_5)b + (f_6 - f_2)c + (f_7 - f_3)d + (f_4 - f_8)a = 0 \end{aligned} \quad (\text{B.65})$$

Resolvendo este conjunto de equações, tem-se a correção das forças nodais:

$$\begin{Bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ f_8 \end{Bmatrix} = \frac{1}{2t_4} \begin{bmatrix} a^2 & bc & bd - t_4 & -ab & -a^2 & -bc & -bd - t_4 & ab \\ \cdot & 2t_4 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 2t_4 & \cdot & \cdot & \cdot & \cdot & \cdot \\ -ab & ac - t_4 & ad & b^2 & ab & -ac - t_4 & -ad & -b^2 \\ -a^2 & -bc & -bd - t_4 & ab & a^2 & bc & bd - t_4 & -ab \\ \cdot & \cdot & \cdot & \cdot & \cdot & 2t_4 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 2t_4 & \cdot \\ ab & -ac - t_4 & -ad & -b^2 & -ab & ac - t_4 & ad & b^2 \end{bmatrix} \begin{Bmatrix} \tilde{f}_1 \\ \tilde{f}_2 \\ \tilde{f}_3 \\ \tilde{f}_4 \\ \tilde{f}_5 \\ \tilde{f}_6 \\ \tilde{f}_7 \\ \tilde{f}_8 \end{Bmatrix}$$

com $t_4 = a^2 + b^2$.

A matriz acima será chamada de $[Q]$, portanto:

$$\{f\} = [Q] \{\tilde{f}\} \quad (\text{B.66})$$

APÊNDICE C – Código-fonte das bibliotecas desenvolvidas

Este apêndice apresenta o código-fonte das principais bibliotecas desenvolvidas neste trabalho. É importante salientar que se optou por descrever apenas o código-fonte das bibliotecas de cálculo neste apêndice, já que o *SPMTool* se trata basicamente de comandos usados para desenho dos elementos no *AutoCAD*. Este código está disponível no repositório do autor (MELLO, 2021c).

C.1 *Material*

A biblioteca *Material* (MELLO, 2020a) implementa os objetos que constituem os materiais, concreto e armaduras. É dividida em: estruturas de parâmetros dos materiais e classes de materiais uni e biaxiais.

C.1.1 Parâmetros dos materiais

C.1.1.1 Estrutura *Parameters*

Constitui os parâmetros do concreto, calculados a partir dos modelos implementados, pela classe de cálculo *ParameterCalculator*.

Parameters.cs

```
namespace andrefmello91.Material.Concrete;

/// <summary>
///     Concrete parameters struct.
/// </summary>
public struct Parameters : IConcreteParameters, ICloneable<Parameters>
{
    /// <summary>
    ///     The default <see cref="Pressure" /> tolerance.
    /// </summary>
    public static readonly Pressure Tolerance = Pressure.FromPascals(1E-3);

    private Length _aggDiameter;

    /// <summary>
    ///     Calculator for concrete parameters.
    /// </summary>
    private ParameterCalculator _calculator;
```

```

private Pressure _strength;

/// <inheritdoc />
public Length AggregateDiameter
{
    get => _aggDiameter;
    set => _aggDiameter = value.ToUnit(DiameterUnit);
}

/// <inheritdoc />
public Pressure CompressiveStrength => Strength;

/// <inheritdoc />
public bool ConsiderConfinement { get; set; }

/// <inheritdoc />
public double CrackingStrain => TensileStrength / ElasticModule;

/// <inheritdoc />
public LengthUnit DiameterUnit
{
    get => AggregateDiameter.Unit;
    set => ChangeUnit(value);
}

/// <inheritdoc />
public Pressure ElasticModule => _calculator.ElasticModule.ToUnit(StressUnit);

/// <inheritdoc />
public ForcePerLength FractureParameter => _calculator.FractureParameter;

/// <inheritdoc />
public ParameterModel Model
{
    get => _calculator.Model;
    set
    {
        if (value == _calculator.Model)
            return;

        // Change model
        _calculator = ParameterCalculator.GetCalculator(Strength, value, Type);
    }
}

/// <inheritdoc />
/// <remarks>
///     Negative value.
/// </remarks>
public double PlasticStrain => _calculator.PlasticStrain;

/// <inheritdoc />
public Pressure SecantModule => _calculator.SecantModule.ToUnit(StressUnit);

/// <inheritdoc />
public Pressure Strength
{
    get => _strength;
    set

```

```

    {
        _strength          = value.ToUnit(StressUnit);
        _calculator.Strength = _strength;
    }
}

///  

public PressureUnit StressUnit
{
    get => Strength.Unit;
    set => ChangeUnit(value);
}

///  

public Pressure TensileStrength => _calculator.TensileStrength.ToUnit(StressUnit);

///  

public Pressure TransverseModule => (ElasticModule / 2.4).ToUnit(StressUnit);

///  

public AggregateType Type
{
    get => _calculator.Type;
    set => _calculator.Type = value;
}

///  

///  

public double UltimateStrain => _calculator.UltimateStrain;

PressureUnit IUnitConvertible<PressureUnit>.Unit
{
    get => StressUnit;
    set => StressUnit = value;
}

LengthUnit IUnitConvertible<LengthUnit>.Unit
{
    get => DiameterUnit;
    set => DiameterUnit = value;
}

///  

///  

///  

public Parameters(double strength, double aggregateDiameter, ParameterModel model =
    ParameterModel.Default, AggregateType type = AggregateType.Quartzite, bool
    considerConfinement = false, PressureUnit strengthUnit = PressureUnit.Megapascal,
    LengthUnit diameterUnit = LengthUnit.Millimeter)
: this((Pressure) strength.As(strengthUnit), (Length)
    aggregateDiameter.As(diameterUnit), model, type, considerConfinement)
{
}

///  

///  

///  


```

```

/// <param name="type">The <see cref="AggregateType" />.</param>
/// <param name="strength">Concrete compressive strength (positive value).</param>
/// <param name="aggregateDiameter">The maximum diameter of concrete
    aggregate.</param>
/// <param name="model">The <see cref="ParameterModel" />.</param>
/// <param name="considerConfinement">
///     Consider confinement strength of concrete? If set to true, concrete strength
    is
///     increase in case of biaxial compression.
/// </param>
public Parameters(Pressure strength, Length aggregateDiameter, ParameterModel model =
    ParameterModel.Default, AggregateType type = AggregateType.Quartzite, bool
    considerConfinement = false)
{
    _strength          = strength;
    _aggDiameter       = aggregateDiameter;
    _calculator        = ParameterCalculator.GetCalculator(strength, model, type);
    ConsiderConfinement = considerConfinement;
}

/// <summary>
///     Get concrete class C20 (fc = 20 MPa).
/// </summary>
/// <inheritdoc cref="Parameters(Pressure, Length, ParameterModel, AggregateType,
    bool)" />
public static Parameters C20(Length aggregateDiameter, ParameterModel model =
    ParameterModel.MC2010, AggregateType type = AggregateType.Quartzite, bool
    considerConfinement = false) =>
    new(Pressure.FromMegapascals(20), aggregateDiameter, model, type,
        considerConfinement);

/// <summary>
///     Get concrete class C30 (fc = 30 MPa).
/// </summary>
/// <inheritdoc cref="Parameters(Pressure, Length, ParameterModel, AggregateType,
    bool)" />
public static Parameters C30(Length aggregateDiameter, ParameterModel model =
    ParameterModel.MC2010, AggregateType type = AggregateType.Quartzite, bool
    considerConfinement = false) =>
    new(Pressure.FromMegapascals(30), aggregateDiameter, model, type,
        considerConfinement);

/// <summary>
///     Get concrete class C40 (fc = 40 MPa).
/// </summary>
/// <inheritdoc cref="Parameters(Pressure, Length, ParameterModel, AggregateType,
    bool)" />
public static Parameters C40(Length aggregateDiameter, ParameterModel model =
    ParameterModel.MC2010, AggregateType type = AggregateType.Quartzite, bool
    considerConfinement = false) =>
    new(Pressure.FromMegapascals(40), aggregateDiameter, model, type,
        considerConfinement);

/// <summary>
///     Get concrete class C50 (fc = 40 MPa).
/// </summary>
/// <inheritdoc cref="Parameters(Pressure, Length, ParameterModel, AggregateType,
    bool)" />

```

```

public static Parameters C50(Length aggregateDiameter, ParameterModel model =
    ParameterModel.MC2010, AggregateType type = AggregateType.Quartzite, bool
    considerConfinement = false) =>
    new(Pressure.FromMegapascals(50), aggregateDiameter, model, type,
        considerConfinement);

/// <summary>
///     Create a clone of this object with converted units.
/// </summary>
/// <param name="stressUnit">The desired <see cref="PressureUnit" />.</param>
/// <param name="lengthUnit">The desired <see cref="LengthUnit" />.</param>
public Parameters Convert(PressureUnit? stressUnit = null, LengthUnit? lengthUnit =
    null) =>
    new(stressUnit.HasValue ? Strength.ToUnit(stressUnit.Value) : Strength,
        lengthUnit.HasValue ? AggregateDiameter.ToUnit(lengthUnit.Value) :
        AggregateDiameter, Model, Type);

/// <inheritdoc />
public override bool Equals(object? obj) => obj is Parameters other && Equals(other);

/// <inheritdoc />
public override int GetHashCode() => (int) Strength.Megapascals * (int)
    AggregateDiameter.Millimeters;

/// <summary>
///     Get a <see cref="CustomParameters" /> from this object.
/// </summary>
public CustomParameters ToCustomParameters() => new(Strength, TensileStrength,
    ElasticModule, AggregateDiameter, PlasticStrain, UltimateStrain,
    ConsiderConfinement);

/// <inheritdoc />
public override string ToString()
{
    char
        phi = (char) Characters.Phi,
        eps = (char) Characters.Epsilon;

    return
        "ConcreteParameters:\n\n" +
        $"fc={Strength}\n" +
        $"ft={TensileStrength}\n" +
        $"Ec={ElasticModule}\n" +
        $"{{eps}}c={PlasticStrain:0.##E+00}\n" +
        $"{{eps}}cu={UltimateStrain:0.##E+00}\n" +
        $"{{phi}},ag={AggregateDiameter}";
}

/// <inheritdoc />
public Parameters Clone() => new(Strength, AggregateDiameter, Model, Type);

/// <inheritdoc />
public bool Approaches(IConcreteParameters? other, Pressure tolerance) => Model ==
    other?.Model && Strength.Approx(other.Strength, tolerance);

/// <summary>
///     Change <see cref="AggregateDiameter" /> unit.
/// </summary>
/// <param name="unit">The desired <see cref="LengthUnit" />.</param>

```



```

public void ChangeUnit(LengthUnit unit)
{
    if (unit == DiameterUnit)
        return;

    _aggDiameter = _aggDiameter.ToUnit(unit);
}

/// <summary>
///     Change <see cref="Strength" /> unit.
/// </summary>
/// <param name="unit">The desired <see cref="LengthUnit" />.</param>
public void ChangeUnit(PressureUnit unit)
{
    if (unit == StressUnit)
        return;

    _strength = _strength.ToUnit(unit);
}

/// <remarks>
///     <see cref="Strength" /> is compared.
/// </remarks>
/// <inheritdoc />
public int CompareTo(IConcreteParameters? other) =>
    Strength == other?.Strength
        ? 0
        : Strength > other?.Strength
            ? 1
            : -1;

/// <inheritdoc />
public bool Equals(IConcreteParameters? other) => Approaches(other, Tolerance);

/// <inheritdoc />
bool IApproachable<IMaterialParameters, Pressure>.Approaches(IMaterialParameters
    other, Pressure tolerance) => other is IConcreteParameters parameters &&
    Approaches(parameters, tolerance);

/// <inheritdoc />
int IComparable<IMaterialParameters>.CompareTo(IMaterialParameters other) => other is
    IConcreteParameters parameters
        ? CompareTo(parameters)
        : 0;

IUnitConvertible<LengthUnit> IUnitConvertible<LengthUnit>.Convert(LengthUnit unit) =>
    Convert(lengthUnit: unit);

IUnitConvertible<PressureUnit> IUnitConvertible<PressureUnit>.Convert(PressureUnit
    unit) => Convert(unit);

/// <inheritdoc />
bool IEquatable<IMaterialParameters>.Equals(IMaterialParameters other) => other is
    IConcreteParameters parameters && Equals(parameters);

/// <summary>
///     Returns true if objects are equal.
/// </summary>
public static bool operator ==(Parameters left, IConcreteParameters right) =>
    left.Equals(right);

```

```

    /// <summary>
    ///     Returns true if objects are not equal.
    /// </summary>
    public static bool operator !=(Parameters left, IConcreteParameters right) =>
        !left.Equals(right);
}

```

C.1.1.2 Classe *ParameterCalculator* e derivadas

Calcula os parâmetros do concreto a partir dos modelos *Default* (Equação 4.1), ABNT NBR 6118:2014 e Model Code 2010.

ParameterCalculator.cs

```

namespace andrefmello91.Material.Concrete;

/// <summary>
///     Parameter calculator base class.
/// </summary>
internal abstract class ParameterCalculator : IEquatable<ParameterCalculator>
{
    private Pressure _strength;
    private AggregateType _type;

    /// <summary>
    ///     Get initial elastic module.
    /// </summary>
    public Pressure ElasticModule { get; protected set; }

    /// <summary>
    ///     Get fracture parameter.
    /// </summary>
    public virtual ForcePerLength FractureParameter =>
        ForcePerLength.FromNewtonsPerMillimeter(0.075);

    /// <summary>
    ///     Get the <see cref="ParameterModel" />.
    /// </summary>
    public abstract ParameterModel Model { get; }

    /// <summary>
    ///     Get concrete plastic (peak) strain (negative value).
    /// </summary>
    public double PlasticStrain { get; protected set; }

    /// <summary>
    ///     Get secant module.
    /// </summary>
    public virtual Pressure SecantModule => Strength / PlasticStrain;

    /// <summary>
    ///     Concrete strength.
    /// </summary>
    public Pressure Strength
    {
        get => _strength;
    }
}

```

```

        set
        {
            _strength = value;
            CalculateCustomParameters();
        }
    }

    /// <summary>
    ///     Get tensile strength.
    /// </summary>
    public Pressure TensileStrength { get; protected set; }

    /// <summary>
    ///     The type of concrete aggregate.
    /// </summary>
    public AggregateType Type
    {
        get => _type;
        set
        {
            _type = value;
            CalculateCustomParameters();
        }
    }

    /// <summary>
    ///     Get concrete ultimate strain (negative value).
    /// </summary>
    public double UltimateStrain { get; protected set; }

    /// <summary>
    ///     Base parameter calculator class.
    /// </summary>
    /// <param name="strength">Concrete compressive strength (positive value).</param>
    /// <param name="type">The <see cref="AggregateType" />.</param>
    protected ParameterCalculator(Pressure strength, AggregateType type)
    {
        _strength = strength;
        _type = type;
        CalculateCustomParameters();
    }

    /// <summary>
    ///     Get the <see cref="ParameterCalculator" /> based on <see
    ///     cref="ParameterModel" />.
    /// </summary>
    /// <param name="model">The <see cref="ParameterModel" />.</param>
    /// <inheritdoc cref="ParameterCalculator(Pressure, AggregateType)" />
    public static ParameterCalculator GetCalculator(Pressure strength, ParameterModel
        model, AggregateType type) =>
        model switch
        {
            ParameterModel.MC2010 => new MC2010(strength, type),
            ParameterModel.NBR6118 => new NBR6118(strength, type),
            - => new Default(strength, type)
        };

    /// <summary>
    ///     Calculate and update values for custom parameters.
    /// </summary>

```

```

protected abstract void CalculateCustomParameters();

public bool Equals(ParameterCalculator? other) => other is not null && Model ==
    other.Model;
}

```

Default.cs

```

namespace andrefmello91.Material.Concrete;

/// <summary>
///     Parameters calculated according to Modified Compression Field Theory.
/// </summary>
internal class Default : ParameterCalculator
{
    // Strains
    private const double ec = -0.002;
    private const double ecu = -0.0035;

    public override ParameterModel Model => ParameterModel.Default;

    /// <summary>
    ///     Parameter calculator based on Classic DSFM formulation.
    /// </summary>
    /// <inheritdoc />
    internal Default(Pressure strength, AggregateType type = AggregateType.Quartzite) :
        base(strength, type)
    {
    }

    private static Pressure Ec(Pressure strength) => (Pressure) (4732.98 *
        strength.Megapascals.Sqrt()).As(PressureUnit.Megapascal);

    private static Pressure fcr(Pressure strength) => (Pressure) (0.65 *
        Math.Pow(strength.Megapascals, 1D / 3)).As(PressureUnit.Megapascal);

    protected override void CalculateCustomParameters()
    {
        TensileStrength = fcr(Strength);
        ElasticModule = Ec(Strength);
        PlasticStrain = ec;
        UltimateStrain = ecu;
    }
}

```

NBR6118.cs

```

namespace andrefmello91.Material.Concrete;

/// <summary>
///     Parameters calculated according to NBR6118:2014.
/// </summary>
internal class NBR6118 : ParameterCalculator
{
    public override ParameterModel Model => ParameterModel.NBR6118;

    public override Pressure SecantModule => AlphaI(Strength) * ElasticModule;
}

```

```

/// <summary>
///     Parameters calculator based on NBR 6118:2014.
/// </summary>
/// <inheritdoc />
internal NBR6118(Pressure strength, AggregateType type = AggregateType.Quartzite)
    : base(strength, type)
{
}

private static double AlphaE(AggregateType type) =>
    type switch
    {
        AggregateType.Basalt      => 1.2,
        AggregateType.Quartzite   => 1,
        AggregateType.Limestone   => 0.9,
        -                          => 0.7
    };

private static double AlphaI(Pressure strength) => Math.Min(0.8 + 0.2 *
    strength.Megapascals / 80, 1);

private static double ec2(Pressure strength) =>
    strength.Megapascals <= 50
    ? -0.002
    : -0.002 - 0.000085 * (strength.Megapascals - 50).Pow(0.53);

private static Pressure Eci(Pressure strength, AggregateType type) =>
    strength.Megapascals <= 50
    ? Pressure.From(5600, PressureUnit.Megapascal) * AlphaE(type) *
        Math.Sqrt(strength.Megapascals)
    : Pressure.From(21500, PressureUnit.Megapascal) * AlphaE(type) * (0.1 *
        strength.Megapascals + 1.25).Pow(1.0 / 3);

private static double ecu(Pressure strength) =>
    strength.Megapascals <= 50
    ? -0.0035
    : -0.0026 - 0.035 * (0.01 * (90 - strength.Megapascals)).Pow(4);

protected override void CalculateCustomParameters()
{
    TensileStrength = fctm(Strength);
    ElasticModule    = Eci(Strength, Type);
    PlasticStrain    = ec2(Strength);
    UltimateStrain   = ecu(Strength);
}

private Pressure fctm(Pressure strength) =>
    strength.Megapascals <= 50
    ? Pressure.From(0.3, PressureUnit.Megapascal) * strength.Megapascals.Pow(2.0 /
        3)
    : Pressure.From(2.12, PressureUnit.Megapascal) * Math.Log(1 + 0.11 *
        strength.Megapascals);
}

```

MC2010.cs

namespace andrefmello91.Material.Concrete;

/// <summary>

```

///      Parameters calculated according to FIB Model Code 2010.
/// </summary>
internal class MC2010 : ParameterCalculator
{
    /// <summary>
    ///      Array of high strength concrete classes, C50 to C90 (MC2010).
    /// </summary>
    private static readonly double[] Classes =
    {
        50, 55, 60, 70, 80, 90
    };

    /// <summary>
    ///      Array of ultimate strains for each concrete class, C50 to C90 (MC2010).
    /// </summary>
    private static readonly double[] ClassesUltStrains =
    {
        -0.0034, -0.0034, -0.0033, -0.0032, -0.0031, -0.003
    };

    public override ParameterModel Model => ParameterModel.MC2010;

    /// <summary>
    ///      Parameter calculator based on fib Model Code 2010.
    /// </summary>
    /// <inheritdoc />
    internal MC2010(Pressure strength, AggregateType type = AggregateType.Quartzite) :
        base(strength, type)
    {
    }

    private static double AlphaE(AggregateType type) =>
        type switch
        {
            AggregateType.Basalt      => 1.2,
            AggregateType.Quartzite   => 1,
            -                          => 0.9
        };

    private static double ec1(Pressure strength) => -1.6 / 1000 * (0.1 *
        strength.Megapascals).Pow(0.25);

    private static Pressure Ec1(Pressure strength) => strength / ec1(strength);

    private static Pressure Eci(Pressure strength, AggregateType type) =>
        Pressure.From(21500, PressureUnit.Megapascal) * AlphaE(type) * (0.1 *
        strength.Megapascals).Pow(1.0 / 3);

    private static double ecu(Pressure strength)
    {
        switch (strength.Megapascals)
        {
            // Verify fcm
            case < 50:
                return
                    -0.0035;

            case >= 90:
                return

```

```

        -0.003;
    }

    // Get classes and ultimate strains
    // Interpolate values
    if (!Classes.Contains(strength.Megapascals))
        return
            UltimateStrainSpline().Interpolate(strength.Megapascals);

    var i = Array.IndexOf(Classes, strength.Megapascals);

    return
        ClassesUltStrains[i];
}

private static Pressure fctm(Pressure strength) =>
    strength.Megapascals <= 50
        ? Pressure.From(0.3, PressureUnit.Megapascal) * strength.Megapascals.Pow(2.0 /
            3)
        : Pressure.From(2.12, PressureUnit.Megapascal) * Math.Log(1 + 0.1 *
            strength.Megapascals);

/// <summary>
///     Interpolation for ultimate strains.
/// </summary>
private static CubicSpline UltimateStrainSpline() =>
    CubicSpline.InterpolateAkimaSorted(Classes, ClassesUltStrains);

protected override void CalculateCustomParameters()
{
    TensileStrength = fctm(Strength);
    ElasticModule    = Eci(Strength, Type);
    PlasticStrain    = ec1(Strength);
    UltimateStrain   = ecu(Strength);
}
}

```

C.1.1.3 Estrutura *CustomParameters*

Construído a partir de parâmetros do concreto fornecidos pelo usuário.

CustomParameters.cs

```

namespace andrefmello91.Material.Concrete;

/// <summary>
///     Concrete parameters struct.
/// </summary>
public struct CustomParameters : IConcreteParameters, ICloneable<CustomParameters>
{
    private Length _aggDiameter;
    private Pressure _elasticModule;
    private double _plasticStrain;
    private Pressure _strength;
    private Pressure _tensileStrength;
    private double _ultimateStrain;
}

```

```

/// <inheritdoc />
public Length AggregateDiameter
{
    get => _aggDiameter;
    set => _aggDiameter = value.ToUnit(DiameterUnit);
}

/// <inheritdoc />
public bool ConsiderConfinement { get; set; }

/// <inheritdoc />
public double CrackingStrain => TensileStrength / ElasticModule;

/// <inheritdoc />
public LengthUnit DiameterUnit
{
    get => AggregateDiameter.Unit;
    set => ChangeUnit(value);
}

/// <inheritdoc />
public Pressure ElasticModule
{
    get => _elasticModule;
    set => _elasticModule = value.ToUnit(StressUnit);
}

/// <inheritdoc />
public ForcePerLength FractureParameter =>
    ForcePerLength.FromNewtonsPerMillimeter(0.075);

/// <inheritdoc />
public ParameterModel Model
{
    get => ParameterModel.Custom;
    set
    {
    }
}

/// <inheritdoc />
/// <inheritdoc cref="Parameters.PlasticStrain" />
public double PlasticStrain
{
    get => _plasticStrain;
    set => _plasticStrain = -value.Abs();
}

/// <inheritdoc />
public Pressure SecantModule => Strength / PlasticStrain;

/// <inheritdoc />
public Pressure Strength
{
    get => _strength;
    set => _strength = value.ToUnit(StressUnit);
}

/// <inheritdoc />
public PressureUnit StressUnit

```



```

{
    get => Strength.Unit;
    set => ChangeUnit(value);
}

/// <inheritdoc />
public Pressure TensileStrength
{
    get => _tensileStrength;
    set => _tensileStrength = value.ToUnit(StressUnit);
}

/// <inheritdoc />
public Pressure TransverseModule => (ElasticModule / 2.4).ToUnit(StressUnit);

/// <inheritdoc />
public AggregateType Type { get; set; }

/// <inheritdoc />
/// <inheritdoc cref="Parameters.PlasticStrain" />
public double UltimateStrain
{
    get => _ultimateStrain;
    set => _ultimateStrain = -value.Abs();
}

/// <inheritdoc />
Pressure IMaterialParameters.CompressiveStrength => Strength;

PressureUnit IUnitConvertible<PressureUnit>.Unit
{
    get => StressUnit;
    set => StressUnit = value;
}

LengthUnit IUnitConvertible<LengthUnit>.Unit
{
    get => DiameterUnit;
    set => DiameterUnit = value;
}

/// <inheritdoc cref="CustomParameters(Pressure, Pressure, Pressure, Length, double,
double, bool)" />
public CustomParameters(double strength, double tensileStrength, double
    elasticModule, double aggregateDiameter, double plasticStrain = 0.002, double
    ultimateStrain = 0.0035, bool considerConfinement = false, PressureUnit
    strengthUnit = PressureUnit.Megapascal, LengthUnit diameterUnit =
    LengthUnit.Millimeter)
: this((Pressure) strength.As(strengthUnit), (Pressure)
    tensileStrength.As(strengthUnit), (Pressure) elasticModule.As(strengthUnit),
    (Length) aggregateDiameter.As(diameterUnit), plasticStrain, ultimateStrain,
    considerConfinement)
{
}

/// <summary>
///     Custom parameters constructor.
/// </summary>
/// <inheritdoc cref="Parameters(Pressure, Length, ParameterModel, AggregateType,
bool)" />

```

```

/// <param name="tensileStrength">Concrete tensile strength.</param>
/// <param name="elasticModule">Concrete initial elastic module.</param>
/// <param name="plasticStrain">Concrete plastic strain (positive or negative
    value).</param>
/// <param name="ultimateStrain">Concrete ultimate strain (positive or negative
    value).</param>
public CustomParameters(Pressure strength, Pressure tensileStrength, Pressure
    elasticModule, Length aggregateDiameter, double plasticStrain = 0.002, double
    ultimateStrain = 0.0035, bool considerConfinement = false)
{
    _strength          = strength.Abs();
    _tensileStrength    = tensileStrength;
    _elasticModule      = elasticModule;
    _plasticStrain      = -plasticStrain.Abs();
    _ultimateStrain     = -ultimateStrain.Abs();
    _aggDiameter        = aggregateDiameter;
    Type                = AggregateType.Quartzite;
    ConsiderConfinement = considerConfinement;
}

/// <summary>
///     Create a clone of this object with converted units.
/// </summary>
/// <param name="stressUnit">The desired <see cref="PressureUnit" />.</param>
/// <param name="lengthUnit">The desired <see cref="LengthUnit" />.</param>
public CustomParameters Convert(PressureUnit? stressUnit = null, LengthUnit?
    lengthUnit = null) =>
    new(Strength.ToUnit(stressUnit ?? StressUnit), TensileStrength.ToUnit(stressUnit
        ?? StressUnit), ElasticModule.ToUnit(stressUnit ?? StressUnit),
        AggregateDiameter.ToUnit(lengthUnit ?? DiameterUnit), PlasticStrain,
        UltimateStrain);

/// <inheritdoc />
public override bool Equals(object? obj) => obj is CustomParameters other &&
    Equals(other);

/// <inheritdoc />
public override int GetHashCode() => (int) Strength.Megapascals * (int)
    AggregateDiameter.Millimeters;

/// <summary>
///     Get a <see cref="Parameters" /> from this object.
/// </summary>
/// <param name="model">The required <see cref="ParameterModel" />. Not <see
    cref="ParameterModel.Custom" />.</param>
/// <inheritdoc cref="Parameters(Pressure, Length, ParameterModel, AggregateType,
    bool)" />
public Parameters ToParameters(ParameterModel model, AggregateType type =
    AggregateType.Quartzite) =>
    new(Strength, AggregateDiameter, model, type, ConsiderConfinement);

/// <inheritdoc />
public override string ToString()
{
    char
        phi = (char) Characters.Phi,
        eps = (char) Characters.Epsilon;

    return
        "Concrete□Parameters:\n\n" +

```

```

        $"fc_{\u{Strength}}\n" +
        $"ft_{\u{TensileStrength}}\n" +
        $"Ec_{\u{ElasticModule}}\n" +
        $"{\eps}c_{\u{PlasticStrain:0.##E+00}}\n" +
        $"{\eps}cu_{\u{UltimateStrain:0.##E+00}}\n" +
        $"{\phi},ag_{\u{AggregateDiameter}}";
    }

    /// <inheritdoc />
    public CustomParameters Clone() => new(Strength, TensileStrength, ElasticModule,
        AggregateDiameter, PlasticStrain, UltimateStrain);

    /// <inheritdoc />
    public bool Approaches(IConcreteParameters? other, Pressure tolerance) => Model ==
        other?.Model && Strength.Approx(other.Strength, tolerance);

    /// <summary>
    ///     Change <see cref="AggregateDiameter" /> unit.
    /// </summary>
    /// <param name="unit">The desired <see cref="LengthUnit" />.</param>
    public void ChangeUnit(LengthUnit unit)
    {
        if (unit == DiameterUnit)
            return;

        _aggDiameter = _aggDiameter.ToUnit(unit);
    }

    /// <summary>
    ///     Change <see cref="Strength" /> unit.
    /// </summary>
    /// <param name="unit">The desired <see cref="LengthUnit" />.</param>
    public void ChangeUnit(PressureUnit unit)
    {
        if (unit == StressUnit)
            return;

        _strength = _strength.ToUnit(unit);
        _tensileStrength = _tensileStrength.ToUnit(unit);
        _elasticModule = _elasticModule.ToUnit(unit);
    }

    /// <remarks>
    ///     <see cref="Strength" /> is compared.
    /// </remarks>
    /// <inheritdoc />
    public int CompareTo(IConcreteParameters? other) =>
        Strength == other?.Strength
            ? 0
            : Strength > other?.Strength
                ? 1
                : -1;

    /// <inheritdoc />
    public bool Equals(IConcreteParameters? other) => Approaches(other,
        Parameters.Tolerance);

    /// <inheritdoc />

```

```

bool IApproachable<IMaterialParameters, Pressure>.Approaches(IMaterialParameters
    other, Pressure tolerance) => other is IConcreteParameters parameters &&
    Approaches(parameters, tolerance);

/// <inheritdoc />
int IComparable<IMaterialParameters>.CompareTo(IMaterialParameters other) => other is
    IConcreteParameters parameters
    ? CompareTo(parameters)
    : 0;

IUnitConvertible<LengthUnit> IUnitConvertible<LengthUnit>.Convert(LengthUnit unit) =>
    Convert(lengthUnit: unit);

IUnitConvertible<PressureUnit> IUnitConvertible<PressureUnit>.Convert(PressureUnit
    unit) => Convert(unit);

/// <inheritdoc />
bool IEquatable<IMaterialParameters>.Equals(IMaterialParameters other) => other is
    IConcreteParameters parameters && Equals(parameters);

/// <returns>
///     True if objects are equal.
/// </returns>
public static bool operator ==(CustomParameters left, IConcreteParameters right) =>
    left.Equals(right);

/// <returns>
///     True if objects are not equal.
/// </returns>
public static bool operator !=(CustomParameters left, IConcreteParameters right) =>
    !left.Equals(right);
}

```

C.1.1.4 Estrutura *SteelParameters*

Constitui os parâmetros do aço.

SteelParameters.cs

```

namespace andrefmello91.Material.Reinforcement;

/// <summary>
///     Steel parameters struct.
/// </summary>
public struct SteelParameters : IMaterialParameters, IApproachable<SteelParameters,
    Pressure>, IEquatable<SteelParameters>, IComparable<SteelParameters>,
    ICloneable<SteelParameters>
{

    /// <summary>
    ///     The default <see cref="Pressure" /> tolerance.
    /// </summary>
    public static readonly Pressure Tolerance = Pressure.FromPascals(1E-3);

    /// <summary>
    ///     Steel hardening consideration.
    /// </summary>
    public bool ConsiderHardening { get; }
}

```

```

/// <summary>
///     Get hardening module.
/// </summary>
public Pressure HardeningModule { get; private set; }

/// <summary>
///     Get hardening strain.
/// </summary>
public double HardeningStrain { get; }

/// <summary>
///     Get yield strain.
/// </summary>
public double YieldStrain => YieldStress / ElasticModule;

/// <summary>
///     Get yield stress.
/// </summary>
public Pressure YieldStress { get; private set; }

/// <inheritdoc />
public Pressure ElasticModule { get; private set; }

/// <inheritdoc />
public double UltimateStrain { get; }

/// <inheritdoc />
public PressureUnit Unit
{
    get => YieldStress.Unit;
    set => ChangeUnit(value);
}

/// <inheritdoc />
Pressure IMaterialParameters.CompressiveStrength => -YieldStress;

/// <inheritdoc />
double IMaterialParameters.PlasticStrain => YieldStrain;

/// <inheritdoc />
Pressure IMaterialParameters.TensileStrength => YieldStress;

/// <summary>
///     Create steel parameters without hardening consideration.
/// </summary>
/// <param name="yieldStress">Steel yield stress.</param>
/// <param name="elasticModule">Steel elastic module.</param>
/// <param name="ultimateStrain">Steel ultimate strain.</param>
public SteelParameters(Pressure yieldStress, Pressure elasticModule, double
    ultimateStrain = 0.01)
    : this(yieldStress, elasticModule, ultimateStrain, false, Pressure.Zero, 0)
{
    YieldStress      = yieldStress;
    ElasticModule     = elasticModule.ToUnit(yieldStress.Unit);
    UltimateStrain    = ultimateStrain;
}

/// <inheritdoc cref="SteelParameters(Pressure, Pressure, double)" />

```

```

/// <param name="unit">The unit of <paramref name="yieldStress" /> and <paramref
    name="elasticModule" />.</param>
public SteelParameters(double yieldStress, double elasticModule = 210000, double
    ultimateStrain = 0.01, PressureUnit unit = PressureUnit.Megapascal)
    : this((Pressure) yieldStress.As(unit), (Pressure) elasticModule.As(unit),
        ultimateStrain)
{
}

/// <summary>
///     Create steel parameters with hardening consideration.
/// </summary>
/// <inheritdoc cref="SteelParameters(Pressure, Pressure, double)" />
/// <param name="hardeningModule">Steel hardening module.</param>
/// <param name="hardeningStrain">Steel strain at the beginning of hardening.</param>
public SteelParameters(Pressure yieldStress, Pressure elasticModule, Pressure
    hardeningModule, double hardeningStrain, double ultimateStrain = 0.01)
    : this(yieldStress, elasticModule, ultimateStrain, true, hardeningModule,
        hardeningStrain)
{
}

/// <inheritdoc cref="SteelParameters(Pressure, Pressure, Pressure, double, double)"
    />
private SteelParameters(Pressure yieldStress, Pressure elasticModule, double
    ultimateStrain, bool considerHardening, Pressure hardeningModule, double
    hardeningStrain)
{
    ConsiderHardening = considerHardening;
    YieldStress        = yieldStress;
    ElasticModule       = elasticModule;
    UltimateStrain      = ultimateStrain;
    HardeningModule     = hardeningModule;
    HardeningStrain     = hardeningStrain;
}

/// <inheritdoc cref="IUnitConvertible{TUnit}.Convert" />
public SteelParameters Convert(PressureUnit unit)
{
    var param = Clone();

    if (Unit != unit)
        param.ChangeUnit(unit);

    return param;
}

/// <inheritdoc />
public override bool Equals(object? other) => other is SteelParameters parameters &&
    Equals(parameters);

/// <inheritdoc />
public override int GetHashCode() => (int) ElasticModule.Gigapascals * (int)
    YieldStress.Megapascals;

/// <inheritdoc />
public override string ToString()
{
    var epsilon = (char) Characters.Epsilon;

```

```

var msg =
    "SteelParameters:\n" +
    $"fy={YieldStress}\n" +
    $"Es={ElasticModule}\n" +
    $"{epsilon}y={YieldStrain:0.##E+00}";

if (ConsiderHardening)
    msg += "\n\n" +
        "HardeningParameters:\n" +
        $"Es={HardeningModule}\n" +
        $"{epsilon}y={HardeningStrain:0.##E+00}";

return msg;
}

/// <inheritdoc />
public bool Approaches(SteelParameters other, Pressure tolerance)
{
    var basic = YieldStress.Approx(other.YieldStress, tolerance) &&
        ElasticModule.Approx(other.ElasticModule, tolerance) &&
        UltimateStrain.Approx(other.UltimateStrain);

    if (!other.ConsiderHardening)
        return basic;

    return basic && HardeningModule.Approx(other.HardeningModule, tolerance) &&
        HardeningStrain.Approx(other.HardeningStrain);
}

/// <inheritdoc />
public SteelParameters Clone() => new(YieldStress, ElasticModule, UltimateStrain,
    ConsiderHardening, HardeningModule, HardeningStrain);

/// <inheritdoc />
public int CompareTo(SteelParameters other) =>
    YieldStress > other.YieldStress || YieldStress.Approx(other.YieldStress,
        Tolerance) && ElasticModule > other.ElasticModule
    ? 1
    : YieldStress.Approx(other.YieldStress, Tolerance) &&
        ElasticModule.Approx(other.ElasticModule, Tolerance)
    ? 0
    : -1;

/// <inheritdoc />
public bool Equals(SteelParameters other) => Approaches(other, Tolerance);

/// <inheritdoc />
public bool Approaches(IMaterialParameters other, Pressure tolerance) => other is
    SteelParameters parameters && Approaches(parameters, tolerance);

/// <inheritdoc />
public void ChangeUnit(PressureUnit unit)
{
    if (Unit == unit)
        return;

    YieldStress = YieldStress.ToUnit(unit);
    ElasticModule = ElasticModule.ToUnit(unit);
    HardeningModule = HardeningModule.ToUnit(unit);

```

```

}

/// <inheritdoc />
public int CompareTo(IMaterialParameters other) => other is SteelParameters parameters
    ? CompareTo(parameters)
    : 0;

/// <inheritdoc />
public bool Equals(IMaterialParameters other) => other is SteelParameters parameters
    && Equals(parameters);

IUnitConvertible<PressureUnit> IUnitConvertible<PressureUnit>.Convert(PressureUnit
    unit) => Convert(unit);

/// <summary>
///     Returns true if steel parameters are equal.
/// </summary>
public static bool operator ==(SteelParameters left, SteelParameters right) =>
    left.IsEqualTo(right);

/// <summary>
///     Create a steel from this parameters.
/// </summary>
public static implicit operator Steel(SteelParameters parameters) => new(parameters);

/// <summary>
///     Get this steel's parameters.
/// </summary>
public static implicit operator SteelParameters?(Steel? steel) => steel?.Parameters;

/// <summary>
///     Returns true if steel parameters are different.
/// </summary>
public static bool operator !=(SteelParameters left, SteelParameters right) =>
    left.IsNotEqualTo(right);
}

```

C.1.2 Materiais uniaxiais

C.1.2.1 Classe base *Concrete*

Dá suporte às classes *UniaxialConcrete* e *BiaxialConcrete*. A principal propriedade desta classe é uma estrutura do tipo *Parameters* ou *CustomParameters*.

Concrete.cs

```

/// <summary>
///     Base class for concrete object.
/// </summary>
public abstract class Concrete : IEquatable<Concrete>, IComparable<Concrete>
{
    /// <summary>
    ///     Check if concrete is cracked.
    /// </summary>
    /// <returns>

```



```

    ///      <b>True</b> if concrete is cracked.
    /// </returns>
    public abstract bool Cracked { get; }

    /// <summary>
    ///      Check if concrete crushed.
    /// </summary>
    /// <returns>
    ///      <b>True</b> if concrete strain is bigger than <see
    ///      cref="IConcreteParameters.UltimateStrain" />.
    /// </returns>
    public abstract bool Crushed { get; }

    /// <summary>
    ///      Get concrete <see cref="ConstitutiveModel" />.
    /// </summary>
    public ConstitutiveModel Model { get; }

    /// <summary>
    ///      Get concrete <see cref="IConcreteParameters" />.
    /// </summary>
    public IConcreteParameters Parameters { get; }

    /// <summary>
    ///      Check if concrete yielded.
    /// </summary>
    /// <returns>
    ///      <b>True</b> if concrete strain is bigger than <see
    ///      cref="IConcreteParameters.PlasticStrain" />.
    /// </returns>
    public abstract bool Yielded { get; }

    /// <summary>
    ///      Base concrete object.
    /// </summary>
    /// <param name="parameters">Concrete <see cref="IConcreteParameters" />
    ///      object.</param>
    /// <param name="model">The <see cref="ConstitutiveModel" />.</param>
    protected Concrete(IConcreteParameters parameters, ConstitutiveModel model =
        ConstitutiveModel.SMM)
    {
        Parameters = parameters;
        Model      = model;
    }

    /// <inheritdoc />
    public override bool Equals(object? obj) => obj is Concrete concrete &&
        Equals(concrete);

    /// <inheritdoc />
    public override int GetHashCode() => Parameters.GetHashCode();

    /// <inheritdoc />
    public override string ToString() => Parameters.ToString(!);

    /// <inheritdoc />
    public int CompareTo(Concrete? other) => Parameters.CompareTo(other?.Parameters);

```

```

    /// <inheritdoc />
    public virtual bool Equals(Concrete? other) => Model == other?.Model && Parameters ==
        other?.Parameters;

    /// <summary>
    ///     Returns true if parameters and constitutive model are equal.
    /// </summary>
    public static bool operator ==(Concrete? left, Concrete? right) =>
        left.IsEqualTo(right);

    /// <summary>
    ///     Returns true if parameters and constitutive model are different.
    /// </summary>
    public static bool operator !=(Concrete? left, Concrete? right) =>
        left.IsNotEqualTo(right);
}

```

C.1.2.2 Classe *UniaxialConcrete*

Contém as propriedades do material concreto uniaxial, como área, tensão e força.

UniaxialConcrete.cs

```

namespace andrefmello91.Material.Concrete;

/// <summary>
///     Concrete uniaxial class.
/// </summary>
public partial class UniaxialConcrete : Concrete, IUniaxialMaterial,
    ICloneable<UniaxialConcrete>
{

    /// <summary>
    ///     Get concrete <see cref="UniaxialConcrete.Constitutive" />.
    /// </summary>
    private readonly Constitutive _constitutive;

    /// <inheritdoc />
    public override bool Cracked => _constitutive.Cracked;

    /// <inheritdoc />
    public override bool Crushed => Strain <= -Parameters.UltimateStrain.Abs();

    /// <summary>
    ///     Calculate maximum force resisted by concrete (negative value).
    /// </summary>
    public Force MaxForce => -Parameters.Strength * Area;

    /// <summary>
    ///     Calculate current secant module of concrete.
    /// </summary>
    public Pressure SecantModule => _constitutive.SecantModule(Stress, Strain);

    /// <summary>
    ///     Calculate normal stiffness.
    /// </summary>
    public Force Stiffness => Parameters.ElasticModule * Area;
}

```

```

/// <inheritdoc />
public override bool Yielded => Strain <= -Parameters.PlasticStrain.Abs();

/// <summary>
///     The concrete area.
/// </summary>
public Area Area { get; }

/// <summary>
///     The current concrete force.
/// </summary>
public Force Force => Stress * Area;

/// <summary>
///     The concrete strain.
/// </summary>
public double Strain { get; private set; }

/// <summary>
///     The concrete stress.
/// </summary>
public Pressure Stress { get; private set; }

/// <summary>
///     Concrete for uniaxial calculations.
/// </summary>
/// <param name="concreteArea">The concrete cross-section area.</param>
/// <inheritdoc cref="Concrete(ICConcreteParameters, ConstitutiveModel)" />
public UniaxialConcrete(ICConcreteParameters parameters, Area concreteArea,
    ConstitutiveModel model = ConstitutiveModel.SMM)
    : base(parameters, model)
{
    Area = concreteArea;
    _constitutive = Constitutive.From(model, parameters);
}

/// <summary>
///     Set concrete strain and calculate stress, in MPa.
/// </summary>
/// <param name="strain">Current strain.</param>
/// <param name="reinforcement">
///     The <see cref="UniaxialReinforcement" /> (only for
///     <see cref="UniaxialConcrete.DSFMConstitutive" />).
/// </param>
public void Calculate(double strain, UniaxialReinforcement? reinforcement = null)
{
    Strain = strain;
    Stress = CalculateStress(strain, reinforcement);
}

/// <inheritdoc />
public override bool Equals(Concrete? other) => other is UniaxialConcrete &&
    base.Equals(other);

/// <inheritdoc />
public override bool Equals(object? obj) => obj is UniaxialConcrete concrete &&
    base.Equals(concrete);

/// <inheritdoc />

```

```

public override int GetHashCode() => Parameters.GetHashCode();

/// <summary>
///     Calculate stress given strain.
/// </summary>
/// <param name="strain">Current strain.</param>
/// <param name="reinforcement">
///     The <see cref="UniaxialReinforcement" /> (only for
///     <see cref="UniaxialConcrete.DSFMConstitutive" />).
/// </param>
private Pressure CalculateStress(double strain, UniaxialReinforcement? reinforcement
    = null) => _constitutive.CalculateStress(strain, reinforcement);

/// <inheritdoc />
public UniaxialConcrete Clone() => new(Parameters, Area, Model);

/// <inheritdoc />
void IUniaxialMaterial.Calculate(double strain) => Calculate(strain);
}

```

C.1.2.3 Subclasse *Constitutive* e derivadas

Subclasses de *UniaxialConcrete* que implementam os modelos constitutivos do concreto, para o estado uniaxial.

Constitutive.cs

```

namespace andrefmello91.Material.Concrete;

public partial class UniaxialConcrete
{
    /// <summary>
    ///     Base class for concrete constitutive model.
    /// </summary>
    private abstract class Constitutive : IEquatable<Constitutive>
    {
        /// <summary>
        ///     Concrete <see cref="IConcreteParameters" />.
        /// </summary>
        protected readonly IConcreteParameters Parameters;

        /// <summary>
        ///     Check if concrete is cracked.
        /// </summary>
        /// <returns>
        ///     <b>True</b> if concrete is cracked.
        /// </returns>
        public bool Cracked { get; private set; }

        /// <summary>
        ///     The constitutive model of concrete.
        /// </summary>
        public abstract ConstitutiveModel Model { get; }

        // Constructor
        /// <summary>
        ///     Base class for concrete behavior
    }
}

```

```

/// </summary>
/// <param name="parameters">Concrete parameters object.</param>
protected Constitutive(IConcreteParameters parameters) => Parameters = parameters;

/// <summary>
///     Get concrete <see cref="Constitutive" /> object based on the <see
///     cref="ConstitutiveModel" />.
/// </summary>
/// <param name="constitutiveModel">The <see cref="ConstitutiveModel" /> for
///     concrete.</param>
/// <param name="parameters">Concrete <see cref="Parameters" />.</param>
public static Constitutive From(ConstitutiveModel constitutiveModel,
    IConcreteParameters parameters) =>
    constitutiveModel switch
    {
        ConstitutiveModel.DSFM => new DSFMConstitutive(parameters),
        ConstitutiveModel.MCFT => new MCFTConstitutive(parameters),
        -                        => new SMMConstitutive(parameters)
    };

/// <summary>
///     Calculate stress (in MPa) given <paramref name="strain" />.
///     <para>For <seealso cref="UniaxialConcrete" />.</para>
/// </summary>
/// <param name="strain">Current strain.</param>
/// <param name="reinforcement">
///     The <see cref="UniaxialReinforcement" /> reinforcement (only for
///     <see cref="DSFMConstitutive" />).
/// </param>
public Pressure CalculateStress(double strain, UniaxialReinforcement?
    reinforcement = null) =>
    !strain.IsFinite() || strain.ApproxZero()
    ? Pressure.Zero
    : strain > 0
    ? TensileStress(strain, reinforcement)
    : CompressiveStress(strain);

/// <summary>
///     Calculate current secant module.
/// </summary>
/// <param name="stress">Current stress.</param>
/// <param name="strain">Current strain.</param>
public Pressure SecantModule(Pressure stress, double strain) =>
    stress.Abs() <= Material.Concrete.Parameters.Tolerance || strain.Abs() <= 1E-9
    ? Parameters.ElasticModule
    : stress / strain;

public override string ToString() => $"{Model}";

/// <summary>
///     Calculate compressive stress for <see cref="UniaxialConcrete" /> case.
/// </summary>
/// <param name="strain">Compressive strain (negative) in concrete.</param>
protected abstract Pressure CompressiveStress(double strain);

/// <summary>
///     Calculate the tensile strain for cracked state.
/// </summary>
/// <inheritdoc cref="TensileStress" />

```

```

protected abstract Pressure CrackedStress(double strain, UniaxialReinforcement?
    reinforcement = null);

/// <summary>
///     Calculate tensile stress for <see cref="UniaxialConcrete" /> case.
/// </summary>
/// <param name="strain">Tensile strain in concrete.</param>
/// <param name="reinforcement">The <see cref="UniaxialReinforcement" /> (only for
///     <see cref="DSFMConstitutive" />).</param>
protected Pressure TensileStress(double strain, UniaxialReinforcement?
    reinforcement = null)
{
    if (strain.ApproxZero())
        return Pressure.Zero;

    CheckCrackedState(strain);

    return !Cracked
        ? UncrackedStress(strain)
        : CrackedStress(strain, reinforcement);
}

/// <summary>
///     Calculate the tensile strain for uncracked state.
/// </summary>
/// <param name="strain">Tensile strain in concrete.</param>
protected Pressure UncrackedStress(double strain) => Parameters.ElasticModule *
    strain;

/// <summary>
///     Check if concrete is cracked based on <paramref name="strain" />.
/// </summary>
/// <inheritdoc cref="UncrackedStress" />
private void CheckCrackedState(double strain) => Cracked = strain >=
    Parameters.CrackingStrain;

public bool Equals(Constitutive? other) => Model == other?.Model;
}
}

```

MCFT.cs

```

namespace andrefmello91.Material.Concrete;

public partial class UniaxialConcrete
{
    /// <summary>
    ///     MCFT constitutive class.
    /// </summary>
    private class MCFTConstitutive : Constitutive
    {
        public override ConstitutiveModel Model { get; } = ConstitutiveModel.MCFT;

        /// <summary>
        ///     MCFT constitutive object.
        /// </summary>
        /// <inheritdoc cref="Concrete" />
        public MCFTConstitutive(IConcreteParameters parameters) : base(parameters)
        {

```

```

    }

    /// <inheritdoc />
    protected override Pressure CompressiveStress(double strain)
    {
        double
            ec = Parameters.PlasticStrain,
            fc = Parameters.Strength.Megapascals,
            n = strain / ec,
            f = -fc * (2 * n - n * n);

        return
            (Pressure) f.As(PressureUnit.Megapascal);
    }

    /// <inheritdoc />
    protected override Pressure CrackedStress(double strain, UniaxialReinforcement?
        reinforcement = null) =>
        Parameters.TensileStrength / (1 + Math.Sqrt(500 * strain));
    }
}

```

DSFM.cs

```

namespace andrefmello91.Material.Concrete;

public partial class UniaxialConcrete
{
    /// <summary>
    ///     DSFM constitutive class.
    /// </summary>
    private class DSFMConstitutive : Constitutive
    {
        public override ConstitutiveModel Model { get; } = ConstitutiveModel.DSFM;

        /// <summary>
        ///     DSFM constitutive object.
        /// </summary>
        /// <inheritdoc cref="Concrete" />
        /// <param name="considerCrackSlip">Consider crack slip? (default: true)</param>
        public DSFMConstitutive(IConcreteParameters parameters)
            : base(parameters)
        {
        }

        /// <summary>
        ///     Calculate reference length.
        /// </summary>
        /// <inheritdoc cref="TensionStiffening" />
        private static Length ReferenceLength(UniaxialReinforcement? reinforcement) =>
            0.5 * (reinforcement is null
                ? Length.FromMillimeters(21)
                : Length.FromMillimeters(21) + 0.155 * reinforcement.BarDiameter /
                    reinforcement.Ratio);

        /// <summary>
        ///     Calculate tension stiffening coefficient (for DSFM).
        /// </summary>
        /// <inheritdoc cref="TensionStiffening" />
    }
}

```

```

private static double TensionStiffeningCoefficient(UniaxialReinforcement?
    reinforcement) => reinforcement is null
    ? 0
    : 0.25 * reinforcement.BarDiameter.Millimeters / reinforcement.Ratio;

/// <inheritdoc />
protected override Pressure CompressiveStress(double strain)
{
    // Calculate the principal compressive stress in concrete
    // Get strains
    var ec2 = strain;

    // Calculate fp and ep
    var fp = -Parameters.Strength;
    var ep = Parameters.PlasticStrain;

    // Calculate parameters of concrete
    double
        k = ep <= ec2
            ? 1
            : 0.67 - fp.Megapascals / 62,
        n = 0.8 - fp.Megapascals / 17,
        ec2_ep = (ec2 / ep).AsFinite();

    // Calculate the principal compressive stress in concrete
    return
        fp * n * ec2_ep / (n - 1 + ec2_ep.Pow(n * k));
}

/// <inheritdoc />
protected override Pressure CrackedStress(double strain, UniaxialReinforcement?
    reinforcement = null)
{
    // Calculate concrete post-cracking stress associated with tension softening
    var fc1a = TensionSoftening(strain, reinforcement);

    // Calculate concrete postcracking stress associated with tension stiffening
    var fc1b = TensionStiffening(strain, reinforcement);

    // Return maximum
    return
        fc1a >= fc1b
            ? fc1a
            : fc1b;
}

/// <summary>
///     Calculate concrete post-cracking stress associated with tension softening.
/// </summary>
/// <inheritdoc cref="TensionStiffening" />
private Pressure TensionSoftening(double strain, UniaxialReinforcement?
    reinforcement)
{
    double
        Gf = Parameters.FractureParameter.NewtonsPerMillimeter,
        ft = Parameters.TensileStrength.Megapascals,
        ecr = Parameters.CrackingStrain,
        ets = 2.0 * Gf / (ft * ReferenceLength(reinforcement).Millimeters);

    return

```



```

        Parameters.TensileStrength * (1.0 - (strain - ecr) / (ets - ecr));
    }

    /// <summary>
    ///     Calculate concrete post-cracking stress associated with tension stiffening
    ///     (for
    ///     <see cref="Material.Concrete.UniaxialConcrete" />).
    /// </summary>
    /// <param name="strain">The tensile strain to calculate stress.</param>
    /// <param name="reinforcement">The <see cref="UniaxialReinforcement" />.</param>
    private Pressure TensionStiffening(double strain, UniaxialReinforcement?
        reinforcement)
    {
        if (reinforcement is null)
            return Pressure.Zero;

        // Calculate coefficient for tension stiffening effect
        var m = TensionStiffeningCoefficient(reinforcement);

        // Calculate concrete postcracking stress associated with tension stiffening
        var fc1b = Parameters.TensileStrength / (1 + Math.Sqrt(2.2 * m * strain));

        // Check the maximum value of fc1 that can be transmitted across cracks
        var fc1s = reinforcement?.MaximumPrincipalTensileStress() ?? Pressure.Zero;

        // Return minimum
        return
            Min(fc1s, fc1b);
    }
}
}

```

SMM.cs

```

namespace andrefmello91.Material.Concrete;

public partial class UniaxialConcrete
{
    /// <summary>
    ///     MCFT constitutive class.
    /// </summary>
    private class SMMConstitutive : Constitutive
    {
        private readonly double _strengthFunction;

        public override ConstitutiveModel Model => ConstitutiveModel.SMM;

        /// <summary>
        ///     MCFT constitutive object.
        /// </summary>
        /// <inheritdoc cref="Concrete" />
        public SMMConstitutive(IConcreteParameters parameters) : base(parameters) =>
            _strengthFunction = StrengthFunction(Parameters.Strength);

        /// <summary>
        ///     Calculate the strength function for the softening parameter.
        /// </summary>
        /// <param name="strength">The compressive strength of concrete.</param>
    }
}

```

```

private static double StrengthFunction(Pressure strength) => Math.Min(5.8 /
    strength.Megapascals.Sqrt(), 0.9);

/// <inheritdoc />
protected override Pressure CompressiveStress(double strain)
{
    // Calculate softening coefficient
    var soft = _strengthFunction;

    // Calculate peak stress and strain
    var fp = -soft * Parameters.Strength;
    var ep = soft * Parameters.PlasticStrain;

    // Calculate strain ratio:
    var e2_ep = (strain / ep).AsFinite();

    return
        (e2_ep <= 1) switch
        {
            { } when e2_ep < 0 => Pressure.Zero,

            // Pre-peak
            true => fp * (2 * e2_ep - e2_ep * e2_ep),

            // Post-peak
            _ => UnitMath.Max(fp * (1D - ((e2_ep - 1D) / (4D / soft - 1D)).Pow(2)),
                0.5 * fp)
        };
}

/// <inheritdoc />
protected override Pressure CrackedStress(double strain, UniaxialReinforcement?
    reinforcement = null) =>
    Parameters.TensileStrength * (Parameters.CrackingStrain / strain).Pow(0.4);
}
}

```

C.1.2.4 Classe *UniaxialReinforcement*

Contém as propriedades do material armadura uniaxial, como área, tensão e força.

UniaxialReinforcement.cs

```

namespace andrefmello91.Material.Reinforcement;

/// <summary>
///     Uniaxial reinforcement class.
/// </summary>
public class UniaxialReinforcement : IUniaxialMaterial, IUnitConvertible<LengthUnit>,
    IApproachable<UniaxialReinforcement, Length>, IEquatable<UniaxialReinforcement>,
    IComparable<UniaxialReinforcement>, ICloneable<UniaxialReinforcement>
{
    /// <summary>
    ///     The tolerance to consider displacements equal.
    /// </summary>
    public static readonly Length Tolerance = Length.FromMillimeters(1E-3);
}

```

```

/// <summary>
///     Get bar diameter.
/// </summary>
public Length BarDiameter { get; private set; }

/// <summary>
///     Get/set concrete area.
/// </summary>
public Area ConcreteArea { get; set; }

/// <summary>
///     Get number of reinforcing bars.
/// </summary>
public int NumberOfBars { get; }

/// <summary>
///     Get reinforcement ratio in the cross-section.
/// </summary>
public double Ratio => ConcreteArea == Area.Zero
    ? 0
    : Area / ConcreteArea;

/// <summary>
///     Get <see cref="Reinforcement.Steel" /> of this.
/// </summary>
public Steel Steel { get; }

/// <summary>
///     Get normal stiffness.
/// </summary>
public Force Stiffness => Steel.Parameters.ElasticModule * Area;

/// <inheritdoc cref="Reinforcement.Steel.Yielded" />
public bool Yielded => Steel.Yielded;

/// <summary>
///     Get the yield force.
/// </summary>
public Force YieldForce => Area * Steel.Parameters.YieldStress;

/// <summary>
///     The reinforcement area.
/// </summary>
public Area Area { get; private set; }

/// <summary>
///     The current force.
/// </summary>
public Force Force => Area * Steel.Stress;

/// <inheritdoc />
public double Strain => Steel.Strain;

/// <inheritdoc />
public Pressure Stress => Steel.Stress;

/// <inheritdoc />
public LengthUnit Unit
{
    get => BarDiameter.Unit;
}

```

```

        set => ChangeUnit(value);
    }

    /// <param name="concreteArea">The concrete area, in <see cref="AreaUnit" />
    /// compatible to <paramref name="unit" />.</param>
    /// <param name="unit">The <see cref="LengthUnit" /> of <paramref name="barDiameter"
    /// />.</param>
    /// <inheritdoc cref="UniaxialReinforcement" />
    public UniaxialReinforcement(int numberOfBars, double barDiameter, [NotNull] Steel
        steel, double concreteArea = 0, LengthUnit unit = LengthUnit.Millimeter)
        : this(numberOfBars, (Length) barDiameter.As(unit), steel, (Area)
            concreteArea.As(unit.GetAreaUnit()))
    {
    }

    /// <summary>
    ///     Reinforcement for uniaxial calculations
    /// </summary>
    /// <param name="numberOfBars">The number of bars of reinforcement.</param>
    /// <param name="barDiameter">The bar diameter.</param>
    /// <param name="steel">The steel object.</param>
    /// <param name="concreteArea">The concrete area.</param>
    public UniaxialReinforcement(int numberOfBars, Length barDiameter, [NotNull] Steel
        steel, Area concreteArea)
    {
        NumberOfBars = numberOfBars;
        BarDiameter = barDiameter;
        Area = CalculateArea().ToUnit(barDiameter.Unit.GetAreaUnit());
        ConcreteArea = concreteArea;
        Steel = steel;
    }

    /// <inheritdoc cref="IUnitConvertible{TUnit}.Convert" />
    public UniaxialReinforcement Convert(LengthUnit unit) => new(NumberOfBars,
        BarDiameter.ToUnit(unit), Steel.Clone(), ConcreteArea.ToUnit(unit.GetAreaUnit()));

    /// <inheritdoc />
    public override bool Equals(object? other) => other is UniaxialReinforcement
        reinforcement && Equals(reinforcement);

    /// <summary>
    ///     Compare two reinforcement objects.
    ///     <para>Returns true if <see cref="NumberOfBars" /> and <see cref="BarDiameter"
    ///     /> are equal.</para>
    /// </summary>
    /// <param name="other">The other reinforcement object.</param>
    /// <param name="tolerance">The tolerance.</param>
    public bool EqualsNumberAndDiameter(UniaxialReinforcement? other, Length tolerance)
        => other is not null && NumberOfBars == other.NumberOfBars &&
        BarDiameter.Approx(other.BarDiameter, tolerance);

    /// <inheritdoc />
    public override int GetHashCode() => (int) BarDiameter.Millimeters.Pow(NumberOfBars);

    /// <summary>
    ///     Calculate maximum value of tensile strength that can be transmitted across
    ///     cracks.
    /// </summary>
    public Pressure MaximumPrincipalTensileStress() => Ratio *
        (Steel.Parameters.YieldStress - Steel.Stress);

```

```

/// <inheritdoc />
public override string ToString()
{
    var phi = (char) Characters.Phi;

    return
        $"Reinforcement: {NumberOfBars}_{phi}_{BarDiameter}_{Area}\n\n"
        + Steel;
}

/// <summary>
///     Calculated reinforcement area.
/// </summary>
private Area CalculateArea() => 0.25 * NumberOfBars * Constants.Pi * BarDiameter *
    BarDiameter;

/// <inheritdoc />
public bool Approaches(UniaxialReinforcement? other, Length tolerance) => other is
    not null && EqualsNumberAndDiameter(other, tolerance);

/// <inheritdoc />
public UniaxialReinforcement Clone() => new(NumberOfBars, BarDiameter, Steel.Clone(),
    ConcreteArea);

/// <inheritdoc />
public int CompareTo(UniaxialReinforcement? other) =>
    other is null || Area > other.Area
        ? 1
        : Area == other.Area
            ? 0
            : -1;

/// <summary>
///     Compare two reinforcement objects.
///     <para>Returns true if parameters are equal.</para>
/// </summary>
/// <param name="other">The other reinforcement object.</param>
public virtual bool Equals(UniaxialReinforcement? other) => Approaches(other,
    Tolerance);

/// <summary>
///     Set steel strain and stress.
/// </summary>
/// <param name="strain">Current strain.</param>
public void Calculate(double strain) => Steel.Calculate(strain);

/// <inheritdoc />
public void ChangeUnit(LengthUnit unit)
{
    if (Unit == unit)
        return;

    BarDiameter = BarDiameter.ToUnit(unit);
    Area = Area.ToUnit(unit.GetAreaUnit());
}

IUnitConvertible<LengthUnit> IUnitConvertible<LengthUnit>.Convert(LengthUnit unit) =>
    Convert(unit);

```

```

    /// <summary>
    ///     Returns true if steel parameters are equal.
    /// </summary>
    public static bool operator ==(UniaxialReinforcement? left, UniaxialReinforcement?
        right) => left.IsEqualTo(right);

    /// <summary>
    ///     Returns true if steel parameters are different.
    /// </summary>
    public static bool operator !=(UniaxialReinforcement? left, UniaxialReinforcement?
        right) => left.IsNotEqualTo(right);
}

```

C.1.2.5 Classe *Steel*

Tem como propriedade principal uma estrutura do tipo *SteelParameters* e implementa o modelo constitutivo do aço, dando suporte às classes *UniaxialReinforcement* e *WebReinforcement*. Cada objeto destas classes possuirá uma propriedade *Steel*.

Steel.cs

```

namespace andrefmello91.Material.Reinforcement;

/// <summary>
///     Steel class.
/// </summary>
public class Steel : IUnitConvertible<PressureUnit>, IApproachable<Steel, Pressure>,
    IEquatable<Steel>, IComparable<Steel>, ICloneable<Steel>
{
    /// <summary>
    ///     The default <see cref="Pressure" /> tolerance.
    /// </summary>
    public static readonly Pressure Tolerance = Pressure.FromPascals(1E-3);

    /// <summary>
    ///     Get tensile hardening consideration.
    /// </summary>
    private readonly bool _considerHardening;

    /// <summary>
    ///     The steel parameters.
    /// </summary>
    public SteelParameters Parameters { get; }

    /// <summary>
    ///     Get current steel secant module.
    /// </summary>
    public Pressure SecantModule => Strain.ApproxZero()
        ? Parameters.ElasticModule
        : Stress / Strain;

    /// <summary>
    ///     Get current strain.
    /// </summary>
    public double Strain { get; private set; }
}

```

```

/// <summary>
///     Get current stress.
/// </summary>
public Pressure Stress { get; private set; }

/// <summary>
///     Check if steel yielded.
/// </summary>
/// <returns>
///     True if steel strain is bigger than <see cref="SteelParameters.YieldStrain"
///     />.
/// </returns>
public bool Yielded => Strain.Abs() >= Parameters.YieldStrain;

/// <summary>
///     Get the <see cref="PressureUnit" /> that this was constructed with.
/// </summary>
public PressureUnit Unit
{
    get => Parameters.Unit;
    set => ChangeUnit(value);
}

/// <summary>
///     Create a steel object from steel parameters.
/// </summary>
/// <param name="parameters">Steel parameters.</param>
public Steel(SteelParameters parameters) => Parameters = parameters;

/// <inheritdoc cref="Steel(Pressure, Pressure, double)" />
/// <param name="unit">
///     The <see cref="PressureUnit" /> of <paramref name="yieldStress" /> and
///     <paramref name="elasticModule" />.
/// </param>
public Steel(double yieldStress, double elasticModule = 210000, double ultimateStrain
    = 0.01, PressureUnit unit = PressureUnit.Megapascal)
    : this(new SteelParameters(yieldStress, elasticModule, ultimateStrain, unit))
{
}

/// <summary>
///     Steel object with no tensile hardening.
/// </summary>
/// <param name="yieldStress">Steel yield stress.</param>
/// <param name="elasticModule">Steel elastic module.</param>
/// <param name="ultimateStrain">Steel ultimate strain.</param>
public Steel(Pressure yieldStress, Pressure elasticModule, double ultimateStrain =
    0.01)
    : this(new SteelParameters(yieldStress, elasticModule, ultimateStrain))
{
}

/// <inheritdoc cref="Steel(Pressure, Pressure, Pressure, double, double)" />
/// <inheritdoc cref="Steel(double, double, double, PressureUnit)" />
public Steel(double yieldStress, double elasticModule, double hardeningModule, double
    hardeningStrain, double ultimateStrain = 0.01, PressureUnit unit =
    PressureUnit.Megapascal)
    : this((Pressure) yieldStress.As(unit), (Pressure) elasticModule.As(unit),
        (Pressure) hardeningModule.As(unit), hardeningStrain, ultimateStrain)
{
}

```

```

}

/// <summary>
///     Steel object with tensile hardening.
/// </summary>
/// <param name="hardeningModule">Steel hardening module.</param>
/// <param name="hardeningStrain">Steel strain at the beginning of hardening.</param>
/// <inheritdoc cref="Steel(Pressure, Pressure, double)" />
public Steel(Pressure yieldStress, Pressure elasticModule, Pressure hardeningModule,
    double hardeningStrain, double ultimateStrain = 0.01)
    : this(new SteelParameters(yieldStress, elasticModule, hardeningModule,
        hardeningStrain, ultimateStrain))
{
}

/// <summary>
///     Calculate stress, given strain.
/// </summary>
/// <param name="strain">Current strain.</param>
private static Pressure CalculateStress(SteelParameters parameters, double strain)
{
    // Correct value
    strain = strain.AsFinite();

    return parameters.ConsiderHardening switch
    {
        // Failure
        { } when strain.Abs() >= parameters.UltimateStrain => Pressure.Zero,

        // Elastic
        { } when strain.IsBetween(-parameters.YieldStrain, parameters.YieldStrain) =>
            parameters.ElasticModule * strain,

        // Compression yielding
        { } when strain.IsBetween(-parameters.UltimateStrain, -parameters.YieldStrain)
            => -parameters.YieldStress,

        // Tension yielding with no hardening
        false when strain.IsBetween(parameters.YieldStrain, parameters.UltimateStrain)
            => parameters.YieldStress,

        // Tension yielding with hardening
        true when strain.IsBetween(parameters.YieldStrain, parameters.HardeningStrain)
            => parameters.YieldStress,

        // Tension hardening (if considered)
        true when strain.IsBetween(parameters.HardeningStrain,
            parameters.UltimateStrain) => parameters.YieldStress +
            parameters.HardeningModule * (strain - parameters.HardeningStrain),

        // Default
        _ => Pressure.Zero
    };

    // Failure
}

/// <summary>
///     Set steel strain and calculate stress.
/// </summary>

```



```

/// <param name="strain">Current strain.</param>
public void Calculate(double strain)
{
    Strain = strain.AsFinite();
    Stress = CalculateStress(Parameters, strain);
}

/// <inheritdoc cref="IUnitConvertible{TUnit}.Convert" />
public Steel Convert(PressureUnit unit) => new(Parameters.Convert(unit));

/// <inheritdoc />
public override bool Equals(object? other) => other is Steel steel && Equals(steel);

/// <inheritdoc />
public override int GetHashCode() => Parameters.GetHashCode();

/// <inheritdoc />
public override string ToString() => Parameters.ToString();

/// <inheritdoc />
public bool Approaches(Steel? other, Pressure tolerance) => other is not null &&
    Parameters.Approaches(other.Parameters, tolerance);

/// <inheritdoc />
public Steel Clone() => new(Parameters.Clone());

/// <inheritdoc />
public int CompareTo(Steel? other) =>
    other is not null
        ? Parameters.CompareTo(other.Parameters)
        : 1;

/// <inheritdoc />
public virtual bool Equals(Steel? other) => Approaches(other, Tolerance);

/// <inheritdoc />
public void ChangeUnit(PressureUnit unit)
{
    if (Unit == unit)
        return;

    Parameters.ChangeUnit(unit);
    Stress = Stress.ToUnit(unit);
}

IUnitConvertible<PressureUnit> IUnitConvertible<PressureUnit>.Convert(PressureUnit
    unit) => Convert(unit);

/// <summary>
///     Returns true if steel parameters are equal.
/// </summary>
public static bool operator ==(Steel? left, Steel? right) => left.IsEqualTo(right);

/// <summary>
///     Returns true if steel parameters are different.
/// </summary>
public static bool operator !=(Steel? left, Steel? right) => left.IsNotEqualTo(right);

```

}

C.1.3 Materiais biaxiais

C.1.3.1 Classes *BiaxialConcrete* e *SMMConcrete*

Contém as propriedades do material concreto biaxial, como os estados de tensão e deformação. A classe base *BiaxialConcrete* usada para os modelos constitutivos TMCC e MCTP, e a classe *SMMConcrete*, especificamente ao MMA.

BiaxialConcrete.cs

```
namespace andrefmello91.Material.Concrete;

/// <summary>
///     Biaxial concrete for membrane calculations.
/// </summary>
public partial class BiaxialConcrete : Concrete, IBiaxialMaterial,
    ICloneable<BiaxialConcrete>
{

    /// <summary>
    ///     Get/set crack slip consideration.
    /// </summary>
    public bool ConsiderCrackSlip
    {
        get => ConstitutiveEquations.ConsiderCrackSlip;
        set => ConstitutiveEquations.ConsiderCrackSlip = value;
    }

    /// <summary>
    ///     Returns true if concrete is cracked.
    /// </summary>
    public override bool Cracked => ConstitutiveEquations.Cracked;

    /// <inheritdoc />
    public override bool Crushed => PrincipalStrains.Epsilon2 <=
        -Parameters.UltimateStrain.Abs();

    /// <summary>
    ///     Get/set Cs coefficient for concrete softening.
    /// </summary>
    public double Cs
    {
        get => ConstitutiveEquations.Cs;
        set => ConstitutiveEquations.Cs = value;
    }

    /// <summary>
    ///     Get the deviation angle between <see cref="Strains" /> and <see
    ///         cref="PrincipalStrains" />.
    /// </summary>
    public double DeviationAngle { get; protected set; }

    /// <summary>
```

```

///      Get concrete initial stiffness <see cref="Matrix" />.
/// </summary>
/// <remarks>
///      Elements are expressed in <see cref="PressureUnit.Megapascal" />.
/// </remarks>
public MaterialMatrix InitialStiffness
{
    get
    {
        var Ec = Parameters.ElasticModule;

        /// Concrete matrix
        var Dc1 = MaterialMatrix.Zero(Constants.PiOver4, Ec.Unit);
        Dc1[0, 0] = Ec;
        Dc1[1, 1] = Ec;
        Dc1[2, 2] = 0.5 * Ec;

        /// Get transformation matrix
        /// var t = StrainRelations.TransformationMatrix(Constants.PiOver4);

        /// Calculate Dc
        return
            Dc1.ToHorizontal();
    }
}

/// <inheritdoc />
public override bool Yielded => PrincipalStrains.Epsilon2 <=
    -Parameters.PlasticStrain.Abs();

/// <summary>
///      Get concrete <see cref="BiaxialConcrete.Constitutive" />.
/// </summary>
protected Constitutive ConstitutiveEquations { get; }

/// <summary>
///      Calculate current secant module of concrete.
/// </summary>
private (Pressure Ec1, Pressure Ec2) SecantModule
{
    get
    {
        /// Get values
        double
            ec1 = PrincipalStrains.Epsilon1,
            ec2 = PrincipalStrains.Epsilon2;

        Pressure
            fc1 = PrincipalStresses.Sigma1,
            fc2 = PrincipalStresses.Sigma2;

        /// Calculate modules
        Pressure
            Ec1 = ConstitutiveEquations.SecantModule(fc1, ec1),
            Ec2 = ConstitutiveEquations.SecantModule(fc2, ec2);

        return
            (Ec1, Ec2);
    }
}

```

```

/// <summary>
///     Get/set concrete <see cref="PrincipalStrainState" />.
/// </summary>
public PrincipalStrainState PrincipalStrains { get; protected set; }

/// <summary>
///     Get/set concrete <see cref="PrincipalStressState" />.
/// </summary>
public PrincipalStressState PrincipalStresses { get; protected set; }

/// <summary>
///     Get concrete stiffness <see cref="Matrix" />, with elements in <see
///     cref="PressureUnit.Megapascal" />.
/// </summary>
public MaterialMatrix Stiffness
{
    get
    {
        var (Ec1, Ec2) = (SecantModule.Ec1.Megapascals, SecantModule.Ec2.Megapascals);

        var Gc = Ec1 * Ec2 / (Ec1 + Ec2);

        // Concrete matrix
        var array = new double[3, 3];
        array[0, 0] = Ec1;
        array[1, 1] = Ec2;
        array[2, 2] = Gc;

        var Dc1 = new MaterialMatrix(array, PrincipalStrains.Theta1)
        {
            Unit = Parameters.StressUnit
        };

        // Get transformation matrix
        // var t = PrincipalStrains.TransformationMatrix;

        // Calculate Dc
        return
            Dc1.ToHorizontal();
    }
}

/// <summary>
///     Get/set concrete <see cref="StrainState" />.
/// </summary>
public StrainState Strains { get; protected set; }

/// <summary>
///     Get/set concrete <see cref="StressState" />.
/// </summary>
public StressState Stresses { get; protected set; }

/// <summary>
///     Create a concrete object for membrane calculations.
/// </summary>
/// <inheritdoc />
protected BiaxialConcrete(ConcreteParameters parameters, ConstitutiveModel model =
    ConstitutiveModel.MCFT)
    : base(parameters, model) =>

```

```

        ConstitutiveEquations = Constitutive.From(model, parameters);

    /// <inheritdoc cref="BiaxialConcrete(IConcreteParameters, ConstitutiveModel)" />
    public static BiaxialConcrete From(IConcreteParameters parameters, ConstitutiveModel
        model = ConstitutiveModel.SMM) =>
        model switch
        {
            ConstitutiveModel.SMM => new SMMConcrete(parameters),
            -                        => new BiaxialConcrete(parameters, model)
        };

    /// <summary>
    ///     Set concrete <see cref="StressState" /> given <see cref="StrainState" />
    /// </summary>
    /// <param name="strains">Current <see cref="StrainState" /> in concrete.</param>
    /// <param name="reinforcement">The <see cref="WebReinforcement" />.</param>
    /// <param name="referenceLength">The reference length (only for <see
    ///     cref="DSFMConstitutive" />).</param>
    public virtual void Calculate(StrainState strains, WebReinforcement? reinforcement,
        Length? referenceLength = null)
    {
        // Get strains
        Strains = strains.Clone();

        // Calculate principal strains
        PrincipalStrains = Strains.ToPrincipal();

        // Get stresses from constitutive model
        PrincipalStresses = ConstitutiveEquations.CalculateStresses(PrincipalStrains,
            reinforcement, referenceLength).ToPrincipal();
        Stresses          = PrincipalStresses.ToHorizontal();
    }

    /// <inheritdoc />
    public override bool Equals(Concrete? other) => other is BiaxialConcrete &&
        base.Equals(other);

    /// <inheritdoc />
    public override bool Equals(object? obj) => obj is BiaxialConcrete concrete &&
        Equals(concrete);

    /// <inheritdoc />
    public override int GetHashCode() => Parameters.GetHashCode();

    /// <summary>
    ///     Set tensile stress.
    /// </summary>
    /// <param name="fc1">Concrete tensile stress.</param>
    public void SetTensileStress(Pressure fc1)
    {
        // Get compressive stress and theta1
        var fc2      = PrincipalStresses.Sigma2;
        var theta1    = PrincipalStresses.Theta1;

        // Set new state
        PrincipalStresses = new PrincipalStressState(fc1, fc2, theta1);
        Stresses          = StressState.FromPrincipal(PrincipalStresses);
    }

```

```

    /// <inheritdoc />
    void IBiaxialMaterial.Calculate(StrainState strainState) => Calculate(strainState,
        null);

    /// <inheritdoc />
    public BiaxialConcrete Clone() => new(Parameters, Model);
}

```

SMMConcrete.cs

```

namespace andrefmello91.Material.Concrete;

/// <summary>
///     SMM concrete auxiliary class.
/// </summary>
internal class SMMConcrete : BiaxialConcrete
{
    /// <summary>
    ///     Get concrete <see cref="BiaxialConcrete.Constitutive" />.
    /// </summary>
    private new SMMConstitutive ConstitutiveEquations => (SMMConstitutive)
        base.ConstitutiveEquations;

    /// <summary>
    ///     The strain state in the average principal strain direction, not affected
    ///     by Poisson effect.
    /// </summary>
    // private StrainState NotAffectedStrains { get; set; }

    /// <inheritdoc />
    internal SMMConcrete(IconcreteParameters parameters)
        : base(parameters, ConstitutiveModel.SMM)
    {
        Strains = new StrainState(0, 0, 0, Constants.PiOver4);
        Stresses = new StressState(0, 0, 0, Constants.PiOver4);
    }

    /// <summary>
    ///     Calculate the deviation angle for a strain state.
    /// </summary>
    /// <param name="strains">The strain state for the principal direction of
    ///     concrete.</param>
    private static double CalculateDeviationAngle(StrainState strains) => 0.5 *
        (strains.GammaXY / (strains.EpsilonX - strains.EpsilonY)).Atan().AsFinite();

    /// <inheritdoc />
    public override void Calculate(StrainState strains, WebReinforcement? reinforcement,
        Length? referenceLength = null)
    {
        // Update strains
        // var theta = strains.ToPrincipal().Theta1 + DeviationAngle;
        Strains = strains;
        PrincipalStrains = Strains.ToPrincipal();

        // Calculate deviation angle
        DeviationAngle = CalculateDeviationAngle(Strains);

        // Calculate stresses
    }
}

```

```

        Stresses          = ConstitutiveEquations.CalculateStresses(Strains,
            reinforcement, deviationAngle: DeviationAngle);
        PrincipalStresses = Stresses.ToPrincipal();

        // Update stresses
        UpdateStresses(reinforcement);
    }

    /// <summary>
    ///     Update the stress state based in equilibrium on crack.
    /// </summary>
    private void UpdateStresses(WebReinforcement? reinforcement)
    {
        if (reinforcement is null)
            return;

        // Check the maximum value of fc1 that can be transmitted across cracks
        var fc1s = reinforcement.MaximumPrincipalTensileStress(PrincipalStresses.Theta1);

        if (fc1s >= PrincipalStresses.Sigma1)
            return;

        // Recalculate stresses
        var pStresses = PrincipalStresses.Clone();
        PrincipalStresses = new PrincipalStressState(fc1s, pStresses.Sigma2,
            pStresses.Theta1);
        Stresses          = PrincipalStresses.Transform(DeviationAngle);
    }
}

```

C.1.3.2 Subclasse *Constitutive* e derivadas

Subclasses de *BiaxialConcrete* que implementam os modelos constitutivos do concreto, para o estado biaxial.

Constitutive.cs

```

namespace andrefmello91.Material.Concrete;

public partial class BiaxialConcrete
{
    /// <summary>
    ///     Base class for concrete constitutive model.
    /// </summary>
    protected abstract class Constitutive : IEquatable<Constitutive>
    {
        /// <summary>
        ///     Concrete <see cref="IConcreteParameters" />.
        /// </summary>
        protected readonly IConcreteParameters Parameters;

        /// <summary>
        ///     The crack slip consideration.
        /// </summary>
        /// <remarks>
        ///     Only for <see cref="ConstitutiveModel.DSFM" />.
        /// </remarks>
    }
}

```

```

public bool ConsiderCrackSlip { get; set; }

/// <inheritdoc cref="Concrete.Cracked" />
public bool Cracked { get; private set; }

/// <inheritdoc cref="BiaxialConcrete.Cs" />
public double Cs { get; set; } = 0.55;

/// <summary>
///     The constitutive model of concrete.
/// </summary>
public abstract ConstitutiveModel Model { get; }

// Constructor
/// <summary>
///     Base class for concrete behavior
/// </summary>
/// <inheritdoc cref="From" />
protected Constitutive(IConcreteParameters parameters) => Parameters = parameters;

/// <summary>
///     Get concrete <see cref="Constitutive" /> object based on the <see
///     cref="ConstitutiveModel" />.
/// </summary>
/// <param name="constitutiveModel">The <see cref="ConstitutiveModel" /> for
///     concrete.</param>
/// <param name="parameters">Concrete <see cref="IConcreteParameters" />.</param>
public static Constitutive From(ConstitutiveModel constitutiveModel,
    IConcreteParameters parameters) =>
    constitutiveModel switch
    {
        ConstitutiveModel.DSFM => new DSFMConstitutive(parameters),
        ConstitutiveModel.MCFT => new MCFTConstitutive(parameters),
        -                        => new SMMConstitutive(parameters)
    };

/// <summary>
///     Calculate confinement strength factor according to Kupfer et. al. (1969).
/// </summary>
/// <param name="transverseStress">The stress acting on the transverse direction
///     of the analyzed direction.</param>
/// <param name="concreteStrength">Concrete compressive strenght.</param>
internal static double ConfinementFactor(Pressure transverseStress, Pressure
    concreteStrength)
{
    // Get absolute value
    var fcn_fc = (transverseStress / concreteStrength).Abs();

    var c = 1 + 0.92 * fcn_fc - 0.76 * fcn_fc * fcn_fc;

    return
        c.IsFinite() && c > 1 && c < 2
        ? c
        : 1;
}

/// <summary>
///     Calculate concrete <see cref="PrincipalStressState" /> related to <see
///     cref="PrincipalStrainState" />.
/// <para>For <seealso cref="BiaxialConcrete" />.</para>

```



```

/// </summary>
/// <param name="strainsAtAvgPrincipal">
///     The strain state in concrete, at the average principal strain direction of
///     the membrane element.
///     <remarks>
///         The direction X must be the tensile (or largest) strain and Y the
///         compressive (or smaller) one.
///     </remarks>
/// </param>
/// <param name="reinforcement">The <see cref="WebReinforcement" />.</param>
/// <param name="referenceLength">The reference length (only for <see
///     cref="ConstitutiveModel.DSFM" />).</param>
/// <param name="deviationAngle">
///     The deviation angle between applied principal stresses and concrete
///     principal stresses
///     (only for <see cref="ConstitutiveModel.SMM" />).
/// </param>
/// <returns>
///     The <see cref="StressState" /> at the direction of <paramref
///     name="strainsAtAvgPrincipal" />.
/// </returns>
public StressState CalculateStresses(IState<double> strainsAtAvgPrincipal,
    WebReinforcement? reinforcement, Length? referenceLength = null, double
    deviationAngle = 0)
{
    if (strainsAtAvgPrincipal.IsZero)
        return new StressState(0, 0, 0, strainsAtAvgPrincipal.ThetaX);

    // Rotate if necessary
    if (strainsAtAvgPrincipal.X < strainsAtAvgPrincipal.Y)
        strainsAtAvgPrincipal = strainsAtAvgPrincipal.Transform(Constants.PiOver2);

    // Get strains
    double
        ec1 = strainsAtAvgPrincipal.X.AsFinite(),
        ec2 = strainsAtAvgPrincipal.Y.AsFinite(),
        yxy = strainsAtAvgPrincipal.XY.AsFinite();

    // Get the case
    var pCase = strainsAtAvgPrincipal is IPrincipalState<double> pState
        ? pState.Case
        : new PrincipalStrainState(ec1, ec2).Case;

    Pressure fc1, fc2;

    switch (pCase)
    {
        // Verify case
        case PrincipalCase.UniaxialCompression:
        case PrincipalCase.TensionCompression:
            fc1 = TensileStress(ec1, ec2, strainsAtAvgPrincipal.ThetaX,
                reinforcement, referenceLength);
            fc2 = CompressiveStress(ec2, ec1, deviationAngle);
            break;

        case PrincipalCase.UniaxialTension:
        case PrincipalCase.PureTension:
            fc1 = TensileStress(ec1, ec2, strainsAtAvgPrincipal.ThetaX,
                reinforcement, referenceLength);
    }
}

```

```

        fc2 = TensileStress(ec2, ec1, strainsAtAvgPrincipal.ThetaX,
            reinforcement, referenceLength);
        break;

    case PrincipalCase.PureCompression when !Parameters.ConsiderConfinement:
        fc1 = CompressiveStress(ec1, ec2, deviationAngle);
        fc2 = CompressiveStress(ec2, ec1, deviationAngle);
        break;

    case PrincipalCase.PureCompression when Parameters.ConsiderConfinement:
        var conf = ConfinementStresses(strainsAtAvgPrincipal, deviationAngle);
        fc1 = conf.SigmaX;
        fc2 = conf.SigmaY;
        break;

    default:
        return new StressState(0, 0, 0, strainsAtAvgPrincipal.ThetaX);
}

// Calculate shear stress (for SMM)
var tau = Model is ConstitutiveModel.SMM
    ? 0.5 * yxy * (fc1 - fc2) / (ec1 - ec2)
    : Pressure.Zero;

return
    new StressState(fc1, fc2, tau, strainsAtAvgPrincipal.ThetaX);
}

/// <summary>
///     Calculate current secant module.
/// </summary>
/// <param name="stress">Current stress.</param>
/// <param name="strain">Current strain.</param>
public Pressure SecantModule(Pressure stress, double strain) =>
    stress.Abs() <= Material.Concrete.Parameters.Tolerance || strain.Abs() <= 1E-9
        ? Parameters.ElasticModule
        : stress / strain;

/// <inheritdoc />
public override string ToString() => $"{Model}";

/// <summary>
///     Calculate compressive stress for <see
///         cref="Material.Concrete.BiaxialConcrete" /> case.
/// </summary>
/// <param name="strain">The compressive strain (negative) to calculate
///     stress.</param>
/// <param name="transverseStrain">The strain at the transverse direction to
///     <paramref name="strain" />.</param>
/// <param name="deviationAngle">
///     The deviation angle between applied principal stresses and concrete
///     principal stresses
///     (only for <see cref="ConstitutiveModel.SMM" />).
/// </param>
/// <param name="confinementFactor">
///     The confinement factor for pure compression case.
///     <para>See: <seealso cref="ConfinementFactor" /></para>
/// </param>
/// <returns>Compressive stress in MPa</returns>

```

```

protected abstract Pressure CompressiveStress(double strain, double
    transverseStrain, double deviationAngle = 0, double confinementFactor = 1);

/// <summary>
///     Calculate tensile stress for cracked concrete.
/// </summary>
/// <param name="strain">Current tensile strain.</param>
/// <param name="theta1">The angle of <paramref name="strain" /> related to
    horizontal axis, in radians.</param>
/// <param name="referenceLength">The reference length (only for <see
    cref="DSFMConstitutive" />).</param>
/// <param name="reinforcement">The <see cref="WebReinforcement" /> (only for <see
    cref="DSFMConstitutive" />).</param>
protected abstract Pressure CrackedStress(double strain, double theta1,
    WebReinforcement? reinforcement, Length? referenceLength = null);

/// <summary>
///     Check if concrete is cracked for <see
    cref="Material.Concrete.BiaxialConcrete" /> case and set cracked property,
///     from Gupta (1998)
///     formulation.
/// </summary>
/// <param name="fc1">Principal tensile stress.</param>
/// <param name="ec2">Principal compressive strain.</param>
private void CheckCrackedState(Pressure fc1, double ec2)
{
    var ft = Parameters.TensileStrength;
    var ec = Parameters.PlasticStrain;

    // Calculate current cracking stress
    var fcr1 = ft * (1 - ec2 / ec);

    // Verify limits
    var fcr = Max(fcr1, 0.25 * ft);
    fcr = Min(fcr, ft);

    // Verify if concrete is cracked
    Cracked = fc1 >= fcr;
}

/// <summary>
///     Calculate confinement <see cref="PrincipalStressState" />.
/// </summary>
/// <param name="strainsAtAvgPrincipal">
///     The smeared strains in concrete, affected by Poisson effect, at the
    direction of
///     average principal strains.
/// </param>
/// <param name="deviationAngle">The deviation angle between applied principal
    stresses and concrete principal stresses.</param>
private StressState ConfinementStresses(IState<double> strainsAtAvgPrincipal,
    double deviationAngle)
{
    // Get strains
    double
        ec1 = strainsAtAvgPrincipal.X,
        ec2 = strainsAtAvgPrincipal.Y;

    // Calculate initial stresses
    Pressure

```

```

        fc1 = CompressiveStress(ec1, ec2, deviationAngle),
        fc2 = CompressiveStress(ec2, ec1, deviationAngle);

    var tol = Pressure.FromMegapascals(0.01);

    // Iterate to find stresses (maximum 20 iterations)
    for (var it = 1; it <= 20; it++)
    {
        // Calculate confinement factors
        double
            betaL1 = ConfinementFactor(fc2, Parameters.Strength),
            betaL2 = ConfinementFactor(fc1, Parameters.Strength);

        // Calculate iteration stresses
        Pressure
            fc1It = CompressiveStress(ec1, ec2, deviationAngle, betaL1),
            fc2It = CompressiveStress(ec2, ec1, deviationAngle, betaL2);

        // Verify tolerances

        if ((fc1 - fc1It).Abs() <= tol && (fc2 - fc2It).Abs() <= tol)
            break;

        // Update stresses
        fc1 = fc1It;
        fc2 = fc2It;
    }

    return
        new StressState(fc1, fc2, Pressure.Zero, strainsAtAvgPrincipal.ThetaX);
}

/// <summary>
///     Calculate confinement <see cref="PrincipalStressState" />.
/// </summary>
/// <param name="principalStrains">
///     The <see cref="PrincipalStrainState" />, in pure compression.
/// </param>
private PrincipalStressState ConfinementStresses(PrincipalStrainState
    principalStrains)
{
    // Get strains
    double
        ec1 = principalStrains.Epsilon1,
        ec2 = principalStrains.Epsilon2;

    // Calculate initial stresses
    Pressure
        fc1 = CompressiveStress(ec1, ec2),
        fc2 = CompressiveStress(ec2, ec1);

    var tol = Pressure.FromMegapascals(0.01);

    // Iterate to find stresses (maximum 20 iterations)
    for (var it = 1; it <= 20; it++)
    {
        // Calculate confinement factors
        double
            betaL1 = ConfinementFactor(fc2, Parameters.Strength),
            betaL2 = ConfinementFactor(fc1, Parameters.Strength);

```

```

        // Calculate iteration stresses
        Pressure
            fc1It = CompressiveStress(ec1, ec2, betaL1),
            fc2It = CompressiveStress(ec2, ec1, betaL2);

        // Verify tolerances

        if ((fc1 - fc1It).Abs() <= tol && (fc2 - fc2It).Abs() <= tol)
            break;

        // Update stresses
        fc1 = fc1It;
        fc2 = fc2It;
    }

    return
        new PrincipalStressState(fc1, fc2, principalStrains.Theta1);
}

/// <summary>
///     Calculate tensile stress for <see cref="Material.Concrete.BiaxialConcrete"
///     /> case.
/// </summary>
/// <param name="strain">The tensile strain to calculate stress.</param>
/// <param name="transverseStrain">The strain at the transverse direction to
/// <paramref name="strain" />.</param>
/// <param name="theta1">The angle of <paramref name="strain" /> related to
/// horizontal axis, in radians.</param>
/// <param name="reinforcement">The <see cref="WebReinforcement" />.</param>
/// <param name="referenceLength">The reference length (only for <see
/// cref="DSFMConstitutive" />).</param>
private Pressure TensileStress(double strain, double transverseStrain, double
    theta1, WebReinforcement? reinforcement, Length? referenceLength = null)
{
    if (!strain.IsFinite() || !transverseStrain.IsFinite() || strain.ApproxZero())
        return Pressure.Zero;

    // Calculate initial uncracked state
    var fc1 = UncrackedStress(strain, transverseStrain);

    return !Cracked
        ? fc1
        : CrackedStress(strain, theta1, reinforcement, referenceLength);
}

/// <summary>
///     Calculate <see cref="Material.Concrete.BiaxialConcrete" /> tensile stress
///     for uncracked state.
/// </summary>
/// <param name="strain">The compressive strain (negative) to calculate
/// stress.</param>
/// <param name="transverseStrain">The strain at the transverse direction to
/// <paramref name="strain" />.</param>
private Pressure UncrackedStress(double strain, double transverseStrain)
{
    // Get strains
    double
        ec1 = strain,
        ec2 = transverseStrain;

```

```

        // Calculate initial uncracked state
        var fc1 = ec1 * Parameters.ElasticModule;

        // Verify if fc1 cracks concrete
        CheckCrackedState(fc1, ec2);

        return fc1;
    }

    /// <inheritdoc />
    public bool Equals(Constitutive? other) => other is not null && Model ==
        other.Model;
}

```

MCFT.cs

```

namespace andrefmello91.Material.Concrete;

public partial class BiaxialConcrete
{
    /// <summary>
    ///     MCFT constitutive class.
    /// </summary>
    protected class MCFTConstitutive : Constitutive
    {

        public override ConstitutiveModel Model { get; } = ConstitutiveModel.MCFT;

        /// <summary>
        ///     MCFT constitutive object.
        /// </summary>
        /// <inheritdoc cref="Constitutive(ICConcreteParameters)" />
        public MCFTConstitutive(ICConcreteParameters parameters) : base(parameters)
        {
        }

        /// <inheritdoc />
        protected override Pressure CompressiveStress(double strain, double
            transverseStrain, double deviationAngle = 0, double confinementFactor = 1)
        {
            if (!strain.IsFinite() || !transverseStrain.IsFinite() || strain.ApproxZero())
                return Pressure.Zero;

            // Get strains
            double
                ec1 = transverseStrain,
                ec2 = strain,
                ec = Parameters.PlasticStrain;

            var fc = Parameters.Strength;

            // Calculate the maximum concrete compressive stress
            Pressure
                f2maxA = ec1 > 0
                    ? -fc / (0.8 - 0.34 * ec1 / ec)
                    : -fc,
                f2max = f2maxA.Value < 0 && f2maxA.Value.IsFinite()
                    ? Max(f2maxA, -fc) * confinementFactor

```

```

        : -fc * confinementFactor;

        // Calculate the principal compressive stress in concrete
        var n = ec2 / ec;

        return
            f2max * (2 * n - n * n).AsFinite();
    }

    /// <inheritdoc />
    protected override Pressure CrackedStress(double strain, double theta1,
        WebReinforcement? reinforcement, Length? referenceLength = null) =>
        Parameters.TensileStrength / (1 + Math.Sqrt(500 * strain));
    }
}

```

DSFM.cs

```

namespace andrefmello91.Material.Concrete;

public partial class BiaxialConcrete
{
    /// <summary>
    ///     DSFM constitutive class.
    /// </summary>
    protected class DSFMConstitutive : Constitutive
    {
        public override ConstitutiveModel Model { get; }

        //
        /// <summary>
        ///     DSFM constitutive object.
        /// </summary>
        /// <param name="considerCrackSlip">Consider crack slip (default: true)</param>
        /// <inheritdoc cref="Constitutive(IConcreteParameters)" />
        public DSFMConstitutive(IConcreteParameters parameters, bool considerCrackSlip =
            true) : base(parameters) => ConsiderCrackSlip = considerCrackSlip;

        /// <summary>
        ///     Calculate tension stiffening coefficient (for DSFM).
        /// </summary>
        /// <inheritdoc cref="TensionStiffening" />
        private static double TensionStiffeningCoefficient(WebReinforcement?
            reinforcement, double theta1)
        {
            var x = reinforcement?.DirectionX;
            var y = reinforcement?.DirectionY;

            if (reinforcement is null || x is null && y is null)
                return 0;

            // Get reinforcement angles and stresses
            var (thetaNx, thetaNy) = reinforcement.Angles(theta1);

            double den = 0;

            if (x is not null)
            {
                double

```

```

        psx    = x.Ratio,
        phiX   = x.BarDiameter.Millimeters,
        cosNx  = thetaNx.Cos(true);

    den += psx / phiX * cosNx;
}

if (y is not null)
{
    double
        psy    = y.Ratio,
        phiY   = y.BarDiameter.Millimeters,
        cosNy  = thetaNy.Cos(true);

    den += psy / phiY * cosNy;
}

// Return m
return
    0.25 / den;
}

/// <inheritdoc />
protected override Pressure CompressiveStress(double strain, double
    transverseStrain, double deviationAngle = 0, double confinementFactor = 1)
{
    if (!strain.IsFinite() || !transverseStrain.IsFinite() || strain.ApproxZero())
        return Pressure.Zero;

    var fc = Parameters.Strength;

    // Get strains
    double
        ec1 = transverseStrain,
        ec2 = strain,
        ec  = Parameters.PlasticStrain;

    // Calculate beta D
    var betaD = SofteningFactor(ec2, ec1);

    // Calculate fp and ep
    var fp = -betaD * fc * confinementFactor;
    var ep = betaD * ec * confinementFactor;

    // Calculate parameters of concrete
    double
        k = ep <= ec2
            ? 1
            : 0.67 - fp.Megapascals / 62,
        n    = 0.8 - fp.Megapascals / 17,
        ec2_ep = ec2 / ep;

    // Calculate the principal compressive stress in concrete
    return
        fp * n * ec2_ep / (n - 1 + ec2_ep.Pow(n * k));
}

/// <inheritdoc />
protected override Pressure CrackedStress(double strain, double theta1,
    WebReinforcement? reinforcement, Length? referenceLength = null)

```



```

{
    // Calculate concrete post-cracking stress associated with tension softening
    var fc1a = TensionSoftening(strain, referenceLength!.Value);

    // Calculate concrete post-cracking stress associated with tension stiffening.
    var fc1b = TensionStiffening(strain, theta1, reinforcement);

    // Return maximum
    return
        Max(fc1a, fc1b);
}

/// <summary>
///     Calculate compression softening factor (beta D).
/// </summary>
/// <param name="strain">The compressive strain (negative) to calculate
///     stress.</param>
/// <param name="transverseStrain">The strain at the transverse direction to
///     <paramref name="strain" />.</param>
private double SofteningFactor(double strain, double transverseStrain)
{
    // Calculate strain ratio
    var r = Math.Min(-transverseStrain / strain, 400);

    if (r < 0.28) // Cd = 0
        return 1;

    // Calculate Cd and Cs
    var Cd = 0.35 * (r - 0.28).Pow(0.8);

    return
        Math.Min(1.0 / (1 + Cs * Cd), 1);
}

/// <summary>
///     Calculate concrete post-cracking stress associated with tension softening.
/// </summary>
/// <param name="strain">The tensile strain to calculate stress.</param>
/// <param name="referenceLength">The reference length.</param>
private Pressure TensionSoftening(double strain, Length referenceLength)
{
    var ft = Parameters.TensileStrength;

    double
        Gf = Parameters.FractureParameter.NewtonsPerMillimeter,
        ecr = Parameters.CrackingStrain,
        ets = 2.0 * Gf / (ft.Megapascals * referenceLength.Millimeters);

    return
        ft * (1.0 - (strain - ecr) / (ets - ecr));
}

/// <summary>
///     Calculate concrete post-cracking stress associated with tension stiffening
///     (for
///     <see cref="Material.Concrete.BiaxialConcrete" />).
/// </summary>
/// <param name="strain">The tensile strain to calculate stress.</param>
/// <param name="theta1">The angle of maximum principal strain, in radians.</param>
/// <param name="reinforcement">The <see cref="WebReinforcement" />.</param>

```

```

private Pressure TensionStiffening(double strain, double theta1, WebReinforcement?
    reinforcement)
{
    // Calculate coefficient for tension stiffening effect
    var m = TensionStiffeningCoefficient(reinforcement, theta1);

    // Calculate concrete postcracking stress associated with tension stiffening
    var fc1b = Parameters.TensileStrength / (1 + (2.2 * m * strain).Sqrt());

    if (reinforcement is null)
        return fc1b;

    // Check the maximum value of fc1 that can be transmitted across cracks
    var fc1s = reinforcement.MaximumPrincipalTensileStress(theta1);

    // Return minimum
    return
        Min(fc1b, fc1s);
}
}

```

SMM.cs

```

namespace andrefmello91.Material.Concrete;

public partial class BiaxialConcrete
{
    /// <summary>
    ///     SMM constitutive class.
    /// </summary>
    protected class SMMConstitutive : Constitutive
    {
        /// <summary>
        ///     <inheritdoc cref="StrengthFunction" />
        /// </summary>
        private readonly double _strengthFunction;

        public override ConstitutiveModel Model { get; } = ConstitutiveModel.SMM;

        /// <summary>
        ///     MCFT constitutive object.
        /// </summary>
        /// <inheritdoc cref="Constitutive(IConcreteParameters)" />
        public SMMConstitutive(IConcreteParameters parameters) : base(parameters) =>
            _strengthFunction = StrengthFunction(parameters.Strength);

        /// <summary>
        ///     Calculate the deviation angle function for the softening parameter.
        /// </summary>
        /// <param name="deviationAngle">The deviation angle between applied principal
        ///     stresses and concrete principal stresses.</param>
        private static double DeviationFunction(double deviationAngle) => 1D -
            deviationAngle.Abs() / 0.418879;

        /// <summary>
        ///     Calculate the strength function for the softening parameter.
        /// </summary>
        /// <param name="strength">The compressive strength of concrete.</param>

```

```

private static double StrengthFunction(Pressure strength) => Math.Min(5.8 /
    strength.Megapascals.Sqrt(), 0.9);

/// <summary>
///     Calculate the tensile strain function for the softening parameter.
/// </summary>
/// <param name="epsilon1">The average principal tensile strain, not considering
///     Poisson effect.</param>
private static double TensileStrainFunction(double epsilon1) => 1D / (1D + 400 *
    epsilon1)..Sqrt();

/// <inheritdoc />
protected override Pressure CompressiveStress(double strain, double
    transverseStrain, double deviationAngle = 0, double confinementFactor = 1)
{
    if (!strain.IsFinite() || !transverseStrain.IsFinite() || strain.ApproxZero())
        return Pressure.Zero;

    // Calculate softening coefficient
    var soft = (TensileStrainFunction(transverseStrain) * _strengthFunction *
        DeviationFunction(deviationAngle)).AsFinite();

    // Calculate peak stress and strain
    var fp = -soft * Parameters.Strength;
    var ep = soft * Parameters.PlasticStrain;

    // Calculate strain ratio:
    var e2_ep = (strain / ep).AsFinite();

    return
        (e2_ep <= 1) switch
        {
            { } when e2_ep < 0 => Pressure.Zero,

            // Pre-peak
            true => fp * (2 * e2_ep - e2_ep * e2_ep) * confinementFactor,

            // Post-peak
            _ => Max(fp * (1D - ((e2_ep - 1D) / (4D / soft - 1D)).Pow(2)), 0.5 * fp)
                * confinementFactor
        };
}

/// <inheritdoc />
protected override Pressure CrackedStress(double strain, double theta1,
    WebReinforcement? reinforcement, Length? referenceLength = null) =>
    Parameters.TensileStrength * (Parameters.CrackingStrain / strain).Pow(0.4);
}
}

```

C.1.3.3 Classes *WebReinforcementDirection* e *WebReinforcement*

A classe *WebReinforcementDirection* contém as propriedades de uma direção da armadura distribuída, como diâmetro das barras, espaçamento e ângulo da direção. A classe *WebReinforcement* têm como propriedades dois objetos *WebReinforcementDirection*, de modo a constituir uma armadura em malha.

ReinforcementDirection.cs

```

namespace andrefmello91.Material.Reinforcement;

/// <summary>
///     Reinforcement direction class for web reinforcement.
/// </summary>
public class WebReinforcementDirection : IUnitConvertible<LengthUnit>,
    IApproachable<WebReinforcementDirection, Length>,
    IEquatable<WebReinforcementDirection>, IComparable<WebReinforcementDirection>,
    ICloneable<WebReinforcementDirection>
{
    private Length _width;

    /// <summary>
    ///     Get the angle (in radians) of this <see cref="WebReinforcementDirection" />,
    ///     related to horizontal axis.
    /// </summary>
    /// <remarks>The angle is positive if counterclockwise.</remarks>
    public double Angle { get; }

    /// <summary>
    ///     The cross-section area of this reinforcement direction, per stirrup.
    /// </summary>
    public Area Area
    {
        get
        {
            var unit = Unit.GetAreaUnit();

            if (unit is AreaUnit.Undefined)
                unit = AreaUnit.SquareMillimeter;

            return
                (0.25 * NumberOfLegs * Constants.Pi * BarDiameter * BarDiameter)
                .ToUnit(unit);
        }
    }

    /// <summary>
    ///     Get the bar diameter.
    /// </summary>
    public Length BarDiameter { get; private set; }

    /// <summary>
    ///     Get the bar spacing.
    /// </summary>
    public Length BarSpacing { get; private set; }

    /// <summary>
    ///     Get reinforcement capacity reserve for tension.
    ///     <para><see cref="YieldStress" /> - <see cref="Stress" />.</para>
    /// </summary>
    public Pressure CapacityReserve => YieldStress - Stress.Abs();

    /// <summary>
    ///     Get reinforcement initial stiffness (ratio multiplied by steel elastic
    ///     module).
    /// </summary>

```

```

public Pressure InitialStiffness => Ratio * Steel.Parameters.ElasticModule;

/// <summary>
///     Returns true if <see cref="Angle" /> is approximately zero.
/// </summary>
public bool IsHorizontal => Angle.ApproxZero(1E-3);

/// <summary>
///     Returns true if <see cref="Angle" /> is approximately 90 degrees.
/// </summary>
public bool IsVertical => Angle.Approx(Constants.PiOver2, 1E-3);

/// <summary>
///     The number of stirrup legs/ branches.
/// </summary>
public int NumberOfLegs { get; }

/// <summary>
///     Get reinforcement ratio.
/// </summary>
public double Ratio => BarSpacing.ApproxZero(Tolerance) || Width.ApproxZero(Tolerance)
    ? 0
    : Area / (BarSpacing * Width);

/// <summary>
///     Get the steel object.
/// </summary>
public Steel Steel { get; }

/// <summary>
///     Get reinforcement stiffness (ratio multiplied by steel secant module).
/// </summary>
public Pressure Stiffness => Ratio * Steel.SecantModule;

/// <summary>
///     Get reinforcement stress (ratio multiplied by steel stress).
/// </summary>
public Pressure Stress => Ratio * Steel.Stress;

/// <summary>
///     Get/set the cross-section width.
/// </summary>
public Length Width
{
    get => _width;
    set => _width = value.ToUnit(Unit);
}

/// <inheritdoc cref="Reinforcement.Steel.Yielded" />
public bool Yielded => Steel.Yielded;

/// <summary>
///     Get reinforcement yield stress (ratio multiplied by steel yield stress).
/// </summary>
public Pressure YieldStress => Ratio * Steel.Parameters.YieldStress;

/// <summary>
///     Get/set the <see cref="LengthUnit" /> of <see cref="BarDiameter" />, <see
    cref="BarSpacing" /> and
///     <see cref="Width" />.

```

```

/// </summary>
public LengthUnit Unit
{
    get => BarDiameter.Unit;
    set => ChangeUnit(value);
}

/// <summary>
///     Reinforcement direction object for web reinforcement.
/// </summary>
/// <param name="barDiameter">The bar diameter.</param>
/// <param name="barSpacing">The bar spacing.</param>
/// <param name="steelParameters">The steel object (not null).</param>
/// <param name="width">The width of cross-section.</param>
/// <param name="angle">
///     The angle (in radians) of this <see cref="WebReinforcementDirection" />,
///     related to horizontal axis.
/// <para><paramref name="angle" /> is positive if counterclockwise.</para>
/// </param>
/// <param name="numberOfLegs">The number of stirrup legs/ branches. Default:
///     2.</param>
private WebReinforcementDirection(Length barDiameter, Length barSpacing,
    SteelParameters steelParameters, Length width, double angle, int numberOfLegs = 2)
{
    BarDiameter = barDiameter;
    BarSpacing = barSpacing.ToUnit(barDiameter.Unit);
    Steel = steelParameters;
    _width = width.ToUnit(barDiameter.Unit);
    Angle = angle;
    NumberOfLegs = numberOfLegs;
}

/// <inheritdoc cref="From(Length, Length, SteelParameters, Length, double, int)" />
/// <param name="unit">
///     The unit of <paramref name="barDiameter" />, <paramref name="barSpacing" />
///     and
///     <paramref name="width" />.
/// </param>
public static WebReinforcementDirection? From(double barDiameter, double barSpacing,
    SteelParameters steelParameters, double width, double angle, int numberOfLegs =
    2, LengthUnit unit = LengthUnit.Millimeter) =>
    From((Length) barDiameter.As(unit), (Length) barSpacing.As(unit), steelParameters,
        (Length) width.As(unit), angle, numberOfLegs);

/// <summary>
///     Get a <see cref="WebReinforcementDirection" />.
/// </summary>
/// <returns>
///     Null any of <paramref name="barDiameter" /> or <paramref name="barSpacing" />
///     is zero.
/// </returns>
/// <inheritdoc cref="WebReinforcementDirection(Length, Length, SteelParameters,
///     Length, double, int)" select="params" />
public static WebReinforcementDirection? From(Length barDiameter, Length barSpacing,
    SteelParameters steelParameters, Length width, double angle, int numberOfLegs =
    2) =>
    barDiameter.ApproxZero(Tolerance) || barSpacing.ApproxZero(Tolerance)
    ? null
    : new WebReinforcementDirection(barDiameter, barSpacing, steelParameters,
        width, angle, numberOfLegs);

```

```

/// <summary>
///     Set steel strain and stress.
/// </summary>
/// <param name="strain">Current strain.</param>
public void Calculate(double strain) => Steel.Calculate(strain);

/// <inheritdoc cref="IUnitConvertible{TUnit}.Convert" />
public WebReinforcementDirection Convert(LengthUnit unit) =>
    new(BarDiameter.ToUnit(unit), BarSpacing.ToUnit(unit), Steel.Parameters.Clone(),
        Width.ToUnit(unit), Angle);

/// <summary>
///     Calculate the crack spacing at this direction, according to Kaklauskas (2019)
///     expression.
/// </summary>
public Length CrackSpacing() =>
    BarDiameter.ApproxZero(Tolerance) || Ratio.ApproxZero()
    ? Length.FromMillimeters(21)
    : Length.FromMillimeters(21) + 0.155 * BarDiameter / Ratio;

/// <inheritdoc />
public override bool Equals(object? other) => other is WebReinforcementDirection
    reinforcement && Equals(reinforcement);

/// <summary>
///     Compare two reinforcement objects.
/// </summary>
/// <remarks>
///     Returns true if <see cref="BarDiameter" /> and <see cref="BarSpacing" /> are
///     nearly equal.
/// </remarks>
/// <param name="other">The other reinforcement object.</param>
/// <param name="tolerance">The tolerance to consider values being equal.</param>
public virtual bool EqualsDiameterAndSpacing(WebReinforcementDirection? other,
    Length? tolerance = null) => other is not null &&
    BarDiameter.Approx(other.BarDiameter, tolerance ?? Tolerance) &&
    BarSpacing.Approx(other.BarSpacing, tolerance ?? Tolerance);

/// <inheritdoc />
public override int GetHashCode() => NumberOfLegs * (int)
    BarDiameter.Millimeters.Pow(BarSpacing.Millimeters);

/// <inheritdoc />
public override string ToString()
{
    var rho = (char) Characters.Rho;
    var phi = (char) Characters.Phi;

    return
        $"{phi}_{BarDiameter}\n" +
        $"{s}_{BarSpacing}\n" +
        $"{rho}_{Ratio:P}\n" +
        $"Angle_{Angle.ToDegree():0.00}deg\n" +
        Steel;
}

/// <inheritdoc />

```

```

public bool Approaches(WebReinforcementDirection? other, Length tolerance) => other
    is not null && NumberOfLegs == other.NumberOfLegs &&
    EqualsDiameterAndSpacing(other, tolerance);

/// <inheritdoc />
public WebReinforcementDirection Clone() => new(BarDiameter, BarSpacing,
    Steel.Parameters.Clone(), Width, Angle);

/// <inheritdoc />
public int CompareTo(WebReinforcementDirection? other) =>
    other is null || Area > other.Area || BarDiameter > other.BarDiameter ||
    BarDiameter.Approx(other.BarDiameter, Tolerance) && BarSpacing >
    other.BarSpacing
    ? 1
    : Equals(other)
    ? 0
    : -1;

/// <summary>
///     Compare two reinforcement objects.
///     <para>Returns true if parameters are equal.</para>
/// </summary>
/// <param name="other">The other reinforcement object.</param>
public virtual bool Equals(WebReinforcementDirection? other) => Approaches(other,
    Tolerance);

/// <inheritdoc />
public void ChangeUnit(LengthUnit unit)
{
    if (Unit == unit)
        return;

    BarDiameter = BarDiameter.ToUnit(unit);
    BarSpacing = BarSpacing.ToUnit(unit);
    _width = _width.ToUnit(unit);
}

IUnitConvertible<LengthUnit> IUnitConvertible<LengthUnit>.Convert(LengthUnit unit) =>
    Convert(unit);

/// <summary>
///     Returns true if steel parameters are equal.
/// </summary>
public static bool operator ==(WebReinforcementDirection? left,
    WebReinforcementDirection? right) => left.IsEqualTo(right);

/// <summary>
///     Returns true if steel parameters are different.
/// </summary>
public static bool operator !=(WebReinforcementDirection? left,
    WebReinforcementDirection? right) => left.IsNotEqualTo(right);
}

```

WebReinforcement.cs

```

namespace andrefmello91.Material.Reinforcement;

/// <summary>
///     Web reinforcement class.
/// </summary>

```



```

public class WebReinforcement : IBiaxialMaterial, IUnitConvertible<LengthUnit>,
    IApproachable<WebReinforcement, Length>, IComparable<WebReinforcement>,
    IEquatable<WebReinforcement>, ICloneable<WebReinforcement>
{
    private Length _width;

    // Properties
    /// <summary>
    ///     Get the <see cref="WebReinforcementDirection" /> on X direction.
    /// </summary>
    public WebReinforcementDirection? DirectionX { get; }

    /// <summary>
    ///     Get the <see cref="WebReinforcementDirection" /> on Y direction.
    /// </summary>
    public WebReinforcementDirection? DirectionY { get; }

    /// <summary>
    ///     Get initial <see cref="WebReinforcement" /> stiffness <see cref="Matrix" />.
    /// </summary>
    /// <inheritdoc cref="BiaxialConcrete.InitialStiffness" />
    public MaterialMatrix InitialStiffness
    {
        get
        {
            // Steel matrix
            var Ds = MaterialMatrix.Zero();

            if (DirectionX is not null)
                Ds[0, 0] = DirectionX.InitialStiffness;

            if (DirectionY is not null)
                Ds[1, 1] = DirectionY.InitialStiffness;

            if ((DirectionX is null || DirectionX.IsHorizontal) && (DirectionY is null ||
                DirectionY.IsVertical))
                return Ds;

            // Transform
            var t = StrainRelations.TransformationMatrix(DirectionX?.Angle ??
                DirectionY?.Angle - Constants.PiOver2 ?? 0);

            return
                (MaterialMatrix) Ds.Transform(t);
        }
    }

    /// <summary>
    ///     Get cross-section width.
    /// </summary>
    public Length Width
    {
        get => DirectionX?.Width ?? _width;
        set
        {
            if (DirectionX is not null)
                DirectionX.Width = value;

            if (DirectionY is not null)

```

```

        DirectionY.Width = value;
    }
}

/// <summary>
///     Returns true if reinforcement <see cref="DirectionX" /> exists.
/// </summary>
public bool XReinforced => DirectionX is not null && DirectionX.BarDiameter >
    Length.Zero && DirectionX.BarSpacing > Length.Zero;

/// <summary>
///     Returns true if reinforcement <see cref="DirectionX" /> and <see
///     cref="DirectionY" /> exist.
/// </summary>
public bool XYReinforced => XReinforced && YReinforced;

/// <summary>
///     Returns true if reinforcement <see cref="DirectionY" /> exists.
/// </summary>
public bool YReinforced => DirectionY is not null && DirectionY.BarDiameter >
    Length.Zero && DirectionY.BarSpacing > Length.Zero;

/// <summary>
///     Get current <see cref="WebReinforcement" /> stiffness <see cref="Matrix" />
///     with elements in
///     <see cref="PressureUnit.Megapascal" />.
/// </summary>
public MaterialMatrix Stiffness
{
    get
    {
        // Steel matrix
        var Ds = MaterialMatrix.Zero();

        if (DirectionX is not null)
            Ds[0, 0] = DirectionX.Stiffness;

        if (DirectionY is not null)
            Ds[1, 1] = DirectionY.Stiffness;

        if ((DirectionX is null || DirectionX.IsHorizontal) && (DirectionY is null ||
            DirectionY.IsVertical))
            return Ds;

        // Transform
        var t = StrainRelations.TransformationMatrix(DirectionX?.Angle ??
            DirectionY?.Angle - Constants.PiOver2 ?? 0);

        return
            (MaterialMatrix) Ds.Transform(t);
    }
}

/// <summary>
///     Get/set reinforcement <see cref="StrainState" />, at horizontal plane.
/// </summary>
public StrainState Strains { get; private set; }

/// <summary>
///     Get reinforcement <see cref="StressState" />, transformed to horizontal plane.

```

```

/// </summary>
public StressState Stresses
{
    get
    {
        Pressure
            fsx = DirectionX?.Stress ?? Pressure.Zero,
            fsy = DirectionY?.Stress ?? Pressure.Zero;

        var stresses = new StressState(fsx, fsy, Pressure.Zero);

        if ((DirectionX is null || DirectionX.IsHorizontal) && (DirectionY is null ||
            DirectionY.IsVertical))
            return stresses;

        return
            stresses.Transform(-DirectionX?.Angle ?? 0);
    }
}

/// <inheritdoc />
PrincipalStrainState IBiaxialMaterial.PrincipalStrains => (PrincipalStrainState)
    Strains;

/// <inheritdoc />
PrincipalStressState IBiaxialMaterial.PrincipalStresses => (PrincipalStressState)
    Stresses;

/// <inheritdoc />
public LengthUnit Unit
{
    get => DirectionX?.Unit ?? DirectionY?.Unit ?? Width.Unit;
    set => ChangeUnit(value);
}

/// <summary>
///     Create a web reinforcement, with different X and Y directions.
/// </summary>
/// <remarks>
///     Two reinforcing bars are considered at cross-section, one in each lateral
///     face of structural element.
/// </remarks>
/// <param name="directionX">
///     The <see cref="WebReinforcementDirection" /> of X direction. Can be null if
///     there is no
///     reinforcement in this direction.
/// </param>
/// <param name="directionY">
///     The <see cref="WebReinforcementDirection" /> of Y direction. Can be null if
///     there is no
///     reinforcement in this direction.
/// </param>
/// <param name="width">The width of cross-section.</param>
public WebReinforcement(WebReinforcementDirection? directionX,
    WebReinforcementDirection? directionY, Length width)
{
    DirectionX = directionX;
    DirectionY = directionY;
    _width     = width;
}

```

```

        if (DirectionX is not null && DirectionX.Width != width)
            DirectionX.Width = width;

        if (DirectionY is not null && DirectionY.Width != width)
            DirectionY.Width = width;
    }

    private WebReinforcement(WebReinforcementDirection? directionX,
        WebReinforcementDirection? directionY, double width, LengthUnit unit =
            LengthUnit.Millimeter)
        : this(directionX, directionY, (Length) width.As(unit))
    {
    }

    /// <inheritdoc cref="From(Length, Length, SteelParameters, Length, double, int)" />
    /// <param name="unit">
    ///     The <see cref="LengthUnit" /> of <paramref name="barDiameter" />, <paramref
    ///     name="barSpacing" /> and
    ///     <paramref name="width" />.
    /// </param>
    public static WebReinforcement From(double barDiameter, double barSpacing,
        SteelParameters steel, double width, double angleX = 0, int numberOfLegs = 2,
        LengthUnit unit = LengthUnit.Millimeter) =>
        From((Length) barDiameter.As(unit), (Length) barSpacing.As(unit), steel, (Length)
            width.As(unit), angleX, numberOfLegs);

    /// <summary>
    ///     Create a web reinforcement, with equal X and Y directions.
    /// </summary>
    /// <inheritdoc cref="WebReinforcement(WebReinforcementDirection,
    ///     WebReinforcementDirection, Length)" />
    /// <param name="barDiameter">The bar diameter for directions X and Y.</param>
    /// <param name="barSpacing">The bar spacing for directions X and Y.</param>
    /// <param name="steel">The steel objects for directions X and Y.</param>
    /// <param name="width">The width of cross-section.</param>
    /// <param name="angleX">
    ///     The angle (in radians) of <see cref="DirectionX" />, related to horizontal
    ///     axis.
    /// <para><paramref name="angleX" /> is positive if counterclockwise.</para>
    /// </param>
    /// <param name="numberOfLegs">The number of stirrup legs/ branches.</param>
    public static WebReinforcement From(Length barDiameter, Length barSpacing,
        SteelParameters steel, Length width, double angleX = 0, int numberOfLegs = 2) =>
        new(WebReinforcementDirection.From(barDiameter, barSpacing, steel, width, angleX,
            numberOfLegs), WebReinforcementDirection.From(barDiameter, barSpacing,
            steel.Clone(), width, angleX + Constants.PiOver2, numberOfLegs), width);

    /// <inheritdoc
    ///
    ///     cref="XOnly(UnitsNet.Length, UnitsNet.Length, andrefmello91.Material.Reinforcement.SteelParameters
    ///     />
    /// <param name="unit">
    ///     The <see cref="LengthUnit" /> of <paramref name="barDiameter" />, <paramref
    ///     name="barSpacing" /> and
    ///     <paramref name="width" />.
    /// </param>

```

```

public static WebReinforcement XOnly(double barDiameter, double barSpacing,
    SteelParameters steel, double width, double angleX = 0, int numberOfLegs = 2,
    LengthUnit unit = LengthUnit.Millimeter) =>
    new(WebReinforcementDirection.From(barDiameter, barSpacing, steel, width, angleX,
        numberOfLegs, unit), null, width, unit);

/// <summary>
///     Get a <see cref="WebReinforcement" /> with <see cref="DirectionX" /> only.
/// </summary>
/// <inheritdoc cref="From(Length, Length, SteelParameters, Length, double, int)" />
/// <param name="barDiameter">The bar diameter for X direction.</param>
/// <param name="barSpacing">The bar spacing for X direction.</param>
/// <param name="steel">The steel objects for X direction.</param>
public static WebReinforcement XOnly(Length barDiameter, Length barSpacing,
    SteelParameters steel, Length width, double angleX = 0, int numberOfLegs = 2) =>
    new(WebReinforcementDirection.From(barDiameter, barSpacing, steel, width, angleX,
        numberOfLegs), null, width);

/// <inheritdoc
///
    cref="YOnly(UnitsNet.Length, UnitsNet.Length, andrefmello91.Material.Reinforcement.SteelParameters
    />
/// <param name="unit">
///     The <see cref="LengthUnit" /> of <paramref name="barDiameter" />, <paramref
    name="barSpacing" /> and
///     <paramref name="width" />.
/// </param>
public static WebReinforcement YOnly(double barDiameter, double barSpacing,
    SteelParameters steel, double width, double angle = Constants.PiOver2, int
    numberOfLegs = 2, LengthUnit unit = LengthUnit.Millimeter) =>
    new(null, WebReinforcementDirection.From(barDiameter, barSpacing, steel, width,
        angle, numberOfLegs, unit), width, unit);

/// <summary>
///     Get a <see cref="WebReinforcement" /> with <see cref="DirectionY" /> only.
/// </summary>
/// <inheritdoc cref="From(Length, Length, SteelParameters, Length, double, int)" />
/// <param name="barDiameter">The bar diameter for Y direction.</param>
/// <param name="barSpacing">The bar spacing for Y direction.</param>
/// <param name="steel">The steel objects for Y direction.</param>
/// <param name="angleY">
///     The angle (in radians) of <see cref="DirectionY" />, related to horizontal
    axis.
///     <para><paramref name="angleY" /> is positive if counterclockwise.</para>
/// </param>
/// <inheritdoc cref="WebReinforcement(Length, Length, Steel, Length, double)" />
public static WebReinforcement YOnly(Length barDiameter, Length barSpacing,
    SteelParameters steel, Length width, double angleY = Constants.PiOver2, int
    numberOfLegs = 2) =>
    new(null, WebReinforcementDirection.From(barDiameter, barSpacing, steel, width,
        angleY, numberOfLegs), width);

/// <summary>
///     Calculate angles (in radians) related to crack angle.
/// </summary>
/// <param name="theta1">Principal tensile strain angle, in radians.</param>
public (double X, double Y) Angles(double theta1)
{
    // Calculate angles
    double

```

```

        thetaNx = theta1 - (DirectionX?.Angle ?? 0),
        thetaNy = theta1 - (DirectionY?.Angle ?? Constants.PiOver2);

    return
        (thetaNx, thetaNy);
}

/// <inheritdoc cref="IUnitConvertible{TUnit}.Convert" />
public WebReinforcement Convert(LengthUnit unit) => new(DirectionX?.Convert(unit),
    DirectionY?.Convert(unit), _width.ToUnit(unit));

/// <inheritdoc />
public override bool Equals(object? other) => other is WebReinforcement reinforcement
    && Equals(reinforcement);

/// <inheritdoc />
public override int GetHashCode() => DirectionX?.GetHashCode() ?? 1 *
    DirectionY?.GetHashCode() ?? 1 * (int) Width.Millimeters;

/// <summary>
///     Calculate maximum value of principal tensile strength (fc1) that can be
///     transmitted across cracks.
/// </summary>
/// <param name="theta1">Principal tensile strain angle, in radians.</param>
public Pressure MaximumPrincipalTensileStress(double theta1)
{
    if (DirectionX is null && DirectionY is null)
        return Pressure.Zero;

    // Get reinforcement angles and stresses
    var (thetaNx, thetaNy) = Angles(theta1);

    Pressure
        fcx = DirectionX?.CapacityReserve ?? Pressure.Zero,
        fcy = DirectionY?.CapacityReserve ?? Pressure.Zero;

    double
        cosNx = thetaNx.Cos(true),
        cosNy = thetaNy.Cos(true);

    // Check the maximum value of fc1 that can be transmitted across cracks
    return
        fcx * cosNx * cosNx + fcy * cosNy * cosNy;
}

/// <inheritdoc />
public override string ToString() =>
    "Reinforcement_␣(x):␣\n" +
    $"{DirectionX?.ToString()␣??␣"null"}␣\n\n" +
    "Reinforcement_␣(y):␣\n" +
    $"{DirectionY?.ToString()␣??␣"null"}";

/// <inheritdoc />
public bool Approaches(WebReinforcement? other, Length tolerance) => other is not
    null && (DirectionX?.Approaches(other.DirectionX, tolerance) ?? false) &&
    (DirectionY?.Approaches(other.DirectionY, tolerance) ?? false);

/// <inheritdoc />
public void Calculate(StrainState strainsState)
{

```

```

    // Set strains
    Strains = strainsState.ThetaX.Approx(DirectionX?.Angle ?? 0, 1E-3)
        ? strainsState
        : strainsState.Transform((DirectionX?.Angle ?? 0) - strainsState.ThetaX);

    // Calculate stresses
    DirectionX?.Calculate(Strains.EpsilonX);
    DirectionY?.Calculate(Strains.EpsilonY);
}

/// <inheritdoc />
public WebReinforcement Clone() => new(DirectionX?.Clone(), DirectionY?.Clone(),
    Width);

/// <inheritdoc />
public int CompareTo(WebReinforcement? other)
{
    if (other is null)
        return 1;

    int
        x = DirectionX?.CompareTo(other?.DirectionX) ?? -1,
        y = DirectionY?.CompareTo(other?.DirectionY) ?? -1;

    return x switch
    {
        1          => 1,
        0 when y == 0 => 0,
        0 when y > 0  => 1,
        -          => -1
    };
}

/// <summary>
///     Compare two reinforcement objects.
///     <para>Returns true if parameters are equal.</para>
/// </summary>
/// <param name="other">The other reinforcement object.</param>
public virtual bool Equals(WebReinforcement? other) => Approaches(other, Tolerance);

/// <inheritdoc />
public void ChangeUnit(LengthUnit unit)
{
    if (Unit == unit)
        return;

    if (DirectionX is not null)
        DirectionX.Unit = unit;

    if (DirectionY is not null)
        DirectionY.Unit = unit;

    _width = _width.ToUnit(unit);
}

IUnitConvertible<LengthUnit> IUnitConvertible<LengthUnit>.Convert(LengthUnit unit) =>
    Convert(unit);

/// <summary>
///     Returns true if objects are equal.

```

```

    /// </summary>
    public static bool operator ==(WebReinforcement? left, WebReinforcement? right) =>
        left.IsEqualTo(right);

    /// <summary>
    ///     Returns true if objects are different.
    /// </summary>
    public static bool operator !=(WebReinforcement? left, WebReinforcement? right) =>
        left.IsNotEqualTo(right);
}

```

C.2 RCMembrane

A biblioteca *RCMembrane* (MELLO, 2020b) foi descrita no Capítulo 4, sendo formada pelas classes *Membrane* e derivadas, e *MembraneSolver*.

C.2.1 Classe *Membrane* e derivadas

A classe base *Membrane* é principalmente composta pelas propriedades *BiaxialConcrete* e *WebReinforcement*, além dos estados médios de tensão e deformação. Fornece os métodos necessários para o cálculo de tensões e deformações para todas as classes derivadas: *MCFTMembrane*, *DSFMMembrane* e *SMMMembane*.

Membrane.cs

```

namespace andrefmello91.ReinforcedConcreteMembrane;

/// <summary>
///     Membrane element base class.
/// </summary>
public abstract class Membrane : IBiaxialMaterial, IEquatable<Membrane>,
    ICloneable<Membrane>
{

    /// <summary>
    ///     Get/set the average <see cref="PrincipalStrainState" />.
    /// </summary>
    public virtual PrincipalStrainState AveragePrincipalStrains { get; protected set; }

    /// <summary>
    ///     Get/set the average <see cref="StrainState" /> in the membrane element.
    /// </summary>
    public StrainState AverageStrains { get; protected set; }

    /// <summary>
    ///     Get average <see cref="StressState" />.
    /// </summary>
    public StressState AverageStresses => Reinforcement is null
        ? Concrete.Stresses
        : Concrete.Stresses + Reinforcement.Stresses;

```



```

/// <summary>
///     Get <see cref="BiaxialConcrete" /> of the membrane element.
/// </summary>
public BiaxialConcrete Concrete { get; protected set; }

/// <summary>
///     Get initial <see cref="Membrane" /> stiffness <see cref="Matrix" />.
/// </summary>
/// <inheritdoc cref="BiaxialConcrete.InitialStiffness" />
public MaterialMatrix InitialStiffness => Reinforcement is null
    ? Concrete.InitialStiffness
    : (MaterialMatrix) (Concrete.InitialStiffness + Reinforcement.InitialStiffness);

/// <summary>
///     Get reference length.
/// </summary>
public Length ReferenceLength => 0.5 * CrackSpacing();

/// <summary>
///     Get/set <see cref="WebReinforcement" /> of the membrane element.
/// </summary>
public WebReinforcement? Reinforcement { get; }

/// <summary>
///     Get the width of the membrane element.
/// </summary>
public Length Width { get; }

/// <summary>
///     Get current <see cref="Membrane" /> stiffness <see cref="Matrix" />.
/// </summary>
public MaterialMatrix Stiffness => Reinforcement is null
    ? Concrete.Stiffness
    : (MaterialMatrix) (Concrete.Stiffness + Reinforcement.Stiffness);

/// <inheritdoc />
PrincipalStrainState IBiaxialMaterial.PrincipalStrains => AveragePrincipalStrains;

/// <inheritdoc />
PrincipalStressState IBiaxialMaterial.PrincipalStresses =>
    AverageStresses.ToPrincipal();

/// <inheritdoc />
StrainState IBiaxialMaterial.Strains => AverageStrains;

/// <inheritdoc />
StressState IBiaxialMaterial.Stresses => AverageStresses;

/// <inheritdoc cref="Membrane(IConcreteParameters, WebReinforcement?, Length,
    ConstitutiveModel)" />
/// <param name="unit">The <see cref="LengthUnit" /> of <paramref name="width"
    /></param>
protected Membrane(IConcreteParameters concreteParameters, WebReinforcement?
    reinforcement, double width, ConstitutiveModel model, LengthUnit unit =
    LengthUnit.Millimeter)
    : this(concreteParameters, reinforcement, (Length) width.As(unit), model)
{
}

/// <summary>

```

```

///      Base membrane element constructor.
/// </summary>
/// <param name="concreteParameters">Concrete <see cref="Parameters" />
///      object.</param>
/// <param name="reinforcement"><see cref="WebReinforcement" /> object .</param>
/// <param name="width">The width of cross-section.</param>
/// <param name="model">Concrete <see cref="ConstitutiveModel" />.</param>
protected Membrane(ICConcreteParameters concreteParameters, WebReinforcement?
    reinforcement, Length width, ConstitutiveModel model)
{
    // Initiate new materials
    Concrete      = BiaxialConcrete.From(concreteParameters, model);
    Reinforcement = reinforcement;

    Width = width;

    // Set initial strains
    AverageStrains = StrainState.Zero;
}

/// <summary>
///      Calculate the average crack opening.
/// </summary>
/// <param name="reinforcement">The <see cref="WebReinforcement" /> object.</param>
/// <param name="concreteStrains">The <see cref="PrincipalStrainState" /> in
///      concrete.</param>
public static Length CrackOpening(WebReinforcement? reinforcement,
    PrincipalStrainState concreteStrains) =>
    concreteStrains.Epsilon1 <= 0 || concreteStrains.Epsilon1.ApproxZero(1E-9)
        ? Length.Zero
        : concreteStrains.Epsilon1 * CrackSpacing(reinforcement, concreteStrains);

/// <summary>
///      Create a membrane element based on concrete constitutive model.
/// </summary>
/// <param name="considerCrackSlip">Consider crack slip? Only for <see
///      cref="DSFMMembrane" /> (default: true)</param>
/// <inheritdoc cref="Membrane(ICConcreteParameters, WebReinforcement, double,
///      ConstitutiveModel, LengthUnit)" />
public static Membrane From(ICConcreteParameters concreteParameters, WebReinforcement?
    reinforcement, double width, ConstitutiveModel model = ConstitutiveModel.SMM,
    LengthUnit unit = LengthUnit.Millimeter, bool considerCrackSlip = true) =>
    From(concreteParameters, reinforcement, (Length) width.As(unit), model,
        considerCrackSlip);

/// <inheritdoc cref="From(ICConcreteParameters, WebReinforcement?, double,
///      ConstitutiveModel, LengthUnit, bool)" />
public static Membrane From(ICConcreteParameters concreteParameters, WebReinforcement?
    reinforcement, Length width, ConstitutiveModel model = ConstitutiveModel.SMM,
    bool considerCrackSlip = true) =>
    model switch
    {
        ConstitutiveModel.SMM => new SMMMembrane(concreteParameters, reinforcement,
            width),
        ConstitutiveModel.MCFT => new MCFTMembrane(concreteParameters, reinforcement,
            width),
        -                       => new DSFMMembrane(concreteParameters, reinforcement,
            width, considerCrackSlip)
    };

```

```

/// <summary>
///     Calculate the crack spacing in principal strain direction.
/// </summary>
/// <inheritdoc cref="CrackOpening(WebReinforcement, PrincipalStrainState)" />
private static Length CrackSpacing(WebReinforcement? reinforcement,
    PrincipalStrainState concreteStrains)
{
    // Get the angles
    var (cosThetaC, sinThetaC) = concreteStrains.Theta1.DirectionCosines(true);

    // Calculate crack spacings in X and Y
    double
        smx = reinforcement?.DirectionX?.CrackSpacing().Millimeters ?? 21,
        smy = reinforcement?.DirectionY?.CrackSpacing().Millimeters ?? 21,
        sm = 1.00 / (sinThetaC / smx + cosThetaC / smy);

    // Calculate crack spacing
    return
        (Length) sm.As(LengthUnit.Millimeter);
}

/// <summary>
///     Calculate maximum shear stress on crack, in MPa.
/// </summary>
/// <param name="crackOpening">Average crack opening, in mm.</param>
/// <param name="parameters">Concrete parameters.</param>
private static Pressure MaximumShearOnCrack(Length crackOpening, IConcreteParameters
    parameters)
{
    var vcimax = 0.18 * parameters.Strength.Megapascals.Sqrt()
        / (0.31 + 24 * crackOpening.Millimeters /
            (parameters.AggregateDiameter.Millimeters + 16));

    return Pressure.FromMegapascals(vcimax);
}

/// <inheritdoc />
public override bool Equals(object? obj) => obj is Membrane other && Equals(other);

/// <inheritdoc />
public override int GetHashCode() => Concrete.GetHashCode() +
    Reinforcement.GetHashCode();

/// <inheritdoc />
public override string ToString() =>
    "Membrane\n" +
    $"Width_{Width}\n" +
    $"{Concrete}\n" +
    $"{Reinforcement}\n";

/// <summary>
///     Limit tensile principal stress by crack check procedure, by Bentz (2000).
/// </summary>
protected void CrackCheck()
{
    // Verify if concrete is cracked
    if (!Concrete.Cracked)
        return;

    // Get concrete tensile stress

```

```

var f1a = Concrete.PrincipalStresses.Sigma1;

// Get reinforcement angles related to crack
var (thetaNx, thetaNy) = Reinforcement?.Angles(Concrete.PrincipalStrains.Theta1)
    ?? (Concrete.PrincipalStrains.Theta1, Concrete.PrincipalStrains.Theta1 -
        Constants.PiOver2);
double
    cosNx = thetaNx.Cos(true),
    cosNy = thetaNy.Cos(true),
    tanNx = thetaNx.Tan(true),
    tanNy = thetaNy.Tan(true);

// Reinforcement capacity reserve
Pressure
    f1cx = Reinforcement?.DirectionX?.CapacityReserve ?? Pressure.Zero,
    f1cy = Reinforcement?.DirectionY?.CapacityReserve ?? Pressure.Zero;

// Maximum possible shear on crack interface
var vcimaxA = MaximumShearOnCrack();

// Maximum possible shear for biaxial yielding
var vcimaxB = (f1cx - f1cy).Abs() / (tanNx + tanNy);

// Maximum shear on crack
var vcimax = Min(vcimaxA, vcimaxB);

// Biaxial yielding condition
var f1b = f1cx * cosNx * cosNx + f1cy * cosNy * cosNy;

// Maximum tensile stress for equilibrium in X and Y
Pressure
    f1c = f1cx + vcimax * tanNx,
    f1d = f1cy + vcimax * tanNy;

// Calculate the minimum tensile stress
var f1List = new[] { f1a, f1b, f1c, f1d };
var fc1 = f1List.Min();

// Set to concrete
if (fc1 < f1a)
    Concrete.SetTensileStress(fc1);
}

/// <summary>
///     Calculate the average crack opening.
/// </summary>
protected Length CrackOpening() => CrackOpening(Reinforcement,
    Concrete.PrincipalStrains);

/// <summary>
///     Calculate the crack spacing in principal strain direction.
/// </summary>
protected Length CrackSpacing() => CrackSpacing(Reinforcement,
    Concrete.PrincipalStrains);

/// <summary>
///     Calculate maximum shear stress on crack, in MPa.
/// </summary>
protected Pressure MaximumShearOnCrack() => MaximumShearOnCrack(CrackOpening(),
    Concrete.Parameters);

```

```

/// <summary>
///     Calculate <see cref="AverageStresses" /> and <see cref="Stiffness" />, given
///     a known <see cref="StrainState" />.
/// </summary>
/// <param name="appliedStrains">Current applied <see cref="StrainState" />.</param>
public abstract void Calculate(StrainState appliedStrains);

/// <inheritdoc />
public abstract Membrane Clone();

/// <summary>
///     Compare two <see cref="Membrane" /> objects.
///     <para>Returns true if <see cref="Concrete" /> and <see cref="Reinforcement"
///     /> are equal.</para>
/// </summary>
/// <param name="other">The other <see cref="Membrane" /> object.</param>
public bool Equals(Membrane? other) => other is not null && Concrete ==
    other.Concrete && Reinforcement == other.Reinforcement;

/// <summary>
///     Returns true if parameters and constitutive model are equal.
/// </summary>
public static bool operator ==(Membrane? left, Membrane? right) =>
    left.IsEqualTo(right);

/// <summary>
///     Returns true if parameters and constitutive model are different.
/// </summary>
public static bool operator !=(Membrane? left, Membrane? right) =>
    left.IsNotEqualTo(right);
}

```

MCFT.cs

```

namespace andrefmello91.ReinforcedConcreteMembrane;

/// <summary>
///     MCFT class, based on formulation by Vecchio and Collins (1986).
/// </summary>
internal class MCFTMembrane : Membrane
{
    /// <inheritdoc />
    public override PrincipalStrainState AveragePrincipalStrains =>
        Concrete.PrincipalStrains;

    /// <inheritdoc cref="MCFTMembrane(ICConcreteParameters, WebReinforcement, Length)" />
    /// <param name="unit">The <see cref="LengthUnit" /> of <paramref name="width"
    /// </param>
    internal MCFTMembrane(ICConcreteParameters concreteParameters, WebReinforcement?
        reinforcement, double width, LengthUnit unit = LengthUnit.Millimeter)
        : this(concreteParameters, reinforcement, (Length) width.As(unit))
    {
    }

    /// <summary>
    ///     Membrane element for MCFT analysis.
    /// </summary>

```

```

/// <inheritdoc cref="Membrane(IConcreteParameters, WebReinforcement?, Length,
    ConstitutiveModel)" />
internal MCFTMembrane(IConcreteParameters concreteParameters, WebReinforcement?
    reinforcement, Length width)
    : base(concreteParameters, reinforcement, width, ConstitutiveModel.MCFT)
{
}

/// <summary>
///     Calculate <see cref="StressState" /> and <see cref="Membrane.Stiffness" /> by
///     MCFT, given a known
///     <see cref="StrainState" />.
/// </summary>
/// <inheritdoc />
public override void Calculate(StrainState appliedStrains)
{
    AverageStrains = appliedStrains.Clone();

    // Calculate and set concrete and steel stresses
    Concrete.Calculate(AverageStrains, Reinforcement);
    Reinforcement?.Calculate(AverageStrains);

    // Verify if concrete is cracked and check crack stresses to limit  $f_{c1}$ 
    CrackCheck();
}

/// <inheritdoc />
public override Membrane Clone() => new MCFTMembrane(Concrete.Parameters,
    Reinforcement?.Clone(), Width);

/// <inheritdoc />
public override bool Equals(object? obj) => obj is MCFTMembrane other &&
    base.Equals(other);

/// <inheritdoc />
public override int GetHashCode() => base.GetHashCode();
}

```

DSFM.cs

```

namespace andrefmello91.ReinforcedConcreteMembrane;

/// <summary>
///     DSFM class, based on formulation by Vecchio (2000).
/// </summary>
internal class DSFMMembrane : Membrane
{
    public string SlipApproach;

    /// <summary>
    ///     Initial cracking angle.
    /// </summary>
    private double? _thetaIc;

    /// <summary>
    ///     Get/set crack slip consideration.
    /// </summary>
    /// <seealso cref="BiaxialConcrete.ConsiderCrackSlip" />
    public bool ConsiderCrackSlip

```

```

{
    get => Concrete.ConsiderCrackSlip;
    set => Concrete.ConsiderCrackSlip = value;
}

/// <summary>
///     Get/set the crack slip <see cref="StrainState" />.
/// </summary>
public StrainState CrackSlipStrains { get; private set; }

/// <summary>
///     Get current pseudo-stresses, in MPa.
/// </summary>
public StressState PseudoStresses => CrackSlipStrains.IsZero
    ? StressState.Zero
    : Concrete.Stiffness.Solve(CrackSlipStrains);

/// <inheritdoc cref="DSFMMembrane(IConcreteParameters, WebReinforcement, Length,
    bool)" />
/// <param name="unit">The <see cref="LengthUnit" /> of <paramref name="width"
    /></param>
internal DSFMMembrane(IConcreteParameters concreteParameters, WebReinforcement?
    reinforcement, double width, LengthUnit unit = LengthUnit.Millimeter, bool
    considerCrackSlip = true)
    : this(concreteParameters, reinforcement, (Length) width.As(unit),
        considerCrackSlip)
{
}

/// <summary>
///     Membrane element for DSFM analysis.
/// </summary>
/// <inheritdoc cref="Membrane(IConcreteParameters, WebReinforcement?, Length,
    ConstitutiveModel)" />
/// <param name="considerCrackSlip">Consider crack slip? (default: true)</param>
internal DSFMMembrane(IConcreteParameters concreteParameters, WebReinforcement?
    reinforcement, Length width, bool considerCrackSlip = true)
    : base(concreteParameters, reinforcement, width, ConstitutiveModel.DSFM)
{
    // Initiate crack slip strains
    ConsiderCrackSlip = considerCrackSlip;
    CrackSlipStrains = StrainState.Zero;
}

/// <summary>
///     Calculate Cs coefficient for concrete softening basing on the applied <see
    cref="StressState" />.
/// </summary>
/// <param name="stressState">The applied <see cref="StressState" />.</param>
/// <seealso cref="BiaxialConcrete.Cs" />
/// <returns>
///     0.55 if maximum of <see cref="StressState.SigmaX" /> and <see
    cref="StressState.SigmaY" /> is finite and smaller
///     than 0.5, otherwise 0.15.
/// </returns>
private static double CalculateCs(StressState stressState)
{
    var sigmaMax = UnitMath.Max(stressState.SigmaX, stressState.SigmaY);
    var ratio = sigmaMax / stressState.TauXY.Abs();

```

```

        return ratio >= 0.5 || !ratio.IsFinite() && sigmaMax > Pressure.Zero
            ? 0.15
            : 0.55;
    }

    /// <summary>
    ///     Calculate <see cref="StressState" /> and <see cref="Membrane.Stiffness" /> by
    ///     DSFM, given a known
    ///     <see cref="StrainState" />.
    /// </summary>
    /// <inheritdoc />
    public override void Calculate(StrainState appliedStrains)
    {
        // Set strains
        AverageStrains = appliedStrains.Clone();

        // Reduce Cs
        ReduceCs(AverageStresses);

        // Calculate concrete strains
        var cStrains = AverageStrains - CrackSlipStrains;

        // Calculate and set concrete and steel stresses
        Concrete.Calculate(cStrains, Reinforcement, ReferenceLength);
        Reinforcement?.Calculate(AverageStrains);

        // Calculate apparent principal strains
        AveragePrincipalStrains = PrincipalStrainState.FromStrain(AverageStrains);

        // Check stresses on crack
        if (ConsiderCrackSlip)
            CalculateCrackSlip();
        else
            CrackCheck();
    }

    /// <inheritdoc />
    public override Membrane Clone() => new DSFMMembrane(Concrete.Parameters,
        Reinforcement?.Clone(), Width, ConsiderCrackSlip);

    /// <inheritdoc />
    public override bool Equals(object? obj) => obj is DSFMMembrane other &&
        base.Equals(other);

    /// <inheritdoc />
    public override int GetHashCode() => base.GetHashCode();

    /// <summary>
    ///     Calculate and set <see cref="CrackSlipStrains" />.
    /// </summary>
    private void CalculateCrackSlip()
    {
        // Verify if concrete is cracked
        if (!Concrete.Cracked)
            return;

        // Verify the initial crack angle
        InitialCrackAngle();

        // Calculate crack local stresses

```



```

    var vci = ShearAtCrack();

    // Calculate crack slip strains
    CrackSlipStrains = CrackSlip(vci);
}

/// <summary>
///     Calculate crack slip <see cref="StrainState" />.
/// </summary>
/// <param name="vci">Shear stress on crack surface.</param>
private StrainState CrackSlip(Pressure vci)
{
    //Console.WriteLine(vci);
    // Get shear slip strains
    double
        ysa = StressCrackSlip(vci),
        ysb = RotationLagCrackSlip(),
        ys = Math.Max(ysa, ysb);

    SlipApproach = ys == ysa ? "Stress" : "Rotation_lag";

    // Get concrete principal angle
    var thetaC1 = Concrete.PrincipalStrains.Theta1;

    // Calculate direction cosines
    var (cos2ThetaC, sin2ThetaC) = (2 * thetaC1).DirectionCosines();

    // Calculate the shear slip strains
    double
        eslx = -0.5 * ys * sin2ThetaC,
        esly = 0.5 * ys * sin2ThetaC,
        eslxy = ys * cos2ThetaC;

    // Correct strains if gamma is negative
    if (AverageStrains.GammaXY < 0)
    {
        eslx = -eslx;
        esly = -esly;
        eslxy = -eslxy;
    }

    return new StrainState(eslx, esly, eslxy);
}

/// <summary>
///     Set the angle of initial cracking (<see cref="_thetaIc" />).
/// </summary>
private void InitialCrackAngle()
{
    if (!_thetaIc.HasValue && Concrete.Cracked)
        _thetaIc = Concrete.PrincipalStrains.Theta1;
}

/// <summary>
///     Set Cs coefficient for concrete basing on the applied <see cref="StressState" />.
/// </summary>
/// <inheritdoc cref="CalculateCs" />
private void ReduceCs(StressState stressState)
{

```

```

        if (Concrete.Cs.Approx(0.15, 1E-3)) // Already reduced
            return;

        Concrete.Cs = CalculateCs(stressState);
    }

    /// <summary>
    ///     Calculate shear slip strain by rotation lag approach.
    /// </summary>
    private double RotationLagCrackSlip()
    {
        // Get the strains
        double
            ex = AverageStrains.EpsilonX,
            ey = AverageStrains.EpsilonY,
            yxy = AverageStrains.GammaXY;

        // Calculate shear slip strain by rotation lag approach
        double
            thetaIc = _thetaIc ?? Constants.PiOver4,
            dThetaE = AveragePrincipalStrains.Theta1 - thetaIc;

        // Get theta L
        var thetaL = Reinforcement is null || !Reinforcement.XYReinforced
            ? 10.ToRadian()
            : Reinforcement.XYReinforced
            ? 5.ToRadian()
            : 7.5.ToRadian();

        // Correct thetaL if dThetaE < 0
        if (dThetaE < 0)
            thetaL = -thetaL;

        // Get dTheta s
        var dThetaS = dThetaE.Abs() > thetaL.Abs()
            ? dThetaE - thetaL
            : dThetaE;

        var
            thetaS = thetaIc + dThetaS;

        var (cos2ThetaS, sin2ThetaS) = (2 * thetaS).DirectionCosines();

        return
            (yxy * cos2ThetaS + (ey - ex) * sin2ThetaS).Abs();
    }

    /// <summary>
    ///     Calculate shear stress at the crack surface.
    /// </summary>
    private Pressure ShearAtCrack()
    {
        var theta1 = Concrete.PrincipalStrains.Theta1;

        var rx = Reinforcement?.DirectionX?.Clone();
        var ry = Reinforcement?.DirectionY?.Clone();

        // Get the average strains
        double
            ex = AverageStrains.EpsilonX,

```

```

    ey = AverageStrains.EpsilonY;

    // Get concrete tensile stress
    var fc1 = Concrete.PrincipalStresses.Sigma1;

    // Get reinforcement angles
    var (thetaNx, thetaNy) = Reinforcement?.Angles(theta1) ?? (theta1, theta1 -
        Constants.PiOver2);

    // Get reinforcement stresses
    Pressure
        fsx = Reinforcement?.Stresses.SigmaX ?? Pressure.Zero,
        fsy = Reinforcement?.Stresses.SigmaY ?? Pressure.Zero;

    // Calculate cosines and sines
    var (cosNx, sinNx) = thetaNx.DirectionCosines(true);
    var (cosNy, sinNy) = thetaNy.DirectionCosines(true);
    double
        cosNx2 = cosNx * cosNx,
        cosNy2 = cosNy * cosNy;

    // Solve the nonlinear equation by Brent Method
    if (!Brent.TryFindRoot(CrackEquilibrium, 0, 0.005, 1E-9, 1000, out var delCr))
        return Pressure.Zero;

    // Calculate local strains
    double
        esCrx = ex + delCr * cosNx2,
        esCry = ey + delCr * cosNy2;

    // Calculate reinforcement stresses
    // Calculate reinforcement stresses
    rx?.Calculate(esCrx);
    ry?.Calculate(esCry);

    Pressure
        fscrx = rx?.Stress ?? Pressure.Zero,
        fscry = ry?.Stress ?? Pressure.Zero;

    // Calculate shear stress
    return
        (fscrx - fsx) * cosNx * sinNx + (fscry - fsy) * cosNy * sinNy;

    // Function to check equilibrium
    double CrackEquilibrium(double delCrIt)
    {
        // Calculate local strains
        double
            esCrxIt = ex + delCrIt * cosNx2,
            esCryIt = ey + delCrIt * cosNy2;

        // Calculate reinforcement stresses
        rx?.Calculate(esCrxIt);
        ry?.Calculate(esCryIt);

        Pressure
            fscrxIt = rx?.Stress ?? Pressure.Zero,
            fscryIt = ry?.Stress ?? Pressure.Zero;

        // Check equilibrium (must be zero)

```

```

        return
        ((fscrxIt - fsx) * cosNx2 + (fscryIt - fsy) * cosNy2 - fc1).Megapascals;
    }
}

/// <summary>
///     Calculate crack slip strains by stress-based approach (Walraven, 1980).
/// </summary>
/// <param name="vci">Shear stress on crack surface, in MPa.</param>
private double StressCrackSlip(Pressure vci)
{
    if (vci.ApproxZero(StressState.Tolerance))
        return 0;

    // Get concrete principal tensile strain and strength
    var fc = Concrete.Parameters.Strength.Megapascals;

    // Calculate crack spacings and width
    var s = CrackSpacing().Millimeters;

    // Calculate crack width
    var w = CrackOpening().Millimeters;

    // Calculate shear slip strain by stress-based approach
    double
        a = Math.Max(0.234 * w.Pow(-0.707) - 0.2, 0),
        ds = vci.Megapascals / (1.8 * w.Pow(-0.8) + a * fc);

    //Console.WriteLine("\n" + (ds/s) + "\n");

    return
        ds / s;
}

/// <summary>
///     Calculate crack slip strains by stress-based approach (Okamura and Maekawa)
/// </summary>
/// <param name="vci">Shear stress on crack surface, in MPa.</param>
private double StressCrackSlipAlt(Pressure vci)
{
    if (vci.ApproxZero(StressState.Tolerance))
        return 0;

    // Calculate crack spacings and width
    var s = CrackSpacing();

    // Calculate crack width
    var w = CrackOpening();

    // Calculate vciMax and psi ratio
    var psi = vci / MaximumShearOnCrack();

    // Calculate shear slip strain by stress-based approach
    var ds = 0.75 * w * (psi / (psi - 1)).Sqrt();

    return
        ds / s;
}
}

```

SMM.cs

```

namespace andrefmello91.ReinforcedConcreteMembrane;

/// <summary>
///     SMM class, based on formulation by Hsu and Zhu (2002).
/// </summary>
internal class SMMMembrane : Membrane
{
    /// <inheritdoc cref="MCFTMembrane(ICConcreteParameters, WebReinforcement, Length)" />
    /// <param name="unit">The <see cref="LengthUnit" /> of <paramref name="width"
    ///     /></param>
    internal SMMMembrane(ICConcreteParameters concreteParameters, WebReinforcement?
        reinforcement, double width, LengthUnit unit = LengthUnit.Millimeter)
        : this(concreteParameters, reinforcement, (Length) width.As(unit))
    {
    }

    /// <summary>
    ///     Membrane element for SMM analysis.
    /// </summary>
    /// <inheritdoc cref="Membrane(ICConcreteParameters, WebReinforcement?, Length,
    ///     ConstitutiveModel)" />
    internal SMMMembrane(ICConcreteParameters concreteParameters, WebReinforcement?
        reinforcement, Length width)
        : base(concreteParameters, reinforcement, width, ConstitutiveModel.SMM)
    {
    }

    /// <summary>
    ///     Calculate the Poisson coefficients for SMM.
    /// </summary>
    /// <param name="reinforcement">The reinforcement.</param>
    /// <param name="cracked">The cracked state of concrete. True if cracked.</param>
    private static (double v12, double v21) PoissonCoefficients(WebReinforcement?
        reinforcement, bool cracked)
    {
        var v21 = cracked
            ? 0
            : 0.2;

        if (reinforcement is null)
            return (0.2, v21);

        var strains = reinforcement.Strain;

        var esf = Math.Max(strains.EpsilonX, strains.EpsilonY);

        var ey = strains.EpsilonX >= strains.EpsilonY
            ? reinforcement.DirectionX?.Steel.Parameters.YieldStrain
            : reinforcement.DirectionY?.Steel.Parameters.YieldStrain;

        var v12 = esf <= 0 || !ey.HasValue
            ? 0.2
            : 0.2 + 850 * esf;

        return (v12, v21);
    }
}

```

```

/// <summary>
///     Calculate the strain state affected by Poisson ratios.
/// </summary>
/// <param name="strainsAtAvgPrincipal">
///     The strain state in concrete, at the average principal strain direction of the
///     membrane element.
/// </param>
/// <param name="reinforcement">The reinforcement.</param>
/// <param name="cracked">The cracked state of concrete. True if cracked.</param>
/// <returns>
///     The <see cref="StrainState" /> without Poisson effect.
/// </returns>
private static StrainState RemovePoissonEffect(StrainState strainsAtAvgPrincipal,
    WebReinforcement? reinforcement, bool cracked)
{
    // Get initial strains
    var e1i = strainsAtAvgPrincipal.EpsilonX;
    var e2i = strainsAtAvgPrincipal.EpsilonY;

    // Get coefficients
    var (v12, v21) = PoissonCoefficients(reinforcement, cracked);

    // Calculate strains
    var v1 = 1D / (1D - v12 * v21);
    var v2 = v21 * v1;

    var e1 = v1 * e1i + v2 * e2i;
    var e2 = v2 * e1i + v1 * e2i;

    return new StrainState(e1, e2, strainsAtAvgPrincipal.GammaXY,
        strainsAtAvgPrincipal.ThetaX);
}

/// <summary>
///     Calculate <see cref="StressState" /> and <see cref="Membrane.Stiffness" /> by
///     MCFT, given a known
///     <see cref="StrainState" />.
/// </summary>
/// <inheritdoc />
public override void Calculate(StrainState appliedStrains)
{
    AverageStrains = appliedStrains.Clone();
    AveragePrincipalStrains = appliedStrains.ToPrincipal();

    // Remove Poisson effect
    var noPoissonStrain =
        RemovePoissonEffect(AveragePrincipalStrains.Transform(Concrete.DeviationAngle),
            Reinforcement, Concrete.Cracked);

    // Calculate and set concrete and steel stresses
    Concrete.Calculate(noPoissonStrain, Reinforcement);
    Reinforcement?.Calculate(noPoissonStrain.ToHorizontal());
}

/// <inheritdoc />
public override Membrane Clone() => new SMMMembrane(Concrete.Parameters,
    Reinforcement?.Clone(), Width);

/// <inheritdoc />

```

```

    public override bool Equals(object? obj) => obj is SMMMembrane other &&
        base.Equals(other);

    /// <inheritdoc />
    public override int GetHashCode() => base.GetHashCode();
}

```

C.2.2 Classe *MembraneSolver*

Implementa o processo iterativo de solução de elementos de membrana isolados submetidos a um estado de tensão conhecido.

MembraneSolver.cs

```

namespace andrefmello91.ReinforcedConcreteMembrane.Solver;

/// <summary>
///     Simple solver class.
/// </summary>
public class MembraneSolver
{
    /// <summary>
    ///     Default save location for output file.
    /// </summary>
    private const string DefaultLocation = "D:/";

    /// <summary>
    ///     <see cref="Array" /> of average <see cref="PrincipalStrainState" /> results
    ///     for each step.
    /// </summary>
    private readonly List<PrincipalStrainState> _averagePrincipalStrainOutput = new()
    {
        PrincipalStrainState.Zero
    };

    /// <summary>
    ///     <see cref="Array" /> of concrete <see cref="PrincipalStrainState" /> results
    ///     for each step.
    /// </summary>
    private readonly List<PrincipalStrainState> _concretePrincipalStrainOutput = new()
    {
        PrincipalStrainState.Zero
    };

    /// <summary>
    ///     <see cref="Array" /> of <see cref="PrincipalStressState" /> results for each
    ///     step.
    /// </summary>
    private readonly List<PrincipalStressState> _principalStressOutput = new()
    {
        PrincipalStressState.Zero
    };

    /// <summary>
    ///     <see cref="Array" /> of <see cref="StrainState" /> results for each step.
    /// </summary>

```

```

private readonly List<StrainState> _strainOutput = new()
{
    StrainState.Zero
};

/// <summary>
///     <see cref="Array" /> of <see cref="StressState" /> results for each step.
/// </summary>
private readonly List<StressState> _stressOutput = new()
{
    StressState.Zero
};

/// <summary>
///     The applied <see cref="StressState" />.
/// </summary>
private StressState _appliedStresses;

/// <summary>
///     Number of successfully calculated steps.
/// </summary>
private int _calculatedSteps;

/// <summary>
///     The step of concrete cracking.
/// </summary>
private int? _crackStep;

/// <summary>
///     The strain increment of the current step.
/// </summary>
private StrainState _currentIncrement;

/// <summary>
///     The residual <see cref="StressState" /> of current iteration
/// </summary>
private StressState _currentResidual = StressState.Zero;

/// <summary>
///     The secant stiffness <see cref="Matrix" /> of current iteration
/// </summary>
private MaterialMatrix _currentStiffness = null!;

/// <summary>
///     The <see cref="StrainState" /> of current iteration
/// </summary>
private StrainState _currentStrains;

/// <summary>
///     The internal <see cref="StressState" /> of current iteration.
/// </summary>
private StressState _currentStresses;

/// <summary>
///     The initial strain increment of the current step.
/// </summary>
private StrainState _initialIncrement;

/// <summary>
///     Current iteration.

```



```

/// </summary>
private int _iteration;

/// <summary>
///     The residual <see cref="StressState" /> of last iteration
/// </summary>
private StressState _lastResidual = StressState.Zero;

/// <summary>
///     The secant stiffness <see cref="Matrix" /> of last iteration
/// </summary>
private MaterialMatrix _lastStiffness = null!;

/// <summary>
///     The <see cref="StrainState" /> of last iteration
/// </summary>
private StrainState _lastStrains = StrainState.Zero;

/// <summary>
///     The internal <see cref="StressState" /> of last iteration.
/// </summary>
private StressState _lastStresses;

/// <summary>
///     Current step.
/// </summary>
private int _step;

/// <summary>
///     The applied <see cref="StressState" /> at the current step.
/// </summary>
private StressState _stepStresses;

/// <summary>
///     Stop parameter for analysis.
/// </summary>
private bool _stop;

/// <summary>
///     The strain increment for each step.
/// </summary>
/// <remarks>
///     Used only in <see cref="AnalysisControl.Strain" />.
/// </remarks>
private StrainState _strainIncrement;

/// <summary>
///     Write analysis results in console?
/// </summary>
private bool _writeInConsole = true;

/// <summary>
///     The type of analysis control.
/// </summary>
public AnalysisControl Control { get; set; }

/// <summary>
///     Get the cracking <see cref="StressState" />.
/// </summary>
public StressState CrackingStresses { get; private set; }

```

```

/// <summary>
///     The <see cref="Membrane" /> element.
/// </summary>
public Membrane Element { get; }

/// <summary>
///     Maximum number of iterations to perform at each step.
/// </summary>
/// <remarks>
///     Default: 10000
/// </remarks>
public int MaximumIterations { get; set; } = 10000;

/// <summary>
///     The number of steps to perform.
/// </summary>
/// <remarks>
///     Default: 100
/// </remarks>
public int NumberOfSteps { get; set; } = 100;

/// <summary>
///     Solver method.
/// </summary>
public NonlinearSolver Solver { get; set; }

/// <summary>
///     Convergence tolerance for strains.
/// </summary>
/// <remarks>
///     Default: 1E-8
/// </remarks>
public double StrainTolerance { get; set; } = 1E-12;

/// <summary>
///     Convergence tolerance for stresses.
/// </summary>
/// <remarks>
///     Default: 1E-3
/// </remarks>
public double StressTolerance { get; set; } = 1E-3;

/// <summary>
///     Get the ultimate <see cref="StressState" />.
/// </summary>
public StressState UltimateStresses { get; private set; }

/// <summary>
///     Get the analysis type.
/// </summary>
private string AnalysisType => $"{Element.Concrete.Model}";

/// <summary>
///     Get current load factor.
/// </summary>
private double LoadFactor => (double) _step / NumberOfSteps;

/// <summary>
///     Calculate strain convergence.

```

```

/// </summary>
private double StrainConvergence => CalculateConvergence(_currentIncrement.AsArray(),
    _initialIncrement.AsArray());

/// <summary>
///     Calculate stress convergence.
/// </summary>
private double StressConvergence => CalculateConvergence(_currentResidual.AsVector(),
    _stepStresses.AsVector());

/// <summary>
///     Default constructor.
/// </summary>
/// <param name="element">The <see cref="Membrane" /> element to analyze.</param>
/// <param name="solver">The solver method.</param>
/// <param name="control">The type of analysis control.</param>
public MembraneSolver(Membrane element, NonlinearSolver solver =
    NonlinearSolver.NewtonRaphson, AnalysisControl control = AnalysisControl.Stress)
{
    Element = element;
    Solver = solver;
    Control = control;
}

/// <summary>
///     Simple console solver example.
/// </summary>
public static void SolverExample()
{
    // Initiate steel for each direction
    var steelXY = new SteelParameters(276, 200000);

    // Get reinforcement
    var xRef = WebReinforcementDirection.From(6.35, 50.55, steelXY, 70, 0);
    var yRef = WebReinforcementDirection.From(4.7, 49.57, steelXY.Clone(),
        70, 0);
    var reinforcement = new WebReinforcement(xRef, yRef, (Length)
        70.As(LengthUnit.Millimeter));

    // Initiate the membrane
    var membraneMCFT = new MCFTMembrane(new Parameters(14.5, 6), reinforcement, 70);
    var membraneDSFM = new DSFMMembrane(new Parameters(14.5, 6),
        reinforcement.Clone(), 70);

    // Initiate stresses
    var sigma = new StressState(0, 0, 5);

    // Initiate solvers
    var mcftSolver = new MembraneSolver(membraneMCFT);
    var dsfmSolver = new MembraneSolver(membraneDSFM);

    // Solve
    mcftSolver.Solve(sigma);
    dsfmSolver.Solve(sigma);

    // Output results
    mcftSolver.OutputResults(out _);
    dsfmSolver.OutputResults(out _);

    Console.WriteLine("Done! Press any key to exit.");
}

```

```

    Console.ReadKey();
}

private static double CalculateConvergence(IEnumerable<double> numerator,
    IEnumerable<double> denominator) =>
    numerator.Sum(n => n * n) / (1 + denominator.Sum(d => d * d));

/// <summary>
///     Output results in a CSV file.
/// </summary>
/// <param name="filePath">The full path of output file produced.</param>
/// <param name="location">
///     The location to save output file.
///     <para>Default: D:/ </para>
/// </param>
public void OutputResults(out string filePath, string location = DefaultLocation)
{
    // Get readers for result
    string[]
        outputReaders = { "ex", "ey", "exy", "", "fx", "fy", "fxy", "", "ec1", "ec2",
            "theta1e", "", "fc1", "fc2", "theta1f" };

    // Result matrices
    var result = Matrix<double>.Build.Dense(_calculatedSteps, outputReaders.Length,
        double.NaN);

    for (var i = 0; i < _calculatedSteps; i++)
    {
        // Set stress and strain states
        result.SetSubMatrix(i, 0, _strainOutput[i].AsVector().ToRowMatrix());
        result.SetSubMatrix(i, 4, _stressOutput[i].AsVector().ToRowMatrix());
        result.SetSubMatrix(i, 8,
            _concretePrincipalStrainOutput[i].AsVector().ToRowMatrix());
        result.SetSubMatrix(i, 12, _principalStressOutput[i].AsVector().ToRowMatrix());

        // Set angles in degrees (average angle in strains)
        result[i, 10] = _averagePrincipalStrainOutput[i].Theta1.ToDegree();
        result[i, 14] = _principalStressOutput[i].Theta1.ToDegree();
    }

    filePath = ResultFileName(location);

    // Save results
    DelimitedWriter.Write(filePath, result, ";", outputReaders, null, null,
        double.NaN);
}

/// <summary>
///     Solve this element to applied stresses.
/// </summary>
/// <param name="appliedStresses">Applied <see cref="StressState" />.</param>
/// <param name="simulate">Simulate until convergence is not reached
///     (failure).</param>
/// <param name="writeInConsole">Write analysis results in console?</param>
public void Solve(StressState appliedStresses, bool simulate = false, bool
    writeInConsole = true)
{
    // Initialize fields and write a starting message
    Initiate(appliedStresses, writeInConsole);
}

```

```

        // Analyze by steps
        StepAnalysis(simulate);
        AnalysisDone();
    }

    /// <summary>
    ///     Write a ending message in <see cref="Console" />.
    /// </summary>
    private void AnalysisDone()
    {
        if (!_writeInConsole)
            return;

        Console.WriteLine($"{AnalysisType}_analysis_done.\n");
    }

    /// <summary>
    ///     Calculate and update stresses.
    /// </summary>
    private void CalculateStresses()
    {
        Element.Calculate(_currentStrains);

        // Update stresses
        _lastStresses = _currentStresses.Clone();
        _currentStresses = Element.AverageStresses;
    }

    /// <summary>
    ///     Write a message in the step that concrete cracks.
    /// </summary>
    private void ConcreteCrackedCheck()
    {
        if (_crackStep.HasValue || !Element.Concrete.Cracked)
            return;

        // Set and write message
        _crackStep = _step;

        // Set cracking stresses
        CrackingStresses = _currentStresses.Clone();

        if (_writeInConsole)
            Console.WriteLine("Concrete_cracked_at_step{0}", _crackStep.Value);
    }

    /// <summary>
    ///     Write a message after achieving convergence.
    /// </summary>
    private void ConvergenceMessage()
    {
        if (!_writeInConsole)
            return;

        switch (Element)
        {
            case DSFMMembrane dsfm when Element.Concrete.Cracked:
                Console.WriteLine("LS={0}, Iterations={1}, SlipApproach={2}", _step,
                    _iteration, dsfm.SlipApproach);
                break;
        }
    }

```

```

        default:
            Console.WriteLine("LS={0},Iterations={1}", _step, _iteration);
            break;
    }
}

/// <summary>
///     Verify if convergence is reached.
/// </summary>
/// <param name="convergence">Calculated convergence.</param>
/// <param name="tolerance">The convergence tolerance.</param>
/// <param name="minIterations">Minimum number of iterations (default: 2).</param>
private bool ConvergenceReached(double convergence, double tolerance, int
    minIterations = 2) =>
    _iteration >= minIterations && convergence <= tolerance;

/// <summary>
///     Get the stress state for the current step, basing in the analysis control.
/// </summary>
private StressState GetStepStresses() =>
    Control switch
    {
        AnalysisControl.Stress => LoadFactor * _appliedStresses,
        -                       => _currentStiffness * (_currentStrains +
            _strainIncrement)
    };

/// <summary>
///     Initialize auxiliary fields and write a starting message in <see
///     cref="Console" />.
/// </summary>
/// <param name="appliedStresses">Applied <see cref="StressState" />.</param>
/// <param name="writeInConsole">Write analysis results in console?</param>
private void Initiate(StressState appliedStresses, bool writeInConsole)
{
    // Set values
    _appliedStresses = appliedStresses;
    _writeInConsole = writeInConsole;

    // Get initial stresses
    _step = 1;
    _stepStresses = LoadFactor * _appliedStresses;
    _currentStresses = _lastStresses = StressState.Zero;
    _currentIncrement = _initialIncrement = StrainState.Zero;

    // Calculate initial stiffness
    _currentStiffness = Element.InitialStiffness;

    // Calculate initial strains
    _currentStrains = StrainState.Zero;

    // Get the strain increment
    _strainIncrement = _stepStresses / _currentStiffness;

    // Initiate solution values
    _lastStiffness = (MaterialMatrix) _currentStiffness.Clone();

    if (_writeInConsole)
        Console.WriteLine("\nStarting {0} analysis...\n", AnalysisType);
}

```

```

}

/// <summary>
///     Iterate to find solution.
/// </summary>
private void Iterate()
{
    _iteration = 1;

    while (_iteration <= MaximumIterations)
    {
        // Calculate stresses
        CalculateStresses();

        // Verify if concrete cracks in this step and write a message
        ConcreteCrackedCheck();

        // Calculate and update residual
        UpdateResidual();

        // Check stress convergence
        if (ConvergenceReached(StressConvergence, StressTolerance))
            goto ConvergenceReached;

        // If reached max iterations
        if (_iteration == MaximumIterations)
            goto Stop;

        // Update strains
        UpdateStrains();

        // Check strain convergence
        if (ConvergenceReached(StrainConvergence, StrainTolerance))
            goto ConvergenceReached;

        // Update stiffness
        UpdateStiffness();

        _iteration++;
    }

    ConvergenceReached:
    {
        // Write message
        ConvergenceMessage();

        // Set results on output matrices
        SaveResults();

        return;
    }

    // Write message
    Stop:
    StopAnalysis();
}

/// <summary>
///     Returns the output file name and save location.
/// </summary>

```

```

/// <param name="location">The location to save output file.</param>
private string ResultFileName(string location) =>
    $"{location}membrane_result_{AnalysisType}.csv";

/// <summary>
///     Save results in output matrices.
///     <para>
///         See: <see cref="_strainOutput" />, <see cref="_stressOutput" />,
///         <see cref="_concretePrincipalStrainOutput" /> and <see
///         cref="_principalStressOutput" />.
///     </para>
/// </summary>
private void SaveResults()
{
    _strainOutput.Add(Element.AverageStrains);
    _stressOutput.Add(Element.AverageStresses);
    _principalStressOutput.Add(Element.Concrete.PrincipalStresses);

    _concretePrincipalStrainOutput.Add(Element.Concrete.PrincipalStrains);
    _averagePrincipalStrainOutput.Add(Element.AveragePrincipalStrains);
}

/// <summary>
///     Do analysis by steps.
/// </summary>
private void StepAnalysis(bool simulate)
{
    // Initiate steps
    _step = 1;
    while (simulate || _step <= NumberOfSteps)
    {
        // Calculate stresses
        _stepStresses = GetStepStresses();

        // Iterate to find solution
        Iterate();

        switch (_stop)
        {
            // Solution not reached
            case true:

                // Decrement ls to correct output file
                _step--;

                // Set last step
                _calculatedSteps = _step;
                return;

            // Solution reached
            default:

                // Set provisional stresses
                UltimateStresses = _currentStresses;

                if (_step > 930)
                {
                }

                _step++;
        }
    }
}

```



```

        continue;
    }
}

_step = 1;
}

/// <summary>
///     Write a message if convergence is not reached.
/// </summary>
private void StopAnalysis()
{
    // Set stop
    _stop = true;

    if (_writeInConsole)
        Console.WriteLine("LS_{0},{1}", _step, "CONVERGENCE_NOT_REACHED");
}

/// <summary>
///     Update residual <see cref="StressState" />.
/// </summary>
private void UpdateResidual()
{
    // Set new values
    _lastResidual = _currentResidual.Clone();
    _currentResidual = _currentStresses - _stepStresses;
}

/// <summary>
///     Calculate the secant stiffness <see cref="Matrix" /> of current iteration.
/// </summary>
private void UpdateStiffness()
{
    Matrix<double> dK;

    switch (Solver)
    {
        // Newton-Raphson update
        case NonlinearSolver.NewtonRaphson:
            // Get variations
            var de = (_currentStrains - _lastStrains).AsVector();
            var ds = (_currentStresses - _lastStresses).AsVector();

            // Calculate increment
            dK = ds.ToColumnMatrix() * de.ToRowMatrix();

            break;

        // Secant update
        case NonlinearSolver.Secant:
            // Calculate the variation of strains and residual as vectors
            Vector<double>
                dStrain = (_currentStrains - _lastStrains).AsVector(),
                dRes = (_currentResidual - _lastResidual).AsVector();

            // Calculate increment
            dK = ((dRes - _lastStiffness * dStrain) / dStrain.Norm(2)).ToColumnMatrix()
                * dStrain.ToRowMatrix();
    }
}

```

```

        break;

    default:
        return;
}

// Set new values
_lastStiffness = (MaterialMatrix) _currentStiffness.Clone();
_currentStiffness = (MaterialMatrix) (_currentStiffness + new MaterialMatrix(dK));
}

/// <summary>
///     Update strains.
/// </summary>
private void UpdateStrains()
{
    // Calculate increment
    _currentIncrement = -(_currentResidual / _currentStiffness);

    // Calculate increment and set initial increment
    if (_iteration == 1)
        _initialIncrement = _currentIncrement.Clone();

    // Increment strains
    _lastStrains = _currentStrains.Clone();
    _currentStrains += _currentIncrement;
}
}

/// <summary>
///     Solver enumeration.
/// </summary>
public enum NonlinearSolver
{
    /// <summary>
    ///     Solve by Newton-Raphson method.
    /// </summary>
    NewtonRaphson,

    /// <summary>
    ///     Solve by secant method.
    /// </summary>
    Secant
}

/// <summary>
///     Types of analysis control.
/// </summary>
public enum AnalysisControl
{
    /// <summary>
    ///     Control the analysis by increasing the applied stress state.
    /// </summary>
    /// <remarks>
    ///     This cannot predict the full response of the element.
    /// </remarks>
    Stress,

    /// <summary>
    ///     Control the analysis by increasing the shear strain.

```

```

    /// </summary>
    /// <remarks>
    ///     This can predict the full response of the element.
    /// </remarks>
    Strain
}

```

C.3 *FEMAnalysis*

A biblioteca *FEMAnalysis* (MELLO, 2021a) apresenta interfaces para elementos finitos, que podem ser implementadas por outras bibliotecas. Apresenta também classes de análises linear e não-linear.

C.3.1 Interfaces

As interfaces *INumberedElement*, *IFiniteElement*, *IGrip* e *IMonitoredElement* fornecem as propriedades para elementos numerados, elementos finitos, nós e elementos monitorados.

Interfaces.cs

```

namespace andrefmello91.FEMAnalysis;

/// <summary>
///     Interface for numbered elements.
/// </summary>
public interface INumberedElement
{

    /// <summary>
    ///     Get the index of degrees of freedom of this element.
    /// </summary>
    int[] DoFIndex { get; }

    /// <summary>
    ///     The name of this element.
    /// </summary>
    string Name { get; }

    /// <summary>
    ///     Get/set the number of this element.
    /// </summary>
    /// <remarks>
    ///     Enumeration starts at 1.
    /// </remarks>
    int Number { get; set; }
}

/// <summary>
///     Interface for grips.
/// </summary>

```

```

public interface IGrip : INumberedElement, IEquatable<IGrip>, IComparable<IGrip>
{
    /// <summary>
    ///     Get the <see cref="Constraint" /> in this grip.
    /// </summary>
    Constraint Constraint { get; }

    /// <summary>
    ///     Get the <see cref="PlaneDisplacement" /> in this grip.
    /// </summary>
    PlaneDisplacement Displacement { get; set; }

    /// <summary>
    ///     Get the <see cref="PlaneForce" /> in this grip.
    /// </summary>
    PlaneForce Force { get; }

    /// <summary>
    ///     Get the <see cref="PlaneForce" /> reaction in this grip, in case it's
    ///     constrained.
    /// </summary>
    PlaneForce Reaction { get; set; }
}

/// <summary>
///     Interface for finite elements.
/// </summary>
public interface IFiniteElement : INumberedElement, IEquatable<IFiniteElement>,
    IComparable<IFiniteElement>
{
    /// <summary>
    ///     Get the displacement vector, in global coordinate system.
    /// </summary>
    DisplacementVector Displacements { get; }

    /// <summary>
    ///     Get the force vector in this element, in global coordinate system.
    /// </summary>
    ForceVector Forces { get; }

    /// <summary>
    ///     Get the grips of this element.
    /// </summary>
    IGrip[] Grips { get; }

    /// <summary>
    ///     Get stiffness matrix in the global coordinate system.
    /// </summary>
    StiffnessMatrix Stiffness { get; }

    /// <summary>
    ///     Calculate forces in this element after updating displacements at each grip.
    /// </summary>
    void CalculateForces();

    /// <summary>
    ///     Update displacements in this element.
    /// </summary>

```

```

    void UpdateDisplacements();

    /// <summary>
    ///     Update stiffness of this element.
    /// </summary>
    void UpdateStiffness();
}

/// <summary>
///     Interface for monitored elements.
/// </summary>
public interface IMonitoredElement
{

    /// <summary>
    ///     The element monitor.
    /// </summary>
    ElementMonitor? Monitor { get; }

    /// <summary>
    ///     Get/set monitoring of this element.
    /// </summary>
    public bool Monitored { get; set; }

    /// <summary>
    ///     Add the monitored values to monitor for the current load factor.
    /// </summary>
    /// <param name="loadFactor">The current load factor.</param>
    void AddValue(double loadFactor);
}

```

As interfaces *IIteration* e *IMonitoredValue* apresentam as propriedades utilizadas nas iterações e valores monitorados.

Interfaces.cs

```

/// <summary>
///     Interface for iterations.
/// </summary>
public interface IIteration : ICloneable<IIteration>
{

    /// <summary>
    ///     The displacement convergence of this iteration.
    /// </summary>
    double DisplacementConvergence { get; }

    /// <summary>
    ///     The displacement increment vector from external forces of this iteration.
    /// </summary>
    DisplacementVector DisplacementIncrement { get; }

    /// <summary>
    ///     The displacement vector of this iteration.
    /// </summary>
    DisplacementVector Displacements { get; }

    /// <summary>
    ///     The force convergence of this iteration.

```

```

/// </summary>
double ForceConvergence { get; }

/// <summary>
///     The internal force vector of this iteration.
/// </summary>
/// <inheritdoc cref="LoadStep.FullForceVector" />
ForceVector InternalForces { get; }

/// <summary>
///     The number of this iteration.
/// </summary>
int Number { get; set; }

/// <summary>
///     The residual force vector of this iteration.
/// </summary>
/// <inheritdoc cref="LoadStep.FullForceVector" />
ForceVector ResidualForces { get; }

/// <summary>
///     The stiffness matrix of this iteration.
/// </summary>
StiffnessMatrix Stiffness { get; set; }

/// <summary>
///     Calculate the convergence of this iteration.
/// </summary>
/// <param name="appliedForces">The applied forces of the current step, simplified in
    constrained DoFs.</param>
/// <param name="initialIncrement">The displacement increment of the first
    iteration.</param>
void CalculateConvergence(ForceVector appliedForces, DisplacementVector
    initialIncrement);

/// <summary>
///     Check convergence for this iteration.
/// </summary>
/// <param name="parameters">The analysis parameters.</param>
/// <returns>
///     True if this iteration number is equal or bigger than minimum iterations and
    force or displacement convergences are
///     smaller than their respective tolerances.
/// </returns>
bool CheckConvergence(AnalysisParameters parameters);

/// <summary>
///     Check the stop condition for this iteration.
/// </summary>
/// <inheritdoc cref="CheckConvergence" />
/// <returns>
///     True if this iteration number is equal or bigger than maximum number of
    iterations or any of the analysis vectors
///     and matrix contains <see cref="double.NaN" />.
/// </returns>
bool CheckStopCondition(AnalysisParameters parameters);

/// <summary>
///     Increment displacements of this iteration.
/// </summary>

```

```

    /// <param name="displacementIncrement">The vector of displacement increments.</param>
    void IncrementDisplacements(DisplacementVector displacementIncrement);

    /// <summary>
    ///     Update forces in this iteration.
    /// </summary>
    /// <remarks>
    ///     Forces must be simplified in constrained DoFs.
    /// </remarks>
    /// <param name="appliedForces">The vector of applied forces of the current step,
    ///     simplified in constrained DoFs.</param>
    /// <param name="internalForces">The vector of internal forces, simplified in
    ///     constrained DoFs.</param>
    void UpdateForces(ForceVector appliedForces, ForceVector internalForces);
}

/// <summary>
///     Generic interface for monitored values.
/// </summary>
/// <typeparam name="TStruct">The monitored type.</typeparam>
public interface IMonitoredValue<TStruct> : IEquatable<IMonitoredValue<TStruct>>,
    IComparable<IMonitoredValue<TStruct>>
    where TStruct : struct
{
    /// <summary>
    ///     The load factor associated to value.
    /// </summary>
    double LoadFactor { get; }

    /// <summary>
    ///     The monitored value.
    /// </summary>
    TStruct Value { get; }
}

```

C.3.2 Entrada de dados

A entrada de dados é representada pela interface *IFEMInput* e pela classe *FEMInput*, que se tratam de conjuntos de elementos finitos que implementam a interface *IFiniteElement*.

FEMInput.cs

```

namespace andrefmello91.FEMAnalysis;

/// <summary>
///     Finite element input class.
/// </summary>
public interface IFEMInput : IEnumerable<IFiniteElement>
{
    /// <summary>
    ///     Get the index of constrained degrees of freedom.
    /// </summary>
    List<int> ConstraintIndex { get; }
}

```

```

    /// <summary>
    ///     Get the external force <see cref="Vector" />.
    /// </summary>
    ForceVector Forces { get; }

    /// <summary>
    ///     Get the grips of the finite element model.
    /// </summary>
    List<IGrip> Grips { get; }

    /// <summary>
    ///     The monitored elements.
    /// </summary>
    IEnumerable<IMonitoredElement> MonitoredElements { get; }

    /// <summary>
    ///     Get the number of degrees of freedom (DoFs).
    /// </summary>
    int NumberOfDoFs { get; }
}

/// <summary>
///     Default finite element input class.
/// </summary>
/// <inheritdoc cref="IFEMInput{TFiniteElement}" />
public class FEMInput : List<IFiniteElement>, IFEMInput
{
    /// <inheritdoc />
    public List<int> ConstraintIndex { get; }

    /// <inheritdoc />
    public ForceVector Forces { get; }

    /// <inheritdoc />
    public List<IGrip> Grips { get; }

    /// <summary>
    ///     The monitored elements.
    /// </summary>
    public IEnumerable<IMonitoredElement> MonitoredElements => this
        .Where(e => e is IMonitoredElement { Monitored: true })
        .Cast<IMonitoredElement>()
        .Concat(Grips
            .Where(e => e is IMonitoredElement { Monitored: true })
            .Cast<IMonitoredElement>());

    /// <inheritdoc />
    public int NumberOfDoFs { get; }

    /// <inheritdoc cref="FEMInput(IEnumerable{IFiniteElement}, IEnumerable{IGrip})" />
    /// <remarks>
    ///     Grips are taken from <paramref name="elements" />.
    /// </remarks>
    public FEMInput(IEnumerable<IFiniteElement> elements)
        : this(elements, elements.SelectMany(e => e.Grips).Distinct())
    {
    }
}

```



```

/// <summary>
///     Input Data constructor.
/// </summary>
/// <param name="elements">The collection containing all distinct <see
///     cref="IFiniteElement" />'s in the model.</param>
/// <param name="grips">The collection containing all distinct <see cref="IGrip" />'s
///     in the model.</param>
public FEMInput(IEnumerable<IFiniteElement> elements, IEnumerable<IGrip> grips)
    : base(elements)
{
    Grips          = grips.OrderBy(g => g.Number).ToList();
    NumberOfDoFs   = 2 * Grips.Count;
    ConstraintIndex = Grips.GetConstraintIndex().ToList();
    Forces          = this.AssembleExternalForces(false);
}

/// <inheritdoc />
public override string ToString() =>
    $"Number_of_elements: {Count}\n" +
    $"Number_of_grips: {Grips.Count}\n" +
    $"Force_vector: {Forces}\n" +
    $"Constraint_Index: {ConstraintIndex.Select(i => i.ToString()).Aggregate((i, f) => f,
    $"{i} - {f}")}";
}

```

C.3.3 Análise

A classe base *Analysis* fornece as propriedades e métodos principais para as classes derivadas *LinearAnalysis* e *NonlinearAnalysis*. A propriedade principal é um objeto que implementa a interface *IFEMInput*.

Analysis.cs

```

namespace andrefmello91.FEMAnalysis;

/// <summary>
///     Analysis base class.
/// </summary>
public abstract class Analysis
{
    /// <summary>
    ///     Get/set the displacement <see cref="Vector" />.
    /// </summary>
    /// <remarks>
    ///     Components in <see cref="LengthUnit.Millimeter" />.
    /// </remarks>
    public DisplacementVector Displacements { get; protected set; }

    /// <summary>
    ///     Get the input for finite element analysis.
    /// </summary>
    public IFEMInput FemInput { get; }

    /// <inheritdoc cref="IFEMInput.Forces" />
    /// <remarks>

```

```

///      Simplified by constrained DoFs.
/// </remarks>
public ForceVector Forces { get; protected set; }

/// <summary>
///      Get/set global stiffness <see cref="Matrix" />.
/// </summary>
public StiffnessMatrix GlobalStiffness { get; protected set; }

/// <summary>
///      Event to execute when analysis is aborted.
/// </summary>
public abstract event EventHandler? AnalysisAborted;

/// <summary>
///      Event to execute when analysis is complete.
/// </summary>
public abstract event EventHandler? AnalysisComplete;

/// <summary>
///      Base analysis constructor.
/// </summary>
/// <param name="femInput">The <see cref="IFEMInput" /> for finite element
///      analysis.</param>
protected Analysis(IFEMInput femInput)
{
    FemInput      = femInput;
    Displacements = DisplacementVector.Zero(FemInput.NumberOfDoFs);
    Forces        = FemInput.Forces.Simplified(FemInput.ConstraintIndex);
    GlobalStiffness = femInput.AssembleStiffness();
}

/// <summary>
///      Calculate the <see cref="Vector" /> of support reactions.
/// </summary>
public ForceVector GetReactions() => (ForceVector) (GlobalStiffness * Displacements -
    Forces);

/// <inheritdoc />
public override string ToString() =>
    $"{FemInput}\n" +
    "Global_Stiffness:\n" +
    $"{GlobalStiffness}\n" +
    "Displacement_Vector:\n" +
    $"{Displacements}";

/// <summary>
///      Invoke the event.
/// </summary>
/// <param name="handler">The handler.</param>
protected void Invoke(EventHandler? handler) => handler?.Invoke(this,
    EventArgs.Empty);

/// <inheritdoc cref="Invoke" />
protected void Invoke<TEventArgs>(EventHandler<TEventArgs>? handler, TEventArgs
    eventArgs) where TEventArgs : EventArgs =>
    handler?.Invoke(this, eventArgs);
}

```

C.3.3.1 Linear

A classe *LinearAnalysis* fornece o método *Execute*, que executa a análise linear.

LinearAnalysis.cs

```
namespace andrefmello91.FEMAnalysis;

/// <summary>
///     Linear analysis class.
/// </summary>
public class LinearAnalysis : Analysis
{
    /// <inheritdoc />
    public override event EventHandler? AnalysisAborted;

    /// <inheritdoc />
    public override event EventHandler? AnalysisComplete;

    /// <summary>
    ///     Linear analysis constructor.
    /// </summary>
    /// <inheritdoc />
    public LinearAnalysis(IFEMInput femInput)
        : base(femInput)
    {
    }

    /// <summary>
    ///     Execute the analysis.
    /// </summary>
    /// <param name="loadFactor">The load factor to multiply <see cref="Analysis.Forces"
    ///     /> (default: 1).</param>
    public void Execute(double loadFactor = 1)
    {
        // Set force vector
        if (!loadFactor.Approx(1))
            Forces = (ForceVector) (Forces * loadFactor);

        // Solve
        Displacements = GlobalStiffness.Solve(Forces);

        // Set displacements to grips
        FemInput.Grips.SetDisplacements(Displacements);

        // Update element displacements
        FemInput.UpdateDisplacements();

        // Calculate element forces
        FemInput.CalculateForces();

        // Set Reactions
        FemInput.Grips.SetReactions(GetReactions());

        // Invoke complete event
        Invoke(AnalysisComplete);
    }
}
```

C.3.3.2 Não-linear

A classe *NonlinearAnalysis* fornece o método *Execute*, que executa a análise não-linear, e os métodos auxiliares necessários.

NonlinearAnalysis.cs

```
namespace andrefmello91.FEMAnalysis;

/// <summary>
///     Nonlinear analysis class
/// </summary>
public class NonlinearAnalysis : Analysis, IEnumerable<LoadStep>
{
    /// <summary>
    ///     The list of step results.
    /// </summary>
    protected readonly List<LoadStep> Steps = new();

    /// <summary>
    ///     Set true to execute analysis until convergence is not achieved (structural
    ///     failure).
    /// </summary>
    private readonly bool _simulate;

    /// <summary>
    ///     Field to store the DoF index for monitored displacements.
    /// </summary>
    protected int? MonitoredIndex;

    /// <inheritdoc cref="List{T}.this[int]" />
    public LoadStep this[int index] => Steps[index];

    /// <inheritdoc cref="List{T}.this[int]" />
    public LoadStep this[Index index] => Steps[index];

    /// <summary>
    ///     The current step result.
    /// </summary>
    public LoadStep CurrentStep => Steps[^1];

    /// <summary>
    ///     The last step result.
    /// </summary>
    public LoadStep LastStep => Steps.Count > 1
        ? Steps[^2]
        : CurrentStep;

    /// <summary>
    ///     The analysis parameters.
    /// </summary>
    public AnalysisParameters Parameters { get; }

    /// <summary>
    ///     Get/set when to stop analysis.
    /// </summary>
    public bool Stop => CurrentStep.Stop;
```

```

/// <summary>
///     Get/set the stop message.
/// </summary>
public string StopMessage { get; protected set; } = string.Empty;

/// <inheritdoc />
public override event EventHandler? AnalysisAborted;

/// <inheritdoc />
public override event EventHandler? AnalysisComplete;

/// <summary>
///     Event to execute when the current load step is aborted.
/// </summary>
/// <remarks>
///     The aborted step is passed to event args.
/// </remarks>
public event EventHandler<StepEventArgs>? StepAborted;

/// <summary>
///     Event to execute when the current load step converges.
/// </summary>
/// <remarks>
///     The converged step is passed to event args.
/// </remarks>
public event EventHandler<StepEventArgs>? StepConverged;

/// <summary>
///     Nonlinear analysis constructor with default parameters.
/// </summary>
/// <param name="nonlinearInput">The finite element input.</param>
/// <param name="monitoredIndex">The index of a degree of freedom to monitor, if
    wanted.</param>
/// <param name="simulate">Execute the analysis until convergence is not
    reached.</param>
public NonlinearAnalysis(IFEMInput nonlinearInput, int? monitoredIndex = null, bool
    simulate = false)
    : this(nonlinearInput, AnalysisParameters.Default, monitoredIndex, simulate)
{
}

/// <summary>
///     Nonlinear analysis constructor.
/// </summary>
/// <param name="parameters">The analysis parameters.</param>
/// <inheritdoc cref="NonlinearAnalysis(IFEMInput, int?, bool)" />
public NonlinearAnalysis(IFEMInput nonlinearInput, AnalysisParameters parameters,
    int? monitoredIndex = null, bool simulate = false)
    : base(nonlinearInput)
{
    Parameters      = parameters;
    MonitoredIndex  = monitoredIndex;
    _simulate       = simulate;
}

/// <summary>
///     Get the step increment.
/// </summary>
/// <param name="numberOfSteps">The number of load steps.</param>
public static double StepIncrement(int numberOfSteps) => 1D / numberOfSteps;

```

```

/// <summary>
///     Calculate the stiffness increment for nonlinear analysis.
/// </summary>
/// <param name="solver">The nonlinear solver.</param>
/// <returns>
///     The <see cref="andrefmello91.OnPlaneComponents.StiffnessMatrix" /> increment
///     with current unit.
/// </returns>
/// <inheritdoc cref="TangentIncrement" />
public static StiffnessMatrix StiffnessIncrement(Iteration currentIteration,
    Iteration lastIteration, NonLinearSolver solver) =>
    solver switch
    {
        NonLinearSolver.Secant => SecantIncrement(currentIteration, lastIteration),
        -                        => TangentIncrement(currentIteration, lastIteration)
    };

/// <summary>
///     Calculate the convergence.
/// </summary>
/// <param name="numerator">
///     The residual forces of the current iteration or the initial displacement
///     increment of the
///     current step.
/// </param>
/// <param name="denominator">
///     The applied forces of the current step or the displacement increment of the
///     current
///     iteration.
/// </param>
internal static double CalculateConvergence(IEnumerable<double> numerator,
    IEnumerable<double> denominator)
{
    double
        num = numerator.Sum(n => n * n),
        den = 1 + denominator.Sum(n => n * n);

    return
        num / den;
}

/// <inheritdoc cref="CalculateConvergence" />
internal static double CalculateConvergence<TQuantity,
    TUnit>(QuantityVector<TQuantity, TUnit> numerator, QuantityVector<TQuantity,
    TUnit> denominator, IEnumerable<int>? constraintIndexes = null)
    where TQuantity : struct, IQuantity<TUnit>
    where TUnit : Enum
{
    var unit = numerator.Unit;

    var dd = denominator.Unit.Equals(unit)
        ? denominator
        : denominator.Convert(unit);

    return
        CalculateConvergence(numerator.Values, dd.Values);
}

/// <summary>

```

```

///      Calculate the secant stiffness increment.
/// </summary>
/// <inheritdoc cref="TangentIncrement(andrefmello91.FEMAnalysis.IIteration,
    andrefmello91.FEMAnalysis.IIteration)" />
private static StiffnessMatrix SecantIncrement(IIteration currentIteration,
    IIteration lastIteration)
{
    // Calculate the variation of displacements and residual as vectors
    var dU = currentIteration.Displacements - lastIteration.Displacements;
    var dR = currentIteration.ResidualForces - lastIteration.ResidualForces;

    var unit = dR.Unit.Per(dU.Unit);

    Matrix<double> stiffness = lastIteration.Stiffness.Unit == unit
        ? lastIteration.Stiffness
        : lastIteration.Stiffness.Convert(unit);

    var inc = ((dR - stiffness * (Vector<double>) dU) / dU.Norm(2)).ToColumnMatrix() *
        dU.ToRowMatrix();

    return
        new StiffnessMatrix(inc, unit);
}

/// <summary>
///      Calculate the tangent stiffness increment.
/// </summary>
/// <param name="currentIteration">The current iteration.</param>
/// <param name="lastIteration">The last solved iteration.</param>
/// <returns>
///      The <see cref="StiffnessMatrix" /> increment with current unit.
/// </returns>
private static StiffnessMatrix TangentIncrement(IIteration currentIteration,
    IIteration lastIteration)
{
    // Get variations
    var dF = (ForceVector) (currentIteration.InternalForces -
        lastIteration.InternalForces);
    var dU = (DisplacementVector) (currentIteration.Displacements -
        lastIteration.Displacements);

    return
        dF / dU;
}

/// <summary>
///      Execute the full analysis.
/// </summary>
/// <param name="loadFactor">The load factor to multiply <see cref="Analysis.Forces"
    /> (default: 1).</param>
public void Execute(double loadFactor = 1)
{
    // Get force vector
    if (!loadFactor.Approx(1))
        Forces = (ForceVector) (Forces * loadFactor);

    // Analysis by steps
    while (true)
    {
        // Add new step

```

```

        ExecuteStep();

        if (Stop || !_simulate && CurrentStep >= Parameters.NumberOfSteps)
            return;
    }
}

/// <summary>
///     Add a load step and execute it.
/// </summary>
/// <remarks>
///     Load step is not executed when analysis is aborted or complete.
/// </remarks>
public void ExecuteStep()
{
    // Check conditions
    if (Steps.Any() && (Stop || !_simulate && CurrentStep >= Parameters.NumberOfSteps))
        return;

    // Add new step
    NewStep();

    // Iterate
    CurrentStep.Iterate(FemInput);

    // Verify if convergence was not reached
    if (Stop)
    {
        Invoke(StepAborted, new StepEventArgs(CurrentStep));
        Invoke(AnalysisAborted);
        CorrectResults();
        return;
    }

    // Set step results
    SetStepResults(MonitoredIndex);

    // Check if analysis is done
    if (_simulate || CurrentStep <= Parameters.NumberOfSteps)
        return;

    // Set Reactions
    FemInput.Grips.SetReactions(GetReactions());

    // Invoke analysis complete event
    Invoke(AnalysisComplete);
}

/// <summary>
///     Generate an <see cref="FEMOutput" /> from analysis results.
/// </summary>
public FEMOutput GenerateOutput() => new(Steps, FemInput.MonitoredElements.Select(e
    => e.Monitor!));

/// <summary>
///     Correct results from last step after not achieving convergence.
/// </summary>
protected virtual void CorrectResults()
{
    StopMessage = $"Convergence not reached at {CurrentStep}";
}

```

```

        // Set displacements from last (current now) step
        FemInput.Grips.SetDisplacements(LastStep.FinalDisplacements);
        FemInput.UpdateDisplacements();

        // Calculate element forces
        FemInput.CalculateForces();

        // Set Reactions
        FemInput.Grips.SetReactions(GetReactions());
    }

    /// <inheritdoc cref="LoadStep.SetResults" />
    protected virtual void SetStepResults(int? monitoredIndex)
    {
        CurrentStep.SetResults(FemInput, monitoredIndex);
        Invoke(StepConverged, new StepEventArgs(CurrentStep));
    }

    /// <summary>
    ///     Create a new load step.
    /// </summary>
    /// <param name="incrementLoad">Increment load of the new step?</param>
    private void NewStep(bool incrementLoad = true) =>
        Steps.Add(Steps.Any()
            ? LoadStep.FromLastStep(CurrentStep, incrementLoad)
            : LoadStep.InitialStep(FemInput, Parameters));

    /// <inheritdoc />
    public IEnumerator<LoadStep> GetEnumerator() => Steps.GetEnumerator();

    /// <inheritdoc />
    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}

```

A classe depende também das auxiliares: *AnalysisParameters*, que representa os parâmetros de solução não-linear; *LoadStep*, que implementa os métodos e guarda os resultados de cada passo de carga; *Iteration*, que contém os métodos e guarda os resultados de cada iteração, dentro de um passo de carga.

AnalysisParameters.cs

```

i»;namespace andrefmello91.FEMAnalysis;

/// <summary>
///     Nonlinear analysis parameters struct.
/// </summary>
/// <param name="Solver">The nonlinear equation solver.</param>
/// <param name="NumberOfSteps">The number of steps to execute.</param>
/// <param name="MaxIterations">The maximum number of iterations.</param>
/// <param name="MinIterations">The minimum number of iterations.</param>
/// <param name="ForceTolerance">The convergence tolerance for residual forces.</param>
/// <param name="DisplacementTolerance">The convergence tolerance for displacement
    increments.</param>
public record AnalysisParameters(NonLinearSolver Solver, int NumberOfSteps, int
    MaxIterations, int MinIterations, double ForceTolerance, double
    DisplacementTolerance)
{

```

```

/// <summary>
///     Get the default analysis parameters.
/// </summary>
/// <remarks>
///     Solver: <see cref="NonLinearSolver.NewtonRaphson" />.
///     <para>NumberOfSteps: 100.</para>
///     <para>MaxIterations: 2000.</para>
///     <para>MinIterations: 2.</para>
///     <para>ForceTolerance: 1E-3</para>
///     <para>DisplacementTolerance: 1E-4</para>
/// </remarks>
public static AnalysisParameters Default { get; } =
    new(NonLinearSolver.NewtonRaphson, 100, 2000, 2, 1E-3, 1E-4);

/// <summary>
///     Get/set the convergence tolerance for displacement increments.
/// </summary>
/// <remarks>
///     Default: 1E-6
/// </remarks>
public double DisplacementTolerance { get; set; } = DisplacementTolerance;

/// <summary>
///     Get/set the convergence tolerance for residual forces.
/// </summary>
/// <remarks>
///     Default: 1E-3
/// </remarks>
public double ForceTolerance { get; set; } = ForceTolerance;

/// <summary>
///     Get/set the maximum number of iterations.
/// </summary>
/// <remarks>
///     Default: 2000
/// </remarks>
public int MaxIterations { get; set; } = MaxIterations;

/// <summary>
///     Get/set the minimum number of iterations.
/// </summary>
/// <remarks>
///     Default: 2
/// </remarks>
public int MinIterations { get; set; } = MinIterations;

/// <summary>
///     Get/set the number of steps to execute.
/// </summary>
/// <remarks>
///     Default: 50
/// </remarks>
public int NumberOfSteps { get; set; } = NumberOfSteps;

/// <summary>
///     The nonlinear equation solver.
/// </summary>
public NonLinearSolver Solver { get; set; } = Solver;
}

```

LoadStep.cs

```

namespace andrefmello91.FEMAnalysis;

/// <summary>
///     Generic class for load step.
/// </summary>
public class LoadStep : IEnumerable<IIteration>
{
    /// <summary>
    ///     The vector of full applied forces.
    /// </summary>
    /// <remarks>
    ///     Simplified at constrained DoFs.
    /// </remarks>
    protected readonly ForceVector FullForceVector;

    /// <summary>
    ///     Auxiliary iteration list.
    /// </summary>
    protected readonly List<IIteration> Iterations = new();

    private double _loadFactor;

    /// <summary>
    ///     Get the iteration at this index.
    /// </summary>
    public IIteration this[int index] => Iterations[index];

    /// <summary>
    ///     Get the iteration at this index.
    /// </summary>
    public IIteration this[Index index] => Iterations[index];

    /// <summary>
    ///     The status of this step. True if convergence was reached.
    /// </summary>
    public bool Converged { get; protected set; }

    /// <summary>
    ///     The results of the ongoing iteration.
    /// </summary>
    public IIteration CurrentIteration => Iterations[^1];

    /// <summary>
    ///     The displacement convergence of this step.
    /// </summary>
    public double DisplacementConvergence => CurrentIteration.DisplacementConvergence;

    /// <summary>
    ///     The total displacement increment at this step.
    /// </summary>
    public DisplacementVector DisplacementIncrement => (DisplacementVector)
        (FinalDisplacements - InitialDisplacements);

    /// <summary>
    ///     The displacement vector at the end of this step.

```

```

/// </summary>
public DisplacementVector FinalDisplacements => Iterations.Last().Displacements;

/// <summary>
///     Get the first iteration of the current step.
/// </summary>
public IIteration FirstIteration { get; }

/// <summary>
///     The force convergence of this step.
/// </summary>
public double ForceConvergence => CurrentIteration.ForceConvergence;

/// <summary>
///     The force vector of this step.
/// </summary>
/// <inheritdoc cref="FullForceVector" />
public ForceVector Forces => (ForceVector) (LoadFactor * FullForceVector);

/// <summary>
///     The displacement vector at the beginning of this step.
/// </summary>
public DisplacementVector InitialDisplacements { get; }

/// <summary>
///     The results of the current solution (last solved iteration [i - 1]).
/// </summary>
public IIteration LastIteration => Iterations.Count > 1
    ? Iterations[^2]
    : Iteration.From(FullForceVector.Count, this is SimulationStep);

/// <summary>
///     The load factor of this step.
/// </summary>
public virtual double LoadFactor => _loadFactor;

/// <summary>
///     The load factor increment of this step.
/// </summary>
public virtual double LoadFactorIncrement => StepIncrement(Parameters.NumberOfSteps);

/// <summary>
///     The monitored displacement of this step.
/// </summary>
public MonitoredDisplacement? MonitoredDisplacement { get; protected set; }

/// <summary>
///     The number of this step.
/// </summary>
public int Number { get; set; }

/// <summary>
///     The analysis parameters.
/// </summary>
public AnalysisParameters Parameters { get; }

/// <summary>
///     The results of the last solution (penultimate solved iteration [i - 2]).
/// </summary>
public IIteration PenultimateIteration => Iterations.Count > 2

```

```

        ? Iterations[^3]
        : Iteration.From(FullForceVector.Count, this is SimulationStep);

    /// <summary>
    ///     The current stiffness matrix of this step (stiffness of the current iteration.
    /// </summary>
    public StiffnessMatrix Stiffness => Iterations.Last().Stiffness;

    /// <summary>
    ///     Get/set when to stop analysis.
    /// </summary>
    /// <remarks>
    ///     If true, convergence was not reached at this load step.
    /// </remarks>
    public bool Stop { get; private set; }

    /// <summary>
    ///     Create a step object.
    /// </summary>
    /// <param name="loadFactor">The load factor of this step.</param>
    /// <param name="number">The number of this step.</param>
    /// <param name="fullForceVector">The full applied force vector of the model.</param>
    /// <param name="simulate">Set true if the performed analysis is a simulation.</param>
    protected LoadStep(ForceVector fullForceVector, double loadFactor, AnalysisParameters
        parameters, int number = 0, bool simulate = false)
        : this(number, fullForceVector, loadFactor,
            DisplacementVector.Zero(fullForceVector.Count),
            StiffnessMatrix.Zero(fullForceVector.Count), parameters, simulate)
    {
    }

    /// <summary>
    ///     Create a step object.
    /// </summary>
    /// <param name="initialDisplacements">The initial displacement vector of this
    ///     step.</param>
    /// <param name="stiffness">The stiffness matrix of this step.</param>
    /// <param name="parameters">The analysis parameters.</param>
    /// <param name="simulate">Set true if the performed analysis is a simulation.</param>
    protected LoadStep(int number, ForceVector fullForceVector, double loadFactor,
        DisplacementVector initialDisplacements, StiffnessMatrix stiffness,
        AnalysisParameters parameters, bool simulate = false)
    {
        Number                = number;
        FullForceVector        = fullForceVector;
        _loadFactor            = loadFactor;
        InitialDisplacements    = initialDisplacements;
        Parameters             = parameters;
        FirstIteration          = Iteration.From(initialDisplacements,
            ForceVector.Zero(fullForceVector.Count), stiffness, simulate, loadFactor);
        Iterations.Add(FirstIteration);
    }

    /// <summary>
    ///     Create a load step for a nonlinear analysis procedure.
    /// </summary>
    /// <param name="fullForceVector">The full applied force vector of the model.</param>
    /// <param name="loadFactor">The load factor of this step.</param>
    /// <param name="initialDisplacements">The initial displacement vector of this
    ///     step.</param>

```

```

/// <param name="stiffness">The stiffness matrix of this step.</param>
/// <param name="parameters">The analysis parameters.</param>
/// <param name="stepNumber">The number of the load step.</param>
/// <param name="simulate">Set true if the performed analysis is a simulation.</param>
public static LoadStep From(ForceVector fullForceVector, double loadFactor,
    DisplacementVector initialDisplacements, StiffnessMatrix stiffness,
    AnalysisParameters parameters, int stepNumber, bool simulate = false) =>
    simulate switch
    {
        false => new LoadStep(stepNumber, fullForceVector, loadFactor,
            initialDisplacements, stiffness, parameters, simulate),
        _ => new SimulationStep(stepNumber, fullForceVector, loadFactor,
            initialDisplacements, stiffness, parameters)
    };

/// <summary>
///     Create a load step for a nonlinear analysis procedure.
/// </summary>
/// <param name="femInput">The finite element input.</param>
/// <param name="loadFactor">The load factor of the first step. </param>
/// <param name="parameters">The analysis parameters.</param>
/// <param name="stepNumber">The number of the load step.</param>
/// <param name="simulate">Set true if the performed analysis is a simulation.</param>
public static LoadStep From(IFEMInput femInput, double loadFactor, AnalysisParameters
    parameters, int stepNumber, bool simulate = false) =>
    simulate switch
    {
        false => new LoadStep(femInput.AssembleExternalForces(), loadFactor,
            parameters, stepNumber, simulate),
        _ => new SimulationStep(femInput.AssembleExternalForces(), loadFactor,
            parameters, stepNumber)
    };

/// <summary>
///     Create a load step from the last load step.
/// </summary>
/// <param name="lastStep">The last load step.</param>
/// <param name="incrementLoad">
///     Increment load of the new step? If it's a <see cref="SimulationStep" />, load
///     is not
///     increased.
/// </param>
public static LoadStep FromLastStep(LoadStep lastStep, bool incrementLoad = true)
{
    if (lastStep is SimulationStep simulationStep)
        return SimulationStep.FromLastStep(simulationStep);

    var newStep = From(lastStep.FullForceVector, lastStep.LoadFactor,
        lastStep.FinalDisplacements, lastStep.Stiffness, lastStep.Parameters,
        lastStep.Number + 1);

    if (incrementLoad)
        newStep.IncrementLoad();

    return newStep;
}

/// <summary>

```

```

///      Do the initial load step of a nonlinear analysis procedure.
/// </summary>
/// <inheritdoc>
    cref="From(ForceVector, double, DisplacementVector, StiffnessMatrix, AnalysisParameters, int, bool)"
/// </inheritdoc>
/// <returns>
///      The initial <see cref="LoadStep" />.
/// </returns>
public static LoadStep InitialStep(IFEMInput femInput, AnalysisParameters parameters)
{
    var lf = StepIncrement(parameters.NumberOfSteps);

    var step = From(femInput, lf, parameters, 1);

    var iteration = step.CurrentIteration;

    // Get the initial stiffness and force vector simplified
    iteration.Stiffness = femInput.AssembleStiffness();

    // Update internal forces
    iteration.UpdateForces(step.Forces, ForceVector.Zero(femInput.NumberOfDoFs));

    return step;
}

/// <summary>
///      Get the accumulated displacement increment from the beginning of this load
///      step until a final index.
/// </summary>
/// <param name="finalIndex">The final index to consider increments.</param>
public DisplacementVector AccumulatedDisplacementIncrement(Index finalIndex)
{
    var iterations = Iterations.Where(i => i.Number > 0).ToArray();

    DisplacementVector accD;

    try
    {
        accD = (DisplacementVector) (iterations[finalIndex].Displacements -
            InitialDisplacements);
    }
    catch
    {
        accD = DisplacementVector.Zero(InitialDisplacements.Count);
    }

    return accD;
}

/// <summary>
///      Increment forces in this step by a custom load factor increment.
/// </summary>
/// <param name="loadFactorIncrement">The increment of the load factor.</param>
public void IncrementLoad(double loadFactorIncrement) => _loadFactor +=
    loadFactorIncrement;

/// <summary>
///      Iterate to find solution.
/// </summary>
/// <param name="femInput">The finite element input.</param>

```

```

public virtual void Iterate(IFEMInput femInput)
{
    // Initiate first iteration
    foreach (var iteration in Iterations)
        iteration.Number = 0;

    // Iterate
    do
    {
        // Update stiffness
        UpdateStiffness();

        // Add iteration
        NewIteration();

        // Update displacements
        UpdateDisplacements();

        // Update elements and forces
        UpdateElements(femInput);
        UpdateForces(femInput);

        // Calculate convergence
        CurrentIteration.CalculateConvergence(Forces,
            FirstIteration.DisplacementIncrement);

    } while (!IterativeStop());

    if (!Stop && Parameters.Solver is NonLinearSolver.ModifiedNewtonRaphson)
        UpdateStiffness();
}

/// <summary>
///     Set step results after achieving convergence.
/// </summary>
public void SetResults(IFEMInput femInput, int? monitoredIndex = null)
{
    if (!monitoredIndex.HasValue)
        return;

    // Get displacement
    var disp = FinalDisplacements[monitoredIndex.Value];

    // Set to step
    MonitoredDisplacement = new MonitoredDisplacement(disp, LoadFactor);

    foreach (var element in femInput.MonitoredElements)
        element.AddValue(LoadFactor);
}

/// <inheritdoc />
public override string ToString() => $"Load_{step}_{Number}";

/// <summary>
///     Check if iterative procedure must stop by achieving convergence or achieving
///     the maximum number of iterations.
/// </summary>
/// <returns>
///     True if convergence is reached or the maximum number of iterations is reached.

```



```

/// </returns>
protected bool IterativeStop()
{
    // Check if one stop condition is reached
    Stop = CurrentIteration.CheckStopCondition(Parameters);

    // Check convergence
    Converged = CurrentIteration.CheckConvergence(Parameters);

    return
        Stop || Converged;
}

/// <summary>
///     Add a new iteration in this load step.
/// </summary>
/// <param name="simulate">Set true if the performed analysis is a simulation.</param>
protected void NewIteration(bool simulate = false)
{
    if (CurrentIteration.Number > 0)
        Iterations.Add(this.Any()
            ? this.Last().Clone()
            : Iteration.FromStepResult(this, simulate));

    // Increase iteration count
    CurrentIteration.Number++;

    // Free space and maintain only last 10 iterations
    if (Iterations.Count > 10)
        Iterations.RemoveAt(0);
}

/// <summary>
///     Update displacements in elements and calculate internal forces.
/// </summary>
protected void UpdateElements(IFEMInput femInput)
{
    // Update elements
    femInput.Grips.SetDisplacements(CurrentIteration.Displacements);
    femInput.UpdateDisplacements();
    femInput.CalculateForces();
}

/// <summary>
///     Update forces and calculate convergence.
/// </summary>
protected void UpdateForces(IFEMInput femInput) =>
    CurrentIteration.UpdateForces(Forces, femInput.AssembleInternalForces());

/// <summary>
///     Calculate the secant stiffness <see cref="Matrix{T}" /> of current iteration.
/// </summary>
protected void UpdateStiffness()
{
    // If analysis stopped or solver is modified newton raphson and step didn't
    // converge yet
    if (CurrentIteration.Number <= 1 || Stop || !Converged && Parameters.Solver is
        NonLinearSolver.ModifiedNewtonRaphson)
        return;
}

```

```

        CurrentIteration.Stiffness = (StiffnessMatrix) (CurrentIteration.Stiffness +
            StiffnessIncrement(CurrentIteration, LastIteration, Parameters.Solver));
    }

    /// <summary>
    ///     Increment forces in this step by the default load factor increment.
    /// </summary>
    private void IncrementLoad() =>
        IncrementLoad(StepIncrement(Parameters.NumberOfSteps));

    /// <summary>
    ///     Update displacements.
    /// </summary>
    private void UpdateDisplacements()
    {
        var curIt = CurrentIteration;
        var lastIt = LastIteration;

        // Calculate increment from residual
        var dUr = (DisplacementVector) (-curIt.Stiffness.Solve(lastIt.ResidualForces));
        curIt.IncrementDisplacements(dUr);

        // Update displacements in grips and elements
        // femInput.Grips.SetDisplacements(curIt.Displacements);
        // femInput.UpdateDisplacements();
    }

    /// <inheritdoc />
    public IEnumerator<IIteration> GetEnumerator() => Iterations.GetEnumerator();

    /// <inheritdoc />
    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

    /// <summary>
    ///     Check the step number.
    /// </summary>
    /// <returns>
    ///     True if the step number is equal to the right number.
    /// </returns>
    public static bool operator ==(LoadStep left, int right) => left.Number == right;

    /// <summary>
    ///     Get the number of a step.
    /// </summary>
    /// <returns>
    ///     <see cref="LoadStep.Number" />
    /// </returns>
    public static explicit operator int(LoadStep loadStep) => loadStep.Number;

    /// <inheritdoc cref="op_Equality" />
    /// <returns>
    ///     True if the step number is bigger than the right number.
    /// </returns>
    public static bool operator >(LoadStep left, int right) => left.Number > right;

    /// <inheritdoc cref="op_Equality" />
    /// <returns>
    ///     True if the step number is bigger or equal to the right number.
    /// </returns>

```

```

public static bool operator >=(LoadStep left, int right) => left.Number >= right;

/// <inheritdoc cref="op_Equality" />
/// <returns>
///     True if the step number is not equal to the right number.
/// </returns>
public static bool operator !=(LoadStep left, int right) => left.Number != right;

/// <inheritdoc cref="op_Equality" />
/// <returns>
///     True if the step number is smaller than the right number.
/// </returns>
public static bool operator <(LoadStep left, int right) => left.Number < right;

/// <inheritdoc cref="op_Equality" />
/// <returns>
///     True if the step number is smaller or equal to the right number.
/// </returns>
public static bool operator <=(LoadStep left, int right) => left.Number <= right;
}

```

Iteration.cs

```

namespace andrefmello91.FEMAnalysis;

/// <summary>
///     Class for nonlinear iteration results.
/// </summary>
public class Iteration : IIteration, ICloneable<Iteration>
{
    /// <inheritdoc />
    public double DisplacementConvergence { get; protected set; }

    /// <inheritdoc />
    public virtual DisplacementVector DisplacementIncrement { get; private set; }

    /// <inheritdoc />
    public DisplacementVector Displacements { get; protected set; }

    /// <inheritdoc />
    public double ForceConvergence { get; protected set; }

    /// <inheritdoc />
    public ForceVector InternalForces { get; set; }

    /// <inheritdoc />
    public int Number { get; set; }

    /// <inheritdoc />
    public ForceVector ResidualForces { get; private set; }

    /// <inheritdoc />
    public StiffnessMatrix Stiffness { get; set; }

    /// <inheritdoc cref="From(int,bool)" />
    protected Iteration(int numberOfDoFs)
        : this(DisplacementVector.Zero(numberOfDoFs), ForceVector.Zero(numberOfDoFs),
              StiffnessMatrix.Zero(numberOfDoFs))
    {

```

```

}

/// <inheritdoc cref="From(DisplacementVector, ForceVector, StiffnessMatrix, bool)" />
protected Iteration(DisplacementVector displacements, ForceVector residualForces,
    StiffnessMatrix stiffness)
{
    Displacements          = displacements;
    ResidualForces         = residualForces;
    Stiffness              = stiffness;
    InternalForces          = ForceVector.Zero(displacements.Count);
    DisplacementIncrement = DisplacementVector.Zero(displacements.Count);
}

/// <summary>
///     Create an iteration object.
/// </summary>
/// <param name="numberOfDoFs">The number of degrees of freedom.</param>
/// <param name="simulate">Set true if the performed analysis is a simulation.</param>
public static IIteration From(int numberOfDoFs, bool simulate = false) => simulate
    switch
    {
        false => new Iteration(numberOfDoFs),
        _      => new SimulationIteration(numberOfDoFs)
    };

/// <summary>
///     Create an iteration object.
/// </summary>
/// <param name="displacements">The displacement vector of this iteration.</param>
/// <param name="residualForces">The residual force vector of this iteration.</param>
/// <param name="stiffness">The stiffness matrix of this iteration.</param>
/// <param name="simulate">Set true if the performed analysis is a simulation.</param>
/// <param name="loadFactor">The load factor if <paramref name="simulate" /> is
    true.</param>
public static IIteration From(DisplacementVector displacements, ForceVector
    residualForces, StiffnessMatrix stiffness, bool simulate = false, double
    loadFactor = 0) => simulate switch
    {
        false => new Iteration(displacements, residualForces, stiffness),
        _      => new SimulationIteration(displacements, residualForces, stiffness,
            loadFactor)
    };

/// <summary>
///     Create an iteration from a load step result.
/// </summary>
/// <param name="loadStep">A calculated load step.</param>
/// <param name="simulate">Set true if the performed analysis is a simulation.</param>
public static IIteration FromStepResult(LoadStep loadStep, bool simulate = false) =>
    simulate switch
    {
        false => new Iteration(loadStep.FinalDisplacements,
            ForceVector.Zero(loadStep.FinalDisplacements.Count), loadStep.Stiffness),
        _      => new SimulationIteration(loadStep.FinalDisplacements,
            ForceVector.Zero(loadStep.FinalDisplacements.Count), loadStep.Stiffness,
            loadStep.LoadFactor)
    };

/// <inheritdoc />
public override string ToString() => $"Iteration_{Number}";

```

```

/// <inheritdoc />
public Iteration Clone() => new((DisplacementVector) Displacements.Clone(),
    (ForceVector) ResidualForces.Clone(), (StiffnessMatrix) Stiffness.Clone()) {
    Number = Number };

/// <inheritdoc />
public void CalculateConvergence(ForceVector appliedForces, DisplacementVector
    initialIncrement)
{
    ForceConvergence = NonlinearAnalysis.CalculateConvergence(ResidualForces,
        appliedForces);
    DisplacementConvergence =
        NonlinearAnalysis.CalculateConvergence(DisplacementIncrement,
            initialIncrement);
}

/// <inheritdoc />
public virtual bool CheckConvergence(AnalysisParameters parameters) =>
    Number >= parameters.MinIterations &&
    (ForceConvergence <= parameters.ForceTolerance ||
        DisplacementConvergence <= parameters.DisplacementTolerance);

/// <inheritdoc />
public bool CheckStopCondition(AnalysisParameters parameters) =>
    Number >= parameters.MaxIterations || ResidualForces.ContainsNaNOrInfinity() ||
    Displacements.ContainsNaNOrInfinity() || Stiffness.ContainsNaN();

/// <inheritdoc />
public void IncrementDisplacements(DisplacementVector displacementIncrement)
{
    DisplacementIncrement = displacementIncrement;
    Displacements = (DisplacementVector) (Displacements +
        displacementIncrement);
}

/// <inheritdoc />
public void UpdateForces(ForceVector appliedForces, ForceVector internalForces)
{
    InternalForces = internalForces;
    ResidualForces = (ForceVector) (internalForces - appliedForces);
}

/// <inheritdoc />
IIteration ICloneable<IIteration>.Clone() => Clone();

/// <summary>
///     Check the iteration number.
/// </summary>
/// <returns>
///     True if the iteration number is equal to the right number.
/// </returns>
public static bool operator ==(Iteration left, int right) => left.Number == right;

/// <summary>
///     Get the number of a iteration.
/// </summary>
/// <returns>
///     <see cref="Iteration.Number" />
/// </returns>

```

```

public static explicit operator int(Iteration iteration) => iteration.Number;

/// <inheritdoc cref="op_Equality" />
/// <returns>
///     True if the iteration number is bigger than the right number.
/// </returns>
public static bool operator >(Iteration left, int right) => left.Number > right;

/// <inheritdoc cref="op_Equality" />
/// <returns>
///     True if the iteration number is bigger or equal to the right number.
/// </returns>
public static bool operator >=(Iteration left, int right) => left.Number >= right;

/// <inheritdoc cref="op_Equality" />
/// <returns>
///     True if the iteration number is not equal to the right number.
/// </returns>
public static bool operator !=(Iteration left, int right) => left.Number != right;

/// <inheritdoc cref="op_Equality" />
/// <returns>
///     True if the iteration number is smaller than the right number.
/// </returns>
public static bool operator <(Iteration left, int right) => left.Number < right;

/// <inheritdoc cref="op_Equality" />
/// <returns>
///     True if the iteration number is smaller or equal to the right number.
/// </returns>
public static bool operator <=(Iteration left, int right) => left.Number <= right;
}

```

C.3.4 Saída de dados

A classe *FEMOutput* contém os resultados de todos os passos de carga e os deslocamentos monitorados em cada um deles. Também fornece o método *Export*, que exporta os resultados de deslocamentos e de todos os elementos monitorados em arquivos *CSV*.

FEMOutput.cs

```

namespace andrefmello91.FEMAnalysis;

/// <summary>
///     Output data class.
/// </summary>
public class FEMOutput : List<MonitoredDisplacement>
{

    private readonly List<ElementMonitor> _monitors;

    /// <summary>
    ///     Values of calculated step results.
    /// </summary>
    public List<LoadStep> LoadStepResults { get; }
}

```

```

/// <summary>
///     Output data constructor.
/// </summary>
/// <param name="loadStepResults">The values of step results.</param>
/// <param name="elementMonitors">The element monitors.</param>
public FEMOutput(IEnumerable<LoadStep> loadStepResults, IEnumerable<ElementMonitor>
    elementMonitors)
    : base(loadStepResults.Where(ls => ls.MonitoredDisplacement.HasValue).Select(ls =>
        ls.MonitoredDisplacement!.Value))
{
    LoadStepResults = loadStepResults
        .Where(ls => ls.Converged)
        .ToList();

    _monitors = elementMonitors.ToList();
}

/// <summary>
///     Export output data to a csv file.
/// </summary>
/// <param name="outputPath">The output file save location.</param>
/// <param name="fileName">The filename, without extension.</param>
/// <param name="unit">The required <see cref="LengthUnit" /> of
    displacements.</param>
/// <param name="delimiter">The delimiter for csv file.</param>
public void Export(string outputPath, string fileName = "FEM_Output", LengthUnit unit
    = LengthUnit.Millimeter, string delimiter = ";")
{
    // Get displacements and load factors as vectors
    var disps = this
        .Select(m => m.Displacement.ToUnit(unit).Value)
        .ToVector();

    var lfs = this
        .Select(m => m.LoadFactor)
        .ToVector();

    // Create a matrix
    var result = Matrix<double>.Build.DenseOfColumnVectors(lfs, disps);

    // Create headers
    var headers = new[] { "Load_Factor", $"Displacement_{unit.Abbrev()}" };

    // Set full save location
    var fullPath = $"{outputPath.TrimEnd('\u002F', '\u005C')}/{fileName}.csv";

    // Export
    DelimitedWriter.Write(fullPath, result, delimiter, headers);

    // Export monitors
    foreach (var monitor in _monitors)
        monitor.Export(outputPath, fileName, delimiter);
}
}

```

MonitoredDisplacement.cs

```
namespace andrefmello91.FEMAnalysis;
```

```
/// <summary>
```

```

///      Monitored displacement struct.
/// </summary>
public struct MonitoredDisplacement : IMonitoredValue<Length>,
    IEquatable<MonitoredDisplacement>, IComparable<MonitoredDisplacement>
{
    /// <summary>
    ///      Get the displacement value, at current <see cref="LoadFactor" />.
    /// </summary>
    public Length Displacement { get; }

    /// <summary>
    ///      Get the load factor associated to <see cref="Displacement" />.
    /// </summary>
    public double LoadFactor { get; }

    /// <inheritdoc />
    Length IMonitoredValue<Length>.Value => Displacement;

    /// <summary>
    ///      Monitored displacement constructor.
    /// </summary>
    /// <param name="displacement">The displacement value, at current <paramref
    name="loadFactor" />.</param>
    /// <param name="loadFactor">The load factor associated to <paramref
    name="displacement" />.</param>
    public MonitoredDisplacement(Length displacement, double loadFactor)
    {
        Displacement = displacement;
        LoadFactor    = loadFactor;
    }

    /// <inheritdoc />
    public override string ToString() =>
        $"Displacement={Displacement}\n" +
        $"LoadFactor={LoadFactor:0.00}";

    /// <inheritdoc />
    public int CompareTo(MonitoredDisplacement other) =>
        LoadFactor.Approx(other.LoadFactor, 1E-6)
        ? 0
        : LoadFactor > other.LoadFactor
        ? 1
        : -1;

    /// <inheritdoc />
    public bool Equals(MonitoredDisplacement other) =>
        LoadFactor.Approx(other.LoadFactor, 1E-6) &&
        Displacement.Approx(other.Displacement, PlaneDisplacement.Tolerance);

    /// <inheritdoc />
    int IComparable<IMonitoredValue<Length>>.CompareTo(IMonitoredValue<Length>? other) =>
        other is MonitoredDisplacement md
        ? CompareTo(md)
        : 0;

    /// <inheritdoc />
    bool IEquatable<IMonitoredValue<Length>>.Equals(IMonitoredValue<Length>? other) =>
        other is MonitoredDisplacement md && Equals(md);

```

```

    public static implicit operator MonitoredValue(MonitoredDisplacement
        monitoredDisplacement) => new(monitoredDisplacement.Displacement.Value,
        monitoredDisplacement.LoadFactor);
}

```

C.3.5 Extensões

A classe estática *Extensions* apresenta métodos que podem ser utilizados por objetos que implementam as interfaces *IFiniteElement*, *IGrip* e *IFEMInput*.

Extensions.cs

```

namespace andrefmello91.FEMAnalysis;

/// <summary>
///     Extensions for <see cref="IGrip" /> and <see cref="IFiniteElement" />.
/// </summary>
public static class Extensions
{
    /// <summary>
    ///     Assemble the global displacement vector.
    /// </summary>
    public static DisplacementVector AssembleDisplacements(this IFEMInput femInput) =>
        new(femInput.Grips.Select(x => x.Displacement));

    /// <summary>
    ///     Assemble the global external force vector.
    /// </summary>
    /// <param name="simplify">Simplify the vector at constraint indexes?</param>
    public static ForceVector AssembleExternalForces(this IFEMInput femInput, bool
        simplify = true)
    {
        var forces = new ForceVector(femInput.Grips.Select(x => x.Force));

        return simplify
            ? forces.Simplified(femInput.ConstraintIndex)
            : forces;
    }

    /// <summary>
    ///     Assemble the global internal force vector.
    /// </summary>
    /// <inheritdoc cref="AssembleExternalForces" />
    public static ForceVector AssembleInternalForces(this IFEMInput femInput, bool
        simplify = true)
    {
        var fi = ForceVector.Zero(femInput.NumberOfDoFs);

        foreach (var element in femInput)
        {
            var dofIndex = element.DoFIndex;

            for (var i = 0; i < dofIndex.Length; i++)
            {
                // DoF index
                var j = dofIndex[i];

```

```

        // Add values
        fi[j] += element.Forces[i];
    }
}

return simplify
    ? (ForceVector) fi.Simplified(femInput.ConstraintIndex, (double?) null)
    : fi;
}

/// <summary>
///     Assemble the global stiffness matrix.
/// </summary>
public static StiffnessMatrix AssembleStiffness(this IFEMInput femInput)
{
    var stiffness = StiffnessMatrix.Zero(femInput.NumberOfDoFs);
    stiffness.ConstraintIndex = femInput.ConstraintIndex;

    foreach (var element in femInput)
    {
        var dofIndex = element.DoFIndex;

        for (var i = 0; i < dofIndex.Length; i++)
        {
            // Global index
            var k = dofIndex[i];

            for (var j = 0; j < dofIndex.Length; j++)
            {
                // Global index
                var l = dofIndex[j];

                stiffness[k, l] += element.Stiffness[i, j];
            }
        }
    }

    return stiffness;
}

/// <summary>
///     Calculate forces in each element in this collection after updating
///     displacements in grips.
/// </summary>
/// <inheritdoc cref="IFiniteElement.CalculateForces" />
public static void CalculateForces<TFiniteElement>(this IEnumerable<TFiniteElement>
    elements)
    where TFiniteElement : IFiniteElement
{
    foreach (var element in elements)
        element.CalculateForces();
}

/// <summary>
///     Return an element of a collection, in given <paramref name="number" />.
/// </summary>
/// <param name="elements">The collection of <see cref="INumberedElement"
/// </s>s.</param>

```

```

/// <param name="number">The number of the element wanted.</param>
public static TNumberedElement? GetByNumber<TNumberedElement>(this
    IEnumerable<TNumberedElement> elements, int number)
    where TNumberedElement : INumberedElement =>
    elements.FirstOrDefault(element => number == element.Number);

/// <summary>
///     Get the indexes of constrained degrees of freedom from a collection of grips.
/// </summary>
/// <param name="grips">The collection containing all distinct grips of the finite
///     element model.</param>
public static IEnumerable<int> GetConstraintIndex(this IEnumerable<IGrip> grips)
{
    foreach (var grip in grips)
    {
        // Get DoF indexes
        var index = grip.DoFIndex;
        int
            i = index[0],
            j = index[1];

        var constraint = grip.Constraint;

        if (constraint.X)

            // There is a support in X direction
            yield return i;

        if (constraint.Y)

            // There is a support in Y direction
            yield return j;
    }
}

/// <summary>
///     Assemble the element displacement vector from it's grips.
/// </summary>
/// <param name="element">The finite element.</param>
public static DisplacementVector GetDisplacements(this IFiniteElement element) =>
    new(element.Grips.Select(g => g.Displacement));

/// <summary>
///     Get the displacement <see cref="Vector" /> from this element's grips.
/// </summary>
/// <returns>
///     The displacement <see cref="Vector" />, with components in <see
///     cref="LengthUnit.Millimeter" />.
/// </returns>
public static DisplacementVector GetDisplacementsFromGrips([NotNull] this
    IFiniteElement element) =>
    new(element.Grips
        .SelectMany(g => new[] { g.Displacement.X, g.Displacement.Y }));

/// <summary>
///     Get global indexes of the degrees of freedom of a collection of grips.
/// </summary>
/// <param name="grips">A collection of <see cref="IGrip" />'s.</param>
public static IEnumerable<int> GlobalIndexes(params IGrip[] grips)
{

```

```

        foreach (var grip in grips)
        {
            var n = 2 * grip.Number;

            yield return n - 2;
            yield return n - 1;
        }
    }

    /// <summary>
    ///     Set displacements to an <see cref="IGrip" /> from the global displacement
    ///     <see cref="Vector" />.
    /// </summary>
    /// <param name="grip">The <see cref="IGrip" /> to set displacements.</param>
    /// <param name="globalDisplacementVector">The global displacement vector.</param>
    public static void SetDisplacements([NotNull] this IGrip grip, [NotNull]
        DisplacementVector globalDisplacementVector)
    {
        var x = globalDisplacementVector[grip.DoFIndex[0]];
        var y = globalDisplacementVector[grip.DoFIndex[1]];

        grip.Displacement = new PlaneDisplacement(x, y);
    }

    /// <summary>
    ///     Set displacements to a collection of <see cref="IGrip" />'s from the global
    ///     displacement <see cref="Vector" />.
    /// </summary>
    /// <param name="grips">The collection of <see cref="IGrip" />'s to set
    ///     displacements.</param>
    /// <param name="globalDisplacementVector">The global displacement vector.</param>
    public static void SetDisplacements([NotNull] this IEnumerable<IGrip> grips,
        [NotNull] DisplacementVector globalDisplacementVector)
    {
        foreach (var grip in grips)
            grip.SetDisplacements(globalDisplacementVector);
    }

    /// <summary>
    ///     Set support reactions to an <see cref="IGrip" /> from the global reaction
    ///     <see cref="Vector" />.
    /// </summary>
    /// <param name="grip">The <see cref="IGrip" /> to set support reactions.</param>
    /// <param name="globalReactionVector">The global reaction vector.</param>
    public static void SetReactions([NotNull] this IGrip grip, [NotNull] ForceVector
        globalReactionVector)
    {
        // Verify if grip is free
        if (grip.Constraint.Direction is ComponentDirection.None)
            return;

        var x = globalReactionVector[grip.DoFIndex[0]];
        var y = globalReactionVector[grip.DoFIndex[1]];

        grip.Reaction = new PlaneForce(x, y);
    }

    /// <summary>
    ///     Set support reactions to a collection of <see cref="IGrip" />'s from the
    ///     global reaction <see cref="Vector" />.

```

```

/// </summary>
/// <param name="grips">The collection of <see cref="IGrip" />'s to set support
    reactions.</param>
/// <param name="globalReactionVector">The global reaction vector.</param>
public static void SetReactions([NotNull] this IEnumerable<IGrip> grips, [NotNull]
    ForceVector globalReactionVector)
{
    foreach (var grip in grips)
        grip.SetReactions(globalReactionVector);
}

/// <summary>
///     Update the displacement vector for this element.
/// </summary>
/// <inheritdoc cref="UpdateStiffness" />
public static void UpdateDisplacements([NotNull] this IEnumerable<IFiniteElement>
    elements)
{
    foreach (var element in elements)
        element.UpdateDisplacements();
}

/// <summary>
///     Update the stiffness of each element in this collection.
/// </summary>
/// <param name="elements">The collection of finite elements.</param>
public static void UpdateStiffness(this IEnumerable<IFiniteElement> elements)
{
    foreach (var element in elements)
        element.UpdateStiffness();
}
}

```

C.4 *SPMElements*

A biblioteca *SPMElements* (MELLO, 2021b) implementa os elementos finitos do MBP a partir das interfaces *IFiniteElement* e *IGrip*, da biblioteca *FEMAnalysis*. A programação dos elementos é baseada na formulação apresentada nos Apêndices A e B.

C.4.1 Classe base

A interface *ISPMElement* apresenta as propriedades dos elementos, sendo que a propriedade principal representa a geometria do elemento. A interface *INonlinearSPMElement* contém as propriedades específicas dos elementos não-lineares.

ISPMElement.cs

```

namespace andrefmello91.SPMElements;

/// <summary>

```

```

///      Interface for SPM elements
/// </summary>
public interface ISPMElement : IFiniteElement
{
    /// <summary>
    ///      The nodes of this element.
    /// </summary>
    new Node[] Grips { get; }

    /// <summary>
    ///      The <see cref="ElementModel" /> of this SPM element.
    /// </summary>
    ElementModel Model { get; }

    /// <summary>
    ///      The name of this element.
    /// </summary>
    string Name { get; }
}

/// <summary>
///      Generic interface for SPM elements
/// </summary>
/// <typeparam name="TGeometry">The struct that represents the geometry of the
///      element.</typeparam>
public interface ISPMElement<out TGeometry> : ISPMElement
    where TGeometry : struct, IEquatable<TGeometry>, IComparable<TGeometry>
{
    /// <summary>
    ///      The geometry of this element.
    /// </summary>
    TGeometry Geometry { get; }
}

/// <summary>
///      Interface for nonlinear SPM elements.
/// </summary>
public interface INonlinearSPMElement : ISPMElement, IMonitoredElement
{
    /// <summary>
    ///      True if concrete is cracked in this element.
    /// </summary>
    bool ConcreteCracked { get; }

    /// <summary>
    ///      True if concrete crushed in this element.
    /// </summary>
    bool ConcreteCrushed { get; }

    /// <summary>
    ///      True if concrete yielded in this element.
    /// </summary>
    bool ConcreteYielded { get; }

    /// <summary>
    ///      True if steel yielded in this element.
    /// </summary>

```

```

    bool SteelYielded { get; }

    /// <summary>
    ///     Execute when the state of this element changes.
    /// </summary>
    event EventHandler<StateEventArgs>? StateChanged;
}

```

A classe base *SPMElement* implementa a interface *ISPMElement* e fornece as propriedades necessárias aos elementos *Stringer* e *Panel*.

SPMElement.cs

```

namespace andrefmello91.SPMElements;

/// <summary>
///     Base class for SPM elements.
/// </summary>
/// <inheritdoc cref="ISPMElement{TGeometry}" />
public abstract class SPMElement<TGeometry> : ISPMElement<TGeometry>,
    IEquatable<SPMElement<TGeometry>>, IComparable<SPMElement<TGeometry>>
    where TGeometry : struct, IEquatable<TGeometry>, IComparable<TGeometry>
{

    /// <summary>
    ///     Get the grip numbers of this element.
    /// </summary>
    public int[] GripNumbers => Grips.Select(g => g.Number).ToArray();

    /// <summary>
    ///     Get the maximum local force at this element.
    /// </summary>
    public abstract Force MaxForce { get; }

    /// <inheritdoc />
    public ElementMonitor? Monitor { get; protected set; }

    /// <inheritdoc />
    public abstract bool Monitored { get; set; }

    /// <summary>
    ///     Get the displacement vector, in local coordinate system.
    /// </summary>
    protected DisplacementVector LocalDisplacements { get; set; }

    /// <summary>
    ///     Get the force vector, in local coordinate system.
    /// </summary>
    protected ForceVector LocalForces { get; set; }

    /// <summary>
    ///     Get the local stiffness matrix.
    /// </summary>
    protected StiffnessMatrix LocalStiffness { get; set; }

    /// <summary>
    ///     Get the transformation matrix to transform from local to global coordinate
    ///     systems.
    /// </summary>

```

```

protected Matrix<double> TransformationMatrix { get; set; }

/// <inheritdoc />
public DisplacementVector Displacements { get; protected set; }

/// <inheritdoc />
public int[] DoFIndex => GlobalIndexes(Grips).ToArray();

/// <inheritdoc />
public ForceVector Forces { get; protected set; }

/// <inheritdoc />
public TGeometry Geometry { get; }

/// <inheritdoc />
public Node[] Grips { get; }

/// <summary>
///     The <see cref="ElementModel" /> of this SPM element.
/// </summary>
public ElementModel Model => this switch
{
    NLStringer or NLPanel => ElementModel.Nonlinear,
    -                      => ElementModel.Elastic
};

/// <inheritdoc />
public abstract string Name { get; }

/// <inheritdoc />
public int Number { get; set; }

/// <inheritdoc />
public StiffnessMatrix Stiffness { get; protected set; }

/// <inheritdoc />
IGrip[] IFiniteElement.Grips => Grips.Cast<IGrip>().ToArray();

/// <summary>
///     SPM element base constructor.
/// </summary>
/// <param name="geometry">The element's geometry.</param>
/// <param name="grips">The collection of grips of this element.</param>
protected SPMElement(TGeometry geometry, IEnumerable<Node> grips)
{
    Geometry          = geometry;
    Grips              = grips.ToArray();
    LocalForces        = ForceVector.Zero(Grips.Length);
    LocalDisplacements = DisplacementVector.Zero(Grips.Length);
    Forces              = ForceVector.Zero(2 * Grips.Length);
    Displacements      = DisplacementVector.Zero(2 * Grips.Length);
}

/// <inheritdoc />
public override bool Equals(object? obj) => obj is SPMElement<TGeometry> spmElement
    && Equals(spmElement);

/// <inheritdoc />
public override int GetHashCode() => Geometry.GetHashCode();

```

```

    /// <inheritdoc />
    public int CompareTo(SPMElement<TGeometry>? other) =>
        other?.Geometry.CompareTo(other.Geometry) ?? 0;

    /// <inheritdoc />
    public bool Equals(SPMElement<TGeometry>? other) => other is not null &&
        Geometry.Equals(other.Geometry);

    /// <inheritdoc />
    public virtual void CalculateForces()
    {
        // Calculate local forces
        LocalForces = LocalStiffness * LocalDisplacements;

        // Approximate small values to zero
        Forces = (ForceVector) (TransformationMatrix.Transpose() * LocalForces);
    }

    /// <inheritdoc />
    public virtual void UpdateDisplacements()
    {
        Displacements      = this.GetDisplacementsFromGrips();
        LocalDisplacements = (DisplacementVector) (TransformationMatrix * Displacements);
    }

    /// <inheritdoc />
    public abstract void UpdateStiffness();

    /// <inheritdoc />
    int IComparable<IFiniteElement>.CompareTo(IFiniteElement? other) => other is
        SPMElement<TGeometry> spmElement
        ? CompareTo(spmElement)
        : 0;

    /// <inheritdoc />
    bool IEquatable<IFiniteElement>.Equals(IFiniteElement? other) => other is
        SPMElement<TGeometry> spmElement && Equals(spmElement);

    /// <summary>
    ///     Returns true if arguments are equal.
    /// </summary>
    public static bool operator ==(SPMElement<TGeometry>? left, SPMElement<TGeometry>?
        right) => left?.IsEqualTo(right);

    /// <summary>
    ///     Returns true if arguments are different.
    /// </summary>
    public static bool operator !=(SPMElement<TGeometry>? left, SPMElement<TGeometry>?
        right) => left?.IsNotEqualTo(right);
}

```

C.4.2 Elemento nó

A classe *Node* implementa a interface *IGrip* e representa as regiões nodais do MBP. Tem como propriedades principais posição, forças aplicadas e deslocamentos.

Node.cs

```

namespace andrefmello91.SPMElements;

/// <summary>
///     Node class.
/// </summary>
public class Node : IGrip, IMonitoredElement, IEquatable<Node>, IComparable<Node>
{

    /// <summary>
    ///     Get/set the <see cref="LengthUnit" /> of <see cref="Displacement" />.
    /// </summary>
    public LengthUnit DisplacementUnit
    {
        get => Displacement.Unit;
        set => Displacement.ChangeUnit(value);
    }

    /// <summary>
    ///     Get/set the <see cref="UnitsNet.Units.ForceUnit" /> of <see cref="Force" />.
    /// </summary>
    public ForceUnit ForceUnit
    {
        get => Force.Unit;
        set => Force.ChangeUnit(value);
    }

    /// <summary>
    ///     Get/set the <see cref="LengthUnit" /> of <see cref="Position" />.
    /// </summary>
    public LengthUnit GeometryUnit
    {
        get => Position.Unit;
        set => Position.ChangeUnit(value);
    }

    /// <summary>
    ///     Returns true if the node is free.
    /// </summary>
    public bool IsFree => Constraint.Direction is ComponentDirection.None;

    /// <summary>
    ///     Get the position of the node.
    ///     <para>See: <see cref="Point" />.</para>
    /// </summary>
    public Point Position { get; }

    /// <summary>
    ///     Get the node type (<see cref="NodeType" />).
    /// </summary>
    public NodeType Type { get; }

    /// <summary>
    ///     Get/set <see cref="OnPlaneComponents.Constraint" /> condition.
    /// </summary>
    public Constraint Constraint { get; set; } = Constraint.Free;

    /// <summary>
    ///     Get/set nodal <see cref="PlaneDisplacement" />

```

```

/// </summary>
public PlaneDisplacement Displacement { get; set; } = PlaneDisplacement.Zero;

/// <inheritdoc />
public int[] DoFIndex => GlobalIndexes(this).ToArray();

/// <summary>
///     Get/set applied <see cref="Force" />.
/// </summary>
public PlaneForce Force { get; set; } = PlaneForce.Zero;

/// <inheritdoc />
public string Name => $"Node_{Number}";

/// <inheritdoc />
public int Number { get; set; }

/// <inheritdoc />
public PlaneForce Reaction { get; set; }

/// <inheritdoc />
public ElementMonitor? Monitor { get; private set; }

/// <summary>
///     Get/set monitoring of this node.
/// </summary>
public bool Monitored
{
    get => Monitor is not null;
    set
    {
        if (value)
            Monitor ??= new GripMonitor(Name, DisplacementUnit);
        else
            Monitor = null;
    }
}

/// <summary>
///     Node object.
/// </summary>
/// <param name="position">The <seealso cref="Point" /> position.</param>
/// <param name="type">The <see cref="NodeType" />.</param>
/// <param name="displacementUnit">The <see cref="LengthUnit" /> of <see
    cref="Displacement" />.</param>
public Node(Point position, NodeType type, LengthUnit displacementUnit =
    LengthUnit.Millimeter)
{
    // Get the position
    Position = position;

    // Get type
    Type = type;

    // Set units
    DisplacementUnit = displacementUnit;
}

/// <summary>

```

```

///      Returns true if <paramref name="other" /> is a <see cref="Node" /> and both
        positions are equal.
/// </summary>
/// <param name="other">The other <see cref="Node" /> object.</param>
public override bool Equals(object? other) => other is Node otherNode &&
    Equals(otherNode);

/// <summary>
///      Return the angle, related to horizontal axis, of a line that connects this to
        <paramref name="otherNode" /> (in
///      radians).
/// </summary>
/// <inheritdoc cref="GetDistance" />
public double GetAngle(Node? otherNode) => otherNode is not null ?
    Position.GetAngle(otherNode.Position) : 0;

/// <summary>
///      Return the distance to another <see cref="Node" />.
/// </summary>
/// <param name="otherNode">The other <see cref="Node" /> object.</param>
public Length GetDistance(Node? otherNode) => otherNode is not null ?
    Position.GetDistance(otherNode.Position) : Length.Zero;

/// <inheritdoc />
public override int GetHashCode() => Position.GetHashCode();

/// <inheritdoc />
public override string ToString()
{
    var msgstr =
        $"Node_{Number}\n" +
        $"Position:\t({Position.X},{Position.Y})\n" +
        $"DoFIndex:\t{DoFIndex.Select(i=>i.ToString()).Aggregate((i,f)=>_{i} -
            {f})}";

    // Read applied forces
    if (!Force.IsZero)
        msgstr +=
            "\n\n" +
            "Applied_forces:\n" +
            $"{Force}";

    // Get supports
    if (!IsFree)
        msgstr +=
            "\n\n" +
            $"Constraints:\t{Constraint.Direction}";

    // Get displacements
    if (!Displacement.IsZero)
        msgstr +=
            "\n\n" +
            "Displacements:\n" +
            $"{Displacement}";

    return msgstr;
}

/// <inheritdoc />
public int CompareTo(Node? other) => other is null

```

```

        ? 1
        : Position.CompareTo(other.Position);

    /// <summary>
    ///     Returns true if both nodes positions are equal.
    /// </summary>
    /// <param name="other">The other <see cref="Node" /> object.</param>
    public bool Equals(Node? other) => other is not null && Position == other.Position;

    /// <inheritdoc />
    int IComparable<IGrip>.CompareTo(IGrip? other) => other is Node node
        ? CompareTo(node)
        : 0;

    /// <inheritdoc />
    bool IEquatable<IGrip>.Equals(IGrip? other) => other is Node node && Equals(node);

    /// <inheritdoc />
    public void AddValue(double loadFactor) => Monitor?.AddMonitoredValue(loadFactor,
        this);

    /// <summary>
    ///     Returns true if both nodes positions are equal.
    /// </summary>
    public static bool operator ==(Node? left, Node? right) => left is not null &&
        left.Equals(right);

    /// <summary>
    ///     Returns true if both nodes positions are different.
    /// </summary>
    public static bool operator !=(Node? left, Node? right) => left is not null &&
        !left.Equals(right);
}

```

C.4.3 Elemento biela

C.4.3.1 Geometria

A estrutura *StringerGeometry* fornece as propriedades geométricas de uma biela, como seção transversal (pela estrutura *CrossSection*), pontos de início e fim, ângulo e comprimento.

StringerGeometry.cs

```

namespace andrefmello91.SPMElements.StringerProperties;

/// <summary>
///     Stringer geometry struct.
/// </summary>
public struct StringerGeometry : IUnitConvertible<LengthUnit>,
    IApproachable<StringerGeometry, Length>, IEquatable<StringerGeometry>,
    IComparable<StringerGeometry>, ICloneable<StringerGeometry>
{
    private CrossSection _section;

```

```

/// <summary>
///     The stringer angle, in radians.
/// </summary>
public double Angle { get; }

/// <summary>
///     Get the center <see cref="Point" /> of <see cref="Stringer" />.
/// </summary>
public Point CenterPoint { get; }

/// <summary>
///     Get the connected <see cref="Point" />'s.
/// </summary>
public Point[] ConnectedPoints => new[] { InitialPoint, CenterPoint, EndPoint };

/// <summary>
///     Get the cross-section.
/// </summary>
public CrossSection CrossSection
{
    get => _section;
    set => _section = value.Convert(Unit);
}

/// <summary>
///     Get the final <see cref="Point" /> of <see cref="Stringer" />.
/// </summary>
public Point EndPoint { get; }

/// <summary>
///     Get the initial <see cref="Point" /> of <see cref="Stringer" />.
/// </summary>
public Point InitialPoint { get; }

/// <summary>
///     The stringer length.
/// </summary>
public Length Length { get; private set; }

/// <summary>
///     Get/set the <see cref="LengthUnit" /> that this was constructed with.
/// </summary>
public LengthUnit Unit
{
    get => InitialPoint.Unit;
    set => ChangeUnit(value);
}

/// <inheritdoc cref="StringerGeometry(Point, Point, Length, Length)" />
/// <param name="unit">
///     The <see cref="LengthUnit" /> of <paramref name="width" />, <paramref
///     name="height" /> and nodes' coordinates.
///     <para>Default: <seealso cref="LengthUnit.Millimeter" />.</para>
/// </param>
public StringerGeometry(Point initialPoint, Point endPoint, double width, double
    height, LengthUnit unit = LengthUnit.Millimeter)
    : this(initialPoint.Convert(unit), endPoint, (Length) width.As(unit), (Length)
        height.As(unit))
{

```

```

}

/// <inheritdoc cref="StringerGeometry(Point, Point, CrossSection)" />
/// <param name="width">The stringer width.</param>
/// <param name="height">The stringer height.</param>
public StringerGeometry(Point initialPoint, Point endPoint, Length width, Length
    height)
    : this(initialPoint, endPoint, new CrossSection(width, height))
{
}

/// <summary>
///     Stringer geometry object.
/// </summary>
/// <param name="initialPoint">
///     The initial <see cref="Point" /> of the <see cref="Stringer" />.
/// </param>
/// <param name="endPoint">
///     The final <see cref="Point" /> of the <see cref="Stringer" />.
/// </param>
/// <param name="crossSection">The stringer cross-section.</param>
public StringerGeometry(Point initialPoint, Point endPoint, CrossSection crossSection)
{
    InitialPoint = initialPoint;
    EndPoint      = endPoint.Convert(initialPoint.Unit);
    CenterPoint   = initialPoint.MidPoint(endPoint);

    // Calculate length and angle
    Length = initialPoint.GetDistance(endPoint);
    Angle  = initialPoint.GetAngle(endPoint);

    // Set values
    _section = crossSection;
}

/// <summary>
///     Convert this <see cref="StringerGeometry" /> object to another <see
///     cref="LengthUnit" />.
/// </summary>
/// <param name="unit">The desired <see cref="LengthUnit" />.</param>
public StringerGeometry Convert(LengthUnit unit) =>
    unit == Unit
        ? this
        : new StringerGeometry(InitialPoint.Convert(unit), EndPoint.Convert(unit),
            CrossSection.Convert(unit));

/// <summary>
///     Divide a <see cref="StringerGeometry" /> in a <paramref name="number" /> of
///     new ones.
/// </summary>
/// <param name="number">The number of new <see cref="StringerGeometry" />'s. Must be
///     bigger than 1.</param>
/// <returns>
///     An empty collection if <paramref name="number" /> is smaller than 2.
/// </returns>
public IEnumerable<StringerGeometry> Divide(int number)
{
    if (number <= 1)
        yield break;

```

```

var iPt = InitialPoint;
var ePt = EndPoint;

// Calculate distances
var dx = (ePt.X - iPt.X) / number;
var dy = (ePt.Y - iPt.Y) / number;

for (var i = 0; i < number; i++)
{
    // Get end point
    ePt = new Point(iPt.X + dx, iPt.Y + dy);

    // Return a divided geometry
    yield return new StringerGeometry(iPt, ePt, CrossSection);

    // Set initial point
    iPt = ePt;
}
}

/// <inheritdoc />
public override bool Equals(object obj) => obj is StringerGeometry other &&
    Equals(other);

/// <summary>
///     Returns true if <see cref="CrossSection" /> of <paramref name="other" />
///     coincide to this object.
/// </summary>
/// <inheritdoc cref="CompareTo" />
public bool EqualsCrossSection(StringerGeometry other) => CrossSection ==
    other.CrossSection;

/// <inheritdoc />
public override int GetHashCode() => InitialPoint.GetHashCode() *
    EndPoint.GetHashCode();

/// <inheritdoc />
public override string ToString() =>
    $"Lenght_{Length}\n" +
    $"Angle_{Angle.ToDegree():0.00}_deg\n" +
    CrossSection;

/// <inheritdoc />
public bool Approaches(StringerGeometry other, Length tolerance) =>
    InitialPoint.Approaches(other.InitialPoint, tolerance) &&
    EndPoint.Approaches(other.EndPoint, tolerance) ||
    InitialPoint.Approaches(other!.EndPoint, tolerance) &&
    EndPoint.Approaches(other.InitialPoint, tolerance);

/// <inheritdoc />
public StringerGeometry Clone() => new(InitialPoint, EndPoint, CrossSection.Clone());

/// <summary>
///     Compare this <see cref="StringerGeometry" /> to <paramref name="other" />,
///     based on <see cref="CenterPoint" />.
/// </summary>
/// <remarks>
///     See: <seealso cref="Point.CompareTo" />.
/// </remarks>

```

```

    /// <param name="other">The <see cref="StringerGeometry" /> to compare.</param>
    public int CompareTo(StringerGeometry other) =>
        CenterPoint.CompareTo(other.CenterPoint);

    /// <summary>
    ///     Returns true if <see cref="InitialPoint" /> and <seealso cref="EndPoint" />
    ///     of <paramref name="other" /> coincide.
    /// </summary>
    /// <inheritdoc cref="CompareTo" />
    public bool Equals(StringerGeometry other) => Approaches(other, Point.Tolerance);

    /// <summary>
    ///     Change the <see cref="LengthUnit" /> of this.
    /// </summary>
    /// <param name="unit">The <see cref="LengthUnit" /> to convert.</param>
    public void ChangeUnit(LengthUnit unit)
    {
        if (Unit == unit)
            return;

        InitialPoint.ChangeUnit(unit);
        EndPoint.ChangeUnit(unit);
        CenterPoint.ChangeUnit(unit);
        CrossSection.ChangeUnit(unit);

        Length = Length.ToUnit(unit);
    }

    IUnitConvertible<LengthUnit> IUnitConvertible<LengthUnit>.Convert(LengthUnit unit) =>
        Convert(unit);

    /// <summary>
    ///     Returns true if objects are equal.
    ///     <para>See: <seealso cref="Equals(StringerGeometry)" />.</para>
    /// </summary>
    public static bool operator ==(StringerGeometry left, StringerGeometry right) =>
        left.Equals(right);

    /// <summary>
    ///     Returns true if objects are different.
    ///     <para>See: <seealso cref="Equals(StringerGeometry)" />.</para>
    /// </summary>
    public static bool operator !=(StringerGeometry left, StringerGeometry right) =>
        !left.Equals(right);
}

```

CrossSection.cs

```

namespace andrefmello91.SPMElements.StringerProperties;

/// <summary>
///     Stringer cross-section struct.
/// </summary>
public struct CrossSection : IUnitConvertible<LengthUnit>, IApproachable<CrossSection,
    Length>, IEquatable<CrossSection>, IComparable<CrossSection>,
    ICloneable<CrossSection>
{
    private Length _width, _height;

```

```

/// <summary>
///     Get the cross-section area.
/// </summary>
public Area Area => (Width * Height).ToUnit(Unit.GetAreaUnit());

/// <summary>
///     Get/set the cross-section height.
/// </summary>
public Length Height
{
    get => _height;
    set => _height = value.ToUnit(Unit);
}

/// <summary>
///     Get/set the cross-section width.
/// </summary>
public Length Width
{
    get => _width;
    set => _width = value.ToUnit(Unit);
}

/// <inheritdoc />
public LengthUnit Unit
{
    get => _width.Unit;
    set => ChangeUnit(value);
}

/// <summary>
///     Stringer cross-section constructor.
/// </summary>
/// <param name="width">The stringer width.</param>
/// <param name="height">The stringer height.</param>
public CrossSection(Length width, Length height)
{
    _width = width;
    _height = height.ToUnit(_width.Unit);
}

/// <param name="unit">
///     The <see cref="LengthUnit" /> of <paramref name="width" />, <paramref
///     name="height" /> and nodes' coordinates.
///     <para>Default: <seealso cref="LengthUnit.Millimeter" />.</para>
/// </param>
/// <inheritdoc cref="CrossSection(Length, Length)" />
public CrossSection(double width, double height, LengthUnit unit =
    LengthUnit.Millimeter)
    : this((Length) width.As(unit), (Length) height.As(unit))
{
}

/// <inheritdoc cref="IUnitConvertible{TUnit}.Convert" />
public CrossSection Convert(LengthUnit unit) => new(Width.ToUnit(unit),
    Height.ToUnit(unit));

/// <inheritdoc />
public override string ToString() =>
    $"Width={Width}\n" +

```

```

        $"Height_{Width}={Height}\n";

    /// <inheritdoc />
    public bool Approaches(CrossSection other, Length tolerance) =>
        Width.Approx(other.Width, tolerance) && Height.Approx(other.Height, tolerance);

    /// <inheritdoc />
    public CrossSection Clone() => new(Width, Height);

    /// <inheritdoc />
    public int CompareTo(CrossSection other) =>
        Width == other.Width && Height == other.Height
        ? 0
        : Width > other.Width || Width >= other.Width && Height > other.Height
        ? 1
        : -1;

    /// <inheritdoc />
    public bool Equals(CrossSection other) => Approaches(other, Point.Tolerance);

    /// <inheritdoc />
    public void ChangeUnit(LengthUnit unit)
    {
        if (Unit == unit)
            return;

        _width = _width.ToUnit(unit);
        _height = _height.ToUnit(unit);
    }

    IUnitConvertible<LengthUnit> IUnitConvertible<LengthUnit>.Convert(LengthUnit unit) =>
        Convert(unit);

    /// <summary>
    ///     Returns true if objects are equal.
    /// </summary>
    public static bool operator ==(CrossSection left, CrossSection right) =>
        left.Equals(right);

    /// <summary>
    ///     Returns true if objects are different.
    /// </summary>
    public static bool operator !=(CrossSection left, CrossSection right) =>
        !left.Equals(right);
}

```

C.4.3.2 Linear

A classe *Stringer* tem como propriedade principal o objeto de geometria *Stringer-Geometry* e fornece as propriedades básicas do elemento biela, como a matriz rigidez e os vetores de força e deslocamento (grau 6). Conta também com as propriedades de materiais uniaxiais (*UniaxialConcrete* e *UniaxialReinforcement*).

Stringer.cs

```

namespace andrefmello91.SPMElements;

```

```

/// <summary>
///     Stringer base class with linear properties.
/// </summary>
public class Stringer : SPMElement<StringerGeometry>
{
    /// <summary>
    ///     Get the <see cref="UniaxialConcrete" /> of this stringer.
    /// </summary>
    public UniaxialConcrete Concrete { get; protected set; }

    /// <summary>
    ///     Get crack openings in start, mid and end nodes.
    /// </summary>
    public virtual Length[] CrackOpenings { get; }

    /// <summary>
    ///     Get the initial <see cref="Node" /> of this stringer.
    /// </summary>
    public Node Grip1 => Grips[0];

    /// <summary>
    ///     Get the center <see cref="Node" /> of this stringer.
    /// </summary>
    public Node Grip2 => Grips[1];

    /// <summary>
    ///     Get the end <see cref="Node" /> of this stringer.
    /// </summary>
    public Node Grip3 => Grips[2];

    /// <inheritdoc />
    public override Force MaxForce => LocalForces.AbsoluteMaximum();

    /// <inheritdoc />
    public override bool Monitored
    {
        get => Monitor is not null;
        set
        {
            if (value)
                Monitor ??= new StringerMonitor(Name);
            else
                Monitor = null;
        }
    }

    /// <inheritdoc />
    public override string Name => $"Stringer_{Number}";

    /// <summary>
    ///     Get normal forces acting in the stringer.
    /// </summary>
    public (Force N1, Force N3) NormalForces => (-LocalForces[0], LocalForces[2]);

    /// <summary>
    ///     Get the <see cref="UniaxialReinforcement" /> of this element.
    /// </summary>
    public UniaxialReinforcement? Reinforcement { get; protected set; }
}

```

```

/// <summary>
///     Get the state of forces acting at this stringer.
/// </summary>
public StringerForceState State
{
    get
    {
        var (N1, N3) = NormalForces;

        if (N1.ApproxZero(PlaneForce.Tolerance) && N3.ApproxZero(PlaneForce.Tolerance))
            return StringerForceState.Unloaded;

        if (N1 >= Force.Zero && N3 >= Force.Zero)
            return StringerForceState.PureTension;

        if (N1 <= Force.Zero && N3 <= Force.Zero)
            return StringerForceState.PureCompression;

        return StringerForceState.Combined;
    }
}

/// <summary>
///     Elastic stringer object.
/// </summary>
/// <inheritdoc cref="From" />
protected Stringer(Node grip1, Node grip2, Node grip3, CrossSection crossSection)
    : base(new StringerGeometry(grip1.Position, grip3.Position, crossSection), new[] {
        grip1, grip2, grip3 })
{
}

/// <inheritdoc cref="Stringer(Node, Node, Node, CrossSection)" />
/// <inheritdoc cref="From" />
protected Stringer(Node grip1, Node grip2, Node grip3, CrossSection crossSection,
    IConcreteParameters concreteParameters, ConstitutiveModel model =
    ConstitutiveModel.MCFT, UniaxialReinforcement? reinforcement = null)
    : this(grip1, grip2, grip3, crossSection)
{
    Reinforcement = reinforcement;
    Concrete = new UniaxialConcrete(concreteParameters, GetConcreteArea(this),
        model);

    if (Reinforcement is not null)
        Reinforcement.ConcreteArea = Concrete.Area;

    // Calculate matrices
    TransformationMatrix = CalculateTransformationMatrix(Geometry.Angle);
    LocalStiffness = CalculateStiffness(Concrete.Stiffness, Geometry.Length);
    Stiffness = (StiffnessMatrix)
        LocalStiffness.Transform(TransformationMatrix);
}

/// <summary>
///     Create a <see cref="Stringer" /> based on element model.
/// </summary>
/// <param name="grip1">The initial <see cref="Node" /> of the <see cref="Stringer"
    />.</param>

```

```

/// <param name="grip2">The center <see cref="Node" /> of the <see cref="Stringer"
/>.</param>
/// <param name="grip3">The final <see cref="Node" /> of the <see cref="Stringer"
/>.</param>
/// <param name="crossSection">The stringer cross-section.</param>
/// <param name="concreteParameters">The concrete <see cref="IConcreteParameters"
/>.</param>
/// <param name="model">The concrete <see cref="ConstitutiveModel" />.</param>
/// <param name="reinforcement">The <see cref="UniaxialReinforcement" /> of this
stringer.</param>
/// <inheritdoc cref="As" />
public static Stringer From(Node grip1, Node grip2, Node grip3, CrossSection
    crossSection, IConcreteParameters concreteParameters, ConstitutiveModel model =
    ConstitutiveModel.MCFT, UniaxialReinforcement? reinforcement = null, ElementModel
    elementModel = ElementModel.Elastic) =>
    elementModel switch
    {
        ElementModel.Elastic => new Stringer(grip1, grip2, grip3, crossSection,
            concreteParameters, model, reinforcement),
        - => new NLStringer(grip1, grip2, grip3, crossSection,
            concreteParameters, model, reinforcement)
    };

/// <summary>
///     Create a <see cref="Stringer" /> from a collection of <paramref name="nodes"
/> and known positions of initial and
///     final grips.
/// </summary>
/// <param name="nodes">The collection containing all <see cref="Node" />'s of SPM
model.</param>
/// <param name="grip1Position">The position of initial <see cref="Node" /> of the
<see cref="Stringer" />.</param>
/// <param name="grip3Position">The position of final <see cref="Node" /> of the <see
cref="Stringer" />.</param>
/// <inheritdoc cref="From" />
public static Stringer FromNodes([NotNull] IEnumerable<Node> nodes, Point
    grip1Position, Point grip3Position, CrossSection crossSection,
    IConcreteParameters concreteParameters, ConstitutiveModel model =
    ConstitutiveModel.MCFT, UniaxialReinforcement? reinforcement = null, ElementModel
    elementModel = ElementModel.Elastic) =>
    From(nodes.GetByPosition(grip1Position),
        nodes.GetByPosition(grip1Position.MidPoint(grip3Position)),
        nodes.GetByPosition(grip3Position), crossSection, concreteParameters, model,
        reinforcement, elementModel);

/// <summary>
///     Calculate local stiffness <see cref="Matrix" />.
/// </summary>
/// <param name="concreteStiffness">The stiffness of concrete cross-section.</param>
/// <param name="stringerLength">The length of the stringer.</param>
protected static StiffnessMatrix CalculateStiffness(Force concreteStiffness, Length
    stringerLength)
{
    // Calculate the constant factor of stiffness
    var k = concreteStiffness.Newtons / (3 * stringerLength.Millimeters);

    // Calculate the local stiffness matrix
    var stiffness = k * new double[,]
    {
        { 7, -8, 1 },
    }

```

```

        { -8, 16, -8 },
        { 1, -8, 7 }
    }.ToMatrix();

    return
        new StiffnessMatrix(stiffness);
}

/// <summary>
///     Calculate the transformation matrix based on the <paramref name="angle" /> of
///     a stringer.
/// </summary>
/// <param name="angle">The angle of the stringer, related to horizontal axis.</param>
protected static Matrix<double> CalculateTransformationMatrix(double angle)
{
    // Get the direction cosines
    var (l, m) = angle.DirectionCosines();

    // Obtain the transformation matrix
    return new[,]
    {
        { 1, m, 0, 0, 0, 0 },
        { 0, 0, 1, m, 0, 0 },
        { 0, 0, 0, 0, 1, m }
    }.ToMatrix();
}

/// <summary>
///     Get the proper concrete area for a <paramref name="stringer" />.
/// </summary>
internal static Area GetConcreteArea(Stringer stringer) =>
    stringer switch
    {
        NLStringer => stringer.Geometry.CrossSection.Area -
            (stringer.Reinforcement?.Area ?? Area.Zero),
        -
            => stringer.Geometry.CrossSection.Area
    };

/// <summary>
///     Create a <see cref="Stringer" /> object based in this nonlinear stringer.
/// </summary>
/// <param name="elementModel">The <see cref="ElementModel" />.</param>
/// <returns>
///     <see cref="Stringer" />
/// </returns>
public Stringer As(ElementModel elementModel) =>
    elementModel switch
    {
        ElementModel.Elastic when this is NLStringer => new Stringer(Grip1,
            Grip2, Grip3, Geometry.CrossSection, Concrete.Parameters, Concrete.Model,
            Reinforcement?.Clone()),
        ElementModel.Nonlinear when this is not NLStringer => new NLStringer(Grip1,
            Grip2, Grip3, Geometry.CrossSection, Concrete.Parameters, Concrete.Model,
            Reinforcement?.Clone()),
        -
            => this
    };

/// <inheritdoc />
public override string ToString()
{

```

```

var msgstr =
    $"Stringer_{Number}\n\n" +
    $"Grips:_{GripNumbers[0]}_{GripNumbers[1]}_{GripNumbers[2]}\n" +
    $"DoFIndex:_{DoFIndex.Select(i=>i.ToString()).Aggregate((i,f)=>_{f} -
        {f})}\n" +
    $"{Geometry}";

if (Reinforcement is not null)
    msgstr += $" \n\n{Reinforcement}";

return msgstr;
}

/// <inheritdoc />
public override void UpdateStiffness()
{
    // Not needed in linear element.
}
}

```

C.4.3.3 Não-linear

A classe *NLStringer* é derivada da classe *Stringer* e fornece as propriedades não-lineares do elemento biela, bem como os métodos de cálculo de forças a partir dos deslocamentos.

NLStringer.cs

```

namespace andrefmello91.SPMElements;

/// <summary>
///     Nonlinear stringer class.
/// </summary>
internal class NLStringer : Stringer, INonlinearSPMElement
{
    // Auxiliary fields
    private readonly Matrix<double> _bMatrix;
    private bool _concreteCracked;
    private bool _concreteCrushed;
    private bool _concreteYielded;

    private Force _n1, _n3;
    private bool _steelYielded;

    /// <inheritdoc />
    public override Length[] CrackOpenings => Strains
        .Select(eps => CrackOpening(Reinforcement, eps,
            Concrete.Parameters.CrackingStrain))
        .ToArray();

    /// <summary>
    ///     Get the strain <see cref="Vector" />.
    /// </summary>
    internal Vector<double> Strains => _bMatrix * (Vector<double>)
        (LocalDisplacements.Unit is LengthUnit.Millimeter
        ? LocalDisplacements

```



```
        : LocalDisplacements.Convert(LengthUnit.Millimeter));

    /// <summary>
    ///     The cross section at the end node.
    /// </summary>
    private RCCrossSection EndCrossSection { get; }

    /// <summary>
    ///     The cross section at the initial node.
    /// </summary>
    private RCCrossSection InitialCrossSection { get; }

    /// <summary>
    ///     Check if concrete is cracked in this stringer.
    /// </summary>
    /// <inheritdoc cref="Material.Concrete.Concrete.Cracked" />
    public bool ConcreteCracked
    {
        get => _concreteCracked;
        private set
        {
            if (_concreteCracked)
                return;

            _concreteCracked = value;

            if (value)
                OnStateChanged();
        }
    }

    /// <inheritdoc />
    public bool ConcreteCrushed
    {
        get => _concreteCrushed;
        private set
        {
            if (_concreteCrushed)
                return;

            _concreteCrushed = value;

            if (value)
                OnStateChanged();
        }
    }

    /// <inheritdoc />
    public bool ConcreteYielded
    {
        get => _concreteYielded;
        private set
        {
            if (_concreteYielded)
                return;

            _concreteYielded = value;

            if (value)
                OnStateChanged();
        }
    }
}
```

```

    }
}

/// <inheritdoc />
public bool SteelYielded
{
    get => _steelYielded;
    private set
    {
        if (_steelYielded)
            return;

        _steelYielded = value;

        if (value)
            OnStateChanged();
    }
}

/// <inheritdoc />
public event EventHandler<StateEventArgs>? StateChanged;

/// <summary>
///     Nonlinear stringer object.
/// </summary>
/// <inheritdoc
///     cref="Stringer(Node, Node, Node, CrossSection, IConcreteParameters,
///     ConstitutiveModel, UniaxialReinforcement)" />
internal NLStringer(Node grip1, Node grip2, Node grip3, CrossSection crossSection,
    IConcreteParameters concreteParameters, ConstitutiveModel model =
    ConstitutiveModel.MCFT, UniaxialReinforcement? reinforcement = null)
    : base(grip1, grip2, grip3, crossSection)
{
    Reinforcement = reinforcement;
    Concrete = new UniaxialConcrete(concreteParameters, GetConcreteArea(this),
        model);

    if (Reinforcement is not null)
        Reinforcement.ConcreteArea = Concrete.Area;

    InitialCrossSection = new RCCrossSection(Concrete.Clone(), Reinforcement?.Clone());
    EndCrossSection = InitialCrossSection.Clone();

    _bMatrix = CalculateBMatrix(Geometry.Length);

    // Calculate matrices
    TransformationMatrix = CalculateTransformationMatrix(Geometry.Angle);
    LocalStiffness = CalculateStiffness(Concrete.Stiffness, Geometry.Length);
    Stiffness = (StiffnessMatrix)
        LocalStiffness.Transform(TransformationMatrix);
}

/// <summary>
///     Calculate B Matrix based on stringer length.
/// </summary>
/// <param name="stringerLength">The length of the stringer.</param>
private static Matrix<double> CalculateBMatrix(Length stringerLength) =>
    1D / stringerLength.Millimeters * new double[,]
    {
        { -3, 4, -1 },
    }
}

```

```

        { -1, 0, 1 },
        { 1, -4, 3 }
    }.ToMatrix();

    /// <summary>
    ///     Calculate the average crack opening.
    /// </summary>
    /// <param name="reinforcement">The <see cref="UniaxialReinforcement" />.</param>
    /// <param name="strain">The strain.</param>
    /// <param name="concreteCrackingStrain">
    ///     The cracking strain of concrete.
    ///     <seealso cref="IConcreteParameters.CrackingStrain" />.
    /// </param>
    private static Length CrackOpening(UniaxialReinforcement? reinforcement, double
        strain, double concreteCrackingStrain) =>
        strain <= concreteCrackingStrain
            ? Length.Zero
            : strain * CrackSpacing(reinforcement);

    /// <summary>
    ///     Calculate the crack spacing at <paramref name="reinforcement" />, according
    ///     to Kaklauskas (2019)
    ///     expression.
    ///     <para> $sm = 21 \text{ mm} + 0.155 \phi / \rho$ </para>
    /// </summary>
    /// <param name="reinforcement">The <see cref="UniaxialReinforcement" />.</param>
    private static Length CrackSpacing(UniaxialReinforcement? reinforcement) =>
        reinforcement is null || reinforcement.BarDiameter.ApproxZero(Point.Tolerance) ||
            reinforcement.Ratio.ApproxZero()
            ? Length.FromMillimeters(21)
            : Length.FromMillimeters(21) + 0.155 * reinforcement.BarDiameter /
                reinforcement.Ratio;

    /// <summary>
    ///     Check state changes.
    /// </summary>
    private void CheckStates()
    {
        ConcreteCracked = InitialCrossSection.Concrete.Cracked ||
            EndCrossSection.Concrete.Cracked;
        ConcreteYielded = InitialCrossSection.Concrete.Yielded ||
            EndCrossSection.Concrete.Yielded;
        ConcreteCrushed = InitialCrossSection.Concrete.Crushed ||
            EndCrossSection.Concrete.Crushed;
        Steelyielded = Reinforcement is not null &&
            (InitialCrossSection.Reinforcement!.Yielded ||
                EndCrossSection.Reinforcement!.Yielded);
    }

    private void OnStateChanged([CallerMemberName] string? stateName = null) =>
        StateChanged?.Invoke(this, new StateEventArgs(stateName!));

    /// <inheritdoc />
    public void AddValue(double loadFactor) => Monitor?.AddMonitoredValue(loadFactor,
        this);

    /// <inheritdoc />
    public override void CalculateForces()
    {
        // Get strains

```

```

    var eps = Strains;

    // Calculate normal forces
    InitialCrossSection.Calculate(eps[0]);
    EndCrossSection.Calculate(eps[2]);

    Force
        n1 = InitialCrossSection.Force,
        n3 = EndCrossSection.Force;

    // Update forces
    LocalForces = new ForceVector(new[] { -n1, n1 - n3, n3 });

    Forces = (ForceVector) (TransformationMatrix.Transpose() * LocalForces);

    CheckStates();
}
}

```

C.4.4 Elemento painel

C.4.4.1 Geometria

A geometria do painel é representada pela estrutura *PanelGeometry*, que tem como propriedades: vértices (pela estrutura *Vertices*), bordas laterais (pela estrutura *Edge*), posições dos graus de liberdade e largura.

PanelGeometry.cs

```

namespace andrefmello91.SPMElements.PanelProperties;

/// <summary>
///     Panel geometry struct.
/// </summary>
public struct PanelGeometry : IUnitConvertible<LengthUnit>, IApproachable<PanelGeometry,
    Length>, IEquatable<PanelGeometry>, IComparable<PanelGeometry>,
    ICloneable<PanelGeometry>
{
    private Length _width;

    /// <summary>
    ///     Get panel dimensions (a, b, c, d).
    /// </summary>
    public (Length a, Length b, Length c, Length d) Dimensions { get; private set; }

    /// <summary>
    ///     Get <see cref="Edge" /> 1 (base edge).
    /// </summary>
    public Edge Edge1 { get; }

    /// <summary>
    ///     Get <see cref="Edge" /> 2 (right edge).
    /// </summary>
    public Edge Edge2 { get; }
}

```

```

/// <summary>
///     Get <see cref="Edge" /> 3 (top edge).
/// </summary>
public Edge Edge3 { get; }

/// <summary>
///     Get <see cref="Edge" /> 4 (left edge).
/// </summary>
public Edge Edge4 { get; }

/// <summary>
///     Get edges' lengths as an array.
/// </summary>
public Length[] EdgeLengths => Edges.Select(e => e.Length).ToArray();

/// <summary>
///     Get edges as an array.
/// </summary>
public Edge[] Edges => new[] { Edge1, Edge2, Edge3, Edge4 };

/// <summary>
///     Get grip positions as an array.
/// </summary>
public Point[] GripPositions => Edges.Select(e => e.CenterPoint).ToArray();

/// <summary>
///     Returns true if this geometry is rectangular.
/// </summary>
public bool IsRectangular => Vertices.IsRectangular;

/// <summary>
///     Get edges' stringer dimensions as an array.
/// </summary>
public Length[] StringerDimensions => Edges.Select(e =>
    e.StringerDimension).ToArray();

/// <summary>
///     Get vertices of panel.
/// </summary>
public Vertices Vertices { get; }

/// <summary>
///     Get panel width.
/// </summary>
public Length Width
{
    get => _width;
    set => _width = value.ToUnit(Unit);
}

/// <summary>
///     Get the <see cref="LengthUnit" /> that this was constructed with.
/// </summary>
public LengthUnit Unit
{
    get => Vertices.Unit;
    set => ChangeUnit(value);
}

```

```

/// <summary>
///     Create a panel geometry object.
/// </summary>
/// <param name="vertices">Panel <see cref="PanelProperties.Vertices" />
///     object.</param>
/// <param name="width">Panel width.</param>
public PanelGeometry(Vertices vertices, Length width)
{
    Vertices = vertices;
    _width = width.ToUnit(vertices.Unit);

    // Get edges
    Edge1 = new Edge(vertices.Vertex1, vertices.Vertex2);
    Edge2 = new Edge(vertices.Vertex2, vertices.Vertex3);
    Edge3 = new Edge(vertices.Vertex3, vertices.Vertex4);
    Edge4 = new Edge(vertices.Vertex4, vertices.Vertex1);

    Dimensions = CalculateDimensions(Vertices);
}

/// <inheritdoc cref="PanelGeometry(Vertices,Length)" />
/// <param name="unit">
///     The <see cref="LengthUnit" /> of <paramref name="width" /> and <paramref
///     name="vertices" />' coordinates.
/// </param>
public PanelGeometry(Vertices vertices, double width, LengthUnit unit =
    LengthUnit.Millimeter)
    : this(vertices, (Length) width.As(unit))
{
}

/// <summary>
///     Calculate panel dimensions (a, b, c, d).
/// </summary>
public static (Length a, Length b, Length c, Length d) CalculateDimensions(Vertices
    vertices)
{
    var x = vertices.XCoordinates;
    var y = vertices.YCoordinates;

    // Calculate the necessary dimensions of the panel
    Length
        a = 0.5 * (x[1] + x[2] - x[0] - x[3]),
        b = 0.5 * (y[2] + y[3] - y[0] - y[1]),
        c = 0.5 * (x[2] + x[3] - x[0] - x[1]),
        d = 0.5 * (y[1] + y[2] - y[0] - y[3]);

    return (a, b, c, d);
}

/// <summary>
///     Create a panel geometry object from a collection of points.
/// </summary>
/// <param name="vertices">The collection of the four <see cref="Point" /> vertices,
///     in any order.</param>
/// <param name="width">Panel width.</param>
/// <exception cref="ArgumentException">If <paramref name="vertices" /> doesn't
///     contain 4 points.</exception>
public static PanelGeometry From(IEnumerable<Point> vertices, Length width) =>
    new(Vertices.From(vertices), width);

```

```

/// <inheritdoc cref="From(IEnumerable{Point}, Length)" />
/// <param name="unit">
///     The <see cref="LengthUnit" /> of <paramref name="width" /> and <paramref
///     name="vertices" />' coordinates.
/// </param>
public static PanelGeometry From(IEnumerable<Point> vertices, double width,
    LengthUnit unit = LengthUnit.Millimeter) => new(Vertices.From(vertices), width,
    unit);

/// <inheritdoc cref="IUnitConvertible{TUnit}.Convert" />
public PanelGeometry Convert(LengthUnit unit) => new(Vertices.Convert(unit),
    Width.ToUnit(unit));

/// <summary>
///     Return <see cref="Dimensions" /> in <see cref="LengthUnit.Millimeter" />.
/// </summary>
public (double a, double b, double c, double d) DimensionsInMillimeters() =>
    (Dimensions.a.Millimeters, Dimensions.b.Millimeters, Dimensions.c.Millimeters,
    Dimensions.d.Millimeters);

/// <summary>
///     Divide a <see cref="PanelGeometry" /> object into new ones.
/// </summary>
/// <remarks>
///     The object must be rectangular, otherwise a collection containing only it is
///     returned.
/// </remarks>
/// <param name="rows">The required number of rows.</param>
/// <param name="columns">The required number of columns.</param>
public IEnumerable<PanelGeometry> Divide(int rows, int columns)
{
    var width = Width;

    return Vertices.Divide(rows, columns).Select(v => new PanelGeometry(v, width));
}

/// <inheritdoc />
public override bool Equals(object obj) => obj is PanelGeometry other &&
    Equals(other);

/// <inheritdoc />
public override int GetHashCode() => Vertices.GetHashCode();

/// <inheritdoc />
public override string ToString() =>
    $"{Vertices}\n" +

    // Edges.Select(e => $"{nameof(e)}\n{e}\n").Aggregate((i, f) => i + f) +
    $"Width={Width}";

/// <inheritdoc />
public bool Approaches(PanelGeometry other, Length tolerance) =>
    Vertices.Approaches(other.Vertices, tolerance);

/// <inheritdoc />
public PanelGeometry Clone() => new(Vertices.Clone(), Width);

/// <inheritdoc />
public int CompareTo(PanelGeometry other) => Vertices.CompareTo(other.Vertices);

```

```

/// <summary>
///     Returns true if all <see cref="Vertices" /> are equal.
/// </summary>
/// <param name="other">The other <see cref="PanelGeometry" /> to compare.</param>
public bool Equals(PanelGeometry other) => Approaches(other, Point.Tolerance);

/// <summary>
///     Change the <see cref="LengthUnit" /> of this.
/// </summary>
/// <param name="unit">The <see cref="LengthUnit" /> to convert.</param>
public void ChangeUnit(LengthUnit unit)
{
    if (Unit == unit)
        return;

    _width = _width.ToUnit(unit);

    Vertices.ChangeUnit(unit);

    Edge1.ChangeUnit(unit);
    Edge2.ChangeUnit(unit);
    Edge3.ChangeUnit(unit);
    Edge4.ChangeUnit(unit);

    Dimensions = CalculateDimensions(Vertices);
}

IUnitConvertible<LengthUnit> IUnitConvertible<LengthUnit>.Convert(LengthUnit unit) =>
    Convert(unit);

/// <summary>
///     Returns true if arguments are equal.
/// </summary>
public static bool operator ==(PanelGeometry left, PanelGeometry right) =>
    left.Equals(right);

/// <summary>
///     Returns true if arguments are different.
/// </summary>
public static bool operator !=(PanelGeometry left, PanelGeometry right) =>
    !left.Equals(right);
}

```

Vertices.cs

```

namespace andrefmello91.SPMElements.PanelProperties;

/// <summary>
///     Panel vertices struct.
/// </summary>
public struct Vertices : IUnitConvertible<LengthUnit>, IApproachable<Vertices, Length>,
    IEquatable<Vertices>, IComparable<Vertices>, ICloneable<Vertices>
{
    /// <summary>
    ///     Get <see cref="Vertices" /> approximated center point.
    /// </summary>
    public Point CenterPoint { get; }
}

```



```

/// <summary>
///     Returns true if this forms a rectangular geometry.
/// </summary>
public bool IsRectangular => Rectangular(this);

/// <summary>
///     Get vertex 1 (base left vertex).
/// </summary>
public Point Vertex1 { get; }

/// <summary>
///     Get vertex 2 (base right vertex).
/// </summary>
public Point Vertex2 { get; }

/// <summary>
///     Get vertex 3 (top right vertex).
/// </summary>
public Point Vertex3 { get; }

/// <summary>
///     Get vertex 4 (top left vertex).
/// </summary>
public Point Vertex4 { get; }

/// <summary>
///     Get X coordinates of vertices as an array.
/// </summary>
public Length[] XCoordinates => AsArray().Select(v => v.X).ToArray();

/// <summary>
///     Get Y coordinates of vertices as an array.
/// </summary>
public Length[] YCoordinates => AsArray().Select(v => v.Y).ToArray();

/// <summary>
///     Get the <see cref="LengthUnit" /> of vertices' coordinates.
/// </summary>
public LengthUnit Unit
{
    get => Vertex1.Unit;
    set => ChangeUnit(value);
}

/// <summary>
///     Create a panel vertices object.
/// </summary>
/// <param name="vertex1">The base left vertex.</param>
/// <param name="vertex2">The base right vertex.</param>
/// <param name="vertex3">The upper right vertex.</param>
/// <param name="vertex4">The upper left vertex.</param>
public Vertices(Point vertex1, Point vertex2, Point vertex3, Point vertex4)
{
    Vertex1 = vertex1;
    Vertex2 = vertex2.Convert(vertex1.Unit);
    Vertex3 = vertex3.Convert(vertex1.Unit);
    Vertex4 = vertex4.Convert(vertex1.Unit);

    CenterPoint = CalculateCenterPoint(Vertex1, Vertex2, Vertex3, Vertex4);
}

```

```

/// <summary>
///     Calculate <see cref="Vertices" /> approximated center point.
/// </summary>
/// <inheritdoc cref="Vertices(Point, Point, Point, Point)" />
public static Point CalculateCenterPoint(Point vertex1, Point vertex2, Point vertex3,
    Point vertex4)
{
    // Calculate the approximated center point
    var pt1 = vertex1.MidPoint(vertex3);
    var pt2 = vertex2.MidPoint(vertex4);

    return pt1.MidPoint(pt2);
}

/// <summary>
///     Create a panel vertices object from a collection of points.
/// </summary>
/// <param name="vertices">The collection of the four <see cref="Point" /> vertices,
///     in any order.</param>
/// <exception cref="ArgumentException">If <paramref name="vertices" /> doesn't
///     contain 4 points.</exception>
public static Vertices From(IEnumerable<Point> vertices)
{
    if (vertices.Count() != 4)
        throw new ArgumentException("Vertices must contain 4 points.",
            nameof(vertices));

    // Order points
    var verts = vertices.OrderBy(p => p.Y).ThenBy(p => p.X).ToList();

    // Get correct order
    var (v1, v2) = verts[1].X > verts[0].X
        ? (verts[0], verts[1])
        : (verts[1], verts[0]);

    var (v4, v3) = verts[3].X > verts[2].X
        ? (verts[2], verts[3])
        : (verts[3], verts[2]);

    return
        new Vertices(v1, v2, v3, v4);
}

/// <summary>
///     Returns true if <paramref name="vertices" /> form a rectangular geometry.
/// </summary>
/// <param name="vertices">The <see cref="Vertices" /> object to check.</param>
public static bool Rectangular(Vertices vertices) =>
    vertices.Vertex1.Y.Approx(vertices.Vertex2.Y, Point.Tolerance) &&
    vertices.Vertex4.Y.Approx(vertices.Vertex3.Y, Point.Tolerance) &&
    vertices.Vertex1.X.Approx(vertices.Vertex4.X, Point.Tolerance) &&
    vertices.Vertex2.X.Approx(vertices.Vertex3.X, Point.Tolerance);

/// <summary>
///     Get vertices as an array.
/// </summary>
public Point[] AsArray() => new[] { Vertex1, Vertex2, Vertex3, Vertex4 };

/// <summary>

```

```

    /// Convert this <see cref="Vertices" /> object to another <see cref="LengthUnit"
    /// />.
    /// </summary>
    /// <param name="unit">The desired <see cref="LengthUnit" />.</param>
    public Vertices Convert(LengthUnit unit) =>
        unit == Unit
            ? Clone()
            : new Vertices(Vertex1.Convert(unit), Vertex2.Convert(unit),
                Vertex3.Convert(unit), Vertex4.Convert(unit));

    /// <summary>
    /// Divide a <see cref="Vertices" /> object into new ones.
    /// </summary>
    /// <remarks>
    /// This must be rectangular, otherwise an empty collection is returned.
    /// </remarks>
    /// <param name="rows">The required number of rows.</param>
    /// <param name="columns">The required number of columns.</param>
    public IEnumerable<Vertices> Divide(int rows, int columns)
    {
        if (!IsRectangular)
            yield break;

        // Get distances
        var dx = (Vertex2.X - Vertex1.X) / columns;
        var dy = (Vertex4.Y - Vertex1.Y) / rows;

        for (var r = 0; r < rows; r++)
        {
            // Get initial vertex for this row
            var v1 = new Point(Vertex1.X, Vertex1.Y + r * dy);

            for (var c = 0; c < columns; c++)
            {
                // Get other vertices
                var v2 = new Point(v1.X + dx, v1.Y);
                var v3 = new Point(v1.X + dx, v1.Y + dy);
                var v4 = new Point(v1.X, v1.Y + dy);

                // Return
                yield return new Vertices(v1, v2, v3, v4);

                // Set initial vertex for next column
                v1 = v2;
            }
        }
    }

    /// <inheritdoc />
    public override bool Equals(object obj) => obj is Vertices other && Equals(other);

    /// <inheritdoc />
    public override int GetHashCode() => AsArray().Sum(point => point.GetHashCode());

    /// <inheritdoc />
    public override string ToString() =>
        $"Vertex_1:\t({Vertex1.X.Value:0.00},\t{Vertex1.Y.Value:0.00})\n" +
        $"Vertex_2:\t({Vertex2.X.Value:0.00},\t{Vertex2.Y.Value:0.00})\n" +
        $"Vertex_3:\t({Vertex3.X.Value:0.00},\t{Vertex3.Y.Value:0.00})\n" +
        $"Vertex_4:\t({Vertex4.X.Value:0.00},\t{Vertex4.Y.Value:0.00})\n" +

```

```

    $"Coordinates_{in}_{Unit}";

    /// <inheritdoc />
    public bool Approaches(Vertexes other, Length tolerance) =>
        Vertex1.Approaches(other.Vertex1, tolerance) && Vertex2.Approaches(other.Vertex2,
            tolerance) &&
        Vertex3.Approaches(other.Vertex3, tolerance) && Vertex4.Approaches(other.Vertex4,
            tolerance);

    /// <inheritdoc />
    public Vertexes Clone() => From(AsArray());

    /// <inheritdoc />
    public int CompareTo(Vertexes other) => CenterPoint.CompareTo(other.CenterPoint);

    /// <summary>
    ///     Returns true if all vertices are equal.
    /// </summary>
    /// <param name="other">The other <see cref="Vertexes" /> to compare.</param>
    public bool Equals(Vertexes other) => Approaches(other, Point.Tolerance);

    /// <summary>
    ///     Change the <see cref="LengthUnit" /> of vertices' coordinates.
    /// </summary>
    /// <param name="unit">The <see cref="LengthUnit" /> to convert.</param>
    public void ChangeUnit(LengthUnit unit)
    {
        if (Unit == unit)
            return;

        Vertex1.ChangeUnit(unit);
        Vertex2.ChangeUnit(unit);
        Vertex3.ChangeUnit(unit);
        Vertex4.ChangeUnit(unit);

        Unit = unit;
    }

    IUnitConvertible<LengthUnit> IUnitConvertible<LengthUnit>.Convert(LengthUnit unit) =>
        Convert(unit);

    /// <summary>
    ///     Returns true if arguments are equal.
    /// </summary>
    public static bool operator ==(Vertexes left, Vertexes right) => left.Equals(right);

    /// <summary>
    ///     Returns true if arguments are different.
    /// </summary>
    public static bool operator !=(Vertexes left, Vertexes right) => !left.Equals(right);
}

```

Edge.cs

```

namespace andrefmello91.SPMElements.PanelProperties;

/// <summary>
///     Panel edge struct.
/// </summary>

```

```

public struct Edge : IUnitConvertible<LengthUnit>, IApproachable<Edge, Length>,
    IEquatable<Edge>, IComparable<Edge>
{
    /// <summary>
    ///     Get angle related to horizontal axis, in radians.
    /// </summary>
    public double Angle { get; }

    /// <summary>
    ///     Get center point of this <see cref="Edge" />.
    /// </summary>
    public Point CenterPoint { get; }

    /// <summary>
    ///     Get the final vertex of this <see cref="Edge" />.
    /// </summary>
    public Point FinalVertex { get; }

    /// <summary>
    ///     Get the initial vertex of this <see cref="Edge" />.
    /// </summary>
    public Point InitialVertex { get; }

    /// <summary>
    ///     Get length of this edge.
    /// </summary>
    public Length Length { get; private set; }

    /// <summary>
    ///     Get the stringer dimension of this edge, in mm.
    /// </summary>
    public Length StringerDimension { get; private set; }

    /// <summary>
    ///     Get the <see cref="LengthUnit" /> that this was constructed with.
    /// </summary>
    public LengthUnit Unit
    {
        get => InitialVertex.Unit;
        set => ChangeUnit(value);
    }

    /// <summary>
    ///     Panel edge constructor.
    /// </summary>
    /// <param name="initialVertex">The initial vertex.</param>
    /// <param name="finalVertex">The final vertex.</param>
    public Edge(Point initialVertex, Point finalVertex)
    {
        InitialVertex      = initialVertex;
        FinalVertex         = finalVertex.Convert(initialVertex.Unit);
        CenterPoint         = initialVertex.MidPoint(finalVertex);
        Length               = initialVertex.GetDistance(finalVertex);
        Angle                = initialVertex.GetAngle(finalVertex);
        StringerDimension   = Length.Zero;
    }

    /// <inheritdoc cref="IUnitConvertible{TUnit}.Convert" />

```

```

public Edge Convert(LengthUnit unit) => new(InitialVertex.Convert(unit),
    FinalVertex.Convert(unit));

/// <inheritdoc />
public override bool Equals(object obj) => obj is Edge other && Equals(other);

/// <inheritdoc />
public override int GetHashCode() => InitialVertex.GetHashCode() *
    FinalVertex.GetHashCode();

/// <summary>
///     Set stringer dimension in this edge.
/// </summary>
/// <param name="height">The height of the <seealso cref="Stringer" />, in <paramref
    name="unit" /> considered.</param>
/// <param name="unit">The <see cref="LengthUnit" /> of <paramref name="height"
    />.</param>
public void SetStringerDimension(double height, LengthUnit unit =
    LengthUnit.Millimeter) =>
    SetStringerDimension((Length) height.As(unit));

/// <param name="height">The height of the <seealso cref="Stringer" />.</param>
/// <inheritdoc cref="SetStringerDimension(double, LengthUnit)" />
public void SetStringerDimension(Length height) => StringerDimension =
    height.ToUnit(Unit);

/// <inheritdoc />
public override string ToString() =>
    $"InitialVertex: {InitialVertex.X:0.00}, {InitialVertex.Y:0.00}\n" +
    $"FinalVertex: {FinalVertex.X:0.00}, {FinalVertex.Y:0.00}\n" +
    $"Length: {Length}\n" +
    $"Angle: {Angle:0.00} rad";

/// <inheritdoc />
public bool Approaches(Edge other, Length tolerance) =>
    InitialVertex.Approaches(other.InitialVertex, tolerance) &&
    FinalVertex.Approaches(other.FinalVertex, tolerance) ||
    InitialVertex.Approaches(other.FinalVertex, tolerance) &&
    FinalVertex.Approaches(other.InitialVertex, tolerance);

/// <inheritdoc />
public int CompareTo(Edge other) => CenterPoint.CompareTo(other.CenterPoint);

/// <summary>
///     Returns true if edges are equal.
/// </summary>
/// <param name="other">The other <see cref="Edge" /> object.</param>
public bool Equals(Edge other) => InitialVertex == other.InitialVertex && FinalVertex
    == other.FinalVertex;

/// <summary>
///     Change the <see cref="LengthUnit" /> of this.
/// </summary>
/// <param name="unit">The <see cref="LengthUnit" /> to convert.</param>
public void ChangeUnit(LengthUnit unit)
{
    if (Unit == unit)
        return;

    InitialVertex.ChangeUnit(unit);

```

```

        CenterPoint.ChangeUnit(unit);
        FinalVertex.ChangeUnit(unit);

        Length          = Length.ToUnit(unit);
        StringerDimension = StringerDimension.ToUnit(unit);
    }

    IUnitConvertible<LengthUnit> IUnitConvertible<LengthUnit>.Convert(LengthUnit unit) =>
        Convert(unit);

    /// <summary>
    ///     Returns true if arguments are equal.
    /// </summary>
    public static bool operator ==(Edge left, Edge right) => left.Equals(right);

    /// <summary>
    ///     Returns true if arguments are different.
    /// </summary>
    public static bool operator !=(Edge left, Edge right) => !left.Equals(right);
}

```

C.4.4.2 Linear

A classe *Panel* tem como propriedade principal o objeto de geometria *PanelGeometry* e fornece as propriedades básicas do elemento painel, como a matriz rigidez e os vetores de força e deslocamento (grau 8). Conta também com as propriedades de materiais biaxiais (*BiaxialConcrete* e *WebReinforcement*).

Panel.cs

```

namespace andrefmello91.SPMElements;

/// <summary>
///     Base panel class with linear properties.
/// </summary>
public class Panel : SPMElement<PanelGeometry>
{

    /// <summary>
    ///     Get average <see cref="PrincipalStressState" />.
    /// </summary>
    public PrincipalStressState AveragePrincipalStresses =>
        PrincipalStressState.FromStress(AverageStresses);

    /// <summary>
    ///     Get average <see cref="StressState" />.
    /// </summary>
    public virtual StressState AverageStresses
    {
        get
        {
            // Get the dimensions as a vector
            var lsV = Geometry.EdgeLengths.Select(l => l.Millimeters).ToVector();

            // Calculate the shear stresses
            var tau = LocalForces / (lsV * Geometry.Width.Millimeters);

```

```

        // Calculate the average stress
        var tauAvg = (-tau[0] + tau[1] - tau[2] + tau[3]) / 4;

        return new StressState(0, 0, tauAvg);
    }
}

/// <summary>
///     Get <see cref="BiaxialConcrete" /> of this.
/// </summary>
public BiaxialConcrete Concrete { get; protected set; }

/// <summary>
///     Get average concrete <see cref="PrincipalStrainState" />.
/// </summary>
public virtual PrincipalStrainState ConcretePrincipalStrains { get; }

/// <summary>
///     Get average concrete <see cref="PrincipalStressState" />.
/// </summary>
public virtual PrincipalStressState ConcretePrincipalStresses
{
    get
    {
        Pressure sig2;
        double theta1;

        // Get shear stress
        var tau = AverageStresses.TauXY;

        // Get steel strengths
        Pressure
            fyx = Reinforcement?.DirectionX?.Steel.Parameters.YieldStress ??
                Pressure.Zero,
            fyy = Reinforcement?.DirectionY?.Steel.Parameters.YieldStress ??
                Pressure.Zero;

        // Get ratios
        double
            psx = Reinforcement?.DirectionX?.Ratio ?? 0,
            psy = Reinforcement?.DirectionY?.Ratio ?? 0;

        if (psx.ApproxZero(1E-6) || psy.ApproxZero(1E-6) || psx.Approx(psy, 1E-4) &&
            fyx.Approx(fyy, StressState.Tolerance))
        {
            sig2 = -2 * tau.Abs();

            theta1 = tau >= Pressure.Zero
                ? Constants.PiOver4
                : -Constants.PiOver4;
        }
        else
        {
            // Calculate theta
            theta1 = Math.Atan(Math.Sqrt(psx * fyx / (psy * fyy)));
            theta1 = tau >= Pressure.Zero
                ? theta1
                : theta1 - Constants.PiOver2;
        }
    }
}

```



```

        // Calculate stress
        var tanTheta1 = theta1.Tan();
        sig2 = -tau * (tanTheta1 + 1D / tanTheta1);
    }

    return new PrincipalStressState(Pressure.Zero, sig2, theta1);
}
}

/// <summary>
///     Get the average crack opening in concrete.
/// </summary>
public virtual Length CrackOpening { get; }

/// <summary>
///     Get the center <see cref="SPMElements.Node" /> of bottom edge.
/// </summary>
public Node Grip1 => Grips[0];

/// <summary>
///     Get the center <see cref="SPMElements.Node" /> of right edge.
/// </summary>
public Node Grip2 => Grips[1];

/// <summary>
///     Get the center <see cref="SPMElements.Node" /> of top edge.
/// </summary>
public Node Grip3 => Grips[2];

/// <summary>
///     Get the center <see cref="SPMElements.Node" /> of left edge.
/// </summary>
public Node Grip4 => Grips[3];

/// <inheritdoc />
public override Force MaxForce => Forces.AbsoluteMaximum();

/// <inheritdoc />
public override bool Monitored
{
    get => Monitor is not null;
    set
    {
        if (value)
            Monitor ??= new PanelMonitor(Name);
        else
            Monitor = null;
    }
}

/// <inheritdoc />
public override string Name => $"Panel_{Number}";

/// <summary>
///     Get <see cref="WebReinforcement" /> of this.
/// </summary>
public WebReinforcement? Reinforcement { get; protected set; }

/// <summary>
///     Elastic panel object.

```

```

/// </summary>
/// <inheritdoc cref="From" />
protected Panel(Node grip1, Node grip2, Node grip3, Node grip4, PanelGeometry
    geometry)
    : base(geometry, new[] { grip1, grip2, grip3, grip4 })
{
}

/// <summary>
///     Elastic panel object.
/// </summary>
/// <inheritdoc cref="From" />
protected Panel(Node grip1, Node grip2, Node grip3, Node grip4, PanelGeometry
    geometry, IConcreteParameters concreteParameters, ConstitutiveModel model =
    ConstitutiveModel.MCFT, WebReinforcement? reinforcement = null)
    : this(grip1, grip2, grip3, grip4, geometry)
{
    Concrete = BiaxialConcrete.From(concreteParameters, model);

    Reinforcement = reinforcement;

    if (Reinforcement is not null)
        Reinforcement.Width = Geometry.Width;

    // Initiate stiffness
    TransformationMatrix = CalculateTransformationMatrix(Geometry);
    LocalStiffness = CalculateStiffness(Geometry,
        Concrete.Parameters.TransverseModule);
    Stiffness = (StiffnessMatrix)
        LocalStiffness.Transform(TransformationMatrix);
}

/// <summary>
///     Create a <see cref="Panel" /> from an element model.
/// </summary>
/// <param name="grip1">The center <see cref="Node" /> of bottom edge</param>
/// <param name="grip2">The center <see cref="Node" /> of right edge</param>
/// <param name="grip3">The center <see cref="Node" /> of top edge</param>
/// <param name="grip4">The center <see cref="Node" /> of left edge</param>
/// <param name="geometry">The <seealso cref="PanelGeometry" />.</param>
/// <param name="concreteParameters">The concrete parameters <see cref="Parameters"
/// >.</param>
/// <param name="model">The concrete <see cref="ConstitutiveModel" />.</param>
/// <param name="reinforcement">The <see cref="WebReinforcement" />.</param>
/// <inheritdoc cref="As" />
public static Panel From(Node grip1, Node grip2, Node grip3, Node grip4,
    PanelGeometry geometry, IConcreteParameters concreteParameters, ConstitutiveModel
    model = ConstitutiveModel.MCFT, WebReinforcement? reinforcement = null,
    ElementModel elementModel = ElementModel.Elastic) =>
    elementModel switch
    {
        ElementModel.Elastic => new Panel(grip1, grip2, grip3, grip4, geometry,
            concreteParameters, model, reinforcement),
        - => new NLPanel(grip1, grip2, grip3, grip4, geometry,
            concreteParameters, model, reinforcement)
    };

/// <summary>
///     Create a <see cref="Panel" /> from a collection of <paramref name="nodes" />
    and the panel geometry.

```

```

/// </summary>
/// <param name="nodes">The collection containing all <see cref="Node" />'s in SPM
    model.</param>
/// <inheritdoc cref="From" />
public static Panel FromNodes(IEnumerable<Node> nodes, PanelGeometry geometry,
    IConcreteParameters concreteParameters, ConstitutiveModel model =
    ConstitutiveModel.MCFT, WebReinforcement? reinforcement = null, ElementModel
    elementModel = ElementModel.Elastic)
{
    var nds = geometry.Edges.Select(e => nodes.GetByPosition(e.CenterPoint)).ToArray();

    return From(nds[0], nds[1], nds[2], nds[3], geometry, concreteParameters, model,
        reinforcement, elementModel);
}

/// <summary>
///     Calculate panel stiffness
/// </summary>
/// <param name="geometry">The <see cref="PanelGeometry" />.</param>
/// <param name="concreteTransverseModule">The transverse elastic module of
    concrete.</param>
private static StiffnessMatrix CalculateStiffness(PanelGeometry geometry, Pressure
    concreteTransverseModule) => geometry.IsRectangular
    ? RectangularPanelStiffness(geometry, concreteTransverseModule)
    : NonRectangularPanelStiffness(geometry, concreteTransverseModule);

/// <summary>
///     Calculate transformation matrix
/// </summary>
private static Matrix<double> CalculateTransformationMatrix(PanelGeometry geometry)
{
    // Get the transformation matrix
    // Direction cosines
    var (m1, n1) = geometry.Edge1.Angle.DirectionCosines();
    var (m2, n2) = geometry.Edge2.Angle.DirectionCosines();
    var (m3, n3) = geometry.Edge3.Angle.DirectionCosines();
    var (m4, n4) = geometry.Edge4.Angle.DirectionCosines();

    // T matrix
    return new[, ]
    {
        { m1, n1, 0, 0, 0, 0, 0, 0 },
        { 0, 0, m2, n2, 0, 0, 0, 0 },
        { 0, 0, 0, 0, m3, n3, 0, 0 },
        { 0, 0, 0, 0, 0, 0, m4, n4 }
    }.ToMatrix();
}

/// <summary>
///     Calculate panel stiffness if panel is not rectangular.
///     <para>See: <seealso cref="PanelGeometry.IsRectangular" /></para>
/// </summary>
/// <inheritdoc cref="CalculateStiffness" />
private static StiffnessMatrix NonRectangularPanelStiffness(PanelGeometry geometry,
    Pressure concreteTransverseModule)
{
    // Get the dimensions
    double[]
        x = geometry.Vertices.XCoordinates.Select(cx => cx.Millimeters).ToArray(),
        y = geometry.Vertices.YCoordinates.Select(cy => cy.Millimeters).ToArray();

```

```

var (a, b, c, d) = geometry.DimensionsInMillimeters();

double
    w = geometry.Width.Millimeters,
    l1 = geometry.Edge1.Length.Millimeters,
    l2 = geometry.Edge2.Length.Millimeters,
    l3 = geometry.Edge3.Length.Millimeters,
    l4 = geometry.Edge4.Length.Millimeters;

// Calculate Gc
var Gc = concreteTransverseModule.Megapascals;

// Equilibrium parameters
double
    c1 = x[1] - x[0],
    c2 = x[2] - x[1],
    c3 = x[3] - x[2],
    c4 = x[0] - x[3],
    s1 = y[1] - y[0],
    s2 = y[2] - y[1],
    s3 = y[3] - y[2],
    s4 = y[0] - y[3],
    r1 = x[0] * y[1] - x[1] * y[0],
    r2 = x[1] * y[2] - x[2] * y[1],
    r3 = x[2] * y[3] - x[3] * y[2],
    r4 = x[3] * y[0] - x[0] * y[3];

// Kinematic parameters
double
    t1 = -b * c1 - c * s1,
    t2 = a * s2 + d * c2,
    t3 = b * c3 + c * s3,
    t4 = -a * s4 - d * c4;

// Matrices to calculate the determinants
var km1 = new[,]
{
    { c2, c3, c4 },
    { s2, s3, s4 },
    { r2, r3, r4 }
}.ToMatrix();

var km2 = new[,]
{
    { c1, c3, c4 },
    { s1, s3, s4 },
    { r1, r3, r4 }
}.ToMatrix();

var km3 = new[,]
{
    { c1, c2, c4 },
    { s1, s2, s4 },
    { r1, r2, r4 }
}.ToMatrix();

var km4 = new[,]
{
    { c1, c2, c3 },

```

```

        { s1, s2, s3 },
        { r1, r2, r3 }
    }.ToMatrix();

    // Calculate the determinants
    double
        k1 = km1.Determinant(),
        k2 = km2.Determinant(),
        k3 = km3.Determinant(),
        k4 = km4.Determinant();

    // Calculate kf and ku
    double
        kf = k1 + k2 + k3 + k4,
        ku = -t1 * k1 + t2 * k2 - t3 * k3 + t4 * k4;

    // Calculate D
    var D = 16 * Gc * w / (kf * ku);

    // Get the vector B
    var B = new[]
    {
        -k1 * l1, k2 * l2, -k3 * l3, k4 * l4
    }.ToVector();

    // Get the stiffness matrix
    var stiffness = B.ToColumnMatrix() * D * B.ToRowMatrix();

    return
        new StiffnessMatrix(stiffness);
}

/// <summary>
///     Calculate panel stiffness if panel is rectangular.
///     <para>See: <seealso cref="PanelGeometry.IsRectangular" /></para>
/// </summary>
/// <inheritdoc cref="CalculateStiffness" />
private static StiffnessMatrix RectangularPanelStiffness(PanelGeometry geometry,
    Pressure concreteTransverseModule)
{
    // Get the dimensions
    double
        w = geometry.Width.Millimeters,
        a = geometry.Dimensions.a.Millimeters,
        b = geometry.Dimensions.b.Millimeters;

    // Calculate the parameters of the stiffness matrix
    double
        a_b = a / b,
        b_a = b / a;

    // Calculate Gc
    var Gc = concreteTransverseModule.Megapascals;

    // Calculate the stiffness matrix
    var stiffness = Gc * w * new[,]
    {
        { a_b, -1, a_b, -1 },
        { -1, b_a, -1, b_a },
        { a_b, -1, a_b, -1 },
    }

```

```

        { -1, b_a, -1, b_a }
    }.ToMatrix();

    return
        new StiffnessMatrix(stiffness);
}

/// <summary>
///     Create a <see cref="Stringer" /> object based in this nonlinear stringer.
/// </summary>
/// <returns>
///     <see cref="Panel" />
/// </returns>
public Panel As(ElementModel model) =>
    model switch
    {
        ElementModel.Elastic when this is NLPanel => new Panel(Grip1, Grip2,
            Grip3, Grip4, Geometry, Concrete.Parameters, Concrete.Model,
            Reinforcement?.Clone()),
        ElementModel.Nonlinear when this is not NLPanel => new NLPanel(Grip1, Grip2,
            Grip3, Grip4, Geometry, Concrete.Parameters, Concrete.Model,
            Reinforcement?.Clone()),
        - => this
    };

/// <inheritdoc />
public override int GetHashCode() => Geometry.GetHashCode();

/// <summary>
///     Set stringer dimensions on edges.
///     <para>See: <see cref="Edge.SetStringerDimension" /></para>
/// </summary>
/// <param name="stringers">The collection containing all of the stringers.</param>
public void SetStringersDimensions([NotNull] IEnumerable<Stringer> stringers)
{
    // Get dimensions
    var hs = Grips
        .Select(g => 0.5 * stringers.First(str =>
            g.Equals(str.Grip2)).Geometry.CrossSection.Height)
        .ToArray();

    // Set on edges
    Geometry.Edge1.SetStringerDimension(hs[0]);
    Geometry.Edge2.SetStringerDimension(hs[1]);
    Geometry.Edge3.SetStringerDimension(hs[2]);
    Geometry.Edge4.SetStringerDimension(hs[3]);
}

/// <inheritdoc />
public override string ToString()
{
    var msgstr =
        $"Panel_{Number}\n\n" +
        $"Grips:_{GripNumbers[0]}_{GripNumbers[1]}_{GripNumbers[2]}_{GripNumbers[3]}\n" +
        $"DoFIndex:_{DoFIndex.Select(i=>i.ToString()).Aggregate((i,f)=>f+"{i} - {f}")}\n" +
        $"_{Geometry}";

    if (Reinforcement is not null)

```

```

        msgstr += $"\\n\\n{Reinforcement}";

        return msgstr;
    }

    /// <inheritdoc />
    public override void UpdateStiffness()
    {
        // Not needed in linear element.
    }
}

```

C.4.4.3 Não-linear

A classe *NLPanel* é derivada da classe *Panel* e fornece as propriedades não-lineares do elemento painel, bem como os métodos de cálculo de forças a partir dos deslocamentos.

NLPanel.cs

```

namespace andrefmello91.SPMElements;

/// <summary>
///     Nonlinear panel class.
/// </summary>
internal class NLPanel : Panel, INonlinearSPMElement
{
    // Auxiliary fields
    private readonly Matrix<double> _baMatrix;
    private bool _concreteCracked;
    private bool _concreteCrushed;
    private bool _concreteYielded;
    private bool _steelYielded;

    /// <summary>
    ///     The average strains in this panel.
    /// </summary>
    public StrainState AverageStrains => 0.25 * IntegrationPoints
        .Aggregate(StrainState.Zero, (state, membrane) => state + membrane.AverageStrains);

    /// <inheritdoc />
    public override StressState AverageStresses => 0.25 * IntegrationPoints
        .Aggregate(StressState.Zero, (state, membrane) => state +
            membrane.AverageStresses);

    /// <inheritdoc />
    public override PrincipalStrainState ConcretePrincipalStrains =>
        (PrincipalStrainState) (0.25 * IntegrationPoints
            .Aggregate(StrainState.Zero, (state, membrane) => state +
                membrane.Concrete.Strains));

    /// <inheritdoc />
    public override PrincipalStressState ConcretePrincipalStresses =>
        (PrincipalStressState) (0.25 * IntegrationPoints
            .Aggregate(StressState.Zero, (state, membrane) => state +
                membrane.Concrete.Stresses));

    /// <inheritdoc />

```

```

public override Length CrackOpening
{
    get
    {
        var cStrains = ConcretePrincipalStrains;

        return cStrains.Epsilon1 > Concrete.Parameters.CrackingStrain
            ? Membrane.CrackOpening(Reinforcement, cStrains)
            : Length.Zero;
    }
}

/// <summary>
///     The concrete stress <see cref="Vector" />.
/// </summary>
/// <inheritdoc cref="Stresses" />
private Vector<double> ConcreteStresses => IntegrationPoints
    .SelectMany(m => m.Concrete.Stresses.ToHorizontal().AsVector())
    .ToVector();

/// <summary>
///     Get <see cref="Membrane" /> integration points.
/// </summary>
private Membrane[] IntegrationPoints { get; }

/// <summary>
///     Reinforcement stress vector.
/// </summary>
/// <inheritdoc cref="Stresses" />
private Vector<double> ReinforcementStresses => IntegrationPoints
    .SelectMany(m => (m.Reinforcement?.Stresses.ToHorizontal() ??
        StressState.Zero).AsVector())
    .ToVector();

/// <summary>
///     Get panel strain <see cref="Vector" />.
/// </summary>
private Vector<double> StrainVector =>
    _baMatrix * (Vector<double>) (Displacements.Unit == LengthUnit.Millimeter
        ? Displacements
        : Displacements.Convert(LengthUnit.Millimeter));

/// <summary>
///     Get panel stress <see cref="Vector" />.
/// </summary>
/// <remarks>
///     Components in <see cref="PressureUnit.Megapascal" />.
/// </remarks>
private Vector<double> Stresses => ConcreteStresses + ReinforcementStresses;

/// <summary>
///     Check if concrete is cracked in this panel.
/// </summary>
/// <inheritdoc cref="Material.Concrete.Concrete.Cracked" />
public bool ConcreteCracked
{
    get => _concreteCracked;
    private set
    {
        if (_concreteCracked)
    }
}

```



```
        return;

        _concreteCracked = value;

        if (value)
            OnStateChanged();
    }
}

///  

public bool ConcreteCrushed
{
    get => _concreteCrushed;
    private set
    {
        if (_concreteCrushed)
            return;

        _concreteCrushed = value;

        if (value)
            OnStateChanged();
    }
}

///  

public bool ConcreteYielded
{
    get => _concreteYielded;
    private set
    {
        if (_concreteYielded)
            return;

        _concreteYielded = value;

        if (value)
            OnStateChanged();
    }
}

///  

public bool SteelYielded
{
    get => _steelYielded;
    private set
    {
        if (_steelYielded)
            return;

        _steelYielded = value;

        if (value)
            OnStateChanged();
    }
}

///  

public event EventHandler<StateEventArgs>? StateChanged;
```

```

/// <summary>
///     Nonlinear panel object.
/// </summary>
/// <inheritdoc />
internal NLPanel(Node grip1, Node grip2, Node grip3, Node grip4, PanelGeometry
    geometry, IConcreteParameters concreteParameters, ConstitutiveModel model =
    ConstitutiveModel.MCFT, WebReinforcement? reinforcement = null)
    : base(grip1, grip2, grip3, grip4, geometry)
{
    Concrete = BiaxialConcrete.From(concreteParameters, model);

    Reinforcement = reinforcement;

    if (Reinforcement is not null)
        Reinforcement.Width = Geometry.Width;

    IntegrationPoints = IntPoints(concreteParameters, reinforcement, geometry.Width,
        model).ToArray();
    _baMatrix = CalculateBa(Geometry);

    InitiateStiffness();
}

/// <summary>
///     Calculate BA matrix.
/// </summary>
/// <param name="geometry">The <see cref="PanelGeometry" />.</param>
private static Matrix<double> CalculateBa(PanelGeometry geometry)
{
    var (a, b, c, d) = geometry.DimensionsInMillimeters();

    // Calculate t1, t2 and t3
    double
        t1 = a * b - c * d,
        t2 = 0.5 * (a * a - c * c) + b * b - d * d,
        t3 = 0.5 * (b * b - d * d) + a * a - c * c;

    // Calculate the components of A matrix
    double
        a_t1 = a / t1,
        b_t1 = b / t1,
        c_t1 = c / t1,
        d_t1 = d / t1,
        a_t2 = a / t2,
        b_t2 = b / t2,
        a_2t1 = a_t1 / 2,
        b_2t1 = b_t1 / 2,
        c_2t1 = c_t1 / 2,
        d_2t1 = d_t1 / 2;

    // Create A matrix
    var A = new[,]
    {
        { d_t1, 0, b_t1, 0, -d_t1, 0, -b_t1, 0 },
        { 0, -a_t1, 0, -c_t1, 0, a_t1, 0, c_t1 },
        { -a_2t1, d_2t1, -c_2t1, b_2t1, a_2t1, -d_2t1, c_2t1, -b_2t1 },
        { -a_t2, 0, a_t2, 0, -a_t2, 0, a_t2, 0 },
        { 0, b_t3, 0, -b_t3, 0, b_t3, 0, -b_t3 }
    }.ToMatrix();
}

```

```

// Calculate the components of B matrix
double
    c_a = c / a,
    d_b = d / b;

// Create B matrix
var B = new[,]
{
    { 1, 0, 0, -c_a, 0 },
    { 0, 1, 0, 0, -1 },
    { 0, 0, 2, 0, 0 },
    { 1, 0, 0, 1, 0 },
    { 0, 1, 0, 0, d_b },
    { 0, 0, 2, 0, 0 },
    { 1, 0, 0, c_a, 0 },
    { 0, 1, 0, 0, 1 },
    { 0, 0, 2, 0, 0 },
    { 1, 0, 0, -1, 0 },
    { 0, 1, 0, 0, -d_b },
    { 0, 0, 2, 0, 0 }
}.ToMatrix();

return
    B * A;
}

/// <summary>
///     Calculate P matrices for concrete and steel
/// </summary>
/// <inheritdoc cref="CalculateBa" />
private static (Matrix<double> Pc, Matrix<double> Ps) CalculateP(PanelGeometry
    geometry)
{
    // Get dimensions
    double[]
        x = geometry.Vertices.XCoordinates.Select(cx => cx.Millimeters).ToArray(),
        y = geometry.Vertices.YCoordinates.Select(cy => cy.Millimeters).ToArray(),
        c = geometry.StringerDimensions.Select(s => s.Millimeters).ToArray();

    var t = geometry.Width.Millimeters;

    // Create P matrices
    var Pc = Matrix<double>.Build.Dense(8, 12);
    var Ps = Matrix<double>.Build.Dense(8, 12);

    // Calculate the components of Pc
    Pc[0, 0] = Pc[1, 2] = t * (y[1] - y[0]);
    Pc[0, 2] = t * (x[0] - x[1]);
    Pc[1, 1] = t * (x[0] - x[1] + c[1] + c[3]);

    Pc[2, 3] = t * (y[2] - y[1] - c[2] - c[0]);
    Pc[2, 5] = Pc[3, 4] = t * (x[1] - x[2]);
    Pc[3, 5] = t * (y[2] - y[1]);

    Pc[4, 6] = Pc[5, 8] = t * (y[3] - y[2]);
    Pc[4, 8] = t * (x[2] - x[3]);
    Pc[5, 7] = t * (x[2] - x[3] - c[1] - c[3]);

    Pc[6, 9] = t * (y[0] - y[3] + c[0] + c[2]);
    Pc[6, 11] = Pc[7, 10] = t * (x[3] - x[0]);

```

```

Pc[7, 11] = t * (y[0] - y[3]);

// Calculate the components of Ps
Ps[0, 0] = Pc[0, 0];
Ps[1, 1] = t * (x[0] - x[1]);

Ps[2, 3] = t * (y[2] - y[1]);
Ps[3, 4] = Pc[3, 4];

Ps[4, 6] = Pc[4, 6];
Ps[5, 7] = t * (x[2] - x[3]);

Ps[6, 9] = t * (y[0] - y[3]);
Ps[7, 10] = Pc[7, 10];

return
    (Pc, Ps);
}

/// <summary>
///     Calculate Q matrix.
/// </summary>
/// <inheritdoc cref="CalculateBa" />
private static Matrix<double> CalculateQ(PanelGeometry geometry)
{
    // Get dimensions
    var (a, b, c, d) = geometry.DimensionsInMillimeters();

    // Calculate t4
    var t4 = a * a + b * b;

    // Calculate the components of Q matrix
    double
        a2      = a * a,
        bc      = b * c,
        bdMt4   = b * d - t4,
        ab      = a * b,
        MbdMt4  = -b * d - t4,
        Tt4     = 2 * t4,
        acMt4   = a * c - t4,
        ad      = a * d,
        b2      = b * b,
        MacMt4  = -a * c - t4;

    // Create Q matrix
    return
        1.0 / Tt4 * new[,]
        {
            { a2, bc, bdMt4, -ab, -a2, -bc, MbdMt4, ab },
            { 0, Tt4, 0, 0, 0, 0, 0, 0 },
            { 0, 0, Tt4, 0, 0, 0, 0, 0 },
            { -ab, acMt4, ad, b2, ab, MacMt4, -ad, -b2 },
            { -a2, -bc, MbdMt4, ab, a2, bc, bdMt4, -ab },
            { 0, 0, 0, 0, 0, Tt4, 0, 0 },
            { 0, 0, 0, 0, 0, 0, Tt4, 0 },
            { ab, MacMt4, -ad, -b2, -ab, acMt4, ad, b2 }
        }.ToMatrix();
}

/// <summary>

```

```

///      Calculate initial material stiffness matrices.
/// </summary>
/// <param name="integrationPoints">The <see cref="Membrane" /> integration points of
    the panel.</param>
private static (Matrix<double> Dc, Matrix<double> Ds)
    InitialMaterialStiffness(IReadOnlyList<Membrane> integrationPoints)
{
    var Dc = Matrix<double>.Build.Dense(12, 12);
    var Ds = Matrix<double>.Build.Dense(12, 12);

    for (var i = 0; i < 4; i++)
    {
        // Get the stiffness
        var Dci = integrationPoints[i].Concrete.InitialStiffness;
        var Dsi = integrationPoints[i].Reinforcement?.InitialStiffness;

        // Set to stiffness
        Dc.SetSubMatrix(3 * i, 3 * i, Dci);

        if (Dsi is null)
            continue;

        Ds.SetSubMatrix(3 * i, 3 * i, Dsi);
    }

    return
        (Dc, Ds);
}

/// <summary>
///      Initiate <see cref="Membrane" /> integration points.
/// </summary>
private static IEnumerable<Membrane> IntPoints(IConcreteParameters
    concreteParameters, WebReinforcement? reinforcement, Length width,
    ConstitutiveModel model)
{
    for (var i = 0; i < 4; i++)
        yield return Membrane.From(concreteParameters, reinforcement?.Clone(), width,
            model);
}

/// <summary>
///      Calculate and set global force vector.
///      <para>See: <see cref="Panel.Forces" />.</para>
/// </summary>
private void CalculateGripForces()
{
    double t0, t1, t2, t3, t4;

    // Get dimensions
    double[]
        x = Geometry.Vertices.XCoordinates.Select(cx => cx.Millimeters).ToArray(),
        y = Geometry.Vertices.YCoordinates.Select(cy => cy.Millimeters).ToArray();

    var (a, b, c, d) = Geometry.DimensionsInMillimeters();
    var s = Geometry.StringerDimensions.Select(st => st.Millimeters).ToArray();
    ;
    var t = Geometry.Width.Millimeters;

    // Get stresses

```

```

Vector<double>
    sigC = ConcreteStresses,
    sigS = ReinforcementStresses,
    sig  = sigC + sigS;

var (sig1, sigC1, sigS1) = (sig.SubVector(0, 3), sigC.SubVector(0, 3),
    sigS.SubVector(0, 3));
var (sig2, sigC2, sigS2) = (sig.SubVector(3, 3), sigC.SubVector(3, 3),
    sigS.SubVector(3, 3));
var (sig3, sigC3, sigS3) = (sig.SubVector(6, 3), sigC.SubVector(6, 3),
    sigS.SubVector(6, 3));
var (sig4, sigC4, sigS4) = (sig.SubVector(9, 3), sigC.SubVector(9, 3),
    sigS.SubVector(9, 3));

// Calculate forces
t1 = y[1] - y[0];
t2 = x[1] - x[0];
t3 = CheckT3(t2 - s[1] - s[3]);

double
    f1 = (sig1[0] * t1 - sig1[2] * t2) * t,
    f2 = (-sigC1[1] * t3 - sigS1[1] * t2 + sig1[2] * t1) * t;

t1 = y[2] - y[1];
t2 = x[2] - x[1];
t3 = CheckT3(t1 - s[2] - s[0]);

double
    f3 = (sigC2[0] * t3 + sigS2[0] * t1 - sig2[2] * t2) * t,
    f4 = (-sig2[1] * t2 + sig2[2] * t1) * t;

t1 = y[2] - y[3];
t2 = x[2] - x[3];
t3 = CheckT3(t2 - s[1] - s[3]);

double
    f5 = (-sig3[0] * t1 + sig3[2] * t2) * t,
    f6 = (sigC3[1] * t3 + sigS3[1] * t2 - sig3[2] * t1) * t;

t1 = y[3] - y[0];
t2 = x[3] - x[0];
t3 = CheckT3(t1 - s[0] - s[2]);

double
    f7 = (-sigC4[0] * t3 - sigS4[0] * t1 - sig4[2] * t2) * t,
    f8 = (sig4[1] * t2 - sig4[2] * t1) * t;

// Correct forces
t0 = 2 * (a * a + b * b);
t1 = (a * (f1 - f5) - b * (f4 - f8)) / t0;
t2 = (c * (f2 - f6) + d * (f3 - f7)) / t0;
t3 = (f3 + f7) * 0.5;
t4 = (f2 + f6) * 0.5;

f1 = a * t1 + b * t2 - t3;
f4 = -b * t1 + a * t2 - t4;
f5 = -a * t1 - b * t2 - t3;
f8 = b * t1 - a * t2 - t4;

Forces = new ForceVector(new[]

```

```

    {
        f1, f2, f3, f4, f5, f6, f7, f8
    });

    // Check value of t3
    static double CheckT3(double value) => value < 0 ? 0 : value;
}

/// <summary>
///     Calculate and set stress vectors.
///     <para>See: <see cref="ConcreteStresses" />, <see cref="ReinforcementStresses"
///     />.</para>
/// </summary>
private void CalculateStresses()
{
    // Get the vector strains and stresses
    var ev = StrainVector;

    // Calculate stresses based on strains
    for (var i = 0; i < 4; i++)
    {
        // Get the strains and stresses
        var e = StrainState.FromVector(ev.SubVector(3 * i, 3));

        // Calculate stresses
        IntegrationPoints[i].Calculate(e);
    }
}

/// <summary>
///     Check state changes.
/// </summary>
private void CheckStates()
{
    ConcreteCracked = IntegrationPoints.Any(m => m.Concrete.Cracked);
    ConcreteYielded = IntegrationPoints.Any(m => m.Concrete.Yielded);
    ConcreteCrushed = IntegrationPoints.Any(m => m.Concrete.Crushed);
    SteelYielded = Reinforcement is not null && IntegrationPoints.Any(m =>
        (m.Reinforcement!.DirectionX?.Yielded ?? false) ||
        (m.Reinforcement!.DirectionY?.Yielded ?? false));
}

/// <summary>
///     Calculate initial stiffness matrix.
/// </summary>
private void InitiateStiffness()
{
    var (Dc, Ds) = InitialMaterialStiffness(IntegrationPoints);
    var (Pc, Ps) = CalculateP(Geometry);

    var Q = CalculateQ(Geometry);

    var QPs = Q * Ps;
    var QPc = Q * Pc;

    var kc = QPc * Dc * _baMatrix;
    var ks = QPs * Ds * _baMatrix;

    Stiffness = new StiffnessMatrix(kc + ks);
}

```

```
private void OnStateChanged([CallerMemberName] string? stateName = null) =>
    StateChanged?.Invoke(this, new StateEventArgs(stateName!));

/// <inheritdoc />
public void AddValue(double loadFactor) => Monitor?.AddMonitoredValue(loadFactor,
    this);

/// <inheritdoc />
public override void CalculateForces()
{
    // Calculate stresses, forces and update stiffness
    CalculateStresses();
    CalculateGripForces();
    CheckStates();
}

/// <inheritdoc />
public override void UpdateDisplacements() =>
    Displacements = this.GetDisplacementsFromGrips();
}
```
