UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE ENGENHARIA MECÂNICA
CURSO DE GRADUAÇÃO EM ENGENHARIA MECÂNICA

Glenda de Melo Luz

Desenvolvimento de uma ferramenta modular para modelagem e simulação de sistemas térmicos

Florianópolis
2025

Glenda de Melo Luz

Desenvolvimento de uma ferramenta modular para modelagem e simulação de sistemas térmicos

Trabalho de Conclusão de Curso submetido ao curso de Graduação em Engenharia Mecânica da Universidade Federal de Santa Catarina como requisito parcial para a obtenção do título de Bacharel em Engenharia Mecânica

Orientador: Jaime Andrés Lozano Cadena
Coorientadores: Jader Riso Barbosa Jr.
Alan Tihiro Dias Nakashima

Florianópolis
2025

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Luz, Glenda de Melo
Desenvolvimento de uma ferramenta modular para modelagem e simulação de sistemas térmicos / Glenda de Melo Luz ; orientador, Jaime Andrés Lozano Cadena, coorientador, Jader Riso Barbosa Júnior, coorientador, Alan Tihiro Dias Nakashima, 2025.
131 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Engenharia Mecânica, Florianópolis, 2025.

Inclui referências.

1. Engenharia Mecânica. 2. Modelagem de sistemas térmicos. 3. Modelagem modular. 4. Simulação numérica. I. Cadena, Jaime Andrés Lozano. II. Barbosa Júnior, Jader Riso. III. Nakashima, Alan Tihiro Dias IV. Universidade Federal de Santa Catarina. Graduação em Engenharia Mecânica. V. Título.

Glenda de Melo Luz

Desenvolvimento de uma ferramenta modular para modelagem e simulação de sistemas térmicos

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do título de Bacharel em Engenharia Mecânica e aprovado em sua forma final pelo Curso de Graduação em Engenharia Mecânica.

Florianópolis, 14 de julho de 2025

Documento assinado digitalmente
Antonio Carlos Valdiero
Data: 05/01/2026 15:32:01-0300
CPF: ***.227.377-**
Verifique as assinaturas em https://v.ufsc.br

Coordenação do curso

Banca examinadora

Documento assinado digitalmente
Jaime Andres Lozano Cadena
Data: 05/01/2026 15:13:29-0300
CPF: ***.171.599-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Jaime Andrés Lozano Cadena, Dr.
Orientador

Eng. Guilherme Peixer
Universidade Federal de Santa Catarina

Prof. Saulo Güths, Dr.
Universidade Federal de Santa Catarina

Florianópolis, 2025

Dedico este trabalho à minha mãe, ao meu irmão e
à dona Honey, pelo amor e apoio incondicional.
Sem vocês, nada disso teria sido possível.

Agradecimentos

Gostaria de agradecer, em primeiro lugar, à minha melhor amiga, minha mãe, que sempre esteve ao meu lado me apoiando e me guiando durante toda essa trajetória. Sou profundamente grata pelos seus conselhos, pelo seu exemplo de força e dedicação, pelas melhores risadas e pela paciência infinita, principalmente nos momentos em que não podia sair junto com você. Obrigada por acreditar em mim sempre, mesmo quando eu mesma duvidava, por ficar feliz por cada conquista minha, mesmo que pequena, e por sempre me aconselhar em cada passo do caminho. Sem o seu amor, carinho e incentivo, eu não teria chegado até aqui.

Sou extremamente grata também ao meu irmão, Gabriel, e à sua esposa, Giovanna, que considero uma irmã. Sempre cuidaram de mim e me proporcionaram momentos e oportunidades incríveis, especialmente ao tornar possível o intercâmbio. Obrigada por todo apoio, carinho e companhia ao longo dessa jornada. Agradeço também à minha irmã de 4 patas, a dona Honey, que, com seus dentinhos sempre para fora, nunca deixou de me alegrar e me fazer companhia. Sou grata também ao meu namorado, Hígor, que me acompanhou durante esses últimos anos, sempre ao meu lado, me ajudando, me tranquilizando e acreditando em mim.

Gostaria também de agradecer aos professores Jaime e Jader pela oportunidade de trabalhar no Polo. A aula de Fundamentos da Termodinâmica com o professor Jader foi decisiva para despertar meu interesse pela área térmica. Já a disciplina de pós-graduação de Sistemas Térmicos, ministrada pelo professor Jaime, foi essencial para a construção deste projeto. Sou especialmente grata ao Tihiro, cuja contribuição foi fundamental para o desenvolvimento deste trabalho, por meio de suas ideias sempre geniais e sugestões. Agradeço pelo modo didático com que explicou conceitos de simulação, engenharia térmica e física, pelo apoio constante e pelas conversas sobre a vida, o universo e tudo mais, que tornaram esses anos tão especiais, sendo uma amizade que levarei para a vida.

Sou grata também à Natália, que me orientou com dedicação na área de refrigeração magnética, compartilhando seu conhecimento tanto na parte experimental quanto na teórica. Agradeço ao Anderson, pela paciência em responder tantas perguntas e ajudar a resolver meus inúmeros problemas. Ao Rogério, minha gratidão pela companhia durante o tempo que dividimos a sala e as diversas disciplinas que cursamos juntos. Agradeço ao Paulo, sempre disposto a esclarecer minhas dúvidas e pelas conversas sempre divertidas e aleatórias. Também sou grata ao Bernardo pelas trocas de ideias e risadas, e ao Eduardo pelo material de programação que tanto me ajudou.

Agradeço ainda a todos os demais membros do Polo, especialmente ao Giga, Jhonatta, Mariah, Mariana, Rafael, Iran, Luis, Militão e Marcus Vinicius, por tornarem o Polo um ambiente acolhedor, inspirador e alegre, tornando o trabalho muito mais agradável.

Meu muito obrigada a todos que ajudaram de alguma forma na realização deste trabalho.

"Sometimes I'll start a sentence, and I don't even know where it's going. I just hope I find it along the way." — Michael Scott

Resumo

O crescimento da demanda global por energia, aliado às crescentes preocupações com as mudanças climáticas, tem estimulado o desenvolvimento de tecnologias energéticas mais eficientes e sustentáveis. Nesse cenário, os sistemas térmicos assumem um papel central, com aplicações que vão desde o aquecimento e a refrigeração residenciais até processos de geração e conversão de energia em grande escala. O desenvolvimento de tais aplicações de forma competitiva requer a utilização de ferramentas de simulação e otimização que permitam a avaliação de diferentes condições de operação e configurações do sistema. Nesse sentido, este trabalho apresenta o desenvolvimento de uma ferramenta computacional modular e extensível para modelagem e simulação de sistemas térmicos, implementada em Python com programação orientada a objetos. A motivação surgiu da necessidade de unificar os modelos numéricos fragmentados desenvolvidos pelo grupo de pesquisa Polo/UFSC ao longo dos anos, criando uma plataforma comum que facilite a integração entre diferentes linhas de pesquisa e a investigação de novas tecnologias e configurações mais eficientes, permitindo a realização de estudos comparativos de desempenho em diversos contextos. O modelo foi estruturado em três classes principais: *Component*, responsável por representar equipamentos individuais; *Connection*, que calcula as propriedades termodinâmicas entre componentes; e *System*, que coordena o processo de solução através do método de Newton-Raphson. Foi implementada uma biblioteca de componentes básicos, incluindo trocadores de calor, turbomáquinas e elementos auxiliares, cada um com suas equações constitutivas específicas. A validação foi realizada através da simulação de dois casos de estudo: uma bomba de calor operando com R134a e um ciclo Rankine com água, ambos baseados nos tutoriais do *software* TESPy. Os resultados obtidos mostraram correspondência com os valores de referência disponíveis, evidenciando que o modelo simplificado implementado é capaz de reproduzir de forma adequada o comportamento termodinâmico dos sistemas analisados. Foram realizadas análises paramétricas para ambos os casos de estudo, que confirmaram o comportamento físico esperado e mostraram o impacto das variáveis operacionais no desempenho dos ciclos. A ferramenta desenvolvida oferece uma solução simplificada e acessível que mantém a precisão necessária para reproduzir o comportamento termodinâmico dos sistemas, ao mesmo tempo em que facilita a compreensão, a manutenção e a extensão futura dos modelos. Dessa forma, constitui uma base inicial e flexível para estudos na área de engenharia térmica, permitindo a incorporação e análise de tecnologias energéticas emergentes, a comparação de diferentes configurações e a exploração de sistemas híbridos, apoiando o desenvolvimento de sistemas térmicos mais eficientes.

Palavras-chave: Modelagem de sistemas térmicos, Simulação numérica, Modelagem modular.

Abstract

The growing global demand for energy, combined with the increasing concerns about climate change, has driven the development of more efficient and sustainable energy technologies. In this scenario, thermal systems play a central role, with applications ranging from residential heating and cooling to large-scale energy generation and conversion processes. The competitive development of such applications requires the use of simulation and optimization tools that enable the evaluation of different operating conditions and system configurations. In this sense, this work presents the development of a modular and extensible computational tool for modeling and simulation of thermal systems, implemented in Python using object-oriented programming. The motivation arose from the need to unify the fragmented numerical models developed by the Polo/UFSC research group over the years, creating a common platform that facilitates integration among different research lines and the investigation of new and more efficient technologies and configurations, enabling comparative performance studies in multiple contexts. The model was structured into three main classes: Component, responsible for representing individual equipment; Connection, which computes thermodynamic properties between components; and System, which coordinates the solution process through the Newton-Raphson method. A library of basic components was implemented, including heat exchangers, turbomachinery, and auxiliary elements, each with their specific constitutive equations. Validation was performed through the simulation of two case studies: a heat pump operating with R134a and a Rankine cycle with water, both based on TESPy software tutorials. The results obtained showed agreement with the available reference values, demonstrating that the simplified model is capable of adequately reproducing the thermodynamic behavior of the analyzed systems. Parametric analyses were performed for both case studies, confirming the expected physical behavior and illustrating the impact of operating variables on cycle performance. The developed tool offers a simplified and accessible solution that maintains the accuracy required to reproduce the thermodynamic behavior of systems while facilitating understanding, maintenance, and future extension of the models. Thus, it constitutes an initial and flexible foundation for studies in the field of thermal engineering, allowing the incorporation and analysis of emerging energy technologies, the comparison of different configurations, and the exploration of hybrid systems, supporting the development of more efficient thermal systems.

Keywords: Thermal systems modeling, Numerical simulation, Modular modeling.

Lista de figuras

Lista de tabelas

Lista de trechos de código

Lista de símbolos

Símbolos

α	Fator de sub-relaxação [-]
Δ	Variação de um parâmetro [-]
δ	Incremento pequeno [-]
ΔT_{log}	Diferença de temperatura média logarítmica [K]
$\dot{E}$	Taxa de energia [W]
E	Energia [J]
ϵ	Tolerância [-]
ε	Efetividade [-]
η	Eficiência [-]
f	Função residual [-]
$f_{energy\_balance}$	Função residual do balanço de energia [-]
f_{eta_s}	Função residual da eficiência isentrópica [-]
f_{pr}	Função residual da razão de pressão [-]
g	Aceleração da gravidade [m s^{-2}]
h	Entalpia específica [J kg^{-1}]
J	Matriz Jacobiana [-]
$\dot{m}$	Vazão mássica [kg s^{-1}]
Ω	Quantidade física generalizada [-]
p	Pressão [Pa]
pr	Razão de pressão [-]
$\dot{Q}$	Taxa de transferência de calor [W]
$\dot{Q}_{net}$	Taxa de transferência de calor líquida [W]
$\dot{S}$	Taxa de entropia [W K^{-1}]
S	Entropia [J $(K)^{-1}$]

s	Entropia específica [J (kg K)$^{-1}$]
T	Temperatura [K]
t	Tempo [s]
T_b	Temperatura absoluta na fronteira [K]
x_{tent}	Solução tentativa [-]
U	Coeficiente global de transferência de calor [W K^{-1} m^{-2}]
U	Função objetivo
UA	Condutância térmica global [W K^{-1}]
V	Velocidade [m s^{-1}]
$\dot{W}$	Potência [W]
$\dot{W}_{net}$	Potência líquida [W]
x	Título [-]
z	Elevação [m]

Subscritos

cold	Lado frio
crit	Crítico
gen	Geração
hot	Lado quente
in	Entrada (*inlet*)
k	Índice da iteração
líq	Líquido saturado
l	Terminal inferior
log	Logarítmico
max	Máximo
min	Mínimo
out	Saída (*outlet*)
sat	Saturação
s	Isentrópico

sys	Sistema
u	Terminal superior
vap	Vapor saturado

Abreviações

COP	Coeficiente de Performance [-]
EES	*Engineering Equation Solver*
HVAC	*Heating, Ventilation, and Air Conditioning*
LMTD	Diferença de Temperatura Média Logarítmica
pr	Razão de pressão [-]
SI	Sistema Internacional de Unidades
TESPy	*Thermal Engineering Systems in Python*
TRNSYS	*Transient System Simulation Tool*
TTD	Diferença de Temperatura Terminal
VCLib	*Vapor Compression Library*
ORC	Ciclo Rankine Orgânico

Sobrescritos

k	Índice da iteração

Sumário

1 Introdução

O aumento da demanda global por energia, aliado às crescentes preocupações com as mudanças climáticas, tem impulsionado o desenvolvimento de tecnologias energéticas mais eficientes e sustentáveis. Segundo o relatório mais recente da Agência Internacional de Energia (*Global Energy Review* (IEA, 2025)), observa-se uma aceleração no crescimento do consumo energético mundial, impulsionada principalmente por fatores como o aumento da demanda por refrigeração devido a eventos climáticos extremos e a maior necessidade de eletricidade em setores como *data centers* e veículos elétricos.

Nesse cenário, os sistemas térmicos desempenham um papel fundamental, com aplicações que abrangem desde aquecimento e refrigeração residencial até geração e conversão de energia em larga escala. Para enfrentar este cenário desafiador, a transição energética existente requer não apenas novas soluções técnicas, mas também ferramentas de modelagem capazes de avaliar diferentes configurações, realizar otimizações e permitir a comparação de tecnologias emergentes. A complexidade crescente dos sistemas energéticos modernos, que frequentemente integram múltiplas tecnologias e fontes de energia, torna essencial o desenvolvimento de ferramentas de análise que possam simular as interações entre diferentes componentes e processos.

O projeto, análise e modelagem de sistemas térmicos constitui uma atividade central na engenharia para o desenvolvimento de novas tecnologias. A simulação de sistemas baseada em componentes é amplamente utilizada tanto na indústria quanto na pesquisa acadêmica para realizar avaliações de desempenho termodinâmico, análises econômicas e otimização de sistemas. Diversos *softwares* comerciais e de código aberto (*open source*) estão disponíveis, incluindo o EES (*Engineering Equation Solver* (F-CHART, 2025)), TRNSYS (*Transient System Simulation Tool* (TESS, 2025)) e VCLib (*Vapor Compression Library* (VERING et al., 2022)). Entre as ferramentas mais recentes disponíveis, o TESPy (*Thermal Engineering Systems in Python* (WITTE, 2025b)) fornece um conjunto de ferramentas de simulação baseado em componentes para engenharia de processos térmicos, permitindo que pesquisadores e engenheiros projetem plantas e simulem operações estacionárias com uma estrutura modular e *open source*.

Apesar das capacidades destes *softwares* existentes, adaptá-los para necessidades específicas de pesquisa exige uma maior flexibilidade na formulação de equações, integração com códigos numéricos já existentes e implementação de abordagens analíticas mais detalhadas ou correlações empíricas derivadas de estudos experimentais. Embora ferramentas como o TESPy ofereçam uma base sólida, sua robustez e complexidade arquitetural podem tornar desafiadora a implementação de modificações significativas ou a integração de outros modelos. A estrutura consolidada dessas ferramentas, embora vantajosa para aplicações convencionais, pode envolver múltiplas dependências na arquitetura que dificultam adaptações específicas. A necessidade de ferramentas especializadas torna-se ainda mais evidente quando se consideram aplicações emergentes ou não convencionais, onde os *softwares* co-

merciais podem não possuir os componentes ou modelos apropriados. Nessas situações, o desenvolvimento de ferramentas próprias, customizadas de acordo com as necessidades do projeto, torna-se uma alternativa viável e frequentemente adotada na literatura.

1.1 Motivação e objetivos

Nesse contexto, o grupo de pesquisa Polo/UFSC tem desenvolvido diversas ferramentas de simulação para investigar tecnologias convencionais e alternativas ao longo dos anos, tais como refrigeradores e condicionadores de ar magnetocalóricos (NAKASHIMA et al., 2022; PEIXER et al., 2023), sistemas de gerenciamento térmico submarino (MILITAO et al., 2024), dispositivos de resfriamento por *spray* (FERREIRA; CARNEIRO; BARBOSA, 2023), processos de liquefação de hidrogênio (SGANZERLA et al., 2024) e células a combustível (DIAS et al., 2024). Embora individualmente eficazes, essas ferramentas especializadas não podem ser facilmente integradas ou comparadas entre si. Esta fragmentação não apenas limita as possibilidades de colaboração entre diferentes linhas de pesquisa, mas também impede a exploração de sistemas híbridos que combinam múltiplas tecnologias. A partir dessa experiência acumulada, surge a necessidade de desenvolver uma plataforma unificada e modular, capaz de representar diferentes sistemas térmicos de forma generalizada. A ideia central é permitir que os modelos previamente desenvolvidos possam ser integrados em um ambiente comum, com estrutura flexível, orientada a objetos, facilitando a exploração de novas tecnologias e permitindo a realização de estudos comparativos entre diferentes arranjos de componentes.

Considerando esta motivação, o objetivo geral deste trabalho é **desenvolver e documentar de forma abrangente um modelo computacional modular e extensível para simulação de sistemas térmicos**, implementado com programação orientada a objetos, de modo que possa ser facilmente adaptado e estendido para novas aplicações, atendendo às necessidades de pesquisa do grupo Polo/UFSC. O modelo é inspirado no TESPy, devido a sua estrutura modular bem definida, abordagem orientada a componentes e flexibilidade inerente de linguagens de código aberto, mas com as simplificações e adaptações necessárias para facilitar a incorporação de modelos desenvolvidos internamente pelo grupo, bem como futuras extensões.

Os objetivos específicos deste trabalho incluem:

- Implementação de uma biblioteca de componentes básicos que abranja os elementos fundamentais encontrados na maioria dos sistemas térmicos, incluindo trocadores de calor, compressores, bombas, válvulas e outros equipamentos auxiliares;
- Criação de uma documentação abrangente e didática que funcione como um manual completo da ferramenta, incluindo diretrizes para extensibilidade que orientem futuros desenvolvimentos, permitindo que novos usuários compreendam os conceitos fundamentais e contribuam com o desenvolvimento;

- Validação da plataforma através da implementação de casos de estudo baseados nos tutoriais do TESPy (bomba de calor e ciclo Rankine), demonstrando a capacidade de simular sistemas térmicos simplificados.

1.2 Visão geral do trabalho

Este trabalho é dividido em 5 capítulos. O Capítulo 2 apresenta a revisão bibliográfica, onde são abordados os fundamentos da modelagem de sistemas térmicos, incluindo a otimização de sistemas e as leis de conservação que governam esses sistemas, e um panorama das ferramentas de simulação existentes, tanto comerciais quanto de código aberto. Também é realizada uma análise detalhada do TESPy, que serviu como base conceitual para o desenvolvimento do *framework* proposto, incluindo seus conceitos de programação orientada a objetos e metodologia de solução. No Capítulo 3, é apresentada a metodologia utilizada no desenvolvimento da ferramenta modular, detalhando a arquitetura geral do modelo baseada em três classes principais (*Component*, *Connection* e *System*), a implementação do método de Newton-Raphson para solução de sistemas não lineares, e a biblioteca de componentes desenvolvida. Esta seção também descreve os padrões de uso e diretrizes para extensibilidade da ferramenta, fornecendo um manual para a manutenção e extensão da ferramenta. O Capítulo 4 é dedicado à apresentação dos resultados obtidos através da validação do modelo desenvolvido. Esta seção é dividida em duas partes principais: a primeira aborda a simulação de uma bomba de calor operando com R134a, e a segunda apresenta a simulação de um ciclo Rankine com água como fluido de trabalho, tendo como base modelos de referência do TESPy. Para ambos os casos, são apresentados os resultados numéricos e análises paramétricas, demonstrando a capacidade da ferramenta em reproduzir corretamente o comportamento termodinâmico dos sistemas analisados. Por fim, o Capítulo 5 apresenta as conclusões do trabalho, destacando os principais resultados alcançados e as contribuições do *framework* desenvolvido para a simulação de sistemas térmicos. Nesta seção também são apresentadas sugestões para trabalhos futuros, incluindo propostas para expansão da biblioteca de componentes e incorporação de novas funcionalidades ao modelo.

2 Revisão Bibliográfica

Este capítulo apresenta uma análise das ferramentas computacionais para modelagem de sistemas térmicos, com foco no *software* TESPy (*Thermal Engineering Systems in Python*). Inicialmente, são apresentados os fundamentos da modelagem de sistemas térmicos, incluindo a definição e características desses sistemas e conceitos de otimização, seguidos das leis de conservação de massa e energia que governam o comportamento desses sistemas. Em seguida, é apresentado um panorama geral das ferramentas de simulação de sistemas térmicos disponíveis, destacando as diferenças entre *software* proprietário e de código aberto, bem como suas aplicações. Posteriormente, são apresentados os conceitos fundamentais da programação orientada a objetos e sua implementação na arquitetura do TESPy, seguidos de uma descrição dos módulos principais da ferramenta e sua metodologia de solução baseada no método de Newton-Raphson multidimensional.

2.1 Modelagem de Sistemas Térmicos

Um sistema, por definição, é um conjunto de componentes que estão conectados de tal forma que influenciam o desempenho uns dos outros (DHAR, 2017). No contexto dos sistemas térmicos, esses componentes interagem principalmente por meio da transferência de calor e do uso de fluidos de trabalho para o transporte de energia, formando processos ou ciclos específicos (PENONCELLO, 2018). Tais sistemas abrangem uma ampla gama de aplicações, que vão desde sistemas de aquecimento e refrigeração residenciais e comerciais até ciclos de potência a vapor para geração de energia elétrica e ciclos de refrigeração por compressão de vapor.

A simulação de sistemas térmicos consiste em prever o desempenho do sistema para diferentes condições operacionais, utilizando informações sobre o projeto e o comportamento dos seus componentes constituintes (JALURIA, 2019). Esse processo envolve submeter um modelo matemático representativo do sistema físico a variados parâmetros de entrada, como propriedades dos fluidos de trabalho, temperaturas, pressões e vazões mássicas, geralmente assumindo regime permanente de operação (STOECKER, 1989), onde se assume que todas as variáveis permanecem constantes no tempo. Para isso, é necessário conhecer os parâmetros de desempenho de cada componente, assim como as equações que descrevem as propriedades termodinâmicas dos fluidos envolvidos. Essas informações são combinadas com os balanços de massa, energia e quantidade de movimento para formar um sistema de equações simultâneas que relacionam as variáveis operacionais do sistema.

A complexidade de sistemas térmicos está relacionada ao fato de que cada componente individual possui seu próprio conjunto de equações e variáveis específicas que governam seu comportamento termodinâmico. Simultaneamente, estes componentes estão interconectados através do escoamento do fluido e transferências de energia, criando interdependências que devem ser satisfeitas. Nesse sentido, a simulação de sistemas térmicos consiste essenci-

almente em combinar os modelos que predizem o desempenho dos diversos componentes em um processo que considera essas interconexões e garante que as leis de conservação de massa, quantidade de movimento e energia sejam satisfeitas pelo sistema como um todo.

Como exemplo dessa interdependência entre componentes, pode-se considerar um sistema simples de refrigeração por compressão de vapor, composto por quatro elementos principais: compressor, condensador, válvula de expansão e evaporador, dispostos em um ciclo fechado conforme ilustrado na Figura 1. Cada componente é descrito por um conjunto específico de equações que relacionam suas variáveis de entrada e saída. No caso do compressor, por exemplo, as equações estabelecem a relação entre pressão de sucção, pressão de descarga, temperatura, eficiência isentrópica e trabalho consumido. Já o condensador é modelado por meio de balanços de energia que associam as temperaturas e vazões dos fluidos quente e frio à taxa de rejeição de calor.

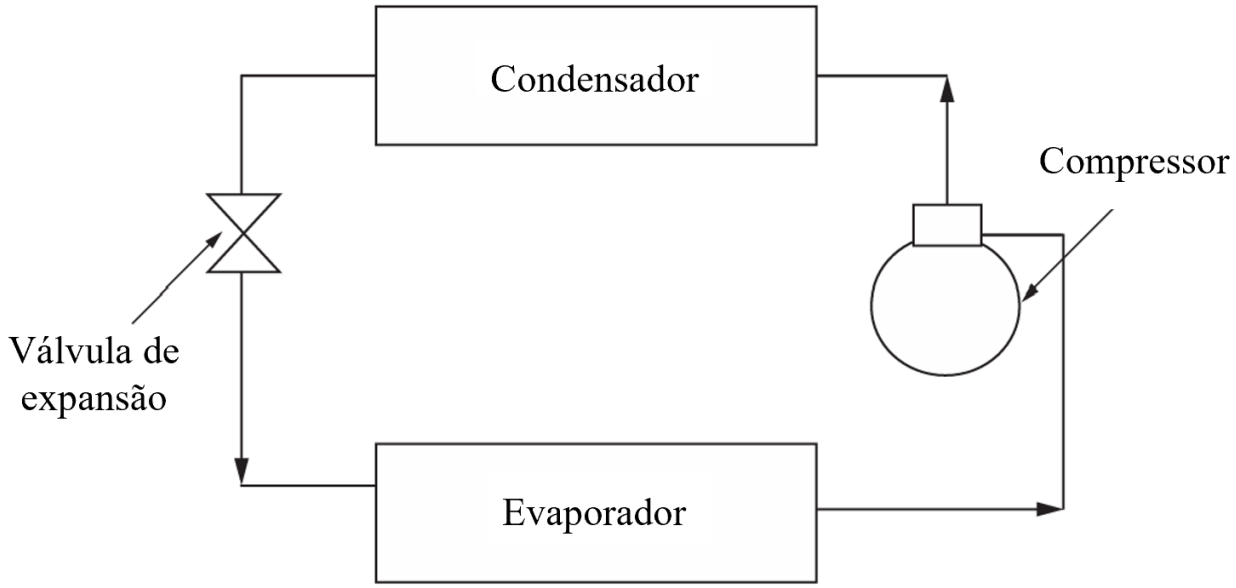

Figura 1 – Representação de um sistema de refrigeração por compressão a vapor adaptado de Dhar (2017).

Entretanto, estes componentes não operam de forma isolada. A saída do compressor (pressão e temperatura elevadas) constitui diretamente a entrada do condensador, enquanto a saída do condensador (líquido saturado) torna-se a entrada da válvula de expansão, e assim sucessivamente ao longo do ciclo. Esta interdependência significa que cada componente influencia o desempenho não apenas do componente subsequente, como seria esperado, mas também daquele que o precede através das restrições impostas pelo ciclo fechado. Por exemplo, uma mudança na eficiência do compressor altera as condições de entrada do condensador que, por sua vez, modifica as condições de operação da válvula de expansão e do evaporador, influenciando, por fim, as condições de entrada do próprio compressor.

Além disso, as conexões entre os componentes impõem restrições adicionais fundamentais que precisam ser satisfeitas simultaneamente. Em regime permanente, a conservação de massa exige que a vazão mássica seja a mesma em todos os componentes do ciclo. Da mesma forma, as condições termodinâmicas nas interfaces entre componentes devem ser

consistentes, ou seja, as pressões e temperaturas de saída de um componente devem coincidir exatamente com as condições de entrada do seguinte. Essas restrições de continuidade são essenciais para a formulação correta do problema de simulação.

Do ponto de vista matemático, a modelagem de um sistema térmico resulta em um conjunto de equações que descrevem seu comportamento físico (JALURIA, 2019). Essas equações podem ser genericamente expressas na forma residual:

$$\begin{aligned} f_1(x_1, x_2, x_3, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, x_3, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, x_3, \ldots, x_n) &= 0 \end{aligned} \tag{2.1}$$

onde x_i representam as variáveis do sistema, como vazões, pressões e temperatura, e f_j as respectivas equações que definem o problema, incluindo tanto as equações específicas de cada componente quanto as restrições das conexões entre eles. Na maioria dos casos práticos, a solução dessas equações exige o uso de métodos numéricos iterativos, como o método de Newton-Raphson, especialmente devido à presença de não linearidades que são comuns em sistemas térmicos, provenientes tanto das equações dos componentes quanto das propriedades termodinâmicas dos fluidos.

A solução da simulação geralmente envolve a resolução de um sistema de n equações com n incógnitas, cuja complexidade cresce rapidamente com o número de componentes e variáveis do sistema. Assim, o problema de simulação pode então ser interpretado como a solução simultânea de todas essas equações, onde se busca encontrar o conjunto único de valores das variáveis que satisfaça simultaneamente todas as equações do sistema, provenientes dos componentes e conexões entre eles. Nesse processo, uma etapa crítica para obter a solução é a escolha de estimativas iniciais adequadas para todas as variáveis, de modo a garantir a convergência do método numérico e evitar soluções fisicamente irreais.

Após a solução bem-sucedida do modelo, procede-se à etapa de avaliação do sistema, que consiste em analisar sistematicamente como variações nos parâmetros de entrada afetam o comportamento global do sistema. Esta análise paramétrica é fundamental para compreender quais variáveis exercem maior influência no desempenho do sistema e quais oferecem maior potencial para uma futura otimização (JALURIA, 2019).

2.1.1 Otimização de Sistemas Térmicos

A crescente competição global exige que os sistemas sejam não apenas funcionais, mas também otimizados em termos de eficiência, custo e desempenho (JALURIA, 2019). A otimização de sistemas térmicos consiste fundamentalmente em determinar os valores das variáveis de projeto que minimizam ou maximizam uma função objetivo específica, de modo a encontrar a configuração ótima dentre as possibilidades viáveis do projeto, respeitando simultaneamente um conjunto de restrições impostas pela física do problema.

Este processo pode ser matematicamente formulado como a busca pelo ponto ótimo dentro de um domínio de soluções aceitáveis, conforme ilustrado na Figura 2. Neste domínio, qualquer ponto representa uma configuração viável do sistema que atende aos requisitos básicos, porém apenas um ponto específico corresponde à solução ótima global. As variáveis x_1 e x_2 representam parâmetros de projeto como a efetividade de um trocador de calor ou condições de operação, como a vazão mássica em um ciclo de refrigeração.

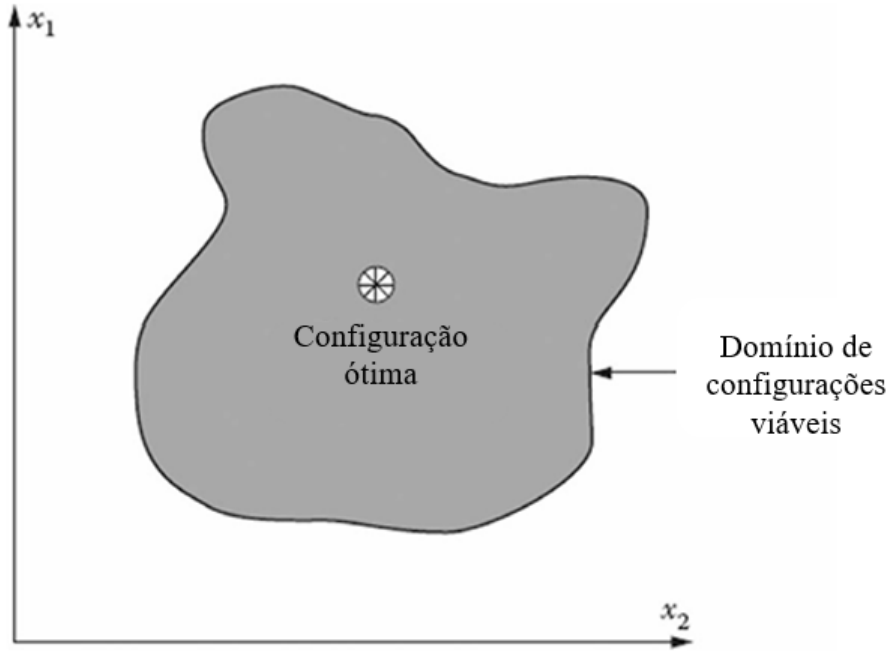

Figura 2 – Representação da configuração ótima dentro do domínio de configurações viáveis, adaptado de Jaluria (2019).

A função objetivo representa o critério de desempenho que se deseja otimizar, frequentemente relacionado a características como peso, volume e eficiência como, por exemplo, o Coeficiente de Performance (COP) em um ciclo de refrigeração. Já as restrições definem os limites que devem ser respeitados pelas variáveis de projeto, assegurando que as soluções permaneçam fisicamente e operacionalmente viáveis. As restrições de igualdade são provenientes das leis de conservação, como a conservação de massa que impõe vazão constante em um ciclo fechado. As restrições de desigualdade definem limites operacionais, como temperaturas máximas dos materiais, pressões de trabalho permitidas pelos equipamentos e dimensões disponíveis.

Matematicamente, o problema de otimização pode ser formulado como a busca pelos valores das variáveis de projeto que otimizam a função objetivo:

$$U = U(x_1, x_2, \ldots, x_n) \tag{2.2}$$

onde U representa a função objetivo e $x_1, x_2, \ldots, x_n$ são as variáveis de projeto do sistema. O objetivo é determinar o conjunto ótimo dessas variáveis que maximize ou minimize U.

Diante da complexidade dos sistemas térmicos e das múltiplas variáveis e restrições envolvidas, diferentes métodos têm sido desenvolvidos para resolver problemas de otimização. Os métodos clássicos baseados em cálculo utilizam derivadas da função objetivo e das

restrições para determinar a localização de mínimos ou máximos. Entretanto, esses métodos apresentam algumas limitações, pois exigem que a função objetivo e as restrições sejam contínuas, bem comportadas e diferenciáveis em todo o domínio de projeto. Para superar essas limitações, diversos outros métodos têm sido desenvolvidos para atender às crescentes demandas por soluções otimizadas na área de sistemas térmicos, como algoritmos genéticos, os quais têm sido empregados para viabilizar a otimização em cenários mais complexos (JALURIA, 2019).

2.1.2 Fundamentos da Modelagem Matemática

Sistemas térmicos podem ser conceitualizados como responsáveis por duas funções básicas: (1) entrega de potência ou (2) transporte de energia utilizando um fluido para aquecimento e/ou resfriamento (PENONCELLO, 2018). A modelagem matemática desses sistemas requer a aplicação das leis fundamentais da mecânica dos fluidos, termodinâmica e transferência de calor.

2.1.2.1 Lei Generalizada de Balanço

A base conceitual da modelagem e projeto de sistemas térmicos é a aplicação da lei generalizada de balanço (PENONCELLO, 2018), ilustrada na Figura 3. Considerando um sistema com uma quantidade física generalizada Ω em um instante de tempo, onde $\dot{\Omega}_{in}$ e $\dot{\Omega}_{out}$ representam o fluxo de Ω entrando e saindo do sistema, respectivamente, o balanço pode ser expresso como:

$$\begin{bmatrix} \text{Taxa de } \Omega \text{ cruzando} \\ \text{a fronteira do sistema e} \\ \text{entrando no sistema} \end{bmatrix} - \begin{bmatrix} \text{Taxa de } \Omega \text{ cruzando} \\ \text{a fronteira do sistema e} \\ \text{saindo do sistema} \end{bmatrix} + \begin{bmatrix} \text{Taxa de } \Omega \text{ sendo} \\ \text{gerada dentro do} \\ \text{sistema} \end{bmatrix} = \begin{bmatrix} \text{Taxa de armazenamento} \\ \text{de } \Omega \text{ no} \\ \text{sistema} \end{bmatrix} \tag{2.3}$$

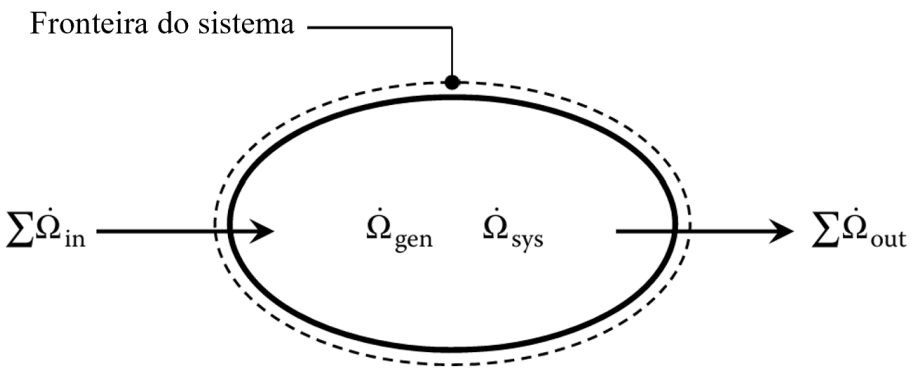

Figura 3 – Representação de um sistema generalizado, adaptado de Penoncello (2018).

Na forma matemática, considerando o termo de armazenamento como uma derivada temporal, esta expressão pode ser escrita como:

$$\sum \dot{\Omega}_{in} - \sum \dot{\Omega}_{out} + \dot{\Omega}_{gen} = \frac{d\Omega_{sys}}{dt} \tag{2.4}$$

Os dois primeiros termos da Equação 2.4 representam a taxa líquida de entrada da quantidade Ω no sistema. O termo de geração $\dot{\Omega}_{gen}$ indica se a quantidade Ω está sendo criada ou destruída dentro do sistema, onde um valor positivo representa geração interna de Ω, enquanto um valor negativo indica consumo ou destruição. Já o termo de armazenamento $\frac{d\Omega_{sys}}{dt}$ representa a taxa de variação da quantidade de Ω dentro do sistema. Se for positivo, significa que Ω está se acumulando, se for negativo, indica que está sendo reduzida.

Neste trabalho, considera-se regime permanente, condição na qual as propriedades do sistema não variam com o tempo. Dessa forma, o termo de armazenamento é anulado:

$$\frac{d\Omega_{sys}}{dt} = 0 \tag{2.5}$$

Portanto, a lei generalizada de balanço em regime permanente é definida como:

$$\sum \dot{\Omega}_{in} - \sum \dot{\Omega}_{out} + \dot{\Omega}_{gen} = 0 \tag{2.6}$$

2.1.2.2 Conservação de Massa

A equação de conservação de massa pode ser desenvolvida fazendo a substituição $\Omega = m$ na Equação 2.4. Nesse caso, massa é uma quantidade conservada, portanto, o termo de geração é zero, resultando na seguinte expressão:

$$\sum \dot{m}_{in} - \sum \dot{m}_{out} = \frac{dm_{sys}}{dt} \tag{2.7}$$

Considerando a condição de regime permanente, onde a vazão mássica que entra no sistema é igual à vazão mássica que sai, não ocorre acúmulo de massa no interior do sistema. Logo, a Equação 2.7 pode ser escrita como:

$$\sum \dot{m}_{in} = \sum \dot{m}_{out} \tag{2.8}$$

2.1.2.3 Conservação de Energia (Primeira Lei da Termodinâmica)

A conservação de energia, também conhecida como primeira lei da termodinâmica, estabelece que a energia não pode ser criada nem destruída, apenas transformada. Para sistemas térmicos, substitui-se $\Omega = E$ na Equação 2.4, onde E representa a energia total do sistema.

Aplicando a lei generalizada de balanço para energia, obtém-se:

$$\sum \dot{E}_{in} - \sum \dot{E}_{out} = \frac{dE_{sys}}{dt} \tag{2.9}$$

Para sistemas abertos, a energia pode ser transferida de duas formas: (1) por calor e trabalho através das fronteiras do sistema, e (2) transportada pelo fluxo de massa. Além da

energia interna, o transporte de massa carrega também energia cinética e energia potencial. Assim, a equação geral de conservação de energia para sistemas abertos pode ser expressa como:

$$\sum(\dot{Q}_{in}+\dot{W}_{in})-\sum(\dot{Q}_{out}+\dot{W}_{out})+\sum\dot{m}_{in}\left(h_{in}+\frac{V_{in}^2}{2}+gz_{in}\right)-\sum\dot{m}_{out}\left(h_{out}+\frac{V_{out}^2}{2}+gz_{out}\right)=\frac{dE_{sys}}{dt} \tag{2.10}$$

onde $\dot{Q}$ representa a taxa de transferência de calor, $\dot{W}$ a taxa de trabalho realizado, $\dot{m}$ a vazão mássica, h a entalpia específica, V a velocidade do fluido e z a elevação. Em aplicações de engenharia térmica, as variações de energia cinética ($\frac{V^2}{2}$) e potencial (gz) são geralmente desprezíveis quando comparadas às mudanças de entalpia do fluido, permitindo a simplificação da equação para:

$$\sum(\dot{Q}_{in}+\dot{W}_{in})-\sum(\dot{Q}_{out}+\dot{W}_{out})+\sum\dot{m}_{in}h_{in}-\sum\dot{m}_{out}h_{out}=\frac{dE_{sys}}{dt} \tag{2.11}$$

Considerando regime permanente, onde as propriedades do sistema não variam com o tempo ($\frac{dE_{sys}}{dt}=0$), a equação de conservação de energia se simplifica para:

$$\sum(\dot{Q}_{in}+\dot{W}_{in})-\sum(\dot{Q}_{out}+\dot{W}_{out})+\sum\dot{m}_{in}h_{in}-\sum\dot{m}_{out}h_{out}=0 \tag{2.12}$$

Esta equação estabelece que, em regime permanente, a soma algébrica das taxas de transferência de calor, trabalho e energia transportada pelo fluxo de massa deve ser nula, constituindo a base fundamental para a modelagem de componentes de sistemas térmicos.

2.1.2.4 Balanço de Entropia (Segunda Lei da Termodinâmica)

A segunda lei da termodinâmica auxilia na identificação de irreversibilidades em processos ou sistemas. Diferentemente da massa e da energia, a entropia não é uma quantidade conservada, pois pode ser gerada devido a fenômenos irreversíveis, como atrito mecânico, atrito viscoso devido ao escoamento do fluido, transferência de calor por meio de diferenças finitas de temperatura, expansões ou contrações abruptas e mistura de fluidos.

Para sistemas abertos, o balanço de entropia é obtido substituindo $\Omega = S$ na Equação 2.4, resultando na forma geral:

$$\sum\dot{S}_{in}-\sum\dot{S}_{out}+\dot{S}_{gen}=\frac{dS_{sys}}{dt} \tag{2.13}$$

Nessa expressão, $\dot{S}_{gen}$ é a taxa de geração de entropia, sempre maior ou igual a zero. Seu valor indica o grau de irreversibilidade do processo:

$$\dot{S}_{\text{gen}}=\begin{cases}>0 & \text{processo real e irreversível}\\ =0 & \text{processo reversível ideal}\\ <0 & \text{impossível fisicamente}\end{cases}$$

A entropia pode ser transferida tanto pelo calor trocado nas fronteiras quanto pelo fluxo de massa que cruza o sistema. Assim, a equação pode ser reescrita separando os termos dessas contribuições.

$$\left(\sum \dot{S}_{in} - \sum \dot{S}_{out}\right)_{Q} + \left(\sum \dot{S}_{in} - \sum \dot{S}_{out}\right)_{fluxo} + \dot{S}_{gen} = \frac{dS_{sys}}{dt} \tag{2.14}$$

onde Q corresponde à entropia transferida devido à troca de calor nas fronteiras do sistema, enquanto o termo *fluxo* refere-se à transferência de entropia associada ao transporte de massa que cruza o contorno. A taxa de transferência de entropia associada ao calor é expressa por:

$$\dot{S}_{Q} = \sum \frac{\dot{Q}}{T_b} \tag{2.15}$$

onde T_b é a temperatura absoluta da fronteira onde ocorre a transferência de calor $\dot{Q}$.

A taxa de transferência de entropia associada ao fluxo de massa corresponde ao produto da vazão mássica pela entropia específica do fluido na fronteira de entrada ou saída. Assim, para cada fluxo de massa que cruza o contorno do sistema, a taxa de entrada ou saída de entropia pode ser calculada como:

$$\dot{S}_{fluxo} = \dot{m}\, s \tag{2.16}$$

onde s é a entropia específica do fluido. Assim, a equação geral do balanço de entropia pode ser reescrita como:

$$\left[\sum \frac{\dot{Q}_{in}}{T_{b,in}} - \sum \frac{\dot{Q}_{out}}{T_{b,out}}\right] + \left[\sum \dot{m}_{in} s_{in} - \sum \dot{m}_{out} s_{out}\right] + \dot{S}_{gen} = \frac{dS_{sys}}{dt} \tag{2.17}$$

Considerando regime permanente, no qual as propriedades do sistema não variam com o tempo ($\frac{dS_{sys}}{dt} = 0$), a equação se simplifica para:

$$\left[\sum \frac{\dot{Q}_{in}}{T_{b,in}} - \sum \frac{\dot{Q}_{out}}{T_{b,out}}\right] + \left[\sum \dot{m}_{in} s_{in} - \sum \dot{m}_{out} s_{out}\right] + \dot{S}_{gen} = 0 \tag{2.18}$$

A análise do balanço de entropia é fundamental não apenas para verificar a conformidade de processos com a segunda lei da termodinâmica, mas também como ferramenta de diagnóstico e otimização em sistemas térmicos. Ao quantificar a taxa de geração de entropia, é possível identificar e localizar fontes de irreversibilidade, permitindo propor melhorias que reduzam perdas e aumentem a eficiência do sistema térmico.

2.2 Ferramentas de Simulação Existentes

A complexidade matemática inerente aos sistemas térmicos, caracterizada por sistemas de equações não lineares acopladas e pela necessidade de avaliar propriedades termodinâmicas dos fluidos de trabalho, tornou o desenvolvimento de ferramentas computacionais especializadas uma necessidade fundamental para a análise e projeto desses sistemas. Nas últimas décadas, as simulações computacionais tornaram-se um pilar central da engenharia moderna, levando ao desenvolvimento de um grande número de ferramentas de simulação para resolver uma ampla gama de problemas de engenharia (RICHTER, 2008).

A partir dessa diversidade de ferramentas, é possível classificar as soluções existentes em duas grandes categorias: *softwares* comerciais (proprietários) e ferramentas de código

aberto (*open-source*). Cada uma dessas abordagens apresenta vantagens e limitações específicas, que impactam diretamente a escolha da ferramenta conforme o contexto de aplicação, como a área industrial, educacional ou de pesquisa.

2.2.1 Ferramentas Comerciais

As ferramentas comerciais são tradicionalmente voltadas para o mercado industrial e consultorias especializadas. Essas soluções, que incluem desde *softwares* baseados em componentes como TRNSYS (TESS, 2025) e VapCyc (SYSTEMS, 2025) até solucionadores de equações como EES (F-CHART, 2025), oferecem diferentes abordagens para modelagem de sistemas térmicos. Ferramentas como TRNSYS fornecem interfaces gráficas avançadas e bibliotecas de componentes com validação extensiva, enquanto o EES oferece um modelo para resolução de equações com uma base de dados de propriedades termodinâmicas abrangente, acompanhadas de suporte técnico dedicado (RICHTER, 2008).

No entanto, essas ferramentas frequentemente apresentam limitações quanto à flexibilidade para modificação ou criação de novos modelos. Os custos de licenciamento podem representar um obstáculo para instituições acadêmicas ou projetos de pesquisa com recursos limitados. Além disso, por utilizarem códigos de implementação fechados, essas ferramentas geralmente não permitem acesso detalhado às formulações matemáticas internas, de modo a limitar a capacidade de realizar análises aprofundadas, como estudos exergéticos detalhados ou implementação de métodos de otimização customizados, ou mesmo compreender os procedimentos de solução numérica adotados (ZODER et al., 2018).

2.2.1.1 Engineering Equation Solver (EES)

O *Engineering Equation Solver* (EES) é um programa solucionador de equações de propósito geral que pode resolver numericamente milhares de equações algébricas e diferenciais não lineares acopladas. O programa também pode ser usado para resolver equações diferenciais e integrais, realizar otimizações, fornecer análises de incerteza, executar regressão linear e não linear, converter unidades, verificar consistência de unidades e gerar gráficos (F-CHART, 2025).

Uma característica do EES é sua base de dados de propriedades termodinâmicas para centenas de fluidos, integrada de forma que pode ser utilizada diretamente com a capacidade de resolução de equações. Diferentemente de *softwares* baseados em componentes, o EES requer que o usuário formule explicitamente todas as equações que descrevem o sistema térmico, oferecendo flexibilidade total na modelagem, mas sem disponibilizar uma biblioteca de componentes pré-configurados para facilitar a construção de modelos.

A ferramenta CoolPack, desenvolvida por Jakobsen, Rasmussen e Andersen (1999), exemplifica tanto as potencialidades quanto as limitações dessa abordagem. O CoolPack foi desenvolvido como uma coleção de programas de simulação relacionados à refrigeração, incluindo análise de ciclos, dimensionamento de sistemas, simulação de componentes e análise de custos do ciclo de vida. O *software* utilizava o EES como base para seus modelos

de simulação, aproveitando suas capacidades de resolução de equações e base de dados de propriedades.

Entretanto, a dependência do EES como plataforma de desenvolvimento revelou limitações significativas para a manutenção e evolução do CoolPack. Conforme os próprios desenvolvedores indicam, a forte dependência do EES, associada a questões como a necessidade de licenciamento e a impossibilidade de modificar aspectos fundamentais da arquitetura do *software*, limitou a capacidade de evolução da ferramenta. Essas dificuldades dificultaram a implementação de melhorias e correções ao longo do tempo, contribuindo para a descontinuação do projeto, que foi substituído por soluções mais modulares e com maior independência entre seus componentes (DTU, 2025).

2.2.1.2 Transient System Simulation Tool (TRNSYS)

O TRNSYS é um ambiente de simulação modular e orientado a componentes, amplamente utilizado para modelagem de sistemas transientes, ou seja, aqueles cujas variáveis variam ao longo do tempo, com ênfase em aplicações térmicas e energéticas. Sua arquitetura é composta por dois elementos principais: o núcleo de simulação (*kernel*), responsável pela leitura dos arquivos de entrada, solução iterativa das equações do sistema e controle da convergência, e uma biblioteca de componentes que representam diferentes elementos de um sistema (TESS, 2025).

A biblioteca padrão do TRNSYS contém aproximadamente 150 modelos, abrangendo desde equipamentos típicos de sistemas térmicos até tecnologias emergentes, como unidades de conversão de energia, turbinas eólicas e eletrolisadores. Os usuários têm a possibilidade de modificar os componentes existentes ou desenvolver novos, ampliando a flexibilidade da ferramenta.

2.2.1.3 VapCyc - Advanced system simulation tool for HVAC applications

O VapCyc (SYSTEMS, 2025) é uma ferramenta para o projeto e análise de ciclos de compressão de vapor operando em regime permanente, com foco específico em sistemas de refrigeração e bombas de calor. Essa especialização permite ao *software* oferecer modelos detalhados e validados para esses tipos de aplicações, incluindo correlações empíricas para transferência de calor em evaporadores e condensadores, uma biblioteca de refrigerantes e ferramentas de otimização para identificação de configurações operacionais ideais.

A arquitetura do VapCyc é baseada em programação orientada a objetos, estruturada em três elementos fundamentais: componentes, portas (*ports*) e junções (*junctions*) (RICHARDSON, 2006). Os componentes representam os equipamentos físicos do sistema, como compressor, condensador, válvula de expansão e evaporador. Cada componente é modelado como uma entidade independente, responsável por satisfazer seus próprios balanços de massa, quantidade de movimento e energia. Todos os componentes devem ser criados seguindo o Padrão de Componentes (*Component Standard*), uma interface de classe que define um conjunto específico de métodos e propriedades obrigatórias. Esse padrão garante

que todos os componentes possam se comunicar de forma uniforme com o *solver* do sistema, independentemente da configuração do ciclo.

As portas constituem a interface entre os componentes e o sistema, definindo os pontos de entrada e saída de escoamento, com suas propriedades termodinâmicas associadas. As junções são pontos dentro do sistema que permitem a interação entre os componentes e o *solver*, funcionando como o mecanismo pelo qual as portas dos componentes são conectadas.

Um exemplo ilustrativo dessa arquitetura pode ser visto na Figura 4, que apresenta um sistema de refrigeração por compressão de vapor composto por quatro componentes: compressor, condensador, válvula de expansão e evaporador, os quais são interligados por quatro junções, indicadas por J. Nesse exemplo, os componentes e sua conectividade são especificados ao *Junction Solver*, formando a estrutura física e computacional do sistema de refrigeração.

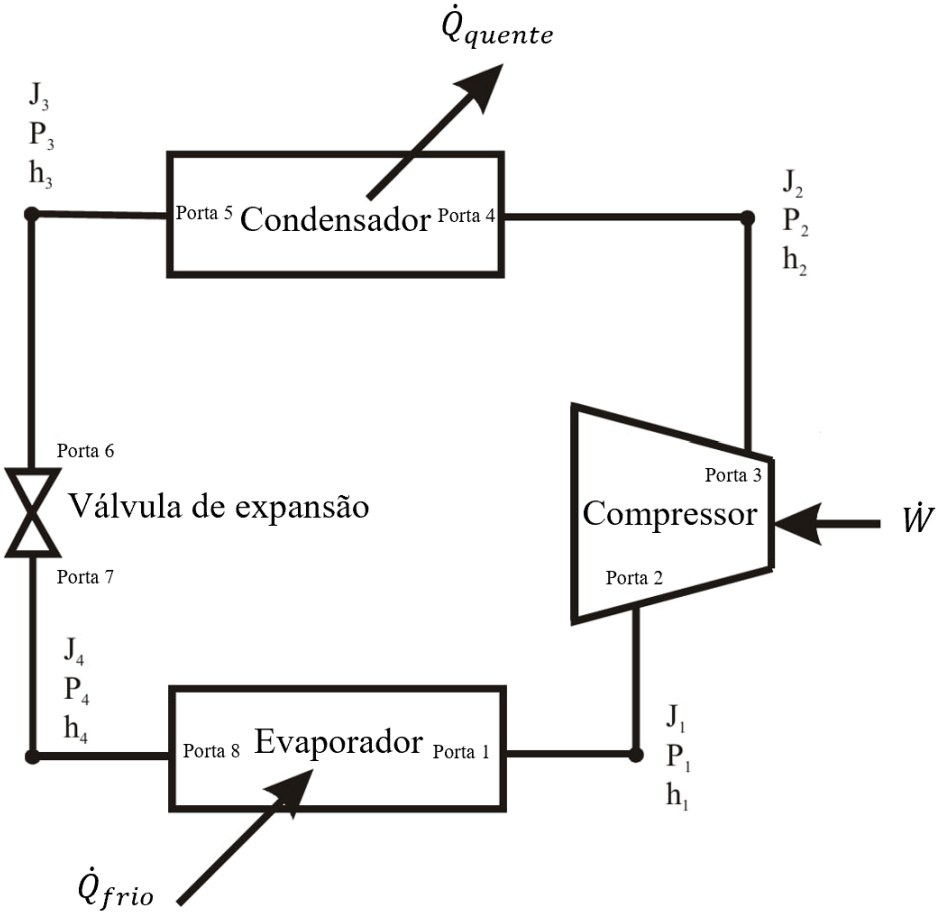

Figura 4 – Representação de um sistema de refrigeração no *framework* VapCyc, adaptado de Richardson (2006).

O processo de solução, denominado *Junction Solver*, adota como variáveis independentes as pressões e as entalpias específicas nas junções. Assim, a solução do sistema é obtida pela satisfação simultânea dos balanços de massa e energia em cada junção, além de um balanço de massa global no caso de sistemas fechados. Para isso, é utilizado o método de Newton-Raphson, com construção explícita da matriz Jacobiana, que contém as derivadas parciais das equações de balanço em relação às variáveis. O processo é iterativo e prossegue

até que os resíduos de todas as equações sejam reduzidos a valores abaixo de um critério de convergência predefinido.

2.2.2 Ferramentas de Código Aberto (open-source)

Por outro lado, o desenvolvimento de *frameworks open-source* representa uma evolução importante no cenário de simulação de sistemas térmicos. Esse movimento tem sido impulsionado pela necessidade de maior transparência, reprodutibilidade científica e acessibilidade, especialmente em ambientes de pesquisa e educação. Diferentemente das soluções comerciais, as ferramentas de código aberto permitem total acesso ao código-fonte, viabilizando a customização de modelos, a integração com outras bibliotecas científicas e o desenvolvimento de novas funcionalidades pelos próprios usuários (ZODER et al., 2018).

No contexto específico de sistemas térmicos, duas ferramentas se destacam: a VCLib (*Vapor Compression Library*), desenvolvida em Modelica para a modelagem dinâmica de bombas de calor, e o TESPy (Thermal Engineering Systems in Python), focado na simulação de ciclos termodinâmicos de energia em regime permanente. Ambas as ferramentas exemplificam abordagens para simulação de sistemas térmicos, oferecendo soluções complementares conforme os requisitos específicos de cada aplicação.

2.2.2.1 VCLib - Vapor Compression Library

A VCLib *(Vapor Compression Library)* é uma biblioteca *open-source* desenvolvida em Modelica, especificamente para modelagem dinâmica e simulação de ciclos de compressão de vapor, com ênfase em bombas de calor e sistemas de refrigeração (VERING et al., 2022). Seu desenvolvimento foi motivado pela necessidade de uma ferramenta educacional e de pesquisa que permitisse a análise de sistemas completos, com diferentes níveis de detalhe e de forma modular e escalável, a partir de uma abordagem orientada a objetos.

A arquitetura da VCLib é estruturada em três camadas hierárquicas: camada de aplicação (*application layer*), camada de interconexão (*interconnection layer*) e camada de componentes (*component layer*).Essa divisão interna (*layers*) define o nível de detalhamento com que o usuário ou desenvolvedor interage com os modelos. A camada de componentes contém os modelos individuais de cada equipamento, como compressores, trocadores de calor e válvulas de expansão, desenvolvidos de forma independente e obedecendo aos balanços fundamentais de massa e energia. A camada de interconexão permite a conexão entre os componentes, formando a topologia do sistema simulado. Assim, os componentes utilizam portas termodinâmicas para comunicação de variáveis como pressão, entalpia e vazão mássica, permitindo a conexão entre eles por meio dessa camada de interconexão. Por fim, a camada de aplicação oferece modelos de sistemas completos e pré-configurados, prontos para uso em análises de desempenho ou controle.

Além dessa divisão interna por camadas, a VCLib organiza seus modelos em diferentes níveis de aplicação (*levels*), os quais correspondem ao escopo ou ao domínio de aplicação da simulação. Esses níveis são: nível de propriedades dos fluidos (*Fluid Property Level*), nível de componentes da bomba de calor (*Heat Pump Component Level*), nível de sistema energético

(*Energy System Level*) e nível de edifícios (*Building and District Level*). Cada nível representa uma escala de modelagem, que vai desde o cálculo isolado de propriedades termodinâmicas dos fluidos até a integração de sistemas completos em aplicações em edifícios.

A relação entre *layers* e *levels* está ilustrada na Figura 5. Enquanto as camadas (*layers*) descrevem a organização interna da biblioteca e o nível de detalhamento na construção dos modelos, os níveis *levels* definem o escopo da aplicação que o usuário pretende simular. A biblioteca estabelece um gradiente de complexidade que varia desde a simplicidade de uso *(simplicity of entry)* na camada de aplicação até o maior nível de detalhamento *(level of detail)* na camada de componentes. Paralelamente, os quatro níveis de aplicação permitem abordar desde propriedades isoladas de refrigerantes até sistemas energéticos urbanos. Essa estrutura hierárquica garante flexibilidade para diferentes perfis de usuários, desde estudantes interessados em análises conceituais até pesquisadores focados em detalhes de modelagem de componentes ou simulações em maiores escalas, permitindo que cada usuário acesse o nível de complexidade apropriado às suas necessidades.

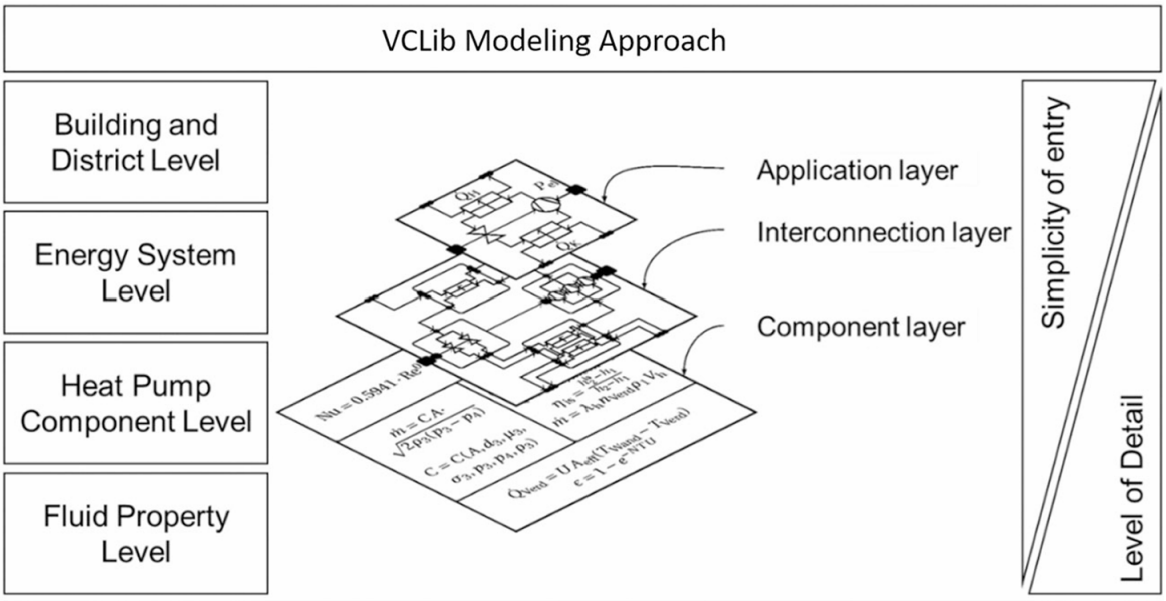

Figura 5 – Abordagem da modelagem do VCLib, retirado de Vering et al. (2022).

2.2.2.2 TESPy - Thermal Engineering Systems in Python

O TESPy (*Thermal Engineering Systems in Python*) (WITTE; TUSCHY, 2020a) é uma ferramenta *open-source* desenvolvida para simulação e otimização de sistemas térmicos em regime permanente, com foco na modelagem de ciclos de potência, sistemas de aquecimento distrital, bombas de calor e outros processos de conversão de energia térmica. Seu desenvolvimento foi motivado pela necessidade de uma alternativa de código aberto flexível, extensível e de fácil integração com bibliotecas da linguagem Python, incluindo ferramentas como a CoolProp (BELL et al., 2014) para o cálculo de propriedades termodinâmicas.

A arquitetura do TESPy é orientada a componentes e estruturada em três módulos principais que definem sua funcionalidade modular. Conforme apresentado na Figura 6, que ilustra o conceito básico de modelagem do TESPy através do exemplo de uma bomba de

calor simples, a ferramenta constrói sistemas baseados em componentes que são conectados usando conexões para formar uma rede topológica.

O módulo *components* contém todos os equipamentos do sistema, os quais são organizados em categorias como turbomáquinas, trocadores de calor, tubulações e válvulas, onde cada componente implementa individualmente seus balanços de massa e energia, além de equações específicas. O módulo *connections*, indicado pelos números de 0 a 4 na figura, gerencia as conexões entre componentes, armazenando as informações sobre as variáveis termodinâmicas dos fluidos como pressão, temperatura, entalpia e vazão mássica, que escoam entre os componentes e definem a topologia do sistema simulado. Por fim, o módulo *network* atua como a unidade central de gerenciamento da simulação, coordenando o pré-processamento como a montagem do sistema de equações e a inicialização das variáveis, além da execução da solução numérica e do pós-processamento dos resultados do sistema.

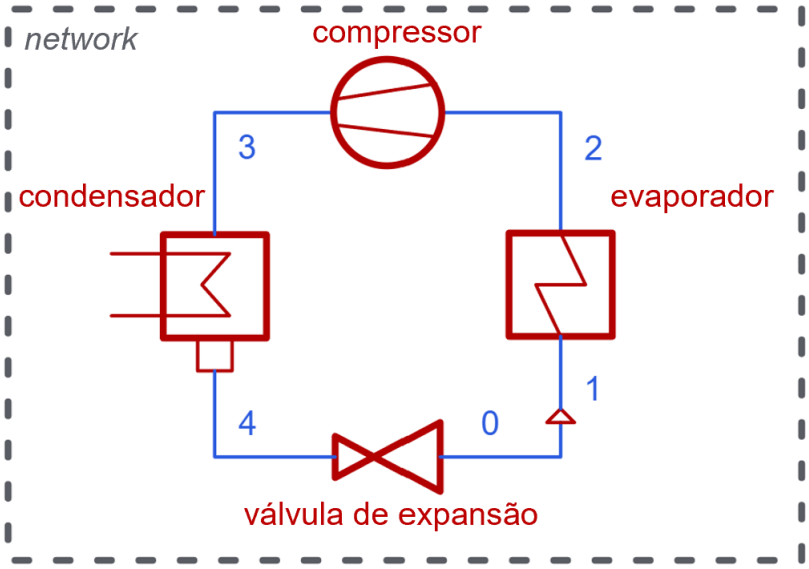

Figura 6 – Estrutura modular do TESPy, exemplificada por uma bomba de calor simples, adaptado de Witte (2025a).

O método de solução utilizado no TESPy baseia-se na formulação de um sistema de equações não lineares provenientes dos balanços de massa e energia e equações específicas de cada componente e de cada conexão do sistema. A resolução dessas equações é realizada por um *solver* interno que implementa o método de Newton-Raphson multidimensional, com construção explícita da matriz Jacobiana para determinar a vazão mássica, pressão, entalpia e composição do fluido de cada conexão. A modelagem de uma planta específica é realizada conectando os componentes para formar uma rede topológica, na qual as condições de operação são definidas pelas propriedades do fluido em cada conexão entre os elementos do sistema.

2.2.3 *Síntese Comparativa das Ferramentas de Simulação*

A análise das ferramentas de simulação existentes mostra que, independentemente de serem comerciais ou de código aberto, a maioria adota uma abordagem comum baseada

em programação orientada a objetos e metodologia orientada a componentes. Essa convergência de metodologia representa a estratégia amplamente adotada para lidar com essa complexidade inerente aos sistemas térmicos. Ao estruturar os modelos de forma orientada a componentes, é possível decompor o sistema global em unidades menores e bem definidas, facilitando tanto a formulação das equações quanto a solução numérica desses sistemas acoplados e não lineares.

Ferramentas como VapCyc, TESPy e VCLib, apesar de suas diferentes implementações e focos de aplicação, compartilham uma arquitetura estrutural fundamentalmente similar, organizada em três elementos principais: componentes, os quais representam os equipamentos físicos do sistema, conexões ou junções, que gerenciam o escoamento de fluidos e propriedades termodinâmicas entre componentes, e, por fim, o sistema ou rede, a qual coordena a solução global. Essas ferramentas também se assemelham na metodologia de solução, utilizando o método de Newton-Raphson para resolver sistemas de equações não lineares, tendo como variáveis principais as propriedades termodinâmicas nas conexões, especialmente pressão e entalpia específica.

Dada sua arquitetura orientada a componentes, sua metodologia de solução baseada no método de Newton-Raphson e sua implementação em Python, o TESPy foi selecionado como base para o desenvolvimento do modelo apresentado neste trabalho. Além dessas características, sua natureza de código aberto viabilizou a adaptação e o desenvolvimento de um modelo próprio, alinhado aos requisitos específicos do trabalho, a simplificação da estrutura original e a criação de uma base mais acessível e facilmente extensível para futuras modificações. A Seção 2.3 apresenta uma descrição mais detalhada da estrutura, das metodologias de solução e das principais características dessa ferramenta.

2.3 TESPy

O TESPy (*Thermal Engineering Systems in Python*), como apresentado na Subseção 2.2.2.2 é uma biblioteca de código aberto desenvolvida para a modelagem, simulação e otimização de sistemas térmicos por meio de uma abordagem modular e orientada a objetos. Essa modularidade é baseada nos princípios da programação orientada a objetos, que permite a construção de componentes reutilizáveis, a adição de novas funcionalidades e a manutenção de uma estrutura de código organizada.

A versatilidade do TESPy tem sido demonstrada em diversas aplicações práticas. Chen et al. (2022) realizaram a otimização paramétrica de uma planta de ciclo Rankine orgânico (ORC) utilizando fontes geotérmicas bifásicas, estabelecendo as bases para o submódulo de otimização do TESPy. Fry, Eagle-Bluestone e Witte (2022) modelaram um sistema de cogeração ORC para exploração geotérmica sedimentar, monitorando o desenvolvimento da potência elétrica e o fornecimento de calor para um sistema de aquecimento distrital ao longo de 40 anos de operação. Zoder et al. (2018) demonstraram a aplicabilidade do TESPy em análises exergéticas de turbinas a gás de ciclo combinado, validando a ferramenta.

2.3.1 Conceitos de Programação Orientada a Objetos

A simulação orientada a objetos permite ganhos significativos em produtividade de programação, manutenção de código e reutilização de modelos. Por meio de uma metodologia de simulação generalizada, é possível representar diferentes sistemas térmicos, modelando seus componentes individuais como objetos que interagem dentro de uma estrutura de sistema coordenada (RICHARDSON, 2006). Aplicando essa abordagem ao contexto de um sistema de compressão de vapor, por exemplo, cada componente físico, como um compressor ou trocador de calor, pode ser modelado como uma entidade independente (objeto), com suas próprias propriedades e métodos, enquanto um módulo de rede (*network*) coordena a interação entre eles para formar e solucionar o sistema.

Essa abordagem modular oferece diversas vantagens, uma vez que os componentes podem herdar de uma classe base comum, permitindo o compartilhamento de código e simplificando tanto o desenvolvimento quanto a manutenção do *software* de simulação. Além disso, quando combinada com uma rotina de solução generalizada no *solver*, essa abordagem permite simular diferentes configurações de sistema sem alterações no algoritmo principal de solução.

A programação orientada a objetos, com seus princípios de organização e estruturação (Phillips (2010)), é aplicada no TESPy para representar componentes físicos e suas interações em sistemas térmicos. A seguir, são apresentados os principais princípios e sua aplicação prática na arquitetura da ferramenta:

- Classe (*Class*): Uma classe define a estrutura e o comportamento de um tipo específico de entidade, no caso, de um componente do sistema. No TESPy, cada classe de componente, como *Compressor*, *Turbine*, *Pump*, define os parâmetros de desempenho e os métodos para cálculo dos balanços de massa e energia. Essas classes estão localizadas dentro do módulo *components*.

- Objeto (*Object*): Um objeto é uma instância concreta de uma classe. Por exemplo, ao criar um compressor específico dentro de uma simulação, o usuário está instanciando um objeto da classe *Compressor*. Cada objeto possui seus próprios dados para os atributos descritos pela classe como eficiência, potência.

- Herança (*Inheritance*): A herança permite que uma classe (classe filha) herde atributos e métodos de outra classe (classe pai), promovendo reutilização de código. O TESPy utiliza herança para compartilhar funcionalidades comuns entre diferentes tipos de componentes. Por exemplo, todas as classes de componentes herdam da classe base *Component*, de modo que métodos e atributos genéricos, como aqueles relacionados as portas de entrada e saída dos componentes, são definidos na classe base e reutilizados pelas classes derivadas como *Compressor* e *Turbine*.

- Encapsulamento *(Encapsulation)*: O encapsulamento agrupa dados, ou seja, atributos, e métodos que operam sobre esses dados dentro de uma única classe, controlando o acesso a esses elementos. No TESPy, essa abordagem garante que os dados internos

dos componentes, como parâmetros de desempenho, só possam ser acessados ou modificados por meio de métodos específicos definidos na própria classe. Isso evita alterações inconsistentes, assegurando a integridade e consistência dos modelos.

- Polimorfismo (*Polymorphism*): Graças ao polimorfismo, diferentes componentes podem ser tratados de forma uniforme pelo *solver* do TESPy, desde que implementem os métodos esperados da classe base. Por exemplo, o método *calc_parameters()*, que executa os cálculos principais do componente no pós-processamento, pode ter implementações distintas em um compressor e em um trocador de calor, mas ambos são chamados da mesma forma pelo *solver* durante o processo de solução.

Essa estrutura orientada a objetos permite que os componentes sejam desenvolvidos de maneira modular e extensível. Além disso, facilita a manutenção do código e a inclusão de novos equipamentos ao longo do tempo, sem a necessidade de alterações significativas na lógica do algoritmo do *solver*. Como consequência, a arquitetura do TESPy proporciona grande flexibilidade na construção de diferentes topologias de sistemas térmicos, mantendo a consistência numérica e a escalabilidade dos modelos.

2.3.2 Descrição dos Módulos

O TESPy apresenta uma arquitetura modular robusta e bem estruturada, com múltiplos níveis hierárquicos. A estrutura hierárquica principal é apresentada a seguir:

Trecho de código 2.1 – Organização hierárquica dos módulos do *software* TESPy

```
tespy/
+-- __init__.py
+-- components/
+-- connections/
+-- networks/
+-- tools/
    +-- __init__.py
    +-- analyses.py
    +-- characteristics.py
    +-- data_containers.py
    +-- document_models.py
    +-- fluid_properties/
    +-- global_vars.py
    +-- helpers.py
    +-- logger.py
```

Cada módulo principal possui responsabilidades específicas e bem definidas, formando uma base sólida para a simulação de sistemas térmicos complexos. Os principais módulos e suas funcionalidades são descritos a seguir:

- ***components/*** - Contém todos os componentes físicos do sistema organizados por categoria:
  - *component.py* - Classe base da qual todos os componentes herdam.
  - *basics/* - Componentes básicos como sumidouro (*sink*) e fonte (*source*).

  - *combustion/* - Componentes de combustão como motor.
  - *heat_exchangers/* - Trocadores de calor como condensador e coletor solar.
  - *nodes/* - Nós e separadores para junção e ramificação de fluxos.
  - *piping/* - Tubulações como tubo e válvula.
  - *reactors/* - Reatores como célula de combustível e eletrolisador de água.
  - *turbomachinery/* - Turbomáquinas como compressor, bomba e turbina.

- ***connections/*** - Gerencia conexões entre componentes.
- ***networks/*** - Sistema central que coordena todo o processo de simulação.
- ***tools/*** - Ferramentas auxiliares:
  - *analyses.py* - Análise exergética de modelos termodinâmicos com a classe *ExergyAnalysis* para cálculo de destruição de exergia, eficiência exergética e geração de diagramas.
  - *characteristics.py* - Implementação de curvas e mapas características dos componentes.
  - *data_containers.py* - Containers para diferentes tipos de dados como propriedades de fluidos, componentes e características de componentes.
  - *document_models.py* - Geração de documentação em LaTeX com tabelas e equações.
  - *fluid_properties/*- Submódulo para cálculo das propriedades dos fluidos.
  - *global_vars.py* - Variáveis globais do sistema com tolerâncias de erro e unidades das propriedades.
  - *helpers.py* - Funções auxiliares como conversão de unidades e método de derivadas numéricas.
  - *logger.py* - Sistema de *logging* e *debug*.
  - *optimization.py* - Ferramentas de otimização baseadas em algoritmo genético.

2.3.3 Metodologia de Solução

O TESPy formula o problema de simulação como um sistema de equações algébricas não lineares derivadas das equações de cada componente e conexão. A solução deste sistema é obtida através do método de Newton-Raphson multidimensional.

O vetor de equações na forma residual $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ é construído pela combinação das equações de balanço de massa e energia, equações específicas para cada componente e equações de propriedades termodinâmicas nas conexões. O vetor de variáveis $\mathbf{x}$ é composto pelas propriedades termodinâmicas de cada conexão: vazão mássica ($\dot{m}$), pressão (p) e entalpia específica (h), formando um sistema com n equações e n variáveis.

O método de Newton-Raphson, detalhado na Subseção 3.1.3, requer o cálculo dos valores residuais das equações e das derivadas parciais em relação a todas as variáveis do

sistema para a construção da matriz jacobiana. A matriz é invertida e multiplicada pelo vetor residual para calcular o incremento das variáveis do sistema:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{J}(\mathbf{x}^{(k)})^{-1} \cdot \mathbf{f}(\mathbf{x}^{(k)}) \tag{2.19}$$

Este processo é repetido até que os valores residuais sejam menores que uma tolerância de erro especificada:

$$||\mathbf{f}(\mathbf{x}^{(k)})|| < \epsilon \tag{2.20}$$

O TESPy constrói a matriz Jacobiana através de diferenciação numérica por diferenças finitas, e a inversão da matriz Jacobiana é realizada utilizando a biblioteca NumPy (HARRIS et al., 2020) através da função *np.linalg.inv*. O algoritmo segue um processo iterativo de inicialização, avaliação dos resíduos, construção da Jacobiana, solução do sistema linear, atualização das variáveis e verificação de convergência. O processo continua até que os resíduos sejam reduzidos abaixo da tolerância de 10^{-6} predefinida pelo TESPy.

2.3.3.1 Considerações sobre a ferramenta

O TESPy representa uma ferramenta robusta e bem consolidada no cenário de simulação de sistemas térmicos de código aberto. Sua maturidade é evidenciada pela extensa validação em diversas aplicações práticas, desde sistemas de potência convencionais até sistemas geotérmicos (WITTE; TUSCHY, 2020a; CHEN et al., 2022; FRY; EAGLE-BLUESTONE; WITTE, 2022). Sua arquitetura orientada a objetos oferece vantagens significativas em termos de modularidade e reutilização de código. A biblioteca de componentes disponível abrange uma ampla diversidade de equipamentos térmicos, proporcionando uma base sólida para a modelagem de diversos tipos de sistemas.

A organização modular do framework oferece flexibilidade e extensibilidade, mas o alto grau de integração entre os módulos requer compreensão prévia das interdependências internas para modificações, como a adição de novos componentes ou ajustes e aprimoramentos na estrutura do *solver*. Nesse contexto, o TESPy configura-se como um excelente ponto de partida para o desenvolvimento de modelos próprios e simplificados, especialmente com a documentação abrangente e tutoriais que o *software* disponibiliza.

3 Metodologia

Conforme apresentado no Capítulo 1, este trabalho tem como objetivo o desenvolvimento de um pacote computacional para a modelagem e simulação de sistemas térmicos. A proposta fundamenta-se na criação de um *framework* modular, flexível e extensível, capaz de representar diferentes componentes interligados em ciclos térmicos, como os de refrigeração e de Rankine.

Neste capítulo, são detalhadas as principais etapas do desenvolvimento do pacote, com ênfase na estrutura orientada a objetos adotada, na arquitetura geral do código e nas classes principais que compõem o modelo. Apresentam-se também a formulação matemática utilizada, os métodos numéricos de solução empregados, e os aspectos práticos da implementação em Python.

Além disso, são descritos os componentes implementados, a lógica dos algoritmos e trechos selecionados do código, de forma a ilustrar como os conceitos físicos e matemáticos foram traduzidos em estrutura computacional. A abordagem adotada busca aliar a consistência técnica à clareza na organização do código, de modo a facilitar a manutenção, a extensão e a aplicação do pacote em futuros estudos e configurações de sistemas térmicos.

3.1 Arquitetura Geral do Modelo

O modelo simplificado desenvolvido neste trabalho adota uma abordagem de programação orientada a objetos implementada em linguagem Python, estruturada em torno de três classes principais que trabalham de forma integrada para representar sistemas termodinâmicos. Esta arquitetura foi inspirada no *framework* do pacote TESPy (*Thermal Engineering Systems in Python*), mas com simplificações estratégicas que mantêm a funcionalidade essencial enquanto reduzem a complexidade do código sem comprometer a precisão dos resultados fundamentais, como detalhado na Seção 3.1.2.

A escolha da programação orientada a objetos para simulação termodinâmica se justifica pela natureza modular dos sistemas térmicos reais, onde equipamentos individuais (componentes) são interconectados através de tubulações (conexões) para formar um sistema completo. Esta abordagem permite que cada elemento do sistema seja modelado independentemente, com suas próprias características e equações governantes, facilitando tanto novos desenvolvimentos quanto a manutenção do código.

Assim, de forma geral, a arquitetura foi desenvolvida com base nos seguintes princípios: modularidade, ao tratar cada componente como uma unidade funcional independente; extensibilidade, permitindo a inclusão de novos componentes sem impactar a estrutura existente; reusabilidade, viabilizada pela padronização dos componentes, permitindo seu uso em diferentes configurações e aplicações; e separação de responsabilidades, garantindo que cada classe tenha uma função específica e bem definida no modelo.

3.1.1 Visão Geral das Classes Principais

A arquitetura do modelo é baseada em três classes principais que estabelecem uma hierarquia clara de responsabilidades. De forma geral, elas podem ser descritas da seguinte forma:

Classe *Component*: Atua como classe base para todos os componentes adicionados no sistema térmico, como trocadores de calor, bombas e válvulas. Esta classe estabelece uma interface comum que deve ser implementada por todos os componentes, incluindo métodos para adição de parâmetros, registro de equações governantes e cálculo de parâmetros específicos de cada equipamento.

Classe *Connection*: Responsável por conectar os componentes do sistema, representando os fluxos de fluido entre eles e calculando as propriedades termodinâmicas em cada ponto de conexão do sistema.

Classe *System*: responsável por administrar o funcionamento do sistema como um todo, gerenciando a lista de componentes e conexões, além de coordenar o processo de solução das equações e do pós-processamento dos resultados. Esta classe implementa o método iterativo de Newton-Raphson multidimensional para resolver o sistema de equações não lineares resultante.

A comunicação entre os objetos do sistema é feita por meio de interfaces bem definidas, o que permite isolar funcionalidades e facilita tanto a identificação e correção dos erros quanto a extensão do código. A Figura 7 a seguir apresenta um diagrama simplificado da arquitetura do modelo, evidenciando a relação e o fluxo de informações entre as classes principais.

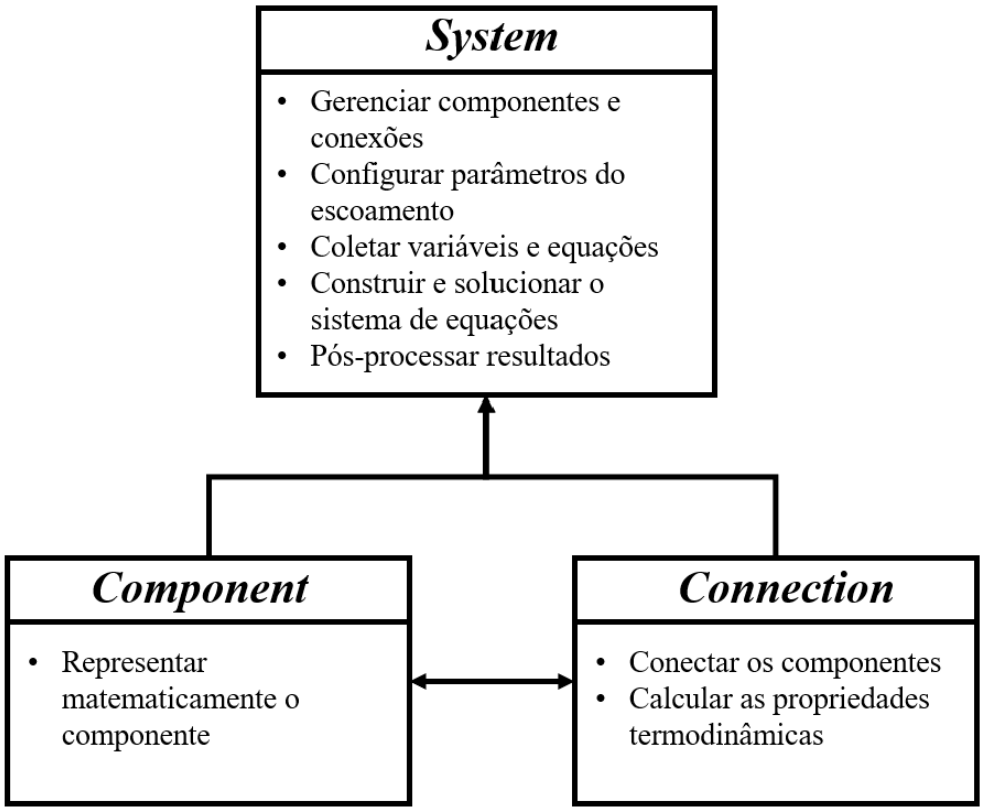

Figura 7 – Diagrama esquemático da estrutura do modelo desenvolvido.

O fluxo de informações no sistema segue um padrão bem estruturado que espelha o comportamento físico dos sistemas térmicos reais. As informações fluem na direção entrada → processamento do componente → saída, onde cada componente recebe propriedades termodinâmicas através de suas conexões de entrada, aplica suas equações governantes específicas, e produz propriedades de saída que são transmitidas para o próximo componente na sequência por meio dos objetos da classe *Connection*.

Essa organização orientada a objetos não apenas facilita o entendimento, a manutenção e a escalabilidade do código, como também permite que o pacote seja adaptado a diferentes aplicações, como a simulação de ciclos de potência, sistemas de refrigeração e novas tecnologias. Além disso, sua estrutura modular, extensível e reutilizável possibilita a incorporação de novos componentes de forma independente, bem como a integração com algoritmos de otimização e outras ferramentas computacionais, ampliando as possibilidades de aplicação em pesquisas e projetos de engenharia térmica.

3.1.2 Comparação com o TESPY

O modelo simplificado desenvolvido neste trabalho representa um refinamento da arquitetura do TESPy, mantendo os princípios fundamentais de programação orientada a objetos enquanto elimina complexidades. Esta simplificação não compromete a precisão dos resultados, mas oferece maior acessibilidade e facilidade de compreensão do código, permitindo que a ferramenta seja estendida sem dificuldades futuramente.

A estrutura original do TESPy compreende diversos arquivos distribuídos em múltiplos módulos, incluindo diversos containers de dados especializados e um sistema avançado de *logging*. Em contraste, o modelo simplificado concentra toda a funcionalidade essencial em apenas quatro arquivos principais: *system.py*, *component.py*, *connection.py* e *utils.py*. Assim como no TESPy, foi mantida a visão organizacional baseada nas três classes principais que estabelecem uma hierarquia clara de responsabilidades e trabalham de forma integrada no sistema termodinâmico. A diferença na complexidade estrutural pode ser evidenciada comparando-se a árvore simplificada do modelo desenvolvido, apresentada no Trecho de código 3.1 com a estrutura ramificada do TESPy original, apresentada no Trecho de código 2.1.

Trecho de código 3.1 – Organização hierárquica dos módulos desenvolvidos neste trabalho

```
thermo_sim/
+-- __init__.py
+-- components/
+-- connections/
+-- system/
+-- utils.py
```

Esta redução foi possível através da eliminação criteriosa de funcionalidades que, embora úteis em contextos específicos, não são tão essenciais para a simulação fundamental de sistemas térmicos. Por exemplo, o sistema complexo de *logging* foi substituído por mensagens diretas, as análises avançadas não críticas foram removidas, e os múltiplos containers de dados foram consolidados em estruturas de dicionários Python mais diretas. O

solver Newton-Raphson, responsável pela solução numérica do sistema, foi simplificado mantendo os componentes essenciais de forma mais clara, incluindo o cálculo de resíduos, matriz Jacobiana e atualização das variáveis. As propriedades dos fluidos são calculadas dentro do próprio módulo das *connections*, por meio da biblioteca CoolProp (BELL et al., 2014), substituindo o módulo de *fluid_properties* do TESPy. Além disso, a compatibilidade conceitual com o TESPy foi preservada, facilitando a migração de componentes e métodos de análise quando necessário e permitindo o aproveitamento da documentação e conhecimento existente.

3.1.3 Método de Solução Newton-Raphson

Sistemas térmicos e termodinâmicos frequentemente envolvem equações não lineares, resultado da combinação de balanços de energia, equações de conservação de massa e equações constitutivas que descrevem as propriedades termodinâmicas dos fluidos. Nesse sentido, a resolução eficiente desses sistemas requer métodos iterativos robustos. Neste trabalho, adota-se o método de Newton-Raphson, cuja formulação e aplicação ao contexto do modelo desenvolvido neste trabalho serão detalhadas nesta seção.

3.1.3.1 Justificativa da Escolha do Método

Diversas ferramentas de simulação na área da engenharia térmica, como o *software Enginnering Equation Solver - EES* (KLEIN, 2003) e o pacote *open-source* TESPy (*Thermal Engineering Systems in Python*) (WITTE; TUSCHY, 2020b), empregam o método de Newton-Raphson ou variantes deste para resolver os sistemas não lineares decorrentes da modelagem de sistemas térmicos. Essa escolha se deve à alta eficiência do método e sua capacidade de apresentar convergência quadrática, o que garante uma aproximação rápida da solução quando as condições iniciais são adequadamente escolhidas (BURDEN; FAIRES; BURDEN, 2016).

3.1.3.2 Formulação Geral

O método de Newton é um procedimento iterativo para a determinação de raízes de funções, ou seja, valores de x para os quais $f(x) = 0$.

Considerando inicialmente uma função com uma única incógnita, a aproximação da solução pode ser entendida sob duas perspectivas: (PETERS; SZEREMETA, 2019)

- Geométrica: por meio da reta tangente à curva da função no ponto atual;
- Analítica: por meio truncamento da série de Taylor;

Na abordagem geométrica, para a função $f(x) = 0$, assumindo que as funções $f(x)$ e $f'(x)$ são contínuas em um intervalo suficientemente próximo da solução desejada, dado um chute inicial x_0, calcula-se o valor da função $f(x_0)$ e sua derivada $f'(x_0)$. A reta tangente $r(x)$ no ponto x_0 é dada por:

$$r(x) = f(x_0) + f'(x_0)(x - x_0). \tag{3.1}$$

Para encontrar a próxima estimativa da raiz, determina-se a interseção dessa reta com o eixo x, ou seja, resolve-se $r(x) = 0$, resultando em:

$$0 = f(x_0) + f'(x_0)(x - x_0), \tag{3.2}$$

que pode ser reorganizada como

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}. \tag{3.3}$$

Esse novo valor x é uma estimativa mais próxima da raiz. Repetindo o processo iterativamente, substitui-se o valor anterior x_k pela nova estimativa x_{k+1}, o que resulta na fórmula recorrência do método de Newton:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}. \tag{3.4}$$

O incremento aplicado a cada iteração, definido como

$$\Delta x = x_{k+1} - x_k = -\frac{f(x_k)}{f'(x_k)}, \tag{3.5}$$

representa a correção aplicada à estimativa atual. Em geral, o método apresenta convergência quadrática, isto é, o erro da aproximação na iteração seguinte é aproximadamente proporcional ao quadrado do erro da iteração atual, o que proporciona uma rápida aproximação à solução correta quando as condições iniciais são adequadas.

A equação 3.4 gera uma sequência de valores x_k, com $k = 0, 1, 2, ...$ que se aproximam da raiz α da função, ou seja, do valor para o qual $f(\alpha) = 0$, conforme ilustrado na Figura 8. Assim, o processo é repetido até que o valor absoluto do incremento $|\Delta x|$ ou da função $|f(x_k)|$ fique abaixo de uma tolerância previamente definida, indicando convergência.

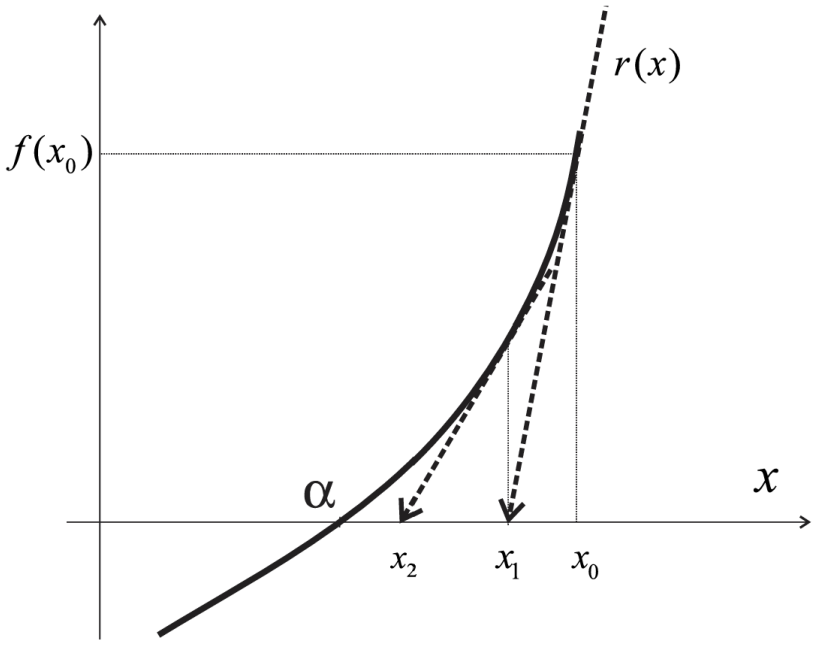

Figura 8 – Interpretação geométrica do método de Newton e sua reta tangente $r(x)$ retirado de Peters e Szeremeta (2019).

Na abordagem analítica do método de Newton, é possível encontrar uma aproximação x_{k+1} para a raiz de $f(x) = 0$ utilizando a expansão de $f(x_{k+1})$ em série de Taylor em torno de um chute inicial x_k. A expansão de Taylor de $f(x_{k+1})$ é dada por:

$$f(x_{k+1}) = f(x_k + \Delta x) = f(x_k) + f'(x_k)\frac{\Delta x^1}{1!} + f''(x_k)\frac{\Delta x^2}{2!} + ... \tag{3.6}$$

Para encontrar a raiz da função, a função $f(x)$ pode ser representada apenas pelos dois primeiros termos da série de Taylor, ou seja, desconsidera-se os termos de ordem superior a 2, resultando na seguinte aproximação de 1ª ordem equivalente à reta $r(x)$ definida pela equação 3.1 :

$$f(x_k + \Delta x) \approx f(x_k) + f'(x_k)\Delta x \tag{3.7}$$

Definindo $x_{k+1} = x_k + \Delta x$ como a próxima estimativa da raiz, impõe-se $f(x_k + \Delta x) = 0$, obtendo-se

$$0 = f(x_k) + f'(x_k)\Delta x \tag{3.8}$$

que pode ser rearranjada para a equação de recorrência encontrada em 3.4:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}. \tag{3.9}$$

Dessa forma, o método de Newton pode ser interpretado como a substituição da função não linear original por sua aproximação linear de primeira ordem (reta tangente) em cada ponto x_k, usando a raiz dessa reta como a nova estimativa x_{k+1} da raiz da função. Essa substituição corresponde ao truncamento da série de Taylor após o termo linear.

De forma análoga ao caso de uma única incógnita $f(x) = 0$, é possível aplicar a mesma estratégia de linearização do método de Newton para resolver relações não lineares entre várias incógnitas, as quais geram um sistema de equações não lineares.

Seja um sistema de n equações não lineares em n incógnitas representado por:

$$\mathbf{f(x)} = \begin{bmatrix} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) \end{bmatrix} = \mathbf{0} \tag{3.10}$$

em que $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$ é o vetor de incógnitas e $\mathbf{f} = [f_1, f_2, \ldots, f_n]^T$ é o vetor de equações do sistema.

Para que o sistema seja determinado e tenha solução única, o número de equações deve ser igual ao número de incógnitas. No contexto de sistemas térmicos, as incógnitas típicas incluem pressões, temperaturas, vazões mássicas e propriedades específicas em diversos pontos do ciclo.

3.1.3.3 Definição do Vetor de Resíduos

A função $\mathbf{f(x)}$ avaliada em um determinado vetor de incógnitas $\mathbf{x}$ fornece o vetor de resíduos do sistema, o qual representa o desvio atual em relação à solução exata do problema.

Nesse sentido, cada componente $f_i(\mathbf{x})$ indica o quanto a i-ésima equação do sistema ainda não está satisfeita para o valor atual das variáveis. O objetivo do processo iterativo é, portanto, minimizar esse vetor de resíduos até que ele se aproxime suficientemente de zero, segundo um critério de convergência previamente estabelecido.

$$f_i(\mathbf{x}) = 0, \quad i = 1, 2, \ldots, n \tag{3.11}$$

Por exemplo:

- Balanço de massa: $f_i = \dot{m}_{\text{entrada}} - \dot{m}_{\text{saída}}$
- Balanço de energia: $f_i = \dot{m}(h_{\text{saída}} - h_{\text{entrada}}) - \dot{Q} + \dot{W}$
- Equações de estado: $f_i = p - p(T, h)$

Durante as iterações, valores diferentes de zero indicam o desvio da solução exata. Quando o sistema está perfeitamente resolvido, todos os resíduos são iguais a 0, $f_i = 0$.

3.1.3.4 Construção da Matriz Jacobiana

Da mesma forma que no caso unidimensional a derivada $f'(x)$ era utilizada para a solução do sistema, no caso de múltiplas incógnitas a matriz Jacobiana $\mathbf{J}$ é usada para agrupar todas as derivadas parciais de primeira ordem de cada equação f_i em relação a cada variável x_j, aplicando o mesmo princípio de truncamento da série de Taylor:

$$J_{ij} = \frac{\partial f_i}{\partial x_j} \tag{3.12}$$

A estrutura geral da matriz Jacobiana para um sistema com n equações e n incógnitas é dada por:

$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \dfrac{\partial f_1}{\partial x_2} & \cdots & \dfrac{\partial f_1}{\partial x_n} \\ \dfrac{\partial f_2}{\partial x_1} & \dfrac{\partial f_2}{\partial x_2} & \cdots & \dfrac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial f_n}{\partial x_1} & \dfrac{\partial f_n}{\partial x_2} & \cdots & \dfrac{\partial f_n}{\partial x_n} \end{bmatrix} \tag{3.13}$$

em que cada linha corresponde a uma equação f_i e a relação desta equação a todas as variáveis, enquanto cada coluna está associada a uma variável x_j do sistema e representa o impacto desta variável em todas as equações do sistema. Assim, o elemento J_{ij} representa a derivada parcial da equação f_i em relação à variável x_j. A matriz Jacobiana pode ser, portanto, usada para interpretar essa sensibilidade local do sistema da seguinte forma:

- Se $J_{ij} > 0$: um aumento em x_j causa um aumento no resíduo f_i;
- Se $J_{ij} < 0$: um aumento em x_j causa uma diminuição no resíduo f_i;
- Se $J_{ij} = 0$: a variável x_j não influencia diretamente a equação f_i;

Assim, expandindo a função vetorial **f(x)** em série de Taylor em torno de um ponto $x^{(k)}$, de forma análoga à equação 3.7 permite aproximar a função por:

$$\mathbf{f}(\mathbf{x^{(k+1)}}) \approx \mathbf{f}(\mathbf{x^{(k)}}) + \mathbf{J}(\mathbf{x^{(k)}})(\mathbf{x^{(k+1)}} - \mathbf{x^{(k)}}), \tag{3.14}$$

onde $\mathbf{J}(\mathbf{x}^{(k)})$ é a matriz Jacobiana da função **f** avaliada em $\mathbf{x}^{(k)}$.

A aplicação do método de Newton consiste em impor que a aproximação linear seja nula no próximo passo iterativo, isto é, $\mathbf{f}(\mathbf{x^{(k+1)}}) \approx 0$. Substituindo na equação (3.14), obtém-se:

$$\mathbf{J}(\mathbf{x^{(k)}})(\mathbf{x^{(k+1)}} - \mathbf{x^{(k)}}) = -\mathbf{f}(\mathbf{x^{(k)}}). \tag{3.15}$$

Definindo o incremento $\mathbf{\Delta x^{(k)}} = \mathbf{x^{(k+1)}} - \mathbf{x^{(k)}}$, chega-se ao sistema linear:

$$\mathbf{J}(\mathbf{x^{(k)}})\mathbf{\Delta x^{(k)}} = -\mathbf{f}(\mathbf{x^{(k)}}), \tag{3.16}$$

onde $\mathbf{x}^{(k)}$ é o vetor solução na iteração k, e $\mathbf{f}(\mathbf{x})$ é o vetor residual contendo todos os resíduos das equações, cuja solução fornece a nova estimativa para a raiz do sistema:

$$\mathbf{x^{(k+1)}} = \mathbf{x^{(k)}} + \mathbf{\Delta x^{(k)}}. \tag{3.17}$$

Essa formulação generaliza de forma direta o método de Newton-Raphson para o caso multidimensional, mantendo a essência da aproximação local por uma função linear. De forma equivalente, o método de Newton-Raphson também pode ser calculado a partir da inversão da matriz $\mathbf{J}(x)$ em cada passo:

$$\mathbf{x^{(k+1)}} = \mathbf{x^{(k)}} - \mathbf{J^{-1}}(\mathbf{x^{(k)}})\mathbf{f}(\mathbf{x^{(k)}}) \tag{3.18}$$

o que resulta da multiplicação da equação (3.16) pela inversa da matriz Jacobiana. No entanto, essa forma é evitada em aplicações computacionais, uma vez que a inversão direta de matrizes é uma operação menos eficiente computacionalmente.

Por essa razão, prefere-se resolver o sistema linear (3.16) em dois passos: obter $\mathbf{\Delta x^{(k)}}$ em um primeiro momento, para em seguida atualizar a solução com base em $\mathbf{x^{(k+1)} = x^{(k)} + \Delta x^{(k)}}$.

No contexto da modelagem de sistemas térmicos, cada equação proveniente de um componente ou conexão como, por exemplo, o balanço de energia em um trocador de calor, contribui para a definição do sistema de equações não lineares. Essas equações devem ter suas derivadas calculadas com relação às variáveis do sistema, como pressão, entalpia específica e vazão mássica, de modo a possibilitar a construção da matriz Jacobiana. Tais derivadas podem ser obtidas de forma analítica ou numérica.

3.1.3.5 Derivadas Analíticas e Numéricas

Sempre que possível, a derivação analítica da matriz Jacobiana é preferível, pois garante maior precisão e eficiência. No entanto, quando as expressões forem muito complexas ou resultantes de modelos complexos ou como tabelas termodinâmicas, pode-se utilizar aproximações numéricas baseadas em diferenças finitas.

No presente trabalho, utiliza-se a fórmula da derivada simétrica para estimar numericamente as derivadas parciais:

$$J_{ij} \approx \frac{f_i(x_j + \delta) - f_i(x_j - \delta)}{2\delta}, \tag{3.19}$$

onde δ representa um pequeno incremento aplicado à variável x_j, mantendo-se todas as demais constantes.

3.1.3.6 Processo Iterativo

O algoritmo de Newton-Raphson pode ser resumido em:

1. Estimativa inicial: Fornece-se, para a primeira iteração, um vetor inicial $\mathbf{x}^{(0)}$ contendo valores iniciais para todas as variáveis do sistema;
2. Cálculo dos resíduos: Avalia-se o vetor de resíduos $\mathbf{f}(\mathbf{x}^{(k)})$, o qual indica quanto cada equação está distante da solução na iteração k atual;
3. Montagem da matriz Jacobiana: Calculam-se todas as derivadas parciais $\frac{\partial f_i}{\partial x_j}$ na estimativa atual;
4. Solução do sistema linear: Resolve-se $\mathbf{J}(\mathbf{x}^{(k)}) \cdot \Delta\mathbf{x}^{(k)} = -\mathbf{f}(\mathbf{x}^{(k)})$ para encontrar o incremento;
5. Atualização: Aplica-se o incremento $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}$ para atualização das variáveis;
6. Verificação de convergência: Repete-se até que os resíduos estejam de acordo com o critério de convergência estabelecido.

3.1.3.7 Critério de Convergência

A convergência é atingida quando a norma dos resíduos atinge um valor inferior a uma tolerância pré-definida:

$$||\mathbf{f}(\mathbf{x}^{(k)})|| < \epsilon \tag{3.20}$$

onde $\mathbf{f}(\mathbf{x}^{(k)})$ representa os resíduos das equações no passo iterativo k. Este critério garante que as equações do modelo (como balanço de energia, relações de eficiência, etc.) estejam próximas de zero dentro de uma tolerância aceitável. Valores usuais para ϵ variam entre 10^{-6} e 10^{-10}, dependendo da precisão desejada. Múltiplos critérios podem ser aplicados simultaneamente:

$$||\mathbf{f}(\mathbf{x}^{(k)})|| < \epsilon_{res} \tag{3.21}$$

$$||\Delta\mathbf{x}^{(k)}|| < \epsilon_{var} \tag{3.22}$$

$$\frac{||\Delta\mathbf{x}^{(k)}||}{||\mathbf{x}^{(k)}||} < \epsilon_{rel} \tag{3.23}$$

onde ϵ_{res}, ϵ_{var} e ϵ_{rel} são as tolerâncias para os resíduos, para a variação absoluta e para a variação relativa, respectivamente. ϵ_{var} controla a mudança absoluta no vetor de incógnitas entre duas iterações consecutivas. Já o critério, ϵ_{rel}, considera a variação das variáveis em

relação ao seu valor atual, normalizando a diferença para evitar problemas em variáveis com magnitudes muito diferentes.

3.1.3.8 Sub-relaxação

Para melhorar a estabilidade, especialmente quando o processo iterativo converge oscilando, ou até mesmo quando diverge, pode-se aplicar um fator de redução na atualização do valor novo, ou seja, não aplicar 100% do incremento calculado:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha \cdot \Delta\mathbf{x}^{(k)}, \quad 0 < \alpha \leq 1 \tag{3.24}$$

Desse modo, valores menores de α suavizam as correções aplicadas, podendo evitar oscilações ou divergência.

3.1.3.9 Step Halving

O *step halving* é uma técnica de estabilização que reduz progressivamente o tamanho do passo quando a solução não melhora ou diverge. O *software* EES utiliza essa técnica, reduzindo o passo em até 20 vezes (KLEIN, 2003). O algoritmo funciona da seguinte forma:

1. Calcula-se a solução tentativa: $\mathbf{x}_{tent} = \mathbf{x}^{(k)} + \alpha \cdot \Delta\mathbf{x}^{(k)}$
2. Avalia-se os novos resíduos: $||\mathbf{f}(\mathbf{x}_{tent})||$
3. Se $||\mathbf{f}(\mathbf{x}_{tent})|| \geq ||\mathbf{f}(\mathbf{x}^{(k)})||$, reduz-se o passo: $\alpha \rightarrow \alpha/2$
4. Repete-se até que $||\mathbf{f}(\mathbf{x}_{tent})|| < ||\mathbf{f}(\mathbf{x}^{(k)})||$ ou $\alpha < \alpha_{min}$ definido

Esta técnica é particularmente útil quando a estimativa inicial está longe da solução.

3.2 *Classe Component*

A classe *Component* representa a classe base para todos os componentes do sistema termodinâmico, fornecendo a estrutura fundamental para modelagem de equipamentos como bombas, turbinas, trocadores de calor e outros dispositivos. Cada componente herda desta classe base os métodos e atributos necessários para a integração do sistema e gerenciamento de parâmetros, como a definição dos pontos de entrada (*inlet*) e saída (*outlet*), a gestão de conexões por meio da integração com a classe *Connection* e a estrutura necessária para o método de solução, gerenciado pela classe *System*.

A partir dessa estrutura básica, cada componente implementa suas próprias características específicas, como as equações de conservação de massa e energia que descrevem seu comportamento termodinâmico, além de parâmetros particulares ao tipo do equipamento. Desse modo, essa abordagem orientada a objetos permite incluir novos tipos de componentes sem a necessidade de alterar a estrutura geral do modelo, promovendo assim sua flexibilidade e escalabilidade.

3.2.1 *Arquitetura e Inicialização*

Uma instância da classe *Component* é inicializada com um nome e cria dicionários para gerenciar suas características. Por padrão, a classe inclui uma única entrada (*inlet*) e uma única saída (*outlet*), que representam portas de fluido que podem ser conectadas a objetos do tipo *Connection*. Subclasses que necessitam de portas adicionais podem personalizar esses dicionários para definir múltiplas entradas e/ou saídas.

Trecho de código 3.2 – Inicialização da Classe *Component*

```
def __init__(self, name, **kwargs):
    self.name = name
    self.parameters = {}      # Parâmetros operacionais do componente
    self.constraints = {}     # Equações obrigatórias
    self.inlets = {"in": None}    # Padrão de única entrada
    self.outlets = {"out": None}  # Padrão de única saída
```

Durante a inicialização, cada componente recebe um nome único no sistema e dicionários para gerenciar seus parâmetros operacionais (como eficiências, potências) em *self.parameters*, suas equações obrigatórias que devem ser adicionadas ao sistema em *self.constraints*, e suas conexões físicas com outros equipamentos por meio da definição dos *inlets* e *outlets*.

3.2.2 *Gerenciamento de Parâmetros*

Após a inicialização básica, a classe *Component* fornece mecanismos para que cada componente defina seus próprios parâmetros e equações obrigatórias. Esses métodos de gerenciamento de parâmetros e equações são uma das funcionalidades centrais desta classe, uma vez que permitem a representação de diferentes tipos de equipamentos de forma modular e adaptável. Cada parâmetro é armazenado com dados que definem seu estado no sistema de equações, incluindo se é uma variável, se possui limites de operação e se está associado a funções específicas de cálculo e derivada.

O método *add_parameter*() permite adicionar parâmetros com características específicas para o sistema de equações. Um trecho dessa função é ilustrado no trecho de código a seguir:

Trecho de código 3.3 – Trecho do método *add_parameter()* para adicionar parâmetros no componente

```
def add_parameter(self, param_name, func=None, deriv=None, value=None, min_val=None,
    max_val=None, num_eq = 0, is_variable = False):

    parameter = {
    "value": float,
    "is_unknown": bool,
    "is_variable": bool,
    "is_added": bool,
    "is_solved": bool,
    "func": equation_function,
    "deriv": derivative_function,
    "min_val": float,
    "max_val": float,
```

```
"num_eq": int
}
```

Cada parâmetro é armazenado como um dicionário contendo:

- *value*: Valor numérico do parâmetro do tipo *float* ou *None*, caso seja um parâmetro desconhecido ou uma variável;
- *is_unknown*: Um dado lógico (*bool*) que indica se o parâmetro não tem valor conhecido;
- *is_variable*: Define se o parâmetro é tratado como uma variável do sistema, ou seja, se será incluído no vetor de incógnitas do método de Newton e terá sua posição em uma coluna da matriz Jacobiana;
- *is_added*: Confirma se o parâmetro foi explicitamente fornecido pelo usuário, sendo essencial para determinar quais equações devem ser adicionadas ao sistema;
- *is_solved*: *Status* que indica se o parâmetro apresenta um valor numérico, ou seja, se foi adicionado pelo usuário, calculado por meio de outras grandezas ou se foi resolvido pelo *solver*;
- $func/deriv$: Funções associadas ao parâmetro para cálculo de resíduos (*func*) e derivadas (*deriv*), usadas na construção do sistema de equações (podem ser *None*, caso o parâmetro não contribua diretamente com equações, mas seja usado apenas para cálculos auxiliares ou pós-processamento.);
- *min_val*/*max_val*: Valores mínimos e máximos permitidos, que garantem que o parâmetro respeite limites físicos, prevenindo valores não físicos como eficiências negativas ou superiores a 100%;
- *num_eq*: Indica o número de equações que o parâmetro contribui para o sistema se for adicionado pelo usuário, sendo igual a 0 ou 1.

Para facilitar o uso e garantir consistência em diferentes partes do código, a classe *Component* implementa também funções auxiliares como *is_unknown*(), *is_variable*(), *is_added*() e *is_solved*(). Essas funções encapsulam essa lógica de verificação do estado dos parâmetros e são utilizadas tanto na montagem do sistema de equações quanto no pós-processamento dos resultados.

Além de armazenar os dados, esse método implementa validação automática de limites físicos no momento em que o parâmetro é definido. Isso ajuda a evitar erros comuns de modelagem, como inserir parâmetros fora de faixas operacionais realistas. Caso o valor atribuído esteja fora dos limites definidos por *min_val* e *max_val*, o método *add_parameter*() interrompe a execução do código e lança um erro do tipo *ValueError*, informando detalhadamente qual foi o parâmetro inválido e os limites que foram violados, de modo a evitar que inconsistências sejam propagadas para a etapa de solução.

É importante destacar a distinção entre os conceitos de incógnita (*unknown*) e variável (*variable)*. Quando um parâmetro do componente é marcado como *is_variable* = *True*, ele

automaticamente se torna uma incógnita do sistema global de equações e terá sua contribuição explícita na matriz Jacobiana. O *solver* ajustará seu valor durante as iterações até que a solução atinja a convergência. Já quando um parâmetro é desconhecido, *is_unknown* = *True*, significa que seu valor ainda não é conhecido. Entretanto, este parâmetro pode ou não ser uma variável, uma vez que ele pode ser calculado diretamente a partir de outros parâmetros conhecidos.

Nesse sentido, essa lógica interna permite que o modelo seja flexível na representação de diferentes tipos de componentes e condições de contorno. Por exemplo, em uma turbina, a eficiência isentrópica (η_s) pode ser conhecida e fornecida como dado de entrada (*is_added* = *True*), contribuindo com uma equação ($func = self.eta\_s\_equation$) para o sistema de variáveis, ou desconhecida (*is_unknown* = *True*) e resolvida com base em outros parâmetros. Esse comportamento pode ser observado no Trecho de código 3.4, que mostra a aplicação do método *add_parameter*() para uma turbina.

Trecho de código 3.4 – Exemplo da aplicação do método *add_parameter()* para uma turbina

```
def __init__(self, name, **kwargs):
    self.add_parameter("eta_s", value=kwargs.get('eta_s', None), min_val=0, max_val=1,
        func=self.eta_s_equation, deriv=self.eta_s_deriv, num_eq=1)
```

Para garantir consistência e facilitar a manutenção do código, adotou-se o padrão de nomeação em que a função associada à equação do parâmetro deve ser nomeada como o próprio parâmetro seguido do sufixo *_equation* (por exemplo, *eta_s_equation*), enquanto a função que calcula sua derivada deve ter o sufixo *_deriv* (por exemplo, *eta_s_deriv*). Essa padronização permite uma associação clara e automática entre parâmetros e suas funções correspondentes, simplificando a estrutura do componente e facilitando o papel de gerenciamento das equações feito pela classe *System*.

Cada componente também pode adicionar equações obrigatórias ao sistema por meio do método *add_constraint*(), observado no Trecho de Código 3.5, que registra uma função residual e suas derivadas correspondentes para uso no *solver* baseado no método de Newton.

Trecho de código 3.5 – Trecho do método *add_constraint()* para registrar equações obrigatórias ao sistema

```
def add_constraint(self, constraint_name, func, deriv=None, num_eq=1, constant_deriv=True
    ):
    self.constraints[constraint_name] = {
        'func': func,
        'deriv': deriv,
        'num_eq': num_eq,
        'constant_deriv': constant_deriv,
    }
```

Enquanto o método *add_parameter*() adiciona equações ao sistema de forma condicional, ou seja, apenas se os parâmetros fornecidos explicitamente pelo usuário (*is_added* = *True*) estiverem associados a funções de equações e derivadas, o método *add_constraint*() é utilizado para registrar equações obrigatórias, que representam comportamentos físicos fundamentais

do componente. Essas equações são sempre adicionadas ao sistema global, independentemente da forma como os parâmetros são definidos.

Um exemplo típico do uso de *add_constraint*() pode ser observado na modelagem de uma válvula de expansão, mostrado no Trecho de código 3.6, em que o processo é assumido como isentálpico. Nesse caso, a igualdade entre as entalpias de entrada e saída não depende de o usuário ter fornecido algum valor, ela é sempre válida fisicamente e, portanto, deve ser sempre imposta ao sistema de equações.

Trecho de código 3.6 – Exemplo da aplicação do método *add_constraint()* para uma válvula isentálpica

```
self.add_constraint(
    "enthalpy_equality",
    func=self.enthalpy_equality_equation,
    deriv=self.enthalpy_equality_deriv,
    num_eq=1
)
```

Assim, enquanto *add_parameter*() permite flexibilidade para ativar ou desativar equações com base na forma como o usuário fornece os dados, o método *add_constraint*() é usado para garantir que certas relações físicas sejam sempre respeitadas.

3.2.3 *Gerenciamento de Conexões*

A classe *Component* implementa alguns métodos para gerenciar conexões entre componentes, permitindo a criação de diferentes sistemas térmicos. Esse gerenciamento é fundamental para estabelecer a topologia do sistema e permitir o compartilhamento de propriedades do fluido entre componentes adjacentes.

A classe *Component* gerencia essas conexões por meio de dicionários *self.inlets* e *self.outlets*, que associam nomes de portas a objetos do tipo *Connection*, representando as ligações físicas com outros componentes do sistema termodinâmico.

Para registrar essas conexões, são utilizados os métodos *register_inlet_connection()* e *register_outlet_connection()*, que vinculam uma *Connection* a uma porta específica do componente, definida por seu nome. No caso padrão com $self.inlets = \{"in" : None\}$ e $self.outlets = \{"out" : None\}$, os termos *in* e *out* são o *inlet_name* e *outlet_name*, respectivamente. Além disso, esses métodos incluem validações que garantem que a porta informada esteja previamente definida no componente, lançando um erro do tipo *ValueError* caso a porta informada não esteja definida, garantindo a integridade da topologia do sistema. Esses métodos são ilustrados no Trecho de código 3.7.

Trecho de código 3.7 – Métodos de registro de conexão de entrada e saída

```
def register_inlet_connection(self, connection, inlet_name):
    if inlet_name not in self.inlets:
        raise ValueError(f"Inlet '{inlet_name}' not defined in component '{self.name}'.")
    self.inlets[inlet_name] = connection

def register_outlet_connection(self, connection, outlet_name):
    if outlet_name not in self.outlets:
```

```
            raise ValueError(f"Outlet '{outlet_name}' not defined in component '{self.name}'.
                ")
        self.outlets[outlet_name] = connection
```

Uma vez registradas, as conexões passam a estar acessíveis para o componente ao qual foram associadas. Isso permite que o próprio componente obtenha os parâmetros termodinâmicos fornecidos por essas conexões, como pressão, temperatura, título, entalpia ou vazão mássica, os quais são essenciais para os cálculos realizados por cada componente. Para isso, é possível utilizar os métodos *get_inlet_parameters*() e *get_outlet_parameters*(), que retornam os dicionários de parâmetros do objeto *Connection* associado à respectiva porta do componente.

Trecho de código 3.8 – Métodos para acesso aos parâmetros de entrada e saída

```
def get_inlet_parameters(self, inlet_name):
    connection = self.inlets.get(inlet_name)
    return connection.parameters if connection else {}

def get_outlet_parameters(self, outlet_name):
    connection = self.outlets.get(outlet_name)
    return connection.parameters if connection else {}
```

Também foram implementados outros métodos como o *get_inlet_connections*() e o método *get_outlet_connections*() para retornar as conexões associadas a cada porta como listas, além do método *print_connections*(), que exibe todas as conexões registradas em um componente, identificando os nomes das portas e os rótulos (*labels*) das conexões. Essa funcionalidade é especialmente útil para verificar se os componentes estão corretamente conectados entre si e para identificar rapidamente falhas de ligação ou configurações incompletas no sistema.

Por meio da implementação dessas funções, esse modelo de gerenciamento de conexões garante que cada componente possa acessar dinamicamente as propriedades termodinâmicas do fluido nas suas conexões de entrada e saída, permitindo que as equações internas de cada equipamento sejam formuladas com base nessas propriedades. Destaca-se também a importância da verificação da existência das portas e lançamento de erro, uma vez que torna o modelo mais robusto e extensível, especialmente para lidar com componentes e sistemas térmicos de maior complexidade.

3.2.4 Métodos de Acesso e Atualização de Parâmetros

A classe *Component* também disponibiliza métodos para acessar e atualizar os valores dos seus parâmetros. O método *get_parameter()* retorna o valor atual de um parâmetro específico, facilitando a leitura dos dados armazenados no componente. Já o método *set_parameter()* permite atualizar o valor de um parâmetro existente, realizando uma validação automática para garantir que o novo valor esteja dentro dos limites definidos previamente (mínimo e máximo). Caso o valor fornecido esteja fora desses limites, um erro é gerado para evitar inconsistências no modelo. Esses métodos, ilustrados a seguir, são fundamentais para

assegurar a integridade dos dados e possibilitar a modificação controlada dos parâmetros durante a simulação.

Trecho de código 3.9 – Métodos para acessar e atualizar parâmetros com validação

```
def get_parameter(self, param_name):
    """Retorna o valor do parâmetro especificado."""
    return self.parameters.get(param_name, {}).get("value")

def set_parameter(self, param_name, value):
    """Atualiza o valor de um parâmetro existente com validação numérica."""
    if param_name in self.parameters:
        param = self.parameters[param_name]
        min_val = param["min_val"]
        max_val = param["max_val"]

        # Valida o valor de acordo com os seus limites
        if min_val is not None and value < min_val:
            raise ValueError(f"Value of {param_name} ({value}) is below the minimum
                allowed ({min_val}).")
        if max_val is not None and value > max_val:
            raise ValueError(f"Value of {param_name} ({value}) is above the maximum
                allowed ({max_val}).")

        # Atualiza o valor
        param["value"] = value
    else:
        raise KeyError(f"Parameter '{param_name}' not found in the component.")
```

3.2.5 *Integração com o Sistema de Equações*

Cada componente é integrado ao sistema de equações global por meio de uma estrutura que mapeia suas contribuições ao *solver*: identifica quais parâmetros são variáveis a serem resolvidas, quais equações são obrigatórias (*constraints*) e quais equações estão associadas aos parâmetros adicionados pelo usuário. Essa integração é viabilizada por métodos auxiliares que identificam o estado de cada parâmetro e localizam a posição de cada elemento dentro da matriz Jacobiana e do vetor de resíduos do sistema.

Os métodos *is_unknown*(), *is_variable*(), *is_added*() e *is_solved*(), como apresentados anteriormente, permitem ao componente avaliar dinamicamente se um determinado parâmetro tem seu valor numérico desconhecido, se ele deve ser incluído como variável na solução ou se já foi fornecido pelo usuário, resolvido numericamente ou calculado a partir de outros parâmetros. Esses métodos são fundamentais tanto durante a construção da matriz de equações quanto no processo iterativo de solução.

Além disso, os métodos *get_J_col*() e *get_eq_index*() são utilizados para mapear parâmetros e equações às suas respectivas posições no sistema global:

- *get_J_col*() : retorna a posição da coluna da matriz Jacobiana associada a uma variável do componente. Essa coluna corresponde à derivada da equação com relação à variável em questão. O nome da variável no sistema é construído no formato "*{nome_do_componente}*

_{nome_do_parâmetro}", garantindo a padronização que facilita o gerenciamento da solução realizada pela classe *System*;

- *get_eq_index*(): retorna o índice da equação associada ao parâmetro, permitindo acessar diretamente a linha correspondente no vetor de resíduos ou na matriz Jacobiana. O nome da equação no gerenciamento realizado pela classe *System* segue o formato "*{nome_do_componente}_{nome_do_parâmetro}_eq*".

Os métodos responsáveis por acessar essas informações de forma padronizada estão implementados conforme o trecho de código a seguir:

Trecho de código 3.10 – Implementações dos métodos *get_J_col*() e *get_eq_index*()

```
def get_J_col(self, param_name, system):
"""Obter o índice da coluna na matriz Jacobiana para um parâmetro especificado."""
    var_name = f"{self.name}_{param_name}"
    if var_name in system.var_index:
        return system.var_index[var_name]
    else:
        raise ValueError(f"Parameter {param_name} in component {self.name} is not a
            variable.")

def get_eq_index(self, param_name, system):
"""Obter o índice da equação associado ao parâmetro especificado."""
    eq_name = f"{self.name}_{param_name}_eq"
    return system.eq_index.get(eq_name)
```

Nessa implementação, o argumento *system* refere-se à instância da classe *System*, que centraliza o gerenciamento do sistema de equações. Essa classe mantém os dicionários:

- *system.var_index*: mapeia os nomes das variáveis (no formato padronizado "*componente_parametro*") para os índices das colunas da matriz Jacobiana;
- *system.eq_index*: mapeia os nomes das equações (no formato "*componente_parametro_eq*") para os índices das linhas da Jacobiana e do vetor de resíduos.

Dessa forma, o componente é capaz de interagir com o sistema de equações de maneira modular e padronizada, contribuindo com equações, variáveis e derivadas de forma consistente e integrada.

A seguir, apresenta-se um exemplo de uso dos métodos *get_J_col*() e *get_eq_index*() dentro de uma função de derivada. O método *get_eq_index*() retorna o índice da linha da equação no sistema, enquanto *get_J_col*() retorna o índice da coluna da variável na matriz Jacobiana, permitindo que a derivada parcial seja atribuída à posição correta:

Trecho de código 3.11 – Exemplo de aplicação dos métodos *get_J_col*() e *get_eq_index*()

```
eq_index = system.eq_index.get(f"{self.name}_eta_s_eq")

if self.is_variable("eta_s", system):
    J_col = self.get_J_col("eta_s", system)
```

```
    deriv = numeric_deriv(residual_func, "eta_s", self)
    system.jacobian[eq_index, J_col] = deriv
```

A classe *Component* também fornece o método *set_parameter_solved()* que marca um parâmetro como resolvido com seu valor final, sendo essencial para a integração com o *solver*. Esse método atualiza o valor do parâmetro e altera o seu *status* para um parâmetro conhecido e resolvido.

Trecho de código 3.12 – Método para marcar um parâmetro como resolvido com o valor final

```
def set_parameter_solved(self, param_name, value):
    """
    Sets a parameter as solved with the given value.
    """

    if param_name in self.parameters:
        self.parameters[param_name]["value"] = value
        self.parameters[param_name]["is_unknown"] = False
        self.parameters[param_name]["is_solved"] = True
```

3.2.6 *Cálculo das Derivadas Parciais*

Para resolver o sistema não linear de equações que descreve o comportamento do sistema térmico, processo descrito na seção 3.1.3, é necessário construir a matriz Jacobiana, composta pelas derivadas parciais das equações de resíduo em relação às variáveis do sistema. Essas derivadas podem ser computadas de duas formas principais: analiticamente, a partir da formulação matemática explícita das equações, ou numericamente, por meio do método das diferenças finitas. Nesse sentido, cada componente deve fornecer as derivadas analíticas ou numéricas de suas equações residuais em relação às variáveis do sistema.

Neste trabalho, sempre que possível as derivadas são implementadas analiticamente, caso contrário, utiliza-se a diferenciação numérica por meio do método *numeric_deriv*, definido no módulo *utils.py*. Essa função perturba ligeiramente o valor de uma variável, com um delta padrão de $1e-4$, e calcula a derivada pelo método da derivada simétrica. Esse método é apresentado no Trecho do código 3.13.

Trecho de código 3.13 – Função para cálculo da derivada numérica

```
def numeric_deriv(func, variable, obj, delta=1e-4):
    value = obj.get_parameter(variable)

    if variable == "pressure" or variable == "enthalpy":
        delta = 1e-1

    obj.parameters[variable]["value"] = value + delta
    f_plus = func()

    obj.parameters[variable]["value"] = value - delta
    f_minus = func()

    obj.parameters[variable]["value"] = value

    return (f_plus - f_minus) / (2 * delta)
```

Para exemplificar o cálculo das derivadas parciais utilizadas na construção da matriz Jacobiana, apresenta-se a seguir o caso de um componente típico de um sistema térmico: a turbina que foi implementada no modelo desenvolvido. O exemplo demonstra a estrutura geral da função de resíduo e a implementação do cálculo das derivadas parciais, incluindo a identificação da equação no sistema e a obtenção das colunas correspondentes no Jacobiano. Assim, considere uma turbina com eficiência isentrópica η_s, cuja equação de resíduo é definida como:

$$f_{\eta_s} = -(h_{out} - h_{in}) + (h_{2s} - h_{in}) \cdot \eta_s \tag{3.25}$$

onde h_{2s} é a entalpia de saída considerando um processo isentrópico, obtida com a função *isentropic_enthalpy* presente no módulo *utils.py* e apresentada a seguir:

Trecho de código 3.14 – Função para cálculo da entalpia de saída considerando um processo isentrópico

```
def isentropic_enthalpy(p_1, h_1, p_2, fluid_data):
    s_1 = CP.PropsSI("S", "H", h_1, "P", p_1, fluid_data)
    return CP.PropsSI("H", "P", p_2, "S", s_1, fluid_data)
```

A função (*func*) que calcula o resíduo da equação de eficiência isentrópica é, seguindo os padrões de nomeação apresentados, *eta_s_equation*. As variáveis *i* e *o* representam, respectivamente, as conexões associadas às portas de entrada (*"in"*) e saída (*"out"*) da turbina, obtidas pelos métodos *self.inlets.get("in")* e *self.outlets.get("out")*. A partir dessas conexões, os parâmetros termodinâmicos são acessados com o método *get_parameter*, como por exemplo *i.get_parameter("enthalpy")* para obter a entalpia na entrada da turbina. A função residual da eficiência isentrópica pode ser observada no Trecho de código 3.15.

Trecho de código 3.15 – Equação de resíduo da eficiência isentrópica para uma turbina

```
def eta_s_equation(self):
    i = self.inlets.get("in")
    o = self.outlets.get("out")

    h_in = i.get_parameter("enthalpy")
    h_out = o.get_parameter("enthalpy")
    p_in = i.get_parameter("pressure")
    p_out = o.get_parameter("pressure")
    fluid = i.get_parameter("fluid")

    eta_s = self.get_parameter("eta_s")

    h_2s = isentropic_enthalpy(p_in, h_in, p_out, fluid)

    residual = -(h_out - h_in) + (h_2s - h_in) * eta_s

    return residual
```

O método (*deriv*) responsável pelo cálculo das derivadas parciais da equação de eficiência isentrópica é, conforme o padrão de nomeação adotado, *eta_s_deriv*. Para cada variável presente na equação (pressão e entalpia, associadas às conexões de entrada e saída, ou eficiência, pertencente ao próprio componente), verifica-se se ela é considerada uma variável no

sistema com o método *is_variable*. Caso seja, sua derivada parcial é calculada numericamente e inserida na matriz Jacobiana na posição correspondente.

Nesse caso específico, optou-se pela diferenciação numérica devido à complexidade da função que calcula a entalpia de saída considerando-se um processo isentrópico. Essa grandeza depende de outras propriedades termodinâmicas e do modelo do fluido, tornando a obtenção da derivada analítica mais complexa. A implementação dessas derivadas é a apresentada a seguir:

Trecho de código 3.16 – Cálculo das derivadas numéricas da equação da eficiência isentrópica η_s

```
def eta_s_deriv(self, system):
    i = self.inlets.get("in")
    o = self.outlets.get("out")

    eq_index = system.eq_index.get(f"{self.name}_eta_s_eq")

    def residual_func():
        return self.eta_s_equation()

    if i.is_variable("pressure", system):
        J_col = i.get_J_col("pressure", system)
        deriv = numeric_deriv(residual_func, "pressure", i)
        system.jacobian[eq_index, J_col] = deriv

    if i.is_variable("enthalpy", system):
        J_col = i.get_J_col("enthalpy", system)
        deriv = numeric_deriv(residual_func, "enthalpy", i)
        system.jacobian[eq_index, J_col] = deriv

    if o.is_variable("pressure", system):
        J_col = o.get_J_col("pressure", system)
        deriv = numeric_deriv(residual_func, "pressure", o)
        system.jacobian[eq_index, J_col] = deriv

    if o.is_variable("enthalpy", system):
        J_col = o.get_J_col("enthalpy", system)
        deriv = numeric_deriv(residual_func, "enthalpy", o)
        system.jacobian[eq_index, J_col] = deriv

    if self.is_variable("eta_s", system):
        J_col = self.get_J_col("eta_s", system)
        deriv = numeric_deriv(residual_func, "eta_s", self)
        system.jacobian[eq_index, J_col] = deriv
```

O índice da equação na matriz do sistema, *eq_index*, é obtido a partir de um dicionário de índices criado durante a construção do sistema de equações. Já os índices das colunas do Jacobiano *(J_col)* indicam a posição de cada variável no vetor de variáveis do sistema, sendo atribuídos durante a construção do modelo a partir da associação entre variáveis e suas conexões.

Como exemplo de derivada analítica, considere a equação de resíduo para a razão de pressões *pr*, presente em diversos componentes:

$$f_{pr} = -p_{out} + p_{in} \cdot pr \tag{3.26}$$

Suas derivadas parciais são dadas por:

$$\frac{\partial r_{pr}}{\partial p_{in}} = pr, \quad \frac{\partial r_{pr}}{\partial p_{out}} = -1, \quad \frac{\partial r_{pr}}{\partial pr} = p_{in} \tag{3.27}$$

No código, essas derivadas são calculadas explicitamente e inseridas na matriz Jacobiana, como no método *pressure_ratio_deriv*, apresentado a seguir:

Trecho de código 3.17 – Cálculo das derivadas analíticas da equação de razão de pressões

```
def pressure_ratio_deriv(self, system):
    i = self.inlets.get("in")
    o = self.outlets.get("out")
    eq_index = system.eq_index.get(f"{self.name}_pr_eq")

    if i.is_variable("pressure", system):
        J_col = i.get_J_col("pressure", system)
        deriv = self.get_parameter("pr")
        system.jacobian[eq_index, J_col] = deriv

    if o.is_variable("pressure", system):
        J_col = o.get_J_col("pressure", system)
        deriv = -1
        system.jacobian[eq_index, J_col] = deriv

    if self.is_variable("pr", system):
        J_col = self.get_J_col("pr", system)
        deriv = i.get_parameter("pressure")
        system.jacobian[eq_index, J_col] = deriv
```

3.2.7 *Verificação e depuração de parâmetros*

A classe *Component* também inclui métodos voltados para a inspeção e depuração dos parâmetros definidos, facilitando o acompanhamento do estado interno do componente durante a modelagem.

O método *list_parameters()* exibe de forma simples e direta a lista de parâmetros com seus respectivos valores atuais, útil para uma visão rápida e geral dos dados disponíveis.

Já o método *print_parameters()* fornece uma saída detalhada de cada parâmetro, incluindo não apenas seu valor, mas também indicadores do seu estado, como se é desconhecido (*is_unknown*), variável (*is_variable*), se foi adicionado (*is_added*) ou já resolvido (*is_solved*). Além disso, são exibidos os limites mínimos e máximos, bem como as funções associadas para cálculo da equação e sua derivada, quando presentes. Essa funcionalidade é essencial para depurar o comportamento do modelo, entender como os parâmetros estão configurados e identificar possíveis inconsistências ou erros no sistema.

Trecho de código 3.18 – Exemplo de saída do método *print_parameters()* para um componente *Compressor* com apenas a eficiência isentrópica (*eta_s*) definida pelo usuário.

```
Component Compressor

Parameter: eta_s
```

```
  Value: 0.85
  Is Unknown: False
  Is Added: True
  Is Variable: False
  Is Solved: True
  Min Value: 0
  Max Value: 1
  Function: eta_s_equation
  Derivative: eta_s_deriv

Parameter: W_dot
  Value: None
  Is Unknown: True
  Is Added: False
  Is Variable: False
  Is Solved: False
  Min Value: None
  Max Value: None
  Function: energy_balance_equation
  Derivative: energy_balance_deriv

Parameter: pr
  Value: None
  Is Unknown: True
  Is Added: False
  Is Variable: False
  Is Solved: False
  Min Value: 0
  Max Value: None
  Function: pressure_ratio_equation
  Derivative: pressure_ratio_deriv
```

Esses métodos complementam as funcionalidades de manipulação dos parâmetros apresentados na 3.2.4, permitindo ao desenvolvedor ou usuário uma análise detalhada do componente.

3.2.8 Cálculo e Atualização Pós-Processamento de Parâmetros

A classe base *Component* define o método *calc_parameters()*, que serve como um ponto central para realizar cálculos e atualizações adicionais nos parâmetros após a solução principal do sistema. Este método funciona como uma etapa de pós-processamento, podendo ser sobrescrito pelas subclasses para implementar comportamentos específicos de cada tipo de componente.

Por exemplo, em uma subclasse que representa uma turbina, o método pode calcular grandezas relevantes como a eficiência isentrópica, razão de pressão e potência, utilizando os valores das variáveis de entrada e saída. A seguir, apresenta-se um trecho exemplificando a implementação desse método para o cálculo da razão de pressões *pr* (*pressure ratio*) entre a entrada e a saída do componente.

Trecho de código 3.19 – Exemplo de implementação do método *calc_parameters()* na classe base e sobrescrição na subclasse

```
class Turbine(Component):
```

```
def calc_parameters(self):
    super().calc_parameters()
    i = self.inlets.get("in")
    o = self.outlets.get("out")

    p_in = i.get_parameter("pressure")
    p_out = o.get_parameter("pressure")

    self.set_parameter("pr", p_out / p_in)
```

3.2.9 *Padrões de Uso e Extensibilidade*

A classe *Component* foi projetada como base para herança, permitindo a criação de componentes especializados e a fácil expansão do *framework*. Classes derivadas como turbomáquinas e trocadores de calor herdam toda a funcionalidade base e podem adicionar:

- Múltiplos *inlets* ou *outlets* dependendo a constituição do equipamento;
- Parâmetros específicos do equipamento, como eficiência isentrópica para turbomáquinas e efetividade para trocadores de calor;
- Equações constitutivas particulares, como balanço de energia, cálculos de eficiência e perda de carga;
- Validações operacionais por meio da definição de limites físicos de operação como razões de pressão máximas e mínimas, eficiências entre 0 e 1.

Os componentes implementados neste trabalho são apresentados na Seção 3.5.

3.2.9.1 *Passo a passo para criação de um novo componente*

1. Chamada ao construtor base da classe *Component*: Toda nova classe de componente deve iniciar com a chamada explícita ao construtor da classe base, garantindo a correta inicialização dos atributos fundamentais.
2. Definição das portas de entrada e saída (*inlets* e *outlets*): Caso o número ou os nomes das portas de entrada e saída sejam diferentes do padrão *self.inlets = {"in": None}* e *self.outlets = {"out": None}*, deve-se sobrescrever esses dicionários na classe, definindo claramente todas as conexões externas esperadas.
3. Criação dos parâmetros do componente: Para cada parâmetro físico ou de projeto (como eficiência, potência ou taxa de transferência de calor), utilizar o método *add_parameter*, especificando:
    - Nome do parâmetro;
    - Valor mínimo e máximo permitido, quando aplicável;
    - Nome da função de equação associada (sufixo *_equation*), caso o parâmetro esteja vinculado a uma equação;

- Nome da função de derivada associada (sufixo *_deriv*), quando houver;
- Número de equações associadas ao parâmetro (*num_eq*), sendo o padrão igual a zero caso nenhuma equação seja vinculada.

4. Definição de restrições obrigatórias: Caso o componente possua equações que devem obrigatoriamente ser satisfeitas (por exemplo, um balanço de energia), adicionar essas restrições usando o método *add_constraint*.

5. Implementação das equações constitutivas: Criar as funções que representem as equações físicas ou empíricas que governam o comportamento do componente, retornando o resíduo da equação. Por padrão, o nome dessas funções deve ser baseado no nome da equação com o sufixo *_equation*, por exemplo: *energy_balance_equation*.

6. Implementação das derivadas das equações: Para cada equação definida, implementar sempre que possível a derivada analítica do resíduo em relação a cada parâmetro envolvido. O nome da função derivada deve seguir o padrão de sufixo *_deriv*, como em *energy_balance_deriv*. A indexação das equações para a montagem do Jacobiano deve ser feita utilizando o índice do sistema de equações, por meio do dicionário *system.eq_index*, que associa o nome da equação ao seu índice. Por convenção, o nome da equação, quando gerenciada pela classe *System*, segue o formato *"nome_do_parâmetro_eq"*. Caso a derivada analítica não seja viável, utilizar o método de diferenciação numérica *numeric_deriv* disponível em *utils.py*.

7. Implementar o método *calc_parameters()*: Caso o componente necessite calcular parâmetros com base em variáveis conhecidas, implementar esse método para garantir o correto processamento dos parâmetros após a solução do sistema.

A seguir, apresenta-se um exemplo de implementação de um novo componente seguindo o passo a passo descrito anteriormente.

Trecho de código 3.20 – Exemplo de implementação de um novo componente com o parâmetro de perda de carga *dp*

```
class NovoComponente(Component):

    def __init__(self, name, **kwargs):
        super().__init__(name, **kwargs)

        self.inlets = {"in": None, "in_2": None} # Adição de mais uma porta de entrada
        self.outlets = {"out": None, "out_2": None}  # Adição de mais uma porta de saída
        # Adição do parâmetro "dp" com uma equação associada
        self.add_parameter(
            "dp",
            value=kwargs.get('dp', None),
            min_value = 0,
            func=self.dp_equation,
            deriv=self.dp_deriv,
            num_eq=1
        )
```

```
    def dp_equation(self):
        """
        Equação residual para perda de carga: p_out - p_in + dp = 0
        """

        i = self.inlets.get("in")
        o = self.outlets.get("out")

        p_in = i.get_parameter("pressure")
        p_out = o.get_parameter("pressure")

        dp = self.get_parameter("dp")

        residual = p_out - p_in + dp

        return residual

    def dp_deriv(self, system):
        """
        Derivadas parciais analíticas para a equação da perda de carga.
        """

        i = self.inlets.get("in")
        o = self.outlets.get("out")

        eq_index = system.eq_index.get(f"{self.name}_dp_eq")

        if eq_index is None:
            print(f"Warning: Could not find equation index for {self.name}_dp_eq")
            return

        # Derivada em relação à pressão de entrada
        if i.is_variable("pressure", system):
            J_col = i.get_J_col("pressure", system)
            deriv = -1 # ou deriv = numeric_deriv(dp_equation, "pressure", i)
            system.jacobian[eq_index, J_col] = deriv

        # Derivada em relação à pressão de saída
        if o.is_variable("pressure", system):
            J_col = o.get_J_col("pressure", system)
            deriv = 1
            system.jacobian[eq_index, J_col] = deriv

        # Derivada em relação ao dp
        if self.is_variable("dp", system):
            J_col = self.get_J_col("dp", system)
            deriv = 1
            system.jacobian[eq_index, J_col] = deriv

    def calc_parameters(self):

        i = self.inlets.get("in")
        o = self.outlets.get("out")

        p_in = i.get_parameter("pressure")
        p_out = o.get_parameter("pressure")

        self.set_parameter("dp", p_out - p_in)
```

Esse processo organizado garante que cada novo componente esteja devidamente integrado ao *framework*, com estrutura consistente de parâmetros, equações e derivadas.

3.3 Classe *Connection*

A classe *Connection* representa as conexões físicas entre os componentes do sistema térmico, sendo responsável por transportar o fluido de trabalho e suas propriedades termodinâmicas da saída de um componente de origem para a entrada de um componente de destino. Esta classe é fundamental para a modelagem do ciclo, pois encapsula todas as informações de estado do fluido em cada ponto do sistema e gerencia as equações que calculam as propriedades termodinâmicas ao longo do sistema. Além disso, a classe oferece uma estrutura para armazenar, calcular, atualizar e resolver os parâmetros termodinâmicos associados a cada interface entre os componentes, assegurando a correta propagação das condições de operação durante a simulação.

3.3.1 Arquitetura e Inicialização

O objeto *Connection* é inicializado por meio de uma ligação bidirecional entre dois componentes, por meio da definição do componente de origem (*source*) e sua porta de saída (*source_outlet)*, assim como o componente de destino (*target)* e sua porta de entrada (*target_inlet*). O atributo *label* pode ser utilizado para nomear a conexão, sendo uma ferramenta importante para a identificação e depuração dos resultados durante a simulação. Ao ser criada, a conexão registra-se automaticamente em ambos os componentes, por meio dos métodos *register_inlet_connection* e *register_outlet_connection*, que garantem que os componentes reconheçam suas conexões de saída e entrada, respectivamente.

Além disso, a classe cria um dicionário chamado *parameters*, o qual armazena os principais parâmetros termodinâmicos e de fluxo associados ao fluido nesta interface, como vazão mássica, temperatura, pressão, entalpia, título e tipo de fluido. O atributo *flow_group*, cujo valor padrão é *"primary"*, permite organizar diferentes caminhos de escoamento dentro do sistema, sendo útil em casos como trocadores de calor com dois circuitos de fluido. Essa inicialização é apresentada no Trecho de código 3.21.

Trecho de código 3.21 – Inicialização da Classe *Connection*

```
def __init__(self, source, source_outlet, target, target_inlet, label=None, flow_group="
    primary"):

    self.source = source
    self.source_outlet = source_outlet
    self.target = target
    self.target_inlet = target_inlet
    self.label = label

    self.flow_group = flow_group
    self.parameters = {}

    # Register connection with components
```

```
self.source.register_outlet_connection(self, source_outlet)
self.target.register_inlet_connection(self, target_inlet)
```

3.3.2 Gerenciamento de Parâmetros

Assim como na classe *Component*, o gerenciamento dos parâmetros na classe *Connection* é feito por meio do método *add_parameter()*. No entanto, diferentemente dos componentes, as conexões possuem um conjunto limitado e bem definido de parâmetros termodinâmicos associados ao escoamento entre dois pontos do sistema. Esses parâmetros são definidos nas unidades do Sistema Internacional (SI), de forma a garantir consistência ao longo da simulação. Os parâmetros permitidos são:

- *m_dot* : vazão mássica [$kg\ s^{-1}$];
- *pressure* : pressão [Pa];
- *enthalpy* : entalpia específica [$J\ kg^{-1}$];
- *temperature* : temperatura [K];
- *quality* : título (fração de vapor entre 0 e 1);
- *fluid* : fluido de trabalho;
- *phase* : fase termodinâmica. Este parâmetro é utilizado apenas para pós-processamento, não estando diretamente envolvido na solução do sistema de equações.

Enquanto na classe *Component* o conjunto de parâmetros é livre e adaptável ao tipo de equipamento, na classe *Connection* somente os parâmetros da lista acima podem ser adicionados. Caso haja uma tentativa de incluir um parâmetro não permitido, o código resulta em erro, evitando inconsistências na modelagem.

Assim como na classe *Component*, esses parâmetros são adicionados com o método *add_parameter*, que também atribui dados a cada um, indicando o seu estado atual, como se foi definido pelo usuário, *is_added*, se é desconhecido, *is_unknown*, uma variável do sistema, *is_variable*, ou um valor já resolvido, *is_solved*, e, quando aplicável, funções associadas para o cálculo de resíduos e derivadas.

Atualmente, apenas os parâmetros *temperature* e *quality* possuem equações de resíduo associadas. Os demais são utilizados na propagação de propriedades termodinâmicas, com base nas combinações de variáveis conhecidas e nas correlações obtidas por meio da biblioteca CoolProp.

Além disso, seguiu-se uma abordagem inspirada no funcionamento da biblioteca TESPy para a definição das variáveis do sistema. Nesse contexto, apenas os parâmetros *m_dot*, *pressure* e *enthalpy* são candidatos a serem tratados como variáveis na solução iterativa do sistema de equações. Nesse sentido, as demais propriedades, como temperatura e título, são calculadas a partir de equações termodinâmicas. Essa distinção garante que o

sistema de equações inclua apenas os graus de liberdade necessários, evitando sobreposição entre variáveis dependentes, conforme ilustrado no Trecho de código 3.22.

Trecho de código 3.22 – Trecho do método *add_parameter()* da Classe *Connection*

```
def add_parameter(self, **kwargs):

    valid_params = ["m_dot", "temperature", "pressure", "enthalpy", "quality", "fluid", "
        phase"]
    for param_name, value in kwargs.items():
        if param_name not in valid_params:
            raise ValueError(f"Invalid parameter '{param_name}' provided.")

        if param_name in ["m_dot", "pressure", "enthalpy"] and is_unknown:
            is_variable = True
```

Dessa forma, dentro do método *add_parameter()*, o atributo *is_variable* é definido como *True* apenas se o parâmetro for um desses três (*m_dot*, *pressure* ou *enthalpy*) e se o seu valor estiver inicialmente desconhecido. Assim, apenas os parâmetros marcados como *is_variable* serão considerados na montagem da matriz Jacobiana durante o processo de solução numérica. Como consequência, somente esses parâmetros participam diretamente do sistema de equações não lineares que será resolvido pelo método de Newton-Raphson.

3.3.3 *Métodos de Acesso e Atualização de Parâmetros*

Assim como na classe *Component*, a classe *Connection* também disponibiliza métodos para acessar e atualizar os valores dos seus parâmetros. Esses métodos, apresentados no Trecho de códgio 3.23, permitem a leitura e a modificação das propriedades termodinâmicas associadas às conexões entre componentes, como pressão, temperatura, entalpia e vazão mássica.

Os métodos implementados são:

- *get_parameter(param_name)*: Retorna o valor atual de um parâmetro específico da conexão, facilitando a leitura de propriedades já definidas ou calculadas durante a simulação.
- *set_parameter(param_name, value)*: Atualiza o valor de um parâmetro existente na conexão. Ao modificar o valor, o método também atualiza automaticamente os estados internos, como o indicador de se o parâmetro é desconhecido. Além disso, quando ocorre alteração no valor, as propriedades termodinâmicas da conexão são recalculadas de forma automática, por meio do método *update_thermodynamic_properties* garantindo consistência com os dados atualizados.

Trecho de código 3.23 – Métodos de acesso e atualização de parâmetros na classe *Connection*

```
def get_parameter(self, param_name):
"""Retorna o valor de um parâmetro especificado."""
return self.parameters.get(param_name, {}).get("value")

def set_parameter(self, param_name, value):
```

```
"""Atualiza o valor de um parâmetro existente."""
    if param_name in self.parameters:
        old_value = self.parameters[param_name]["value"]
    # Atualiza o valor
        self.parameters[param_name]["value"] = value
        self.parameters[param_name]["is_unknown"] = False if value is not None else True

    if value is not None:
        self.parameters[param_name]["is_variable"] = False

        # Se o valor mudou, recalcula as propriedades termodinâmicas
        if old_value != value:
            self.update_thermodynamic_properties()

    elif value is None:
            self.parameters[param_name]["is_added"] = False
    else:
        raise KeyError(f"Parameter '{param_name}' not found in the connection.")
```

3.3.4 *Cálculo de propriedades termodinâmicas*

O sistema implementado utiliza a biblioteca CoolProp para realizar cálculos de propriedades termodinâmicas baseados em diferentes combinações de parâmetros conhecidos. O algoritmo segue uma abordagem sistemática para determinar propriedades ausentes através de relações termodinâmicas estabelecidas no método *update_thermodynamic_properties()*.

O processo de cálculo é estruturado em cinco casos principais, baseados nas propriedades disponíveis:

- Caso 1 - Pressão e Entalpia conhecidas: Quando a pressão (p) e entalpia específica (h) são fornecidas, o sistema calcula a temperatura (T) através da relação termodinâmica:

$$T = f(p, h, \text{fluido}) \tag{3.28}$$

  Adicionalmente, verifica-se se o fluido encontra-se na região bifásica através da função *is_in_two_phase_region()* para determinar o título (x) através da função auxiliar de cálculo do título (*calculate_quality_from_h()*).

- Caso 2 - Temperatura e Título conhecidos: Para sistemas em região bifásica com temperatura (T) e título (x) definidos, as propriedades restantes são calculadas utilizando:

$$p = f(T, x, \text{fluido}) \tag{3.29}$$

$$h = f(T, x, \text{fluido}) \tag{3.30}$$

- Caso 3 - Pressão e Título conhecidos: Similar ao caso anterior, mas utilizando pressão e título como parâmetros de entrada:

$$T = f(p, x, \text{fluido}) \tag{3.31}$$

$$h = f(p, x, \text{fluido}) \tag{3.32}$$

- Caso 4 - Temperatura e Pressão conhecidas: O sistema primeiro verifica se as condições correspondem à região bifásica através da função *is_in_two_phase_region(T, p, fluido)*. Se estiver, ele emite um aviso informando o usuário. Em seguida, para fluidos em estado monofásico, a entalpia é determinada por:

$$h = f(T, p, \text{fluido}) \tag{3.33}$$

  Após o cálculo da entalpia, o título é determinado através da função *calculate_quality_from_h()*.

- Caso 5 - Temperatura e Entalpia conhecidas: Quando a temperatura (T) e entalpia específica (h) são fornecidas, o sistema calcula a pressão (p) através da relação termodinâmica:

$$p = f(T, h, \text{fluido}) \tag{3.34}$$

  Similarmente aos casos anteriores, o título é determinado se o sistema estiver em região bifásica.

3.3.4.1 Detecção de Fases e Determinação do Título

A função da detecção da região bifásica é realizada pelo método *is_in_two_phase_region()*, o qual implementa os seguintes critérios de verificação:

- Verificação dos Limites Críticos: O primeiro critério verifica os pontos relação à região supercrítica:

$$T \geq T_{\text{crit}} \Rightarrow \text{região supercrítica} \tag{3.35}$$

$$p \geq p_{\text{crit}} \Rightarrow \text{região supercrítica} \tag{3.36}$$

  Se qualquer uma das condições for satisfeita, o fluido não é considerado bifásico.

- Verificação por Entalpia: Quando a entalpia é conhecida, o método a compara as entalpias de saturação na temperatura ou pressão correspondente. Se a entalpia estiver compreendida entre os limites de saturação, o ponto pertence à região bifásica.

$$h_{\text{líq}} < h < h_{\text{vap}} \tag{3.37}$$

  onde $h_{\text{líq}}$ e h_{vap} correspondem às entalpias do líquido e do vapor saturado, respectivamente.

- Verificação por Condições de Saturação: Para temperatura e pressão simultaneamente conhecidas, verifica-se a proximidade às condições de saturação com tolerância $\epsilon = 10^{-3}$:

$$\frac{|p - p_{\text{sat}}|}{p_{\text{sat}}} < \epsilon \text{ ou } \frac{|T - T_{\text{sat}}|}{T_{\text{sat}}} < \epsilon \Rightarrow \text{região bifásica} \tag{3.38}$$

  Se qualquer uma das condições for satisfeita, o ponto é classificado como pertencente à região bifásica.

A determinação do título é realizada através do método *calculate_quality_from_h()*, que utiliza a função auxiliar *is_in_two_phase_region()* para verificar se o sistema encontra-se na região bifásica e calcula o título x baseado na entalpia específica e em outra variável de estado termodinâmico.

O algoritmo segue os seguintes passos principais:

- Validação de Entradas: O método primeiro verifica se o fluido e a entalpia foram especificados.
- Verificação da Região Bifásica: O algoritmo primeiro utiliza a função *is_in_two_phase_region()* para verificar se as condições permitem a existência de uma região bifásica e o cálculo do título correspondente.
- Caso 1 - Utilização da Pressão: Quando a pressão (p) é conhecida, as entalpias de saturação são calculadas:

$$h_{\text{líq}} = f(p, x = 0, \text{fluido}) \tag{3.39}$$

$$h_{\text{vap}} = f(p, x = 1, \text{fluido}) \tag{3.40}$$

- Caso 2 - Utilização Temperatura: Quando a temperatura (T) é conhecida, as entalpias de saturação são determinadas por:

$$h_{\text{líq}} = f(T, x = 0, \text{fluido}) \tag{3.41}$$

$$h_{\text{vap}} = f(T, x = 1, \text{fluido}) \tag{3.42}$$

- Cálculo do Título: Com as entalpias de saturação determinadas, o título é calculado pela interpolação linear entre os estados de líquido e vapor saturados:

$$x = \frac{h - h_{\text{líq}}}{h_{\text{vap}} - h_{\text{líq}}} \tag{3.43}$$

  O resultado representa a fração mássica de vapor presente no sistema, sendo que $x = 0$ corresponde ao líquido saturado e $x = 1$, ao vapor saturado.

- Verificação do resultado: O valor calculado de x é limitado ao intervalo físico admissível $0 \leq x \leq 1$. Caso os dados fornecidos não permitam identificar a região bifásica, como em condições supercríticas ou com informações insuficientes, ou em situações de falha no acesso às propriedades termodinâmicas, o método retorna um valor nulo, *None*, indicando que o conceito de título não é aplicável ao ponto em análise.

3.3.5 Integração com o Sistema de Equações

As conexões são integradas ao sistema de equações global por meio de uma estrutura que associa suas propriedades termodinâmicas e equações às variáveis resolvidas pelo *solver*. Assim como na classe *Component*, essa associação é viabilizada por métodos auxiliares, *is_unknown()*, *is_variable()*, *is_added()* e *is_solved()*, que avaliam o *status* de cada parâmetro e *get_J_col()* e *get_eq_index()*, os quais localizam sua posição dentro da matriz Jacobiana e do vetor de resíduos do sistema.

3.3.5.1 Pré-processamento

Antes da definição das equações das conexões que serão adicionadas ao sistema global, é realizado um pré-processamento das conexões por meio do método *preprocess()*. O objetivo desse método é identificar automaticamente quais propriedades termodinâmicas podem ser consideradas como resolvidas, *solved*, com base nas combinações de parâmetros já disponíveis. O algoritmo de pré-processamento analisa três cenários principais:

- Entalpia não definida com Pressão conhecida: Quando a entalpia não foi adicionada mas a pressão está disponível, o sistema verifica se existe temperatura ou título definidos. Se para o caso da temperatura definida o fluido estiver em estado monofásico, o método marca pressão, entalpia e a propriedade complementar (temperatura ou título) como resolvidas, *solved* pois estas combinações permitem determinação única do estado termodinâmico.
- Pressão não definida com Entalpia conhecida: Similar ao caso anterior, mas quando a pressão é a propriedade ausente. O sistema identifica que com entalpia mais temperatura ou título, todas as propriedades principais podem ser determinadas, e as marca como *solved*.
- Ausência de Entalpia e Pressão: Quando nem entalpia nem pressão estão definidas, mas temperatura e título estão disponíveis, o sistema reconhece que esta combinação permite calcular todas as demais propriedades e as marca como *solved*.

Esse pré-processamento é fundamental para evitar a formulação desnecessária de equações de conexão. Caso todas as propriedades termodinâmicas envolvidas em uma conexão já estejam resolvidas após essa etapa, não há necessidade de adicionar equações adicionais ao sistema.

3.3.5.2 Definição das Equações de Conexão

Se, após o pré-processamento, alguma propriedade ainda permanecer não resolvida, o método *get_connection_equations()* é responsável por gerar as equações necessárias para que o *solver* possa determinar o estado termodinâmico completo da conexão.

O método percorre os parâmetros termodinâmicos da conexão e, para cada parâmetro que:

- foi explicitamente adicionado (*is_added* = *True*),
- ainda não está resolvido (*is_solved* = *False*),
- não é uma variável livre no sistema (*is_variable* = *False*),
- e possui uma função de cálculo (*func)* definida,

o método retorna a equação associada ao parâmetro que será incluída no sistema de equações. Cada equação representa o resíduo entre o valor especificado da propriedade e o valor

calculado a partir de outras propriedades, entalpia e pressão, as quais são variáveis do sistema.

Por exemplo, se o usuário especifica apenas a temperatura de uma conexão, o pré-processamento não conseguirá marcar a pressão e a entalpia como resolvidas, pois essas propriedades ainda são desconhecidas. Nesse caso, o método *get_connection_equations()* irá retornar uma equação da forma:

$$f_T = T_{\text{especificada}} - T_{\text{calculada}}(p, h) \tag{3.44}$$

Essa equação será utilizada pelo *solver* para ajustar iterativamente os valores de pressão e entalpia até que a temperatura calculada a partir dessas variáveis coincida com a temperatura definida pelo usuário dentro da tolerância estabelecida.

Esse método associado com o *preprocess* garante que o *solver* tenha as informações necessárias para determinar todas as incógnitas, resultando em um sistema de equações determinado.

3.3.5.3 Mapeamento de Variáveis e Equações

Assim como na classe *Component*, os métodos *get_J_col()* e *get_eq_index()* são utilizados para mapear parâmetros e equações às suas respectivas posições no sistema global:

- *get_J_col():* retorna a posição da coluna da Jacobiana associada a uma variável da conexão. Essa coluna corresponde à derivada da equação com relação à variável em questão. O nome da variável no sistema é construído no formato "*{nome_do_parâmetro}_c{label_da_conexão}*", garantindo a padronização que facilita o gerenciamento da solução realizada pela classe *System*.

- *get_eq_index():* retorna o índice da equação associada ao parâmetro, permitindo acessar diretamente a linha correspondente no vetor de resíduos ou na Jacobiana. O nome da equação segue o formato "*c{label_da_conexão}_{nome_do_parâmetro}_eq*".

3.3.6 Equações e Derivadas

Após o pré-processamento e a identificação dos parâmetros não resolvidos, o método *get_connection_equations()* é responsável por fornecer as equações associadas às conexões ao sistema global. Essas equações possibilitam que o *solver* determine as propriedades termodinâmicas de forma consistente com os parâmetros especificados pelo usuário.

Para cada propriedade com função de cálculo associada, o método retorna uma equação residual que expressa a diferença entre o valor calculado a partir das variáveis do sistema, pressão e entalpia, e o valor especificado. Dois exemplos de equações implementadas são:

- Equação de Temperatura:

$$f_T = T_{\text{calculada}}(p, h) - T_{\text{especificada}} = 0 \tag{3.45}$$

A temperatura calculada é obtida pela função *calc_temperature()*, que utiliza os valores atuais das variáveis de pressão e entalpia.

- Equação de Título:

$$f_x = x_{\text{calculado}}(p, h) - x_{\text{especificado}} = 0 \tag{3.46}$$

Neste caso, a função *calc_quality()* é utilizada para obter o título com base nas variáveis de pressão e entalpia.

As funções implementadas para o cálculo da temperatura e a equação residual da temperatura são mostradas nos Trechos de código 3.24 e 3.25, respectivamente.

Trecho de código 3.24 – Cálculo da temperatura baseado em pressão e entalpia

```
def calc_temperature(self):
"""
Calcula a temperatura baseada na estimativas de entalpia e pressão
"""
fluid = self.parameters.get("fluid", {}).get("value")
if fluid is None:
    print(f"Info: No fluid specified for connection {self.label}.")
    return None

pressure = self.parameters.get("pressure", {}).get("value")
enthalpy = self.parameters.get("enthalpy", {}).get("value")

temperature = CP.PropsSI("T", "P", pressure, "H", enthalpy, fluid)

return temperature
```

Trecho de código 3.25 – Equação residual de temperatura

```
def temperature_equation(self):
    temperature = self.parameters.get("temperature", {}).get("value")
    calculated_temperature = self.calc_temperature()
    residual = calculated_temperature - temperature

    return residual
```

De forma análoga, é retornada a equação associada ao título, *quality_equation*, quando este é definido pelo usuário.

Além de fornecer os resíduos, a classe *Connection* também calcula as derivadas parciais de cada equação em relação às variáveis relevantes, preenchendo as entradas correspondentes na matriz Jacobiana do sistema. Isso é feito por meio de métodos como *temperature_deriv()* e *quality_deriv()*, que usam diferenciação numérica (método *numeric_deriv()*). Essa implementação das derivadas é apresentada a seguir:

$$\frac{\partial f}{\partial p} \quad \text{e} \quad \frac{\partial f}{\partial h} \tag{3.47}$$

Trecho de código 3.26 – Cálculo de derivadas para a equação de temperatura

```
def temperature_deriv(self, system):
    eq_index = system.eq_index.get(f"c{self.label}_temperature_eq")
    if eq_index is None:
```

```
            print(f"Warning: Could not find equation index")
            return

        if self.is_variable("pressure", system):
            J_col = self.get_J_col("pressure", system)
            deriv = numeric_deriv(self.calc_temperature, "pressure", self)
            system.jacobian[eq_index, J_col] = deriv

        if self.is_variable("enthalpy", system):
            J_col = self.get_J_col("enthalpy", system)
            deriv = numeric_deriv(self.calc_temperature, "enthalpy", self)
            system.jacobian[eq_index, J_col] = deriv
```

Essa abordagem modular permite que cada conexão contribua com suas próprias equações e derivadas de maneira sistemática para o *solver*, favorecendo a escalabilidade do *framework*.

3.3.7 *Verificação e depuração de parâmetros*

Assim como a classe *Component*, a classe *Connection* tem o método *print_parameters()*, o qual fornece uma listagem contendo o valor de cada parâmetro da conexão, além de indicadores sobre seu *status*.

3.3.8 *Cálculo das Propriedades no Pós-Processamento*

Após a solução do sistema de equações, as propriedades termodinâmicas adicionais de cada conexão podem ser calculadas com base nas grandezas já resolvidas, como pressão e entalpia. Isso ocorre durante a etapa de pós-processamento, utilizando os resultados numéricos da simulação.

A classe *Connection* contém o método *update_properties_from_pressure_and_enthalpy*, que é responsável por calcular propriedades dependentes, como temperatura e título do fluido (*quality*). Este método utiliza a biblioteca CoolProp para determinar a fase do fluido (*phase*) com base nos valores de pressão e entalpia armazenados na conexão. Em seguida, a temperatura é obtida por meio da função interna *calc_temperature*, e o título pela função *calc_quality*, ambas implementadas na própria classe.

O processo garante que todas as propriedades necessárias para a análise e geração de resultados estejam devidamente atualizadas, mantendo a consistência termodinâmica entre as variáveis. O trecho de código abaixo ilustra a implementação deste procedimento de atualização:

Trecho de código 3.27 – Cálculo das propriedades no pós-processamento

```
def update_properties_from_pressure_and_enthalpy(self):
    """
    Atualiza as propriedades termodinâmicas com base na pressão e entalpia.
    Calcula temperatura, título e fase do fluido.
    """
    pressure = self.parameters.get("pressure", {}).get("value")
    enthalpy = self.parameters.get("enthalpy", {}).get("value")
    fluid = self.parameters.get("fluid", {}).get("value")
```

```
    phase = CP.PhaseSI("P", pressure, "H", enthalpy, fluid)
    self.set_parameter("phase", phase)

    temperature = self.calc_temperature()
    self.set_parameter("temperature", temperature)

    quality = self.calc_quality()
    self.set_parameter("quality", quality)
```

3.4 *Classe System*

A classe *System* representa o núcleo central de gerenciamento do sistema térmico, coordenando e integrando todos os componentes e conexões do sistema térmico. Esta classe implementa o método de Newton-Raphson para a resolução do sistema de equações não lineares e gerencia a estrutura global de variáveis e equações da simulação termodinâmica.

3.4.1 Arquitetura e Inicialização

O sistema mantém uma estrutura que organiza componentes, conexões, variáveis e equações de forma integrada. A inicialização é apresentada no trecho de código abaixo.

Trecho de código 3.28 – Inicialização da Classe *System*

```
def __init__(self):
    self.components = {}
    self.connections = []

    self.variables = []
    self.var_index = {}
    self.eq_index = {}
    self.equations = []
    self.equations_names = []
    self.residuals = np.array([])

    self.jacobian = None
```

Durante a inicialização, os seguintes elementos são criados:

- Dicionário de componentes (*self.components*): armazena todas as instâncias de componentes adicionados ao sistema.
- Lista de conexões (*self.connections*): mantém todas as conexões físicas entre os componentes, representadas por objetos da classe *Connection*.
- Estruturas para variáveis e equações: incluem a lista de variáveis globais (*self.variables*), os índices de mapeamento para variáveis (*self.var_index*) e para equações (*self.eq_index*), além da lista de equações do sistema (*self.equations* e *self.equations_names*).
- Resíduos: o vetor de resíduos (*self.residuals*) é utilizado para armazenar o valor atual dos erros de cada equação durante o processo iterativo de solução.

- Matriz Jacobiana: a matriz Jacobiana (*self.jacobian*) contém as derivadas parciais de cada equação em relação a cada variável, sendo essencial para o método de Newton-Raphson.

Essa estrutura permite que, ao longo da montagem do sistema, novos componentes e conexões sejam adicionados de forma sistemática. Além disso, garante que o sistema consiga construir de maneira eficiente o conjunto de equações não lineares, calcular os resíduos e montar a Jacobiana necessária para o processo de solução.

3.4.2 Gerenciamento de Componentes e Conexões

Após a inicialização da classe *System*, são disponibilizados métodos que permitem a montagem do sistema termodinâmico por meio da adição de componentes, definição de conexões entre eles e especificação das condições do escoamento.

3.4.2.1 Adição de Componentes e Conexões

O método *add_component* permite incluir novos componentes ao sistema. Cada componente adicionado deve possuir um nome único, utilizado como chave de acesso no dicionário *self.components*. Caso um nome duplicado seja detectado, o método lança uma exceção para evitar conflitos de nomenclatura, conforme apresentado no Trecho de código 3.29. Para conexões, o método *add_connection* estabelece as interligações entre componentes, criando a topologia do sistema.

Trecho de código 3.29 – Adição de componentes e conexões ao sistema

```
def add_component(self, component):
    """Adiciona um componente ao sistema."""
    if component.name in self.components:
        raise ValueError(f"Component {component.name} already exists.")
    self.components[component.name] = component
    print(f"Component {component} : {component.name} added to the system.")

def add_connection(self, connection):
    """Adiciona uma conexão ao sistema."""
    self.connections.append(connection)
```

3.4.2.2 Gerenciamento de propriedades termodinâmicas

O gerenciamento de propriedades termodinâmicas é realizado através dos métodos *update_all_thermodynamic_properties* e *update_all_properties_from_pressure_and_enthalpy*, utilizados no pós-processamento, os quais garantem consistência das propriedades em todas as conexões. esses métodos são ilustrados no Trecho de código 3.30.

Trecho de código 3.30 – Métodos para atualização das propriedades termodinâmicas em todas as conexões.

```
def update_all_thermodynamic_properties(self):
    """Atualiza as propriedades termodinâmicas de todas as conexões."""
    for conn in self.connections:
```

```
            conn.update_thermodynamic_properties()

    def update_all_properties_from_pressure_and_enthalpy(self):
        """Atualiza as propriedades termodinâmicas a partir da pressão e entalpia para todas
           as conexões."""
        for conn in self.connections:
            conn.update_properties_from_pressure_and_enthalpy()
```

3.4.2.3 *Definição de Parâmetros do Escoamento*

O método *set_flow* permite a definição dos parâmetros de escoamento para grupos específicos de conexões, denominados *flow groups*. Por padrão, todas as conexões pertencem ao grupo *"primary"*, mas o usuário pode criar grupos adicionais ao projetar sistemas com múltiplos circuitos ou fluidos distintos.

Durante a execução do *set_flow*, o sistema percorre todas as conexões associadas ao grupo especificado e define os parâmetros informados, como o fluido de trabalho (*fluid*), a vazão mássica (*m_dot*) e o título do escoamento (*quality*). É importante ressaltar que o parâmetro *fluid* é obrigatório e não pode ser indefinido (*None*), garantindo que todas as conexões dentro de um mesmo grupo compartilhem o mesmo fluido base, conforme ilustrado no Trecho de código 3.31.

Trecho de código 3.31 – Definição de parâmetros do escoamento

```
def set_flow(self, flow_group="primary", **kwargs):
    """Define parâmetros de fluxo e propaga através do circuito."""
    if "fluid" in kwargs and kwargs["fluid"] is None:
        raise ValueError("Fluid parameter cannot be None.")

    print(f"Setting flow parameters for group '{flow_group}': {kwargs}")

    for conn in self.connections:
        if conn.flow_group == flow_group:
            for param_name, value in kwargs.items():
                if param_name in ["fluid", "m_dot", "quality"]:
                    if conn.parameters.get(param_name) is None or \
                       conn.parameters[param_name].get("is_unknown", True):
                        conn.add_parameter(**{param_name: value})
```

A associação de cada conexão a um *flow group* ocorre no momento em que os parâmetros de escoamento são adicionados, por meio do método *add_parameter* da classe *Connection*. Dessa forma, ao longo da construção do sistema, o usuário pode organizar diferentes circuitos termodinâmicos simplesmente especificando o nome do grupo ao adicionar as conexões.

3.4.2.4 *Análise de Determinação do Sistema*

O método *analyze_system* fornece uma ferramenta essencial para o usuário verificar se o sistema termodinâmico está matematicamente bem determinado, ou seja, se o número de variáveis é igual ao número de equações. Esta análise é fundamental antes de iniciar a resolução numérica, permitindo identificar inconsistências como falta ou excesso de variáveis ou equações.

Ao ser executado, o método realiza as seguintes etapas principais:

- Atualiza as propriedades termodinâmicas de todas as conexões, garantindo que os dados de entrada estejam coerentes.
- Varre todas as conexões e componentes do sistema para identificar quais parâmetros estão definidos como variáveis do sistema (*is_variable*).
- Lista todas as equações associadas às conexões e aos componentes, incluindo tanto as equações provenientes de parâmetros quanto as provenientes de *constraints* dos componentes.
- Calcula o total de variáveis e equações, exibindo um resumo com a contagem de cada tipo.
- Realiza um diagnóstico, informando se o sistema está bem determinado (número de variáveis igual ao número de equações) ou se há inconsistências.

Nesse sentido, o método produz uma análise detalhada que verifica se o número de variáveis corresponde ao número de equações, conforme mostrado no trecho de código 3.32.

Trecho de código 3.32 – Trecho do método de análise do sistema para verificação de restrições

```
def analyze_system(self):
    """
    Analisa o sistema para identificar todas as variáveis e equações.
    """
    # Cálculo dos totais
    total_vars = len(conn_vars) + len(comp_vars) # Variáveis de conexões + variáveis de
        componentes
    total_eqs = len(conn_eqs) + len(comp_eqs) #Equações de conexões + equações de
        componentes

    # Verificação se o sistema está adequadamente restrito
    if total_vars != total_eqs:
        print(f"\nWARNING: System is not properly constrained!")
        print(f"The system has {total_vars} variables but {total_eqs} equations.")
    else:
        print("\nSystem is properly constrained.")
```

3.4.3 *Configuração de Variáveis e Equações*

O processo de solução inicia com a definição das variáveis e das equações que descrevem o sistema termodinâmico. Esta etapa é fundamental para garantir que o número de variáveis seja igual ao número de equações, condição necessária para que o sistema seja solucionável.

3.4.3.1 *Identificação de Variáveis*

A função *setup_variables* percorre todos os componentes e conexões do sistema, registrando as variáveis que devem ser resolvidas durante a simulação. Cada variável recebe um índice único, o que permite organizar corretamente sua posição na matriz Jacobiana, como

apresentado no Trecho de código 3.33. Um ponto importante é que, dentro de um mesmo *flow_group*, as variáveis de vazão mássica (*m_dot*) são compartilhadas entre as conexões que pertencem ao mesmo circuito, garantindo a continuidade de massa ao longo do escoamento.

Trecho de código 3.33 – Exemplo de registro de variável

```
self.variables.append(var_name)
self.var_index[var_name] = J_col
J_col += 1
```

3.4.3.2 *Configuração das Equações*

A função *setup_equations* coleta todas as equações dos componentes e conexões, construindo o sistema global de equações não lineares. O método processa equações baseadas em parâmetros (quando um valor foi especificado, ou seja, "is_added = True"e existe uma função de equação associada) e equações obrigatórias (*constraint*) dos componentes.

Para conexões, são incluídas equações de propriedades termodinâmicas, como relações entre temperatura, pressão e entalpia. Cada equação recebe um índice único no vetor de resíduos e é associada às suas funções de cálculo e derivada correspondentes, como observado no trecho de código abaixo.

Trecho de código 3.34 – Exemplo de adição de equação

```
self.equations.append(func)
self.equation_names.append(eq_name)
self.eq_index[eq_name] = eq_number
eq_number += 1
```

3.4.3.3 *Verificação do Sistema*

Ao final, o sistema verifica se o número total de variáveis é igual ao número de equações. Caso contrário, um erro é lançado no código:

Trecho de código 3.35 – Verificação de sistema determinado

```
if len(self.variables) != len(self.equations):
    raise ValueError("System has {num_vars} variables but {num_eqs} equations")
```

Esta verificação garante que o sistema seja solucionável antes da execução do resolvedor numérico.

3.4.4 *Implementação do Método Newton-Raphson*

A implementação do método de Newton-Raphson na classe *System* segue rigorosamente os passos apresentados na metodologia, estabelecendo uma correspondência direta entre a teoria e a implementação prática.

3.4.4.1 Estimativa Inicial

Antes de iniciar o processo iterativo de solução, é necessário fornecer uma estimativa inicial (*initial guess*) para todas as variáveis livres do sistema. Essa etapa é fundamental para garantir a convergência, especialmente em métodos como Newton-Raphson.

O vetor de estimativas iniciais é construído pela classe *System* por meio do método *initial_guess*. Nesse método, cada variável é tratada conforme sua natureza e sua posição na hierarquia do método. As estimativas podem ser atribuídas em diferentes níveis de prioridade: valores padrão, estimativas globais fornecidas pelo usuário e estimativas específicas por conexão. Quando um mesmo parâmetro é definido em mais de um nível, aplica-se a seguinte ordem de precedência: a estimativa por conexão sobrescreve a estimativa global, que por sua vez sobrescreve o valor padrão.

O processo de geração da estimativa inicial segue as seguintes etapas:

- Atribuição de valores padrão: Temperaturas recebem um valor inicial típico (por exemplo, 300 K), pressões são iniciadas em 101325 Pa, e assim por diante para demais propriedades como vazão mássica, eficiência isentrópica e entalpia.
- Aplicação de estimativas globais (*global_guess*): Caso o usuário forneça um dicionário de estimativas iniciais globais, o método sobrescreve os valores padrão para todas as variáveis do sistema com o mesmo nome de parâmetro.
- Estimativas específicas por conexão (*connection_guess*): Caso o usuário forneça um dicionário com estimativas específicas por conexão, o método identifica as variáveis associadas a cada conexão e ajusta suas estimativas individuais.
- Cálculo de entalpias a partir de outras propriedades: Em casos onde a variável entalpia não tem uma estimativa dada pelo usuário, mas existem valores definidos para pressão e temperatura, pressão e título ou temperatura e título na respectiva conexão, o método calcula a entalpia usando a biblioteca CoolProp.

O trecho de código a seguir exemplifica a estrutura básica do método:

Trecho de código 3.36 – Estrutura geral do método *initial_guess()*.

```
def initial_guess(self, global_guess=None, connection_guess=None):
    x0 = np.zeros(len(self.variables))

    # Valores padrão
    for i, var in enumerate(self.variables):
        if var["param"] == "temperature":
            x0[i] = 300
        elif var["param"] == "pressure":
            x0[i] = 101325
        elif var["param"] == "m_dot":
            x0[i] = 1  # 1 kg/s
        elif var["param"] == "eta_s":
            x0[i] = 0.8  # 80%

    # Global guess
```

```
    if global_guess:
        for param, value in global_guess.items():
            for i, var in enumerate(self.variables):
                if var["param"] == param:
                    x0[i] = value

    # Connection  guess
    if connection_guess:
        for conn_name, conn_guess in connection_guess.items():
            for i, var in enumerate(self.variables):
                if "connection" in var and str(var["connection"].label) == conn_name:
                    if var["param"] in conn_guess:
                        x0[i] = conn_guess[var["param"]]
```

Ao final da execução, o vetor *x0* contém a estimativa inicial para todas as variáveis, respeitando a hierarquia definida no modelo.

3.4.4.2 *Cálculo dos Resíduos*

O método *calculate_residuals* implementa a avaliação $\mathbf{f}(\mathbf{x}^{(k)})$, calculando quanto cada equação se desvia da solução na iteração atual. Este método atualiza primeiro todas as variáveis com os valores atuais e então avalia cada função de equação. A convergência é alcançada quando os resíduos se aproximam de zero, indicando que as equações estão satisfeitas.

No *framework* implementado, o método *calculate_residuals* recebe o vetor atual de variáveis *x* e realiza o cálculo dos valores das equações do sistema, armazenando os resultados em um vetor de resíduos para análise posterior no processo de solução. Um trecho desse método é mostrado a seguir:

Trecho de código 3.37 – Cálculo dos resíduos do sistema

```
def calculate_residuals(self, x):
    # Calcula resíduos para todas as equações
    residuals = np.zeros(len(self.equations))
    for i, eq in enumerate(self.equations):
        residuals[i] = eq()

    self.residuals = residuals
    return residuals
```

Esse método é chamado a cada iteração pelo *solver*, que utiliza os resíduos para ajustar as variáveis e avançar na convergência do sistema.

3.4.4.3 *Montagem da Matriz Jacobiana*

O método *calculate_jacobian* constrói analiticamente a matriz Jacobiana **J** através da chamada sistemática das funções de derivada de cada componente e conexão. O procedimento segue a seguinte estrutura:

- Inicialização da matriz Jacobiana com zeros, de dimensão correspondente ao número de equações por número de variáveis;

- Iteração sobre todos os componentes do sistema para executar as funções derivadas associadas aos parâmetros e restrições. Essas funções preenchem as respectivas posições na matriz Jacobiana;
- Iteração sobre todas as conexões do sistema, chamando as funções derivadas específicas das equações que representam as condições de conexão;
- Retorno da matriz Jacobiana resultante, que será utilizada no processo iterativo de solução do sistema.

A seguir, um trecho do código que implementa esse procedimento:

Trecho de código 3.38 – Cálculo da matriz Jacobiana

```
def calculate_jacobian(self):
    num_vars = len(self.variables)
    num_eqs = len(self.equations)
    self.jacobian = np.zeros((num_eqs, num_vars))

    for comp_name, comp in self.components.items():
        for param_name, param_data in comp.parameters.items():
            if "deriv" in param_data and param_data["deriv"] is not None:
               eq_index = param_data.get("eq_index")
               if eq_index is not None:
                   param_data["deriv"](self)

        if hasattr(comp, 'constraints'):
            for constraint_name, constraint_data in comp.constraints.items():
                if "deriv" in constraint_data and constraint_data["deriv"] is not None:
                    eq_index = constraint_data.get("eq_index")
                    if eq_index is not None:
                        constraint_data["deriv"](self)

    for conn in self.connections:
        if hasattr(conn, 'get_connection_equations'):
            conn_equations = conn.get_connection_equations()
            for eq_name, eq_data in conn_equations.items():
                if "deriv" in eq_data and eq_data["deriv"] is not None:
                    eq_data["deriv"](self)

    return self.jacobian
```

3.4.4.4 *Solução do Sistema Linear*

A cada iteração do método de Newton-Raphson, o sistema linear

$$\mathbf{J}\,\Delta\mathbf{x} = -\mathbf{f} \tag{3.48}$$

é resolvido para obter o vetor de incremento $\Delta\mathbf{x}$ das variáveis do sistema através da função *np.linalg.solve* da biblioteca Numpy. A solução desse sistema permite atualizar as variáveis e avançar na convergência do método.

3.4.4.5 Atualização das Variáveis

O método *update_variables* realiza a atualização dos valores das variáveis do modelo com base no vetor solução x obtido na iteração atual. Para cada variável, o valor correspondente no vetor é atribuído ao parâmetro adequado, considerando a sua hierarquia dentro do sistema: parâmetros associados a grupos de fluxo, conexões ou componentes.

A atualização numérica pode ser expressa como

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha \, \Delta \mathbf{x}^{(k)}, \tag{3.49}$$

onde α é um fator de sub-relaxação utilizado para controlar o passo de atualização e garantir estabilidade no processo iterativo.

Em termos de implementação, o método percorre todas as variáveis, atualizando os valores correspondentes nos objetos de conexão ou componente, conforme o tipo de variável, garantindo assim a consistência do modelo para a próxima iteração.

3.4.4.6 Verificação de Convergência

O algoritmo monitora o resíduo máximo das equações e utiliza critérios de convergência baseados em uma tolerância de resíduos predefinida. Caso o resíduo fique abaixo desse limite, considera-se que a solução convergiu. Além disso, são implementados mecanismos para detectar divergências, permitindo a interrupção antecipada do processo iterativo quando necessário.

3.4.4.7 Algoritmo Principal de Solução

O método *solve* integra todos os elementos anteriores em um algoritmo robusto de solução, implementando o ciclo completo do método de Newton-Raphson com verificações de convergência e divergência, conforme apresentado no Trecho de código 3.39.

Trecho de código 3.39 – Trecho do método principal de solução do sistema

```
def solve(self, max_iterations=20, tolerance=1e-10, global_guess=None,
          connection_guess=None):
    """Resolve o sistema usando método de Newton com Jacobiano analítico."""

    # Inicializa variáveis e equações
    self.setup_variables()
    self.setup_equations()

    # Verifica se o sistema está adequadamente restrito
    num_eqs = len(self.equations)
    num_vars = len(self.variables)

    if num_eqs != num_vars:
        print(f"Warning: System has {num_vars} variables but {num_eqs} equations")

    # Obtém estimativa inicial
    x = self.initial_guess(global_guess=global_guess,
                           connection_guess=connection_guess)

    converged = False
```

```
    # Loop de iterações de Newton
    for iteration in range(max_iterations):
        # Calcula resíduos
        residuals = self.calculate_residuals(x)
        max_residual = np.max(np.abs(residuals))

        # Verifica convergência
        if max_residual < tolerance:
            converged = True
            print(f"System converged after {iteration + 1} iterations!")
            break

        # Calcula Jacobiano analítico
        self.calculate_jacobian()

        try:
            # Resolve sistema linear J * dx = -residuals
            dx = np.linalg.solve(self.jacobian, -residuals)

            # Aplica sub-relaxação se necessário
            alpha = 1.0
            if iteration > 5 and max_residual > 1e3:
                alpha = 0.5

            # Atualiza variáveis
            x = x + alpha * dx
            self.update_variables(x)

        except np.linalg.LinAlgError:
            print("Error: Singular Jacobian matrix.")
            break

    return converged, x
```

3.4.4.8 *Pós-Processamento*

Após a convergência do processo iterativo de solução, o sistema realiza automaticamente o pós-processamento dos resultados obtidos. Essa etapa é fundamental para consolidar e organizar as informações calculadas, garantindo que todas as propriedades termodinâmicas e parâmetros relevantes estejam atualizados e disponíveis para análise.

Inicialmente, o método *update_all_properties_from_pressure_and_enthalpy()* é chamado para recalcular as propriedades termodinâmicas, como temperatura e título, a partir da solução da pressão e entalpia nas conexões do sistema. Em seguida, o método *calc_parameters()* é chamado em cada componente individualmente. Esse método pertence aos objetos da classe *Component* e é responsável por calcular parâmetros que não fazem parte diretamente do vetor de solução, mas que são indispensáveis para uma avaliação completa do sistema. Além disso, nesse estágio, cada parâmetro resolvido tem seus indicadores de estado atualizados, como o *is_unknown* para *False* e *is_solved* para *True*.

Por fim, o método *print_parameters()* gera uma saída formatada dos resultados, organizando as informações em tabelas para conexões e componentes. Essa apresentação facilita a visualização dos dados finais, permitindo ao usuário verificar rapidamente os valores

calculados e analisar o comportamento do sistema modelado.

3.5 Componentes implementados

O *framework* desenvolvido implementa diversos componentes termodinâmicos através de subclasses especializadas da classe *Component*. Cada componente herda a estrutura base e implementa suas equações características, parâmetros e equações específicos. Esta seção apresenta os principais componentes implementados, destacando suas funcionalidades distintas.

3.5.1 Componentes básicos

3.5.1.1 CycleCloser

O componente *CycleCloser*, ilustrado na Figura 9, é utilizado para fechar ciclos termodinâmicos, garantindo a continuidade das propriedades termodinâmicas entre a saída e a entrada do ciclo. Ele impõe duas condições fundamentais:

- Igualdade de entalpia: assegura que a entalpia na saída do ciclo seja igual à entalpia na entrada, mantendo a energia interna do fluido constante no fechamento do ciclo;
- Igualdade de pressão: assegura que a pressão na saída seja igual à pressão na entrada, garantindo consistência na pressão de fechamento do ciclo.

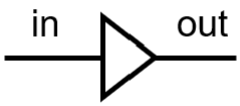

Figura 9 – Representação do componente *CycleCloser*.

3.5.1.1.1 Inicialização da classe

A inicialização dessa subclasse pode ser vista no trecho de código abaixo.

Trecho de código 3.40 – Inicialização da classe *CycleCloser*

```
class CycleCloser(Component):

    def __init__(self, name, **kwargs):

        super().__init__(name, **kwargs)

        self.add_constraint("enthalpy_equality", func=self.enthalpy_equality_equation,
                            deriv=self.enthalpy_equality_deriv, num_eq=1)
        self.add_constraint("pressure_equality", func=self.pressure_equality_equation,
                            deriv=self.pressure_equality_deriv, num_eq=1)
```

3.5.1.1.2 Igualdade de entalpia

A equação da igualdade de entalpia é dada por:

$$f = h_{in} - h_{out} = 0 \tag{3.50}$$

onde h_{in} e h_{out} são as entalpias na entrada e saída, respectivamente.

As derivadas analíticas da equação residual em relação às variáveis são apresentadas a seguir. Essas derivadas são utilizadas na construção da matriz Jacobiana do sistema de equações não lineares resolvido pelo método de Newton-Raphson:

$$\frac{\partial f}{\partial h_{in}} = 1, \quad \frac{\partial f}{\partial h_{out}} = -1 \tag{3.51}$$

3.5.1.1.3 Igualdade de pressão

A equação da igualdade de pressão é dada por:

$$f = p_{in} - p_{out} = 0 \tag{3.52}$$

onde p_{in} e p_{out} são as pressões na entrada e saída, respectivamente.

As derivadas parciais são:

$$\frac{\partial f}{\partial p_{in}} = 1, \quad \frac{\partial f}{\partial p_{out}} = -1 \tag{3.53}$$

3.5.1.2 Sink

O componente *Sink*, mostrado na Figura 10, representa um ponto de descarga no sistema termodinâmico, o sumidouro, onde o fluido entra no componente, mas não possui saída. Ele é utilizado para modelar pontos em que parâmetros do fluido são removidos do sistema, como descargas para o ambiente ou para um reservatório.

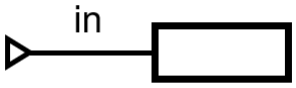

Figura 10 – Representação do componente *Sink*.

3.5.1.2.1 Inicialização da classe

A inicialização desse componente pode ser vista no trecho de código a seguir:

Trecho de código 3.41 – Inicialização da classe *Sink*

```
class Sink(Component):

    def __init__(self, name, **kwargs):
```

```
        super().__init__(name, **kwargs)

        self.inlets = {"in": None}  # Entrada única padrão
        self.outlets = {}           # Sem saída
```

O *Sink* não possui variáveis nem equações adicionais específicas, atuando apenas como um ponto de sumidouro no ciclo com parâmetros termodinâmicos como pressão, entalpia e temperatura que devem ser adicionados pelo método *add_parameter* na conexão associada ao sumidouro. Já para determinar as características do escoamento, o método *set_flow* implementado na classe *System* deve ser utilizado para determinar parâmetros como a vazão mássica, *m_dot*, e especificar o fluido.

3.5.1.3 *Source*

O componente *Source*, ilustrado na Figura 11, representa um ponto de fornecimento de fluido no sistema termodinâmico. Ele possui uma saída única, por onde o fluido entra no ciclo, e não possui entrada. É utilizado para modelar fontes externas de fluido, como reservatórios, alimentações ou condições de contorno.

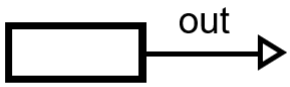

Figura 11 – Representação do componente *Source*.

3.5.1.3.1 *Inicialização da classe*

A inicialização desse componente é apresentada no trecho de código abaixo.

Trecho de código 3.42 – Inicialização da classe *Source*

```
class Source(Component):

    def __init__(self, name, **kwargs):

        super().__init__(name, **kwargs)

        self.inlets = {}              # Sem entradas
        self.outlets = {"out": None} # Saída única padrão
```

Por não possuir entrada, o *Source* não estabelece equações internas adicionais e serve como ponto de inserção do fluido no sistema. A definição das propriedades do fluido na saída deve ser feita pelo método *add_parameter* na conexão associada ao *Source* e pela função *set_flow* implementada na classe *System*, garantindo as condições iniciais para a simulação.

3.5.2 *Trocadores de calor*

Trocadores de calor são dispositivos responsáveis por transferir energia térmica entre dois fluidos a diferentes temperaturas, sem que ocorra mistura entre eles. São amplamente utilizados em sistemas térmicos e termodinâmicos, como ciclos de refrigeração.

No *framework* desenvolvido, foram implementadas duas classes distintas de trocadores de calor:

- *Simple_hex*: modelo simplificado, unidirecional, com apenas uma corrente. Serve para testes e análises iniciais simplificadas.
- *Heat_exchanger*: modelo mais completo, com duas correntes, quente e fria, que permite cálculos de efetividade, condutância térmica, perdas de carga para ambos os fluidos, e diferença média logarítmica de temperatura (LMTD).

3.5.2.1 *Simple_hex*

O *Simple_hex* é um modelo básico de trocador de calor com uma única corrente de fluido. Ele possui uma entrada, *in*, e uma saída, *out*, como apresentado na Figura 12. Esse componente pode funcionar tanto como um sumidouro quanto como uma fonte de calor, dependendo do sinal do parâmetro da taxa de transferência de calor, $\dot{Q}$. Quando $\dot{Q}$ é positivo, ($h_{out} > h_{in}$), o trocador de calor adiciona energia ao fluido, sendo uma fonte de calor. Já quando $\dot{Q}$ é negativo, ($h_{out} < h_{in}$), o trocador de calor remove energia do fluido, atuando como um sumidouro de calor.

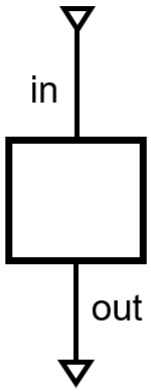

Figura 12 – Representação do componente *Simple_hex*.

3.5.2.1.1 *Inicialização da classe*

O componente *Simple_hex* possui como principais parâmetros a taxa de transferência de calor ($\dot{Q}$) e a razão de pressão (*pr*). Estes parâmetros estão associados às respectivas equações de balanço de energia (*energy balance*) e de razão de pressão (*pressure ratio*). A Listagem 3.43 apresenta a inicialização da classe.

Trecho de código 3.43 – Inicialização da classe *Simple_hex*

```
class Simple_hex(Component):

    def __init__(self, name, **kwargs):

        super().__init__(name,**kwargs)
```

```
self.add_parameter("Q_dot", value=kwargs.get('Q_dot', None),
                   func=self.energy_balance_equation,
                   deriv=self.energy_balance_deriv,
                   num_eq=1)
self.add_parameter("pr", value=kwargs.get('pr', None), min_val=0,
                   func=self.pressure_ratio_equation,
                   deriv=self.pressure_ratio_deriv,
                   num_eq=1)
```

3.5.2.1.2 *Balanço de energia*

O balanço de energia garante que a quantidade de energia transferida para o fluido corresponde à variação de entalpia entre as condições de entrada e saída. A formulação residual da equação é dada por:

$$f_{energy\_balance} = \dot{m}(h_{out} - h_{in}) - \dot{Q} \tag{3.54}$$

As derivadas parciais da equação residual, obtidas de forma analítica, são:

$$\frac{\partial f_{energy\_balance}}{\partial \dot{m}} = h_{out} - h_{in} \tag{3.55}$$

$$\frac{\partial f_{energy\_balance}}{\partial h_{in}} = -\dot{m} \tag{3.56}$$

$$\frac{\partial f_{energy\_balance}}{\partial h_{out}} = \dot{m} \tag{3.57}$$

$$\frac{\partial f_{energy\_balance}}{\partial \dot{Q}} = -1 \tag{3.58}$$

3.5.2.1.3 *Razão de pressão*

A equação de razão de pressão descreve de forma adimensional a perda de pressão ao longo do trocador de calor, representada por:

$$f_{pressure\_ratio} = -p_{out} + pr \cdot p_{in}$$

As derivadas parciais analíticas em relação às variáveis associadas são dadas por:

$$\frac{\partial f_{pressure\_ratio}}{\partial p_{in}} = pr \tag{3.59}$$

$$\frac{\partial f_{pressure\_ratio}}{\partial p_{out}} = -1 \tag{3.60}$$

$$\frac{\partial f_{pressure\_ratio}}{\partial pr} = p_{in} \tag{3.61}$$

3.5.2.2 *Heat_exchanger*

O *Heat_exchanger* representa um trocador de calor com duas correntes distintas: uma quente e outra fria. O trocador possui duas entradas *(in, in_2)* e duas saídas *(out, out_2)*, que

correspondem aos dois lados do trocador, o fluido quente e o fluido frio, como ilustrado na Figura 13. O componente implementa cálculos adicionais como condutância térmica global (UA), diferença de temperatura logarítmica ($\Delta T_{\log}$), efetividade e balanços de energia em ambas as correntes. Neste modelo, adota-se a configuração de fluxo contracorrente, onde os fluidos escoam em direções opostas.

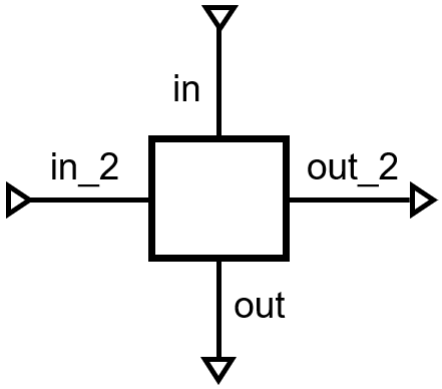

Figura 13 – Representação do componente *Heat_exchanger*.

3.5.2.2.1 Inicialização da classe

O componente *Heat_exchanger* apresenta uma estrutura mais complexa comparado ao *Simple_hex*, incorporando múltiplos parâmetros para uma caracterização mais completa do trocador de calor:

- $\dot{Q}$: Taxa de transferência de calor, calculada através do balanço de energia do lado quente;
- *pr, pr_2*: Razões de pressão para ambos os lados do trocador, modelando perdas de carga;
- *UA*: Condutância térmica global (produto do coeficiente global de transferência de calor pela área);
- *ttd_u, ttd_l*: Diferenças de temperatura terminais superior e inferior, respectivamente;
- *ttd_min*: Menor diferença de temperatura no trocador;
- *td_log*: Diferença de temperatura média logarítmica (LMTD);
- *eff_cold, eff_hot*: Efetividades dos lados frio e quente, respectivamente, limitadas entre 0 e 1.

A inicialização desse componente é apresentada a seguir:

Trecho de código 3.44 – Inicialização da classe *Heat_exchanger*

```
class Heat_exchanger(Component):

    def __init__(self, name, **kwargs):

        super().__init__(name,**kwargs)

        self.inlets = {"in": None, "in_2": None}
        self.outlets = {"out": None, "out_2": None}

        self.add_constraint("energy_balance",  func=self.energy_balance_equation, deriv=
            self.energy_balance_deriv, num_eq=1)
        self.add_parameter("Q_dot", value=kwargs.get('Q_dot', None), func=self.
            energy_balance_hot_equation, deriv=self.energy_balance_hot_deriv, num_eq=1)
        self.add_parameter("pr", value=kwargs.get('pr', None),min_val=0,  func=self.
            pressure_ratio_equation, deriv=self.pressure_ratio_deriv, num_eq=1)
        self.add_parameter("pr_2", value=kwargs.get('pr_2', None),min_val=0,  func=self.
            pressure_ratio_2_equation, deriv=self.pressure_ratio_2_deriv, num_eq=1)
        self.add_parameter("UA", value=kwargs.get('UA', None),min_val=0,  func=self.
            UA_equation, deriv=self.UA_deriv, num_eq=1)

        self.add_parameter("td_log", value=kwargs.get('td_log', None),min_val=0,  func=
            self.calc_td_log, deriv=None, num_eq=0)
        self.add_parameter("ttd_u", value=kwargs.get('ttd_u', None),min_val=0,  func=self
            .ttd_u_equation, deriv=self.ttd_u_deriv, num_eq=1)
        self.add_parameter("ttd_l", value=kwargs.get('ttd_l', None),min_val=0,  func=self
            .ttd_l_equation, deriv=self.ttd_l_deriv, num_eq=1)
        self.add_parameter("ttd_min", value=kwargs.get('ttd_min', None),min_val=0,  func=
            self.ttd_min_equation, deriv=self.ttd_min_deriv, num_eq=1)

        self.add_parameter("eff_cold", value=kwargs.get('eff_cold', None),min_val=0,
            max_val = 1, func=self.eff_cold_equation, deriv=self.eff_cold_deriv, num_eq
            =1)
        self.add_parameter("eff_hot", value=kwargs.get('eff_hot', None),min_val=0,
            max_val = 1, func=self.eff_hot_equation, deriv=self.eff_hot_deriv, num_eq=1)
```

A equação *energy_balance*, definida como *constraint*, garante a conservação global de energia entre as duas correntes, enquanto os demais parâmetros permitem a análise do desempenho térmico e dimensionamento do equipamento.

3.5.2.2.2 Balanço de energia global

A formulação matemática para o resíduo da equação do balanço de energia global é dada por:

$$f_{energy\_balance} = \dot{m}(h_{out} - h_{in}) + \dot{m}_2(h_{out,2} - h_{in,2}) \tag{3.62}$$

Nesta expressão, seguindo o padrão das portas de entrada e saída, o primeiro termo representa o lado quente do trocador de calor, com a vazão mássica $\dot{m}$ e entalpias na entrada e saída (h_{in} e h_{out}, respectivamente). Já o segundo termo representa o lado frio, indicado pelo subscrito 2, com a vazão mássica $\dot{m}_2$ e entalpias correspondentes à entrada e saída ($h_{in,2}$ e $h_{out,2}$).

As expressões analíticas das derivadas parciais em relação às variáveis associadas são apresentadas a seguir:

$$\frac{\partial f_{energy\_balance}}{\partial \dot{m}} = h_{out} - h_{in} \qquad \frac{\partial f_{energy\_balance}}{\partial h_{in}} = -\dot{m} \qquad \frac{\partial f_{energy\_balance}}{\partial h_{out}} = \dot{m} \tag{3.63}$$

$$\frac{\partial f_{energy\_balance}}{\partial \dot{m}_2} = h_{out,2} - h_{in,2} \qquad \frac{\partial f_{energy\_balance}}{\partial h_{in,2}} = -\dot{m}_2 \qquad \frac{\partial f_{energy\_balance}}{\partial h_{out,2}} = \dot{m}_2 \tag{3.64}$$

3.5.2.2.3 Balanço de energia do lado quente

Para auxiliar o cálculo de propriedades como a taxa de transferência de calor ($\dot{Q}$), o modelo também implementa um balanço de energia específico apenas para o lado quente do trocador de calor, com o objetivo de evitar a introdução de uma equação redundante do lado frio. Como o modelo já inclui a equação de balanço de energia global, a conservação de energia entre os dois lados do trocador já está garantida. Caso fossem incluídas equações de balanço de energia para ambos os lados, o sistema de equações resultante seria sobredeterminado, ou seja, teria mais equações do que variáveis, inviabilizando a solução numérica. A equação desse balanço é dada por: :

$$f_{hot\_energy\_balance} = \dot{m}_1(h_{\text{out},1} - h_{\text{in},1}) - \dot{Q} \tag{3.65}$$

As derivadas parciais analíticas da equação são:

$$\frac{\partial f_{hot\_energy\_balance}}{\partial \dot{m}} = h_{out} - h_{in} \tag{3.66}$$

$$\frac{\partial f_{hot\_energy\_balance}}{\partial h_{in}} = -\dot{m} \tag{3.67}$$

$$\frac{\partial f_{hot\_energy\_balance}}{\partial h_{out}} = \dot{m} \tag{3.68}$$

$$\frac{\partial f_{hot\_energy\_balance}}{\partial \dot{Q}} = -1 \tag{3.69}$$

3.5.2.2.4 Razões de pressão em ambas as correntes

Para cada lado do trocador de calor, foi implementada uma equação para representar a razão de pressão entre os pontos de entrada e saída. A equação de razão de pressão do lado quente é formulada como:

$$f_{pr} = -p_{out} + p_{in} \cdot pr \tag{3.70}$$

Já a equação correspondente ao lado frio é:

$$f_{pr_2} = -p_{out,2} + p_{in,2} \cdot pr_2 \tag{3.71}$$

As derivadas parciais obtidas são:

$$\frac{\partial f_{pr}}{\partial p_{in}} = pr \qquad \frac{\partial f_{pr}}{\partial p_{out}} = -1 \qquad \frac{\partial f_{pr}}{\partial pr} = p_{in} \tag{3.72}$$

$$\frac{\partial f_{pr_2}}{\partial p_{in,2}} = pr_2 \qquad \frac{\partial f_{pr_2}}{\partial p_{out,2}} = -1 \qquad \frac{\partial f_{pr_2}}{\partial pr_2} = p_{in,2} \tag{3.73}$$

3.5.2.2.5 Diferenças de temperatura terminais

As diferenças de temperatura nas extremidades do trocador são fundamentais para a análise térmica e são calculadas através de equações específicas que relacionam as temperaturas dos fluidos quente e frio.

Para o cálculo da diferença de temperatura terminal superior e inferior, utilizam-se as seguintes definições:

- Diferença de temperatura terminal superior (ΔT_u) definida como *ttd_u*:

$$\Delta T_u = T_{in,\,(hot)} - T_{out,2\,(cold)} \tag{3.74}$$

- Diferença de temperatura terminal inferior (ΔT_l) definida como *ttd_l*:

$$\Delta T_l = T_{out,\,(hot)} - T_{in,2\,(cold)} \tag{3.75}$$

onde *hot* e *cold* representam os lados quente e frio, respectivamente.

Essas definições refletem a configuração contracorrente, na qual o fluido quente flui de sua entrada para sua saída, enquanto o fluido frio segue no sentido oposto. A diferença de temperatura terminal superior representa a maior diferença de temperatura entre os dois fluidos ao longo do trocador, enquanto a inferior representa a menor.

As equações de resíduo que representam essas diferenças, utilizadas no processo de solução numérica, são formuladas da seguinte maneira:

- Para a diferença de temperatura terminal superior (*ttd_u*):

$$f_{ttd_u} = ttd\_u - T_{in} + T_{out,2} = 0 \tag{3.76}$$

- Para a diferença de temperatura terminal inferior (*ttd_l*):

$$f_{ttd_l} = ttd\_l - T_{out} + T_{in,2} = 0 \tag{3.77}$$

- Para a diferença de temperatura mínima (*ttd_min*), que representa a menor das duas diferenças terminais:

$$f_{ttd\_min} = ttd\_min - \min(ttd\_u, ttd\_l) = 0 \tag{3.78}$$

As derivadas parciais dessas equações utilizam diferenciação numérica devido à complexidade envolvida no cálculo das temperaturas a partir das propriedades termodinâmicas com o método *calc_temperature* implementado na classe *Connection*.

3.5.2.2.6 Diferença de temperatura média logarítmica (LMTD)

A partir dessas diferenças de temperatura terminais, calcula-se a diferença de temperatura média logarítmica ($\Delta T_{\log}$), através do método auxiliar *calc_td_log()*, o qual implementa a seguinte equação:

$$\Delta T_{\log} = \frac{\Delta T_l - \Delta T_u}{\ln\left(\frac{\Delta T_l}{\Delta T_u}\right)} \tag{3.79}$$

Para evitar problemas de indeterminação matemática e garantir a estabilidade numérica durante a solução iterativa, a seguinte condição é aplicada:

$$\Delta T_{\log} = \begin{cases} \text{ttd}_l & \text{se } \text{ttd}_u = \text{ttd}_l \\ \frac{\text{ttd}_l - \text{ttd}_u}{\ln\left(\frac{\text{ttd}_l}{\text{ttd}_u}\right)} & \text{se } \text{ttd}_u \neq \text{ttd}_l \end{cases}$$

Essa abordagem previne singularidades associadas ao denominador nulo que ocorre quando ΔT_u e ΔT_l são iguais.

3.5.2.2.7 Condutância térmica global (UA)

A equação da condutância térmica global relaciona a equação 3.65 da taxa de transferência de calor do lado quente, $\dot{Q}$, com a LMTD, ($\Delta T_{\log}$), através da seguinte equação:

$$f_{UA} = \dot{m}(h_{out} - h_{in}) + UA \cdot \Delta T_{\log} = 0 \tag{3.80}$$

Esta equação expressa que a taxa de transferência de calor do lado quente deve ser igual ao produto da condutância térmica global pela LMTD. O parâmetro UA representa o produto do coeficiente global de transferência de calor (U) pela área de troca térmica (A), sendo fundamental para o dimensionamento e a análise de performance do equipamento.

As derivadas parciais desta equação são calculadas utilizando diferenciação numérica para a maioria dos termos, exceto para as vazões mássicas, $\dot{m}$ onde as derivadas analíticas são:

$$\frac{\partial f_{UA}}{\partial \dot{m}} = h_{out} - h_{in} \tag{3.81}$$

$$\frac{\partial f_{UA}}{\partial \dot{m}_2} = h_{out,2} - h_{in,2} \tag{3.82}$$

O uso de diferenciação numérica para as demais derivadas se deve à complexidade do cálculo da LMTD, que envolve funções logarítmicas e condicionais, tornando as derivadas analíticas mais complexas de calcular.

3.5.2.2.8 Efetividade

A efetividade é uma grandeza adimensional definida como a razão entre a taxa de transferência de calor real em um trocador de calor e a taxa de transferência de calor máxima teoricamente possível.

Para o cálculo das efetividades do lado frio e quente, são definidas as seguintes variações máximas teóricas de entalpia:

$$\Delta h_{\max,cold} = h(T_{in,\,(hot)}, p_{out,2\,(cold)}) - h_{\text{in,2 (cold)}} \tag{3.83}$$

$$\Delta h_{\max,hot} = h_{in,\,(hot)} - h(T_{in,2\,(cold)}, p_{out,\,(hot)}) \tag{3.84}$$

onde p_{cold} e p_{hot} são as pressões de saída nos lados frio e quente, usadas para calcular as entalpias a partir das temperaturas de entrada opostas.

Essas grandezas representam as variações máximas de entalpia teoricamente possíveis nos lados frio e quente, assumindo que cada fluido poderia atingir a temperatura de entrada do fluido oposto, mantendo-se à sua respectiva pressão de saída. São obtidas por meio das funções *calc_dh_max_cold*, *calc_dh_max_hot*, que calculam os valores de $\Delta h_{\max}$.

A efetividade no lado frio é definida como:

$$\varepsilon_{cold} = \frac{\Delta h_{real,\,cold}}{\Delta h_{\max,cold}} = \frac{h_{out,\,cold} - h_{in,\,cold}}{\Delta h_{\max,cold}} \tag{3.85}$$

Para resolver o sistema numérico, utiliza-se a equação residual:

$$f_{eff\_cold} = \varepsilon_{cold} \cdot \Delta h_{\max,cold} - (h_{out,\,2\,(cold)} - h_{in,\,2\,(cold)}) = 0 \tag{3.86}$$

As derivadas parciais obtidas de forma analítica em relação às variáveis envolvidas podem ser expressas como:

$$\frac{\partial f_{eff\_cold}}{\partial h_{in,\,2\,(cold)}} = 1 - \varepsilon_{cold} \tag{3.87}$$

$$\frac{\partial f_{eff\_cold}}{\partial h_{out,\,2\,(cold)}} = -1 \tag{3.88}$$

Similarmente, a efetividade no lado quente é dada por:

$$\varepsilon_{hot} = \frac{\Delta h_{real,\,hot}}{\Delta h_{\max,hot}} = \frac{h_{out,\,hot} - h_{in,\,hot}}{\Delta h_{\max,hot}} \tag{3.89}$$

A equação residual correspondente é expressa por:

$$f_{eff\_hot} = \varepsilon_{hot} \cdot \Delta h_{\max,hot} - (h_{out,\,(hot)} - h_{in,\,(hot)}) = 0 \tag{3.90}$$

De forma análoga, as derivadas parciais obtidas de forma analítica são dadas por:

$$\frac{\partial f_{eff\_hot}}{\partial h_{in,\,(hot)}} = 1 + \varepsilon_{hot} \tag{3.91}$$

$$\frac{\partial f_{eff\_hot}}{\partial h_{out,\,(hot)}} = -1 \tag{3.92}$$

As derivadas em relação às pressões e entalpias que afetam o cálculo das variações máximas de entalpia são calculadas numericamente devido à complexidade das propriedades termodinâmicas.

3.5.3 Turbine

O componente *Turbine*, mostrado na Figura 14, representa uma turbina adiabática, ou seja, sem troca de calor com o ambiente, utilizada para converter energia do fluido em potência mecânica por meio do processo de expansão. No *framework*, essa turbina é modelada com base na eficiência isentrópica, balanço de energia e razão de pressão.

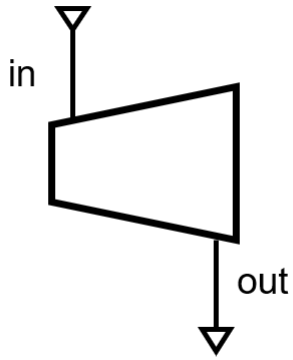

Figura 14 – Representação do componente *Turbine*.

3.5.3.1 Inicialização da classe

O componente *Turbine* é modelado com uma entrada (*in*) e uma saída *(out)* e apresenta os seguintes parâmetros para caracterização do processo de expansão:

- *eta_s*: Eficiência isentrópica, limitada entre 0 e 1, que compara o desempenho do processo real em relação à expansão isentrópica ideal
- *W_dot*: Potência gerada pela turbina, calculada através do balanço de energia
- *pr*: Razão de pressão (p_{out}/p_{in}), sendo sempre menor que 1 para turbinas devido à expansão do fluido

A inicialização desse componente pode ser vista no trecho de código a seguir:

Trecho de código 3.45 – Inicialização da classe *Turbine*

```
class Turbine(Component):

    def __init__(self, name, **kwargs):

        super().__init__(name,**kwargs)

        self.add_parameter("eta_s", value=kwargs.get('eta_s', None), min_val=0, max_val
            =1,
                           func=self.eta_s_equation, deriv=self.eta_s_deriv, num_eq=1)
        self.add_parameter("W_dot", value=kwargs.get('W_dot', None),
                           func=self.energy_balance_equation, deriv=self.
                               energy_balance_deriv, num_eq=1)
        self.add_parameter("pr", value=kwargs.get('pr', None), min_val=0,
                           func=self.pressure_ratio_equation, deriv=self.
                               pressure_ratio_deriv, num_eq=1)
```

3.5.3.2 Balanço de energia

O balanço de energia relaciona a potência líquida extraída do fluido pela turbina com a variação de entalpia do fluido:

$$f_{energy\_balance} = \dot{W} - \dot{m}(h_{in} - h_{out}) \tag{3.93}$$

Para turbinas, $h_{out} < h_{in}$, indicando a expansão do fluido, resultando em $\dot{W} > 0$, em que há geração de potência.

As derivadas parciais obtidas analiticamente são dadas por:

$$\frac{\partial f_{energy\_balance}}{\partial \dot{W}} = 1 \tag{3.94}$$

$$\frac{\partial f_{energy\_balance}}{\partial \dot{m}} = -(h_{\text{in}} - h_{out}) \tag{3.95}$$

$$\frac{\partial f_{energy\_balance}}{\partial h_{in}} = -\dot{m} \tag{3.96}$$

$$\frac{\partial f_{energy\_balance}}{\partial h_{out}} = \dot{m} \tag{3.97}$$

3.5.3.3 Razão de pressão

A razão de pressão é definida com a mesma equação e derivadas analíticas apresentadas em 3.5.2.1.3.

3.5.3.4 Eficiência isentrópica

A eficiência isentrópica relaciona o trabalho real ao trabalho ideal que seria obtido com uma expansão isentrópica:

$$\eta_s = \frac{h_{out} - h_{in}}{h_{out,s} - h_{in}} \tag{3.98}$$

onde:

- $h_{out,s}$: entalpia de saída isentrópica, calculada a partir de p_{in}, h_{in}, p_{out}, fluido
- η_s: eficiência isentrópica da turbina

Para o processo de solução numérica, a equação é reescrita na forma de resíduo:

$$f_{eta\_s} = -(h_{out} - h_{in}) + \eta_s \cdot (h_{out,s} - h_{in}) = 0 \tag{3.99}$$

As derivadas parciais analíticas são dadas por:

$$\frac{\partial f_{eta\_s}}{\partial h_{out}} = -1 \tag{3.100}$$

$$\frac{\partial f_{eta\_s}}{\partial \eta_s} = h_{out,s} - h_{in} \tag{3.101}$$

Como a entalpia de saída em um processo isentrópico $h_{out,s}$ é obtida via chamada à biblioteca CoolProp, suas derivadas parciais em relação a p_{in}, p_{out} e h_{in} são calculadas numericamente usando o método de diferenças finitas.

3.5.4 Compressor

O componente *Compressor*, apresentado na Figura 15, representa um compressor que aumenta a pressão de um fluido de trabalho através do fornecimento de trabalho mecânico, aumentando sua pressão e entalpia. O modelo considera uma compressão adiabática e implementa equações para eficiência isentrópica, balanço de energia e razão de pressão.

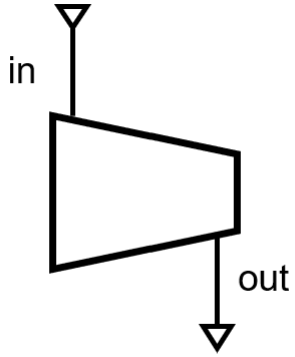

Figura 15 – Representação do componente *Compressor*.

3.5.4.1 Inicialização da classe

O componente *Compressor* é modelado com as portas de entrada e saída padrão da classe base *Component*, e apresenta os seguintes parâmetros para a caracterização do processo de compressão:

- *eta_s*: Eficiência isentrópica, limitada entre 0 e 1, que quantifica a diferença do processo real em relação à compressão isentrópica ideal;
- *W_dot*: Potência consumida pelo compressor, calculada através do balanço de energia;
- *pr*: Razão de pressão (p_{out}/p_{in}), sempre maior que 1 para compressores devido ao aumento de pressão do fluido.

A inicialização da subclasse *Compressor* é apresentada no trecho de código abaixo.

Trecho de código 3.46 – Inicialização da classe *Compressor*

```
class Compressor(Component):

    def __init__(self, name, **kwargs):

        super().__init__(name,**kwargs)

        self.add_parameter("eta_s", value=kwargs.get('eta_s', None), min_val=0, max_val
            =1,
                           func=self.eta_s_equation, deriv=self.eta_s_deriv, num_eq=1)
        self.add_parameter("W_dot", value=kwargs.get('W_dot', None),
                           func=self.energy_balance_equation, deriv=self.
                               energy_balance_deriv, num_eq=1)
        self.add_parameter("pr", value=kwargs.get('pr', None), min_val=0,
                           func=self.pressure_ratio_equation, deriv=self.
                               pressure_ratio_deriv, num_eq=1)
```

3.5.4.2 Balanço de energia

O balanço de energia relaciona a potência consumida com a variação de entalpia do fluido de trabalho:

$$f_{energy\_balance} = \dot{W} - \dot{m} \cdot (h_{out} - h_{in}) \tag{3.102}$$

Para compressores, $h_{out} > h_{in}$ em virtude do processo de compressão, resultando em $\dot{W} < 0$, de forma a representar o consumo de potência.

As derivadas parciais analíticas são:

$$\frac{\partial f_{energy\_balance}}{\partial W_{\text{dot}}} = 1 \tag{3.103}$$

$$\frac{\partial f_{energy\_balance}}{\partial \dot{m}} = -(h_{out} - h_{in}) \tag{3.104}$$

$$\frac{\partial f_{energy\_balance}}{\partial h_{in}} = \dot{m} \tag{3.105}$$

$$\frac{\partial f_{energy\_balance}}{\partial h_{out}} = -\dot{m} \tag{3.106}$$

3.5.4.3 Razão de pressão

A razão de pressão é definida com a mesma equação e derivadas analíticas apresentadas em 3.5.2.1.3.

3.5.4.4 Eficiência isentrópica

A eficiência isentrópica relaciona o trabalho ideal necessário para uma compressão isentrópica com o trabalho real fornecido ao compressor:

$$\eta_s = \frac{h_{out,s} - h_{in}}{h_{out} - h_{in}} \tag{3.107}$$

A equação residual implementada é reescrita como:

$$f_{eta\_s} = \eta_s \cdot (h_{out} - h_{in}) - (h_{out,s} - h_{in}) \tag{3.108}$$

onde:

- $h_{out,s}$: entalpia de saída para compressão isentrópica (calculada via função *isentropic_enthalpy* a partir de p_{in}, h_{in}, p_{out}, fluido);
- η_s: eficiência isentrópica (valor entre 0 e 1).

As derivadas parciais obtidas de forma analítica são:

$$\frac{\partial f_{eta\_s}}{\partial h_{out}} = \eta_s \tag{3.109}$$

$$\frac{\partial f_{eta\_s}}{\partial \eta_s} = h_{out} - h_{in} \tag{3.110}$$

Como a entalpia de saída de um processo isentrópico $h_{out,s}$ é obtida por meio da combinação de outras propriedades termodinâmicas por meio da biblioteca CoolProp, suas derivadas parciais em relação a p_{in}, p_{out} e h_{in} são calculadas numericamente usando o método de diferenças finitas.

3.5.5 Simple_pump

O componente *Simple_pump*, apresentado na Figura 16, representa uma bomba que consome trabalho mecânico para aumentar a pressão do fluido, assumindo uma operação adiabática. O componente implementa modelos idênticos aos do compressor para eficiência isentrópica, balanço de energia e razão de pressão.

Figura 16 – Representação do componente *Simple_pump*.

3.5.5.1 Inicialização da classe

O componente *Simple_pump* incorpora os mesmos parâmetros do compressor para caracterização do processo de bombeamento:

- *eta_s*: Eficiência isentrópica, limitada entre 0 e 1, que quantifica o desvio do processo real em relação ao bombeamento isentrópico ideal
- *W_dot*: Potência consumida pela bomba, calculada através do balanço de energia
- *pr*: Razão de pressão (p_{out}/p_{in}), sempre maior que 1 para bombas devido ao aumento de pressão do fluido

A inicialização da subclasse *Simple_pump* é apresentada a seguir:

Trecho de código 3.47 – Inicialização da classe *Simple_pump*

```
class Simple_pump(Component):

    def __init__(self, name, **kwargs):

        super().__init__(name, **kwargs)

        self.add_parameter("eta_s", value=kwargs.get('eta_s', None), min_val=0, max_val
            =1,
                           func=self.eta_s_equation, deriv=self.eta_s_deriv, num_eq=1)
        self.add_parameter("W_dot", value=kwargs.get('W_dot', None),
                           func=self.energy_balance_equation, deriv=self.
                               energy_balance_deriv, num_eq=1)
        self.add_parameter("pr", value=kwargs.get('pr', None), min_val=0,
                           func=self.pressure_ratio_equation, deriv=self.
                               pressure_ratio_deriv, num_eq=1)
```

3.5.5.2 Balanço de energia

O balanço de energia da bomba é idêntico ao do compressor, seguindo a mesma equação e derivadas analíticas apresentadas na seção 3.5.4.2 do componente *Compressor*.

3.5.5.3 Razão de pressão

A razão de pressão é definida com a mesma equação e derivadas analíticas apresentadas em 3.5.2.1.3.

3.5.5.4 Eficiência isentrópica

A eficiência isentrópica da bomba segue a mesma formulação apresentada para o compressor, conforme definido na seção 3.5.4.4 da eficiência isentrópica do compressor.

3.5.6 Valve

O componente *Valve*, mostrado na Figura 17, representa uma válvula de expansão que reduz a pressão do fluido através de um processo isentálpico, ou seja, em que a entalpia permanece constante. Este componente é fundamental em ciclos de refrigeração e outros sistemas onde é necessária uma expansão controlada do fluido sem realização de trabalho mecânico.

Figura 17 – Representação do componente *Valve*.

3.5.6.1 Inicialização da classe

O componente *Valve* incorpora os seguintes parâmetros para caracterização do processo de expansão:

- *pr*: Razão de pressão (p_{out}/p_{in}), limitada entre 0 e 1 devido à redução de pressão característica de válvulas;
- *enthalpy_equality*: Restrição que garante a conservação de entalpia durante o processo de expansão.

A inicialização é apresentada no trecho de código abaixo.

Trecho de código 3.48 – Inicialização da classe *Valve*

```
class Valve(Component):

    def __init__(self, name, **kwargs):
```

```
        super().__init__(name, **kwargs)

        self.add_parameter("pr", value=kwargs.get('pr', None), min_val=0, max_val=1,
                           func=self.pressure_ratio_equation, deriv=self.
                               pressure_ratio_deriv, num_eq=1)
        self.add_constraint("enthalpy_equality", func=self.enthalpy_equality_equation,
                            deriv=self.enthalpy_equality_deriv, num_eq=1)
```

3.5.6.2 *Razão de pressão*

A razão de pressão para válvulas que representa uma redução de pressão é definida com a mesma equação e derivadas analíticas apresentadas em 3.5.2.1.3.

3.5.6.3 *Igualdade de entalpia*

A restrição de igualdade de entalpia caracteriza o processo isentálpico típico de válvulas de expansão, assegurando que a entalpia permaneça constante durante a expansão. O equilíbrio de entalpia é modelado do mesmo modo feito na seção 3.5.1.1.2 para o componente *Cycle_closer*.

4 Resultados

Nesta seção, são apresentados os resultados obtidos com o *framework* de simulação desenvolvido. O objetivo principal é validar a capacidade do modelo simplificado em representar corretamente diferentes ciclos termodinâmicos, por meio da comparação com casos de referência disponíveis na documentação do TESPy.

Para isso, foram escolhidos dois ciclos com características distintas: uma bomba de calor, representando um sistema de refrigeração, e um ciclo Rankine, representando um ciclo de potência. Inicialmente, serão descritas as simulações realizadas para cada um desses casos, incluindo as condições de contorno, os componentes utilizados e os principais resultados obtidos. Em seguida, são apresentadas análises paramétricas para cada um dos casos, explorando a influência de diferentes variáveis de entrada sobre o desempenho dos ciclos simulados e demonstrando a consistência do modelo desenvolvido com os comportamentos termodinâmicos esperados.

4.1 Simulação da Bomba de Calor

4.1.1 Descrição do Sistema

A bomba de calor opera segundo o ciclo de compressão de vapor, cuja finalidade é transferir energia térmica de uma fonte a baixa temperatura para um reservatório a temperatura mais elevada, mediante o fornecimento de trabalho mecânico ao sistema. O ciclo básico da bomba de calor é composto por quatro processos principais: compressão isentrópica do fluido refrigerante, aumentando sua pressão e temperatura, rejeição de calor no condensador, expansão isentálpica através da válvula de expansão, reduzindo a pressão e temperatura do fluido, e absorção de calor no evaporador.

Para a validação do *framework* desenvolvido, foi selecionado o caso de estudo do tutorial de bomba de calor operando em regime permanente disponível na documentação oficial do TESPy (TESPY, 2025a). O sistema é constituído no modelo por cinco componentes principais:

- Compressor (*Compressor*) : O compressor é responsável pela elevação da pressão e temperatura do fluido refrigerante. No modelo, este componente é caracterizado pela eficiência isentrópica (η_s), que relaciona o trabalho isentrópico ideal com o trabalho real necessário para a compressão;

- Condensador (*Simple_hex*): O condensador atua como trocador de calor responsável pela rejeição de energia térmica do sistema para o ambiente externo. Durante este processo, o fluido refrigerante condensa, passando do estado de vapor superaquecido para líquido saturado;

- Válvula de Expansão (*Valve*): A válvula de expansão promove a redução de pressão do fluido refrigerante através de um processo de expansão isentálpico;

- Evaporador (*Simple_hex*): O evaporador funciona como trocador de calor onde ocorre a absorção de energia térmica da fonte fria, promovendo a evaporação do fluido refrigerante;
- *Cycle Closer* (*CycleCloser*): Componente fictício cuja função é fechar o ciclo termodinâmico, garantindo a conservação do escoamento do fluido de trabalho entre o final e o início do ciclo.

A topologia do sistema de bomba de calor utilizada para esta validação é apresentada na Figura 18.

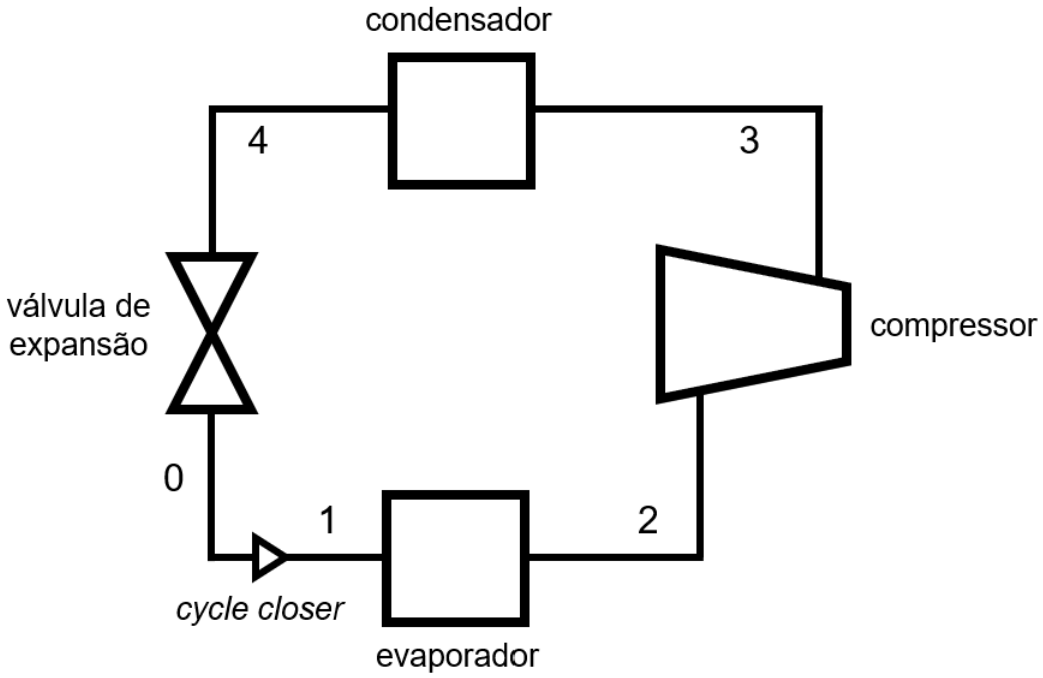

Figura 18 – Representação da topologia da bomba de calor.

4.1.1.1 Condições de Contorno e Parâmetros de Simulação

A Tabela 1 apresenta os principais parâmetros de entrada adotados na simulação, reproduzindo as condições estabelecidas no caso de referência do TESPy.

Tabela 1 – Condições de contorno e parâmetros da simulação da bomba de calor.

Parâmetro	**Valor**
Fluido de trabalho	R134a
Temperatura de entrada no compressor	20 °C
Título na entrada do compressor	1 (vapor saturado)
Temperatura de saída do condensador	80 °C
Título na saída do condensador	0 (líquido saturado)
Eficiência isentrópica do compressor	85 %
Razão de pressão no condensador	0,98
Razão de pressão no evaporador	0,98
Potência térmica rejeitada no condensador	-1 MW

4.1.1.2 Conexões do Sistema

A configuração das conexões entre os componentes, implementada tanto no TESPy quanto no *framework* desenvolvido, é apresentada na Tabela 2. Esta tabela descreve o caminho do fluido através dos diferentes componentes do ciclo.

Tabela 2 – Conexões do sistema de bomba de calor.

Conexão	**De**	**Para**
1	*Cycle Closer* (saída)	Evaporador (entrada)
2	Evaporador (saída)	Compressor (entrada)
3	Compressor (saída)	Condensador (entrada)
4	Condensador (saída)	Válvula de expansão (entrada)
0	Válvula de expansão (saída)	*CycleCloser* (entrada)

4.1.2 Implementação no Modelo

A implementação do sistema no *framework* desenvolvido segue uma arquitetura orientada a objetos que permite a criação modular e flexível de sistemas termodinâmicos. O processo de construção do modelo compreende as seguintes etapas:

4.1.2.1 Importação das classes necessárias

Para simular a bomba de calor, é necessário importar os cinco componentes que constituem o ciclo, representados por 4 subsclasses (*Compressor*, *Simple_hex*, *Valve* e *CycleCloser*), além das classes *Connection* e *System*. Essa processo de importação de módulos é apresentado no Trecho de código 4.1.

Trecho de código 4.1 – Importação dos módulos do *framework*

```
from thermo_sim.components.compressor import Compressor
from thermo_sim.components.heat_exchangers import Simple_hex
from thermo_sim.components.valves import Valve
from thermo_sim.connections import Connection
from thermo_sim.components.basics.cycle_closer import CycleCloser
from thermo_sim.system import System
```

4.1.2.2 Instanciação dos componentes

Cada componente é criado como uma instância de sua respectiva classe, permitindo a configuração de parâmetros específicos durante a inicialização. Os parâmetros também podem ser adicionados pelo método *set_parameter* posteriormente. Esse processo é demonstrado no Trecho de código 4.2.

Trecho de código 4.2 – Criação dos componentes da bomba de calor

```
cycle_closer = CycleCloser("Cycle Closer")
condenser = Simple_hex("Condenser", Q_dot=-1e6, pr=0.98)
expansion_valve = Valve("Expansion valve")
evaporator = Simple_hex("Evaporator", pr=0.98)
compressor = Compressor("Compressor", eta_s=0.85)
```

4.1.2.3 Definição das conexões entre componentes

As conexões estabelecem as ligações físicas entre os componentes, definindo o caminho do fluido através do sistema. Cada conexão especifica explicitamente os componentes e suas portas de origem e destino conforme a Tabela 2. Como o sistema modelado possui apenas um único circuito de fluido, não é necessário especificar o parâmetro *flow_group* nas conexões. As conexões são definidas pelo modelo de acordo com o Trecho de código 4.3.

Trecho de código 4.3 – Definição das conexões do sistema

```
c1 = Connection(source=cycle_closer, source_outlet="out",
target=evaporator, target_inlet="in", label="1")
c2 = Connection(source=evaporator, source_outlet="out",
target=compressor, target_inlet="in", label="2")
c3 = Connection(source=compressor, source_outlet="out",
target=condenser, target_inlet="in", label="3")
c4 = Connection(source=condenser, source_outlet="out",
target=expansion_valve, target_inlet="in", label="4")
c0 = Connection(source=expansion_valve, source_outlet="out",
target=cycle_closer, target_inlet="in", label="0")
```

4.1.2.4 Criação e configuração do sistema

O sistema atua como o núcleo principal que gerencia todos os componentes e conexões, coordenando os cálculos termodinâmicos. O processo de criação e configuração do sistema com a adição de componentes e conexões é apresentado no Trecho de código 4.4.

Trecho de código 4.4 – Configuração do sistema

```
# Criação do Sistema
system = System()
#Adição dos componentes ao sistema
system.add_component(condenser)
system.add_component(expansion_valve)
system.add_component(evaporator)
system.add_component(compressor)
system.add_component(cycle_closer)
#Adição das conexões ao sistema
system.add_connection(c1)
system.add_connection(c2)
system.add_connection(c3)
system.add_connection(c4)
system.add_connection(c0)

system.analyze_system()
```

4.1.2.5 Definição das condições de contorno

As condições de contorno são estabelecidas através da especificação de propriedades termodinâmicas em pontos de conexão e na definição das condições do escoamento, como ilustrado no Trecho de código 4.5. Os valores foram especificados de acordo com a Tabela 1.

Trecho de código 4.5 – Especificação das condições de contorno

```
c2.add_parameter(temperature=293, quality=1) # Saída do evaporador: 20 C , vapor saturado
c4.add_parameter(temperature=353, quality=0) #Saída do condensador: 80 C , líquido
    saturado

system.set_flow(fluid="R134a") # Definição do fluido
```

4.1.2.6 Análise do Sistema

Antes de solucionar o sistema, é possível verificar o sistema de equações formado pelo método *system.analyse_system()*, que examina o número de variáveis e equações, garantindo que o sistema esteja adequadamente determinado. O método de análise percorre todos os componentes e conexões do sistema, identificando as as variáveis do sistema e as equações que formam o sistema de equações. Durante este processo, são geradas mensagens informativas que permitem acompanhar a configuração do sistema de equações gerado pela bomba de calor. As variáveis do sistema podem ser vistas no Trecho de código 4.6.

Trecho de código 4.6 – Análise do sistema de bomba de calor

```
====== SYSTEM ANALYSIS ======
Connection label: 1, setting variable: m_dot flow group: primary
Connection label: 1, setting variable: pressure
Connection label: 1, setting variable: enthalpy
Connection label: 3, setting variable: pressure
Connection label: 3, setting variable: enthalpy
Connection label: 0, setting variable: pressure
Connection label: 0, setting variable: enthalpy
```

Cada componente do sistema contribui com equações específicas baseadas em seus parâmetros configurados e nos princípios físicos que governam seu comportamento:

Trecho de código 4.7 – Equações contribuídas pelos componentes

```
[Condenser] Parameter 'Q_dot' adds 1 eqs
[Condenser] Parameter 'pr' adds 1 eqs
[Expansion valve] Constraint 'enthalpy_equality_equation' adds 1 eq(s)
[Evaporator] Parameter 'pr' adds 1 eqs
[Compressor] Parameter 'eta_s' adds 1 eqs
[Cycle Closer] Constraint 'enthalpy_equality_equation' adds 1 eq(s)
[Cycle Closer] Constraint 'pressure_equality_equation' adds 1 eq(s)
```

O resultado final da análise, apresentado no Trecho de código 4.8, demonstra que o sistema está determinado, com o número de variáveis igual ao número de equações disponíveis:

Trecho de código 4.8 – Resumo da análise do sistema

```
====== SUMMARY ======
Total variables: 7

Connection variables: 7
Component variables: 0
Total equations: 7
Connection equations: 0
Component equations: 7
System is properly constrained.
```

Nesse sentido, a análise revela os seguintes aspectos sobre o sistema analisado:

- Variáveis do sistema: O sistema possui 7 variáveis, todas associadas às conexões, sendo a vazão mássica $\dot{m}$ e as pressões e entalpias das conexões 0, 1 e 3;
- Equações disponíveis: São geradas 7 equações, todas provenientes dos componentes, as quais são mostradas na Tabela 3;
- *Status* do sistema: O sistema está adequadamente restringido (*properly constrained*), indicando que possui solução única.

Este diagnóstico é fundamental para garantir que o modelo matemático seja solucionável antes de iniciar os cálculos iterativos, evitando problemas de convergência e identificando possíveis inconsistências na especificação do sistema.

Tabela 3 – Contribuição dos componentes nas equações do sistema

Componente	**Origem da Equação**
Condensador	Parâmetro *Q_dot* (balanço de energia)
Condensador	Parâmetro *pr* (razão de pressão)
Válvula de expansão	*Constraint enthalpy_equality_equation* (conservação de entalpia)
Evaporador	Parâmetro *pr* (razão de pressão)
Compressor	Parâmetro *eta_s* (eficiência isentrópica)
Cycle Closer	*Constraint enthalpy_equality_equation* (conservação de entalpia)
Cycle Closer	*Constraint pressure_equality_equation* (conservação de pressão)

4.1.3 Solução do Sistema

Após a verificação da consistência matemática, o sistema é solucionado através do método iterativo implementado no *framework*. Para acelerar a convergência e garantir estabilidade numérica, são fornecidas estimativas iniciais para algumas variáveis do sistema. Para o sistema de bomba de calor, foram especificadas as condições iniciais na conexão 1, com uma pressão de 0,3 *MPa*, e na conexão 3, com uma pressão de 2,0 *MPa*. Esse procedimento de fornecer estimativas iniciais e chamar o método de solução do sistema é apresentado no trecho de código abaixo.

Trecho de código 4.9 – Especificação das estimativas iniciais

```
global_guesses = {"m_dot": 5 } # kg/s

connection_guesses = {
"1": {"pressure": 300000}, # Pa
"3": {"pressure": 2000000} # Pa
}

system.solve(max_iterations=10, connection_guess=connection_guesses,  global_guess=
    global_guesses)
```

4.1.3.1 Matriz Jacobiana - Primeira Iteração

Durante o processo iterativo de solução, o *framework* calcula a matriz Jacobiana que relaciona as derivadas parciais dos resíduos das equações em relação às variáveis do sistema. A matriz Jacobiana apresentada na Tabela 4 contém as derivadas parciais dos resíduos das equações em relação às variáveis do sistema na primeira iteração do processo de solução. Cada linha da matriz corresponde a uma equação do componente, enquanto cada coluna representa uma variável do sistema. O elemento da matriz J_{ij} representa a derivada parcial do resíduo da equação f_i em relação à variável x_j.

Tabela 4 – Matriz Jacobiana da primeira iteração do sistema de bomba de calor

Equação / Variável	$\dot{m}$	p_1	h_1	p_3	h_3	p_0	h_0
$\dot{Q}$ Condensador	$2{,}221 \times 10^5$	0	0	0	-5	0	0
pr Condensador	0	0	0	0,98	0	0	0
igualdade de *h* Válvula	0	0	0	0	0	0	-1
pr Evaporador	0	0,98	0	0	0	0	0
η_s Compressor	0	0	0	$-9{,}810 \times 10^{-3}$	0,85	0	0
igualdade de *h Cycle closer*	0	0	-1	0	0	0	1
igualdade de *p Cycle closer*	0	-1	0	0	0	1	0

Os valores apresentados na matriz podem ser interpretados fisicamente da seguinte forma:

- Linha 1 - $\dot{Q}$ Condensador: A equação de balanço de energia no condensador é fortemente influenciada pela vazão mássica $\left(\frac{\partial f_1}{\partial \dot{m}} = 2{,}221 \times 10^5\right)$ e pela entalpia na entrada $\left(\frac{\partial f_1}{\partial h_3} = -5\right)$. Fisicamente, estes valores indicam que a vazão mássica do sistema na primeira iteração foi de 5 kg/s, condizente com a estimativa inicial, e a diferença de entalpia no condensador é cerca de 222 kJ/kg;

- Linha 2 - *pr* Condensador: A relação de pressão do condensador depende diretamente da pressão de entrada $\left(\frac{\partial f_2}{\partial p_3} = 0{,}98\right)$, indicando uma perda de carga de aproximadamente 2% através do condensador;

- Linha 3 - Igualdade de entalpia na Válvula: A conservação de entalpia na válvula de expansão relaciona as entalpias de entrada e saída $\left(\frac{\partial f_3}{\partial h_0} = -1\right)$, confirmando matematicamente o processo isentálpico onde $h_3 = h_0$;

- Linha 4 - *pr* Evaporador: A relação de pressão do evaporador é função da pressão de entrada $\left(\frac{\partial f_4}{\partial p_1} = 0{,}98\right)$, similarmente ao condensador, indicando 2% de perda de carga no evaporador;

- Linha 5 - η_s Compressor: A eficiência isentrópica do compressor relaciona as pressões e entalpias de entrada e saída, com sensibilidade significativa à entalpia de saída $\left(\frac{\partial f_5}{\partial h_3} = 0{,}85\right)$, correspondendo à eficiência isentrópica assumida para o compressor;

- Linhas 6-7 - *Cycle closer*: As equações de conservação de entalpia e pressão no fechador de ciclo garantem a continuidade das propriedades no ciclo fechado,com derivadas que asseguram as condições $h_1 = h_0$ e $p_1 = p_0$.

4.1.3.2 Resultados numéricos

O *framework* convergiu em 5 iterações para a tolerância padrão de 1×10^6 , gerando os resultados impressos no terminal que são apresentados nas Tabelas 5 e 6.

Tabela 5 – Parâmetros das conexões do sistema de refrigeração.

Conexão	$\dot{m}$ **[kg/s]**	p **[bar]**	h **[kJ/kg]**	T **[K]**	x **[-]**	**Fase**
0	8,04	5,81	322,10	293,65	0,52	Bifásica
1	8,04	5,81	322,10	293,65	0,52	Bifásica
2	8,04	5,69	409,67	293,00	1,00	Vapor saturado
3	8,04	26,78	446,45	364,17	-	Vapor superaquecido
4	8,04	26,25	322,10	353,00	0,00	Líquido saturado

Tabela 6 – Parâmetros dos componentes do sistema de refrigeração.

Componente	**Parâmetro [Unidade]**	**Valor**
Condensador	Taxa de transferência de calor $\dot{Q}$	$-1,00 \times 10^6$
	Razão de pressão pr [-]	0,98
Válvula de expansão	Razão de pressão pr [-]	0,22
Evaporador	Taxa de transferência de calor $\dot{Q}$ [W]	$7,04 \times 10^5$
	Razão de pressão pr [-]	0,98
Compressor	Eficiência isentrópica η_s [-]	0,85
	Potência consumida $\dot{W}$ [W]	$2,96 \times 10^5$
	Razão de pressão pr [-]	4,71

Com base nos resultados obtidos, é possível calcular o coeficiente de performance do sistema e verificar o balanço de energia global:

- COP (Coeficiente de Performance):

$$COP = \frac{\dot{Q}_{\text{condensador}}}{\dot{W}_{\text{compressor}}} = \frac{1000}{296} = 3{,}38$$

  Este valor indica que o sistema produz 3,38 kW de energia térmica para cada kW de energia elétrica consumida pelo compressor.

- Balanço de energia:

$$\dot{Q}_{\text{evaporador}} + W_{\text{compressor}} = 704 + 296 = 1000 \text{ kW} = \dot{Q}_{\text{condensador}}$$

  O fechamento perfeito do balanço energético confirma que o modelo respeita rigorosamente o primeiro princípio da termodinâmica, garantindo a consistência física da solução. Esta verificação é fundamental para validar a convergência numérica e a confiabilidade dos resultados obtidos.

Conforme esperado para um ciclo fechado em regime permanente, a vazão mássica $\dot{m}$ é conservada em todas as conexões do sistema. Além disso, os valores de entalpia antes e depois da válvula de expansão (Conexões 4 e 0) respeitam a natureza isentálpica do processo de estrangulamento. Os valores de pressão e entalpia na entrada e saída do compressor e trocadores de calor mostram transferência de energia e quedas de pressão consistentes com os parâmetros definidos.

Além disso, o título x se propaga corretamente através do evaporador e condensador, indicando comportamento de mudança de fase consistente com os princípios termodinâmicos do fluido refrigerante. O balanço de energia entre os componentes principais é preservado, em que o calor rejeitado pelo condensador e o calor absorvido no evaporador estão em acordo com a potência mecânica fornecida ao compressor, seguindo a primeira lei da termodinâmica.

Todos os resultados foram obtidos utilizando o *framework* de simulação modular simplificado desenvolvido em Python, conforme descrito neste artigo, e foram verificados contra um modelo equivalente disponível na biblioteca *open-source* TESPy (TESPY, 2025a). A validação do modelo foi realizada através da comparação de vazões mássicas, pressões, entalpias e temperaturas entre o modelo desenvolvido e o tutorial do TESPy, todas dentro de tolerâncias numéricas aceitáveis, indicando que o modelo fornece uma base para futuras análises de desempenho e otimização de sistemas de bomba de calor.

4.1.3.3 Análise paramétrica

A análise paramétrica consiste em investigar a sensibilidade do desempenho do sistema em relação à variação de parâmetros de projeto e de operação. Neste trabalho, foi avaliado o impacto de três parâmetros principais sobre o COP da bomba de calor: a temperatura de evaporação, a temperatura de condensação e a eficiência isentrópica do compressor. Os resultados obtidos são apresentados na Figura 19.

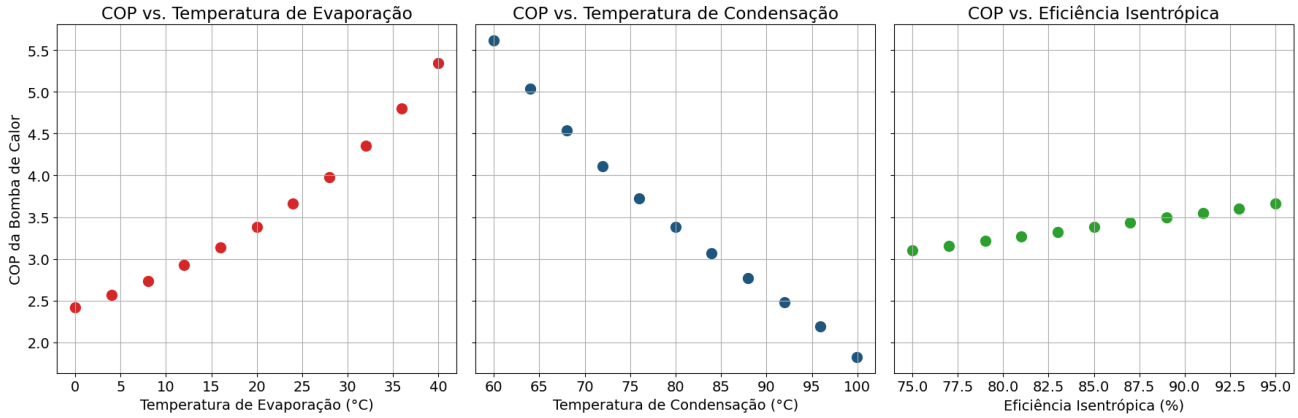

Figura 19 – Análise paramétrica da bomba de calor. Esquerda: Variação da temperatura de evaporação, Centro: Variação da temperatura de condensação, Direita: Variação da eficiência isentrópica do compressor.

Para a análise da temperatura de evaporação, mantiveram-se constantes a temperatura de condensação em 80 °C e a eficiência isentrópica do compressor em 85%, enquanto a

temperatura de evaporação, a qual corresponde à temperatura na saída do evaporador, foi variada de 0 °C a 40 °C. É possível observar que, à medida que essa temperatura aumenta, o COP cresce de forma significativa. Este comportamento ocorre porque elevar a temperatura de evaporação reduz a razão de pressão necessária para o compressor, diminuindo assim o trabalho de compressão requerido para o ciclo. Consequentemente, para a mesma quantidade de calor fornecido ao ambiente por meio do condensador, a potência elétrica ($\dot{W}$) consumida pelo compressor é menor, resultando em um COP mais elevado. Assim, operar com temperaturas de evaporação mais elevadas é desejável sempre que possível, uma vez que aumenta o COP do sistema.

Para a análise da temperatura de condensação, mantiveram-se constantes a temperatura de evaporação em 20 °C e a eficiência isentrópica do compressor em 85%, enquanto a temperatura de condensação, correspondente à temperatura na saída do condensador, foi variada de 60 °C a 100 °C. Verifica-se que, com o aumento da temperatura de condensação, o COP decresce consideravelmente. Esse comportamento ocorre porque a elevação da temperatura de condensação implica em uma maior pressão de condensação, elevando o trabalho requerido pelo compressor para realizar a compressão do fluido. Assim, a potência consumida ($\dot{W}$) cresce e, por consequência, o COP diminui. Isso ressalta a importância de manter a temperatura de condensação o mais baixo possível, por meio de trocadores de calor mais eficientes, por exemplo.

Para a análise da eficiência isentrópica do compressor, mantiveram-se constantes a temperatura de evaporação em 20 °C e a temperatura de condensação em 80 °C, enquanto a eficiência isentrópica foi variada de 75% a 95%. A eficiência isentrópica do compressor representa quão próximo o processo real de compressão se aproxima de uma compressão isentrópica ideal. É possível observar que o aumento da eficiência isentrópica leva a um crescimento moderado do COP. Isso ocorre porque compressores mais eficientes consomem menos trabalho para o mesmo aumento de pressão, diminuindo as perdas associadas às irreversibilidades durante a compressão e, portanto, contribuem para uma melhoria direta da eficiência do ciclo. No entanto, nota-se que a variação do COP com a eficiência isentrópica tende a ser menos acentuada do que nos casos de variação das temperaturas de evaporação e condensação, pois a eficiência isentrópica afeta apenas a parcela de trabalho do compressor, enquanto as variações nas temperaturas impactam também as pressões e entalpias no restante do ciclo.

4.2 Simulação do Ciclo Rankine

4.2.1 Descrição do Sistema

O ciclo Rankine constitui o fundamento termodinâmico das usinas de potência a vapor, sendo amplamente utilizado para a conversão de energia térmica em trabalho mecânico. Este ciclo opera por meio da transferência de calor de uma fonte de alta temperatura para um fluido de trabalho, que sofre mudanças de fase e estado termodinâmico para gerar trabalho. O ciclo básico Rankine é composto por quatro processos principais: expansão isentrópica do

vapor na turbina, de modo a gerar trabalho mecânico, condensação do vapor no condensador, rejeitando calor para o ambiente, compressão isentrópica do líquido saturado na bomba, elevando sua pressão, e aquecimento no gerador de vapor, promovendo a evaporação e superaquecimento do fluido de trabalho. Para a validação do framework desenvolvido, foi selecionado o caso de estudo do tutorial de ciclo Rankine operando em regime permanente disponível na documentação oficial do TESPy (TESPY, 2025b). O sistema é constituído no modelo por sete componentes principais:

- Turbina *(Turbine)*: A turbina é responsável pela expansão do vapor e geração de trabalho mecânico. No modelo, este componente é caracterizado pela eficiência isentrópica (η_s), que relaciona o trabalho isentrópico ideal com o trabalho real extraído durante a expansão;
- Condensador *(Heat_exchanger)*: O condensador atua como trocador de calor responsável pela condensação do vapor proveniente turbina, rejeitando energia térmica para o sistema de resfriamento. Durante este processo, o fluido de trabalho condensa, passando do estado de vapor para líquido saturado. O condensador possui dois circuitos: o lado quente (fluido de trabalho) e o lado frio (água de resfriamento);
- Bomba *(Pump)*: A bomba promove a elevação da pressão do líquido saturado através de um processo de compressão, sendo caracterizado pela eficiência isentrópica;
- Gerador de Vapor *(Simple_hex)*: O gerador de vapor funciona como trocador de calor onde ocorre o aquecimento, vaporização e superaquecimento do fluido de trabalho por meio do fornecimento de energia térmica, promovendo a transição de líquido comprimido proveniente da bomba para vapor superaquecido;
- Fonte de Água de Resfriamento *(Source)*: Componente que representa a entrada de água de resfriamento no sistema, fornecendo o fluido refrigerante necessário para a operação do condensador;
- Sumidouro de Água de Resfriamento *(Sink)*: Componente que representa a saída da água de resfriamento aquecida após sua passagem pelo condensador;
- *Cycle Closer* (*CycleCloser*): Componente fictício cuja função é fechar o ciclo termodinâmico, garantindo a conservação do escoamento do fluido de trabalho entre o final e o início do ciclo.

A topologia do sistema de bomba de calor utilizada para esta validação é apresentada na Figura 20.

4.2.1.1 Condições de Contorno e Parâmetros de Simulação

A Tabela 7 apresenta os principais parâmetros de entrada adotados na simulação, reproduzindo as condições estabelecidas no caso de referência do TESPy.

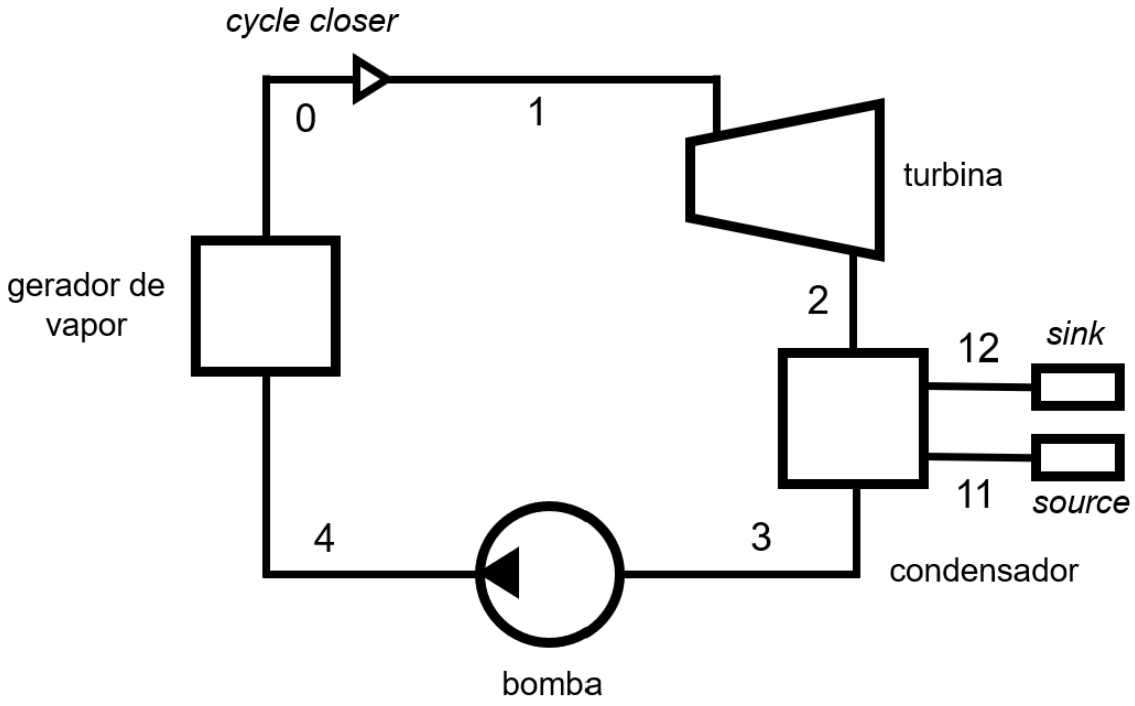

Figura 20 – Representação da topologia do ciclo Rankine.

Tabela 7 – Condições de contorno e parâmetros da simulação do ciclo Rankine.

Parâmetro	**Valor**
Fluido de trabalho	Água
Temperatura de entrada na turbina	600 °C
Pressão de entrada na turbina	150 bar
Vazão mássica do fluido de trabalho	10 kg/s
Eficiência isentrópica da turbina	90%
Eficiência isentrópica da bomba	75%
Razão de pressão no condensador (lado quente)	1
Razão de pressão no condensador (lado frio)	0,98
Título na saída do condensador	0
Razão de pressão no gerador de vapor	0,90
Temperatura da água de resfriamento (entrada *source*)	20 °C
Pressão da água de resfriamento (entrada *source*)	1,2 bar
Temperatura da água de resfriamento (saída *sink*)	30 °C

4.2.1.2 Conexões do Sistema

A configuração das conexões entre os componentes, implementada tanto no TESPy quanto no *framework* desenvolvido, é apresentada na Tabela 8. Esta tabela descreve o caminho do fluido através dos diferentes componentes do ciclo.

Tabela 8 – Conexões do sistema de ciclo Rankine.

Conexão	De	Para
	Circuito Principal	
0	Gerador de vapor (saída)	*Cycle Closer* (entrada)
1	*Cycle Closer* (saída)	Turbina (entrada)
2	Turbina (saída)	Condensador (entrada - lado quente)
3	Condensador (saída - lado quente)	Bomba de alimentação (entrada)
4	Bomba (saída)	Gerador de vapor (entrada)
	Circuito de Água de Resfriamento	
11	*Source* - Fonte de água de resfriamento (saída)	Condensador (entrada - lado frio)
12	Condensador (saída - lado frio)	*Sink* - Sumidouro de água de resfriamento (entrada)

4.2.2 *Implementação no Modelo*

4.2.2.1 *Importação das classes necessárias*

Para simular o ciclo Rankine, é necessário importar os sete componentes que constituem o ciclo, representados por 7 subsclasses, sendo 4 subclasses de componentes específicos (*Turbine*, *Simple_pump*, *Simple_hex* e *Heat_exchanger*) e 3 subclasses de elementos básicos (*CycleCloser*, *Sink* e *Source*), além das classes *Connection* e *System*. Essa processo de importação de módulos é apresentado no Trecho de código 4.10.

Trecho de código 4.10 – Importação dos módulos do *framework*

```
from thermo_sim.components.turbine import Turbine
from thermo_sim.components.pump import Simple_pump
from thermo_sim.components.heat_exchangers import Simple_hex
from thermo_sim.components.heat_exchangers import Heat_exchanger
from thermo_sim.connections import Connection
from thermo_sim.components.basics.cycle_closer import CycleCloser
from thermo_sim.components.basics.sink import Sink
from thermo_sim.components.basics.source import Source
from thermo_sim.system import System
```

4.2.2.2 *Instanciação dos componentes*

Cada componente é criado como uma instância de sua respectiva classe, permitindo a configuração de parâmetros específicos, definidos pela Tabela 7, durante a inicialização, como ilustrado no trecho de código abaixo.

Trecho de código 4.11 – Criação dos componentes do ciclo Rankine

```
cc = CycleCloser('Cycle closer')
sg = Simple_hex('Steam generator', pr = 0.9)
mc = Heat_exchanger('Main condenser', pr=1, pr_2=0.98)
tu = Turbine('Steam turbine', eta_s = 0.9)
fp = Simple_pump('Feed pump', eta_s = 0.75)

cwso = Source('Cooling water source')
cwsi = Sink('Cooling water sink')
```

4.2.2.3 Definição das conexões entre componentes

As conexões estabelecem as ligações físicas entre os componentes, definindo o caminho do fluido através do sistema. Cada conexão especifica explicitamente os componentes e suas portas de origem e destino conforme a Tabela 8. O sistema apresenta dois circuitos de fluido representados pelo *flow_group primary* e o *flow_group secondary*, correspondente ao circuito de água de resfriamento no condensador. O processo de definição das conexões é apresentado no Trecho de código 4.12.

Trecho de código 4.12 – Definição das conexões do sistema

```
c1 = Connection(cc, 'out', tu, 'in', label='1')
c2 = Connection(tu, 'out', mc, 'in', label='2')
c3 = Connection(mc, 'out', fp, 'in', label='3')
c4 = Connection(fp, 'out', sg, 'in', label='4')
c0 = Connection(sg, 'out', cc, 'in', label='0')

c11 = Connection(cwso, 'out', mc, 'in_2', label='11', flow_group = "secondary")
c12 = Connection(mc, 'out_2', cwsi, 'in', label='12', flow_group = "secondary")
```

4.2.2.4 Criação e configuração do sistema

A criação do sistema e adição de componentes e conexões do ciclo Rankine é mostrado no trecho de código a seguir:

Trecho de código 4.13 – Configuração do sistema

```
# Criação do Sistema
system = System()
#Adição dos componentes ao sistema
system.add_component(cc)
system.add_component(sg)
system.add_component(mc)
system.add_component(tu)
system.add_component(fp)
system.add_component(cwso)
system.add_component(cwsi)

#Adição das conexões ao sistema
system.add_connection(c1)
system.add_connection(c2)
system.add_connection(c3)
system.add_connection(c4)
system.add_connection(c0)
system.add_connection(c11)
system.add_connection(c12)
```

4.2.2.5 Definição das condições de contorno

As condições de contorno são estabelecidas através da especificação de propriedades termodinâmicas em pontos de conexão e na definição das condições do escoamento. Os valores foram especificados de acordo com a Tabela 7.

Trecho de código 4.14 – Especificação das condições de contorno

```
c11.add_parameter(temperature = 293, pressure = 120000 ) # Condições da água de
    resfriamento proveniente do componente source
c12.add_parameter(temperature = 303)  # Temperatura de saída do lado frio do condensador
c1.add_parameter(temperature = 873, pressure = 15000000) # Condições da entrada da
    turbina
c2.add_parameter(pressure = 10000) # Pressão da saída da turbina
c3.add_parameter(quality = 0) # Título na saída do condensador

# Definição do fluido
system.set_flow(fluid="Water", m_dot = 10) # Condições do escoamento no circuito pricipal
system.set_flow(fluid="Water", flow_group = "secondary") # Definição do fluido no
    circuito de resfriamento do condensador
```

4.2.2.6 Análise do Sistema

Assim como no caso da bomba de calor, antes de proceder com a solução iterativa, recomenda-se a verificação da formulação matemática do sistema por meio do método *system.analyse_system()*, que realiza a contagem de variáveis e equações para confirmar se o problema está matematicamente determinado. As variáveis do sistema são apresentadas no trecho de código abaixo.

Trecho de código 4.15 – Análise do sistema do ciclo Rankine

```
====== SYSTEM ANALYSIS ======
Connection label: 2, setting variable: enthalpy
Connection label: 3, setting variable: pressure
Connection label: 3, setting variable: enthalpy
Connection label: 4, setting variable: pressure
Connection label: 4, setting variable: enthalpy
Connection label: 0, setting variable: pressure
Connection label: 0, setting variable: enthalpy
Connection label: 11, setting variable: m_dot flow group: secondary
Connection label: 12, setting variable: pressure
Connection label: 12, setting variable: enthalpy
```

Cada componente do sistema contribui com equações específicas baseadas em seus parâmetros configurados e nas equações que governam seu comportamento:

Trecho de código 4.16 – Equações contribuídas pelos componentes e conexões

```
[Connection 3] Equation 'c3_quality_eq' adds 1 eq(s)
[Connection 12] Equation 'c12_temperature_eq' adds 1 eq(s)
[Cycle closer] Constraint 'enthalpy_equality_equation' adds 1 eq(s)
[Cycle closer] Constraint 'pressure_equality_equation' adds 1 eq(s)
[Steam generator] Parameter 'pr' adds 1 eqs
[Main condenser] Parameter 'pr' adds 1 eqs
[Main condenser] Parameter 'pr_2' adds 1 eqs
[Main condenser] Constraint 'energy_balance_equation' adds 1 eq(s)
[Steam turbine] Parameter 'eta_s' adds 1 eqs
[Feed pump] Parameter 'eta_s' adds 1 eqs
```

O resultado final da análise demonstra que o sistema está corretamente especificado, com o número de variáveis igual ao número de equações disponíveis, ilustrado no Trecho de código 4.17.

Trecho de código 4.17 – Resumo da análise do sistema

```
====== SUMMARY ======
Total variables: 10
  - Connection variables: 10
  - Component variables: 0
Total equations: 10
  - Connection equations: 2
  - Component equations: 8

System is properly constrained.
```

Nesse sentido, a análise revela os seguintes aspectos sobre o ciclo Rankine:

- sistema possui 10 variáveis, todas associadas às conexões do circuito. Estas incluem as propriedades termodinâmicas (pressão, entalpia, temperatura) das conexões do ciclo principal (0, 1, 2, 3, 4) e do circuito de água de resfriamento (12), além da vazão mássica $\dot{m}$ da água de resfriamento.
- São geradas 10 equações distribuídas entre: 2 equações de conexões, as quais são relacionadas às condições de contorno especificadas (título na conexão 3 e temperatura na conexão 12) e 8 equações de componentes, que são provenientes dos balanços de massa e energia nos componentes e dos parâmetros específicos, como mostradas na Tabela 9.
- *Status* do sistema: O sistema está adequadamente restringido (*properly constrained*), indicando que possui solução única.

Tabela 9 – Contribuição dos componentes e conexões nas equações do sistema do ciclo Rankine

Componente/Conexão	**Origem da Equação**
	Equações de Componentes
Gerador de vapor	Parâmetro *pr* (razão de pressão)
Condensador	Parâmetro *pr* (razão de pressão - lado quente)
Condensador	Parâmetro *pr_2* (razão de pressão - lado frio)
Condensador	*Constraint energy_balance_equation* (balanço de energia)
Turbina	Parâmetro *eta_s* (eficiência isentrópica)
Bomba	Parâmetro *eta_s* (eficiência isentrópica)
Cycle closer	*Constraint enthalpy_equality_equation* (conservação de entalpia)
Cycle closer	*Constraint pressure_equality_equation* (conservação de pressão)
	Equações de Conexões
Conexão 3	Equação *c3_quality_eq* (especificação do título)
Conexão 12	Equação *c12_temperature_eq* (especificação de temperatura)

4.2.3 Solução do Sistema

Após a verificação da consistência matemática, o sistema é solucionado através do método iterativo implementado no *framework*. Para acelerar a convergência e garantir estabi-

lidade numérica, foi fornecida uma estimativa inicial de 200 $kg\ s^{-1}$ para a variável da vazão mássica:

Trecho de código 4.18 – Especificação das estimativas iniciais

```
global_guesses = {"m_dot": 200} #kg/s

system.solve(max_iterations = 10, global_guess=global_guesses)
```

4.2.3.1 *Matriz Jacobiana - Primeira Iteração*

Durante o processo iterativo de solução, o *framework* calcula a matriz Jacobiana que relaciona as derivadas parciais dos resíduos das equações em relação às variáveis do sistema. A matriz Jacobiana apresentada na Tabela 10 contém as derivadas parciais dos resíduos das equações em relação às variáveis do sistema na primeira iteração do processo de solução.

Tabela 10 – Matriz Jacobiana da primeira iteração do sistema de ciclo Rankine

Equação / Variável	h_2	p_3	h_3	p_4	h_4	p_0	h_0	$\dot{m}_{sec}$	p_{12}	h_{12}
igualdade de *h Cycle closer*	0	0	0	0	0	0	1,0	0	0	0
igualdade de *p Cycle closer*	0	0	0	0	0	1,0	0	0	0	0
pr Gerador de vapor	0	0	0	0,9	0	-1,0	0	0	0	0
pr Condensador	0	-1,0	0	0	0	0	0	0	0	0
pr_2 Condensador	0	0	0	0	0	0	0	0	-1,0	0
Balanço energético condensador	-10,0	0	10,0	0	0	0	0	$4{,}18 \times 10^{4}$	0	$2{,}0 \times 10^{2}$
η_s Turbina	-1,0	0	0	0	0	0	0	0	0	0
η_s Bomba	0	$1{,}04 \times 10^{-3}$	$-7{,}5 \times 10^{-1}$	$-1{,}04 \times 10^{-3}$	$7{,}5 \times 10^{-1}$	0	0	0	0	0
Título Conexão 3	0	-5,0	5,0	0	0	0	0	0	0	0
Temperatura Conexão 12	0	0	0	0	0	0	0	0	$-2{,}18 \times 10^{-7}$	$2{,}39 \times 10^{-4}$

Os valores apresentados na matriz podem ser interpretados fisicamente da seguinte forma:

- Linha 1 - Igualdade de *h Cycle closer*: A equação de igualdade de entalpia para fechamento do ciclo estabelece uma relação direta com a entalpia h_0 $\left(\frac{\partial f_1}{\partial h_0} = 1{,}0\right)$;
- Linha 2 - Igualdade de *p Cycle closer*: A equação de igualdade de pressão para fechamento do ciclo é definida através da pressão p_0 $\left(\frac{\partial f_2}{\partial p_0} = 1{,}0\right)$, garantindo a continuidade do ciclo;
- Linha 3 - *pr* do gerador de vapor: A razão de pressão no gerador de vapor relaciona a pressão de saída $\left(\frac{\partial f_3}{\partial p_4} = 0{,}9\right)$, indicando 10% de perda de carga, e apresenta uma relação com a pressão de saída do gerador de vapor $\left(\frac{\partial f_3}{\partial p_0} = -1{,}0\right)$;
- Linha 4 - *pr* do condensador: A perda de carga no condensador no lado quente mostra a relação com a pressão de saída do condensador p_3 $\left(\frac{\partial f_4}{\partial p_3} = -1{,}0\right)$;
- Linha 5 - pr_2 do condensador: A relação de pressão do lado frio *pr_2* do condensador apresenta uma dependência da pressão de saída do lado frio p_{12} $\left(\frac{\partial f_5}{\partial p_{12}} = -1{,}0\right)$;

- Linha 6 - Balanço de energia do condensador: O balanço de energia apresenta relação com a entalpia de entrada do $\left(\frac{\partial f_6}{\partial h_2} = -10{,}0\right)$ e de saída do condensador $\left(\frac{\partial f_6}{\partial h_3} = 10{,}0\right)$, da vazão mássica do *flow_group* secundário $\left(\frac{\partial f_6}{\partial \dot{m}_{sec}} = 4{,}18 \times 10^4\right)$, e da entalpia de saída da água de resfriamento $\left(\frac{\partial f_6}{\partial h_{12}} = 2{,}0 \times 10^2\right)$;

- Linha 7 - η_s da turbina: A equação da turbina apresenta dependência da entalpia de saída da turbina h_2 $\left(\frac{\partial f_7}{\partial h_2} = -1{,}0\right)$;

- Linha 8 - η_s da bomba: A equação da bomba relaciona as pressões e entalpias de entrada e saída com coeficientes que refletem a eficiência isentrópica: $\left(\frac{\partial f_8}{\partial p_3} = 1{,}04 \times 10^{-3}\right)$, $\left(\frac{\partial f_8}{\partial h_3} = -7{,}5 \times 10^{-1}\right)$, $\left(\frac{\partial f_8}{\partial p_4} = -1{,}04 \times 10^{-3}\right)$ e $\left(\frac{\partial f_8}{\partial h_4} = 7{,}5 \times 10^{-1}\right)$;

- Linha 9 - Título Conexão 3: A especificação do título na conexão 3 apresenta dependência da pressão e entalpia na conexão $\left(\frac{\partial f_9}{\partial p_3} = -5{,}0\right)$ e $\left(\frac{\partial f_9}{\partial h_3} = 5{,}0\right)$, garantindo que o título especificado seja obtido;

- Linha 10 - Temperatura Conexão 12: A especificação de temperatura na conexão 12 apresenta relação com a pressão $\left(\frac{\partial f_{10}}{\partial p_{12}} = -2{,}18 \times 10^{-7}\right)$ e a entalpia $\left(\frac{\partial f_{10}}{\partial h_{12}} = 2{,}39 \times 10^{-4}\right)$ da conexão.

4.2.3.2 Resultados numéricos

O *framework* convergiu em 4 iterações para a tolerância padrão de 1×10^{-6}, gerando os resultados impressos no terminal que são apresentados nas Tabelas 11 e 12.

Tabela 11 – Parâmetros das conexões do sistema de ciclo Rankine.

Conexão	$\dot{m}$ **[kg/s]**	p **[bar]**	h **[kJ/kg]**	T **[K]**	x **[-]**	**Fase**
0	10,0	150,0	3582,7	873,0	-	Vapor superaquecido
1	10,0	150,0	3582,7	873,0	-	Vapor superaquecido
2	10,0	0,10	2261,9	319,0	0,865	Bifásica
3	10,0	0,10	191,8	319,0	0	Líquido saturado
4	10,0	166,7	214,2	320,9	0	Líquido comprimido
11	495,1	1,20	83,4	293,0	0	Líquido saturado
12	495,1	1,18	125,2	303,0	0	Líquido saturado

Tabela 12 – Parâmetros dos componentes do sistema de ciclo Rankine.

Componente	**Parâmetro [Unidade]**	**Valor**
Gerador de vapor	Taxa de transferência de calor $\dot{Q}$ [W]	$3,37 \times 10^7$
	Razão de pressão pr [-]	0,90
Condensador	Taxa de transferência de calor $\dot{Q}$ [W]	$-2,07 \times 10^7$
	Razão de pressão (lado quente) pr [-]	0,98
	Razão de pressão (lado frio) pr_2 [-]	1,00
	Condutância térmica global UA [$W\ K^{-1}$]	$1,01 \times 10^6$
	Diferença de temperatura média logarítmica ΔT_{log} [K]	20,55
	Diferença de temperatura terminal superior ΔT_{sup} [K]	15,96
	Diferença de temperatura terminal inferior ΔT_{inf} [K]	25,96
	Efetividade lado frio ε_{frio} [-]	0,385
	Efetividade lado quente ε_{quente} [-]	0,950
Turbina	Eficiência isentrópica η_s [-]	0,90
	Potência gerada $\dot{W}$ [W]	$1,32 \times 10^7$
	Razão de pressão pr [-]	0,000667
Bomba	Eficiência isentrópica η_s [-]	0,75
	Potência consumida $\dot{W}$ [W]	$2,24 \times 10^5$
	Razão de pressão pr [-]	1666,7

Com base nos resultados obtidos, é possível calcular a eficiência térmica do ciclo e verificar o balanço de energia:

- Eficiência térmica do ciclo (η):

$$\eta = \frac{\dot{W}_{\text{líquido}}}{\dot{Q}_{entrada}} = \frac{\dot{W}_{turbina} - \dot{W}_{bomba}}{\dot{Q}_{gerador\_vapor}} = \frac{13209 - 224}{33686} = \frac{12985}{33686} = 0,385 = 38,5\%$$

  Este valor indica que o ciclo Rankine converte 38,5% da energia térmica fornecida em trabalho.

- Balanço de energia global:

$$\dot{Q}_{gerador\_vapor} - \dot{Q}_{condensador} = \dot{W}_{turbina} - \dot{W}_{bomba}$$

$$33686 - 20701 = 13209 - 224 = 12985 \text{ kW}$$

$$\dot{Q}_{net} = \dot{W}_{net} = 12985 \text{ kW}$$

  O fechamento do balanço energético confirma que o modelo respeita rigorosamente o primeiro princípio da termodinâmica, garantindo a consistência física da solução.

Conforme esperado para um ciclo fechado em regime permanente, a vazão mássica $\dot{m} = 10$ kg/s é conservada em todas as conexões do ciclo principal. As propriedades termodinâmicas nas diferentes seções do ciclo são coerentes com os processos que ocorrem no ciclo Rankine:

- Processo 1→2 (Expansão isentrópica na turbina): A pressão reduz de 150 bar para 0,1 bar, com diminuição correspondente da entalpia de 3582,7 kJ/kg para 2261,9 kJ/kg, resultando em geração de trabalho;

- Processo 2→3 (Rejeição de calor no condensador): O vapor (título 0,865) condensa completamente até líquido saturado (título 0) à pressão constante de 0,1 bar;
- Processo 3→4 (Compressão isentrópica na bomba): A pressão do líquido saturado é elevada de 0,1 bar para 166,7 bar, com um aumento da entalpia de 191,8 kJ/kg para 214,2 kJ/kg;
- Processo 4→0 (Adição de calor no gerador de vapor): O líquido comprimido é aquecido e transformado em vapor até vapor superaquecido, com aumento significativo da entalpia de 214,2 kJ/kg para 3582,7 kJ/kg.

O circuito de água de resfriamento opera adequadamente, com vazão de 495,1 kg/s e aquecimento de 10°C (de 293 K para 303 K), demonstrando capacidade térmica suficiente para rejeitar o calor do condensador.

Assim como no caso da bomba de calor, foi realizada a validação do modelo por meio da comparação de vazões mássicas, pressões, entalpias e temperaturas entre o modelo desenvolvido e o tutorial do TESPy, todas dentro de tolerâncias numéricas aceitáveis.

4.2.3.3 Análise Paramétrica

Para essa análise paramétrica, foi avaliado o impacto de dois parâmetros principais sobre a eficiência térmica e potência líquida do ciclo Rankine: a temperatura do vapor e a pressão do vapor.

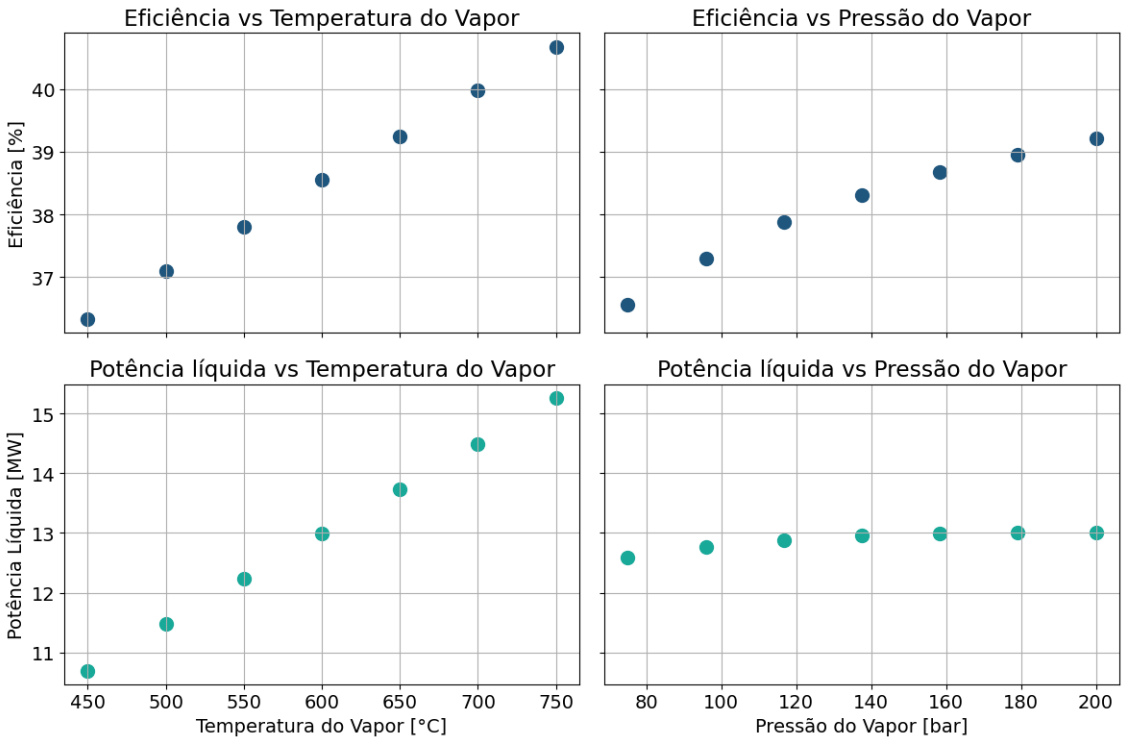

Figura 21 – Análise paramétrica do ciclo Rankine. Esquerda: Variação da temperatura do vapor. Direita: Variação da pressão do vapor.

Para a análise da temperatura do vapor, manteve-se constante a pressão do vapor em 150 bar, enquanto a temperatura do vapor, correspondente à temperatura na entrada da

turbina, foi variada de 450 °C a 750 °C. É possível observar que, à medida que essa temperatura aumenta, tanto a eficiência do ciclo quanto a potência líquida ($\dot{W}_{net}$) crescem de forma significativa. A eficiência aumenta de aproximadamente 36,3% para 40,8%, enquanto a potência líquida cresce de cerca de 10,8 MW para 15,3 MW. Esse comportamento ocorre porque o aumento da temperatura do vapor eleva a entalpia específica na entrada da turbina, proporcionando maior trabalho disponível para expansão. Consequentemente, para a mesma vazão mássica, obtém-se maior trabalho líquido, resultando em uma eficiência e potência mais elevadas.

Para a análise da pressão do vapor, manteve-se constante a temperatura do vapor em 600 °C, enquanto a pressão do vapor, correspondente à pressão na entrada da turbina, foi variada de 75 bar a 200 bar. Verifica-se que, com o aumento da pressão do vapor, a eficiência cresce consideravelmente, aumentando de aproximadamente 36,6% para 39,2%. No entanto, observa-se que a potência líquida permanece relativamente constante em torno de 13,0 MW, com pequenas variações. Esse comportamento ocorre porque o aumento da pressão do vapor eleva a razão de pressão disponível para expansão na turbina, permitindo maior extração de trabalho durante esse processo de expansão. Entretanto, o aumento da pressão também eleva o trabalho requerido pela bomba, o que explica a permanência da potência líquida aproximadamente constante, enquanto a eficiência melhora devido ao maior trabalho extraído.

5 Conclusão

Neste trabalho foi desenvolvido uma ferramenta modular e extensível para simulação de sistemas térmicos, implementado em Python com programação orientada a objetos. A motivação principal surgiu da necessidade identificada no grupo de pesquisa Polo/UFSC de unificar os diferentes modelos numéricos desenvolvidos ao longo dos anos em uma plataforma comum, superando a fragmentação existente e facilitando a integração entre diferentes linhas de pesquisa.

O *framework* foi estruturado em três classes principais (*Component*, *Connection* e *System*) que atuam de forma integrada para representar sistemas termodinâmicos. A arquitetura orientada a objetos adotada permite a criação modular de componentes reutilizáveis, facilitando tanto o desenvolvimento quanto a manutenção do código. Para a solução numérica do sistema de equações não lineares que modelam os sistemas térmicos, foi implementado o método de Newton-Raphson com cálculo analítico e numérico das derivadas parciais para a construção da matriz Jacobiana, garantindo convergência rápida nas simulações realizadas.

Foi implementada uma biblioteca abrangente de componentes básicos que inclui os elementos fundamentais encontrados na maioria dos sistemas térmicos como trocadores de calor (*Simple_hex* e *Heat_exchanger*), turbomáquinas (*Compressor, Turbine, Simple_pump*), válvula de expansão (*Valve*) e componentes auxiliares (*Source, Sink, CycleCloser*). Cada componente implementa suas próprias equações constitutivas e parâmetros específicos, mantendo a consistência física através dos balanços de massa e energia.

A validação do modelo foi realizada através da simulação de dois casos de estudo baseados nos tutoriais do TESPy: uma bomba de calor operando com R134a e um ciclo Rankine com água como fluido de trabalho. Os resultados obtidos apresentaram concordância com os valores de referência, demonstrando que o modelo desenvolvido reproduz corretamente o comportamento termodinâmico dos sistemas simulados. As análises paramétricas realizadas confirmaram o comportamento físico esperado dos sistemas. Para a bomba de calor, verificou-se que o aumento da temperatura de evaporação e a redução da temperatura de condensação resultam em maior COP, enquanto o aumento da eficiência isentrópica do compressor promove melhorias moderadas no desempenho. No ciclo Rankine, o aumento da temperatura e pressão do vapor resultaram em uma maior eficiência do ciclo, com a temperatura exercendo maior influência na potência líquida gerada.

Nesse sentido, a ferramenta desenvolvida atendeu aos objetivos propostos, oferecendo uma plataforma unificada, modular e extensível para simulação de sistemas térmicos. Sua arquitetura simplificada em relação ao TESPy facilitam a compreensão, manutenção e extensão futura dos modelos. A documentação produzida neste trabalho também funciona como um manual, incluindo diretrizes para o desenvolvimento de novos componentes. A estrutura flexível implementada permite não apenas a simulação de sistemas convencionais, mas também a investigação de novas tecnologias e a exploração de diferentes configurações por meio da possibilidade de incorporação de novos componentes e métodos de análise. A ferramenta

estabelece, portanto, uma base que pode evoluir continuamente, atendendo às demandas de pesquisa do grupo Polo/UFSC no desenvolvimento de aplicações em engenharia térmica, contribuindo para soluções energéticas mais eficientes.

5.1 Propostas de trabalhos futuros

A seguir são apresentadas propostas de trabalhos futuros que podem ser desenvolvidos a partir deste modelo:

- Implementar componentes para novas tecnologias como células de combustível e sistemas magnetocalóricos;
- Adicionar componentes para mistura e separação de fluidos;
- Criar um tipo de conexão que transporte massa, concentração de espécies e energia;
- Implementar algoritmos de otimização integrados ao modelo, como algoritmos genéticos;
- Implementar análise exergética para identificação de irreversibilidades;
- Adicionar capacidade de simulação transiente para análise dinâmica dos sistemas;
- Validar o modelo por meio de dados experimentais de sistemas;
- Desenvolver interfaces gráficas para facilitar a simulação de casos;
- Disponibilizar uma versão educacional para a utilização do modelo em disciplinas como EMC5418-Termodinâmica Aplicada e EMC5804-Análise Integrada de Sistemas Térmicos.

Referências

BELL, I. H.; WRONSKI, J.; QUOILIN, S.; LEMORT, V. Pure and pseudo-pure fluid thermophysical property evaluation and the open-source thermophysical property library coolprop. *Industrial & Engineering Chemistry Research*, v. 53, n. 6, p. 2498–2508, 2014.

BURDEN, R. L.; FAIRES, J. D.; BURDEN, A. M. *Análise numérica*. [S.l.]: Cengage Learning, 2016.

CHEN, C.; WITTE, F.; TUSCHY, I.; KOLDITZ, O.; SHAO, H. Parametric optimization and comparative study of an organic rankine cycle power plant for two-phase geothermal sources. *Energy*, Elsevier, v. 252, p. 123910, 2022.

DHAR, P. L. *Thermal System Design and Simulation*. [S.l.]: Academic Press, 2017. ISBN 978-0-12-809449-5.

DIAS, T. Z.; FARIA, P. V.; FERREIRA, J. C. A.; PEIXER, G. F.; LOZANO, J. A.; BARBOSA, J. R. Sensitivity analysis of electrochemical modeling parameters on pemfc polarization curve/efficiency. In: ABCM - BRAZILIAN SOCIETY OF MECHANICAL SCIENCES AND ENGINEERING. *Proceedings of the 20th Brazilian Congress of Thermal Sciences – ENCIT 2024*. Foz do Iguaçu, Brazil, 2024.

DTU, T. U. of D. *CoolPack - A refrigeration toolbox*. 2025. <https://www.ipu.dk/products/coolpack/>.

F-CHART. *Engineering Equation Solver (EES)*. 2025. Disponível em: <https://fchartsoftware.com/ees/>.

FERREIRA, J. C. A.; CARNEIRO, M. V. P.; BARBOSA, J. R. Numerical simulation of full-cone sprays for liquid film formation. In: ABCM - BRAZILIAN SOCIETY OF MECHANICAL SCIENCES AND ENGINEERING. *Proceedings of the 27th International Congress of Mechanical Engineering – COBEM 2023*. Florianópolis, Brazil, 2023.

FRY, N.; EAGLE-BLUESTONE, J.; WITTE, F. Computational modeling of organic rankine cycle combined heat and power for sedimentary geothermal exploitation. *Geothermal Resources Council Transactions*, v. 46, 2022.

HARRIS, C. R.; MILLMAN, K. J.; WALT, S. J. van der; GOMMERS, R.; VIRTANEN, P.; COURNAPEAU, D.; WIESER, E.; TAYLOR, J.; BERG, S.; SMITH, N. J.; KERN, R.; PICUS, M.; HOYER, S.; KERKWIJK, M. H. van; BRETT, M.; HALDANE, A.; RÍO, J. F. del; WIEBE, M.; PETERSON, P.; GÉRARD-MARCHANT, P.; SHEPPARD, K.; REDDY, T.; WECKESSER, W.; ABBASI, H.; GOHLKE, C.; OLIPHANT, T. E. Array programming with NumPy. *Nature*, Springer Science and Business Media LLC, v. 585, n. 7825, p. 357–362, set. 2020. Disponível em: <https://doi.org/10.1038/s41586-020-2649-2>.

IEA. *Global Energy Review 2025*. Paris: International Energy Agency, 2025. Licence: CC BY 4.0. Disponível em: <https://www.iea.org/reports/global-energy-review-2025>.

JAKOBSEN, A.; RASMUSSEN, B.; ANDERSEN, S. Coolpack – simulation tools for refrigeration systems. *Scan Ref*, Scanref, v. 28, n. 4, p. 7–10, 1999. ISSN 0284-0758.

JALURIA, Y. *Design and Optimization of Thermal Systems, Third Edition: with MATLAB Applications*. 3rd. ed. [S.l.]: CRC Press, 2019.

KLEIN, S. *Engineering Equation Solver (EES) for Microsoft Windows: Commercial and Professional Versions*. Middleton, WI, USA, 2003. Version 6.671. Available at <http://www.fchart.com/>.

MILITAO, L. A.; FERNANDES, C. D.; BARBOSA, J. R.; HELDWEIN, M. L.; KASAMA, A. H.; OLIVEIRA, A. M. Assessment of the thermal performance of subsea enclosure geometries. In: ABCM - BRAZILIAN SOCIETY OF MECHANICAL SCIENCES AND ENGINEERING. *Proceedings of the 20th Brazilian Congress of Thermal Sciences – ENCIT 2024*. Foz do Iguaçu, Brazil, 2024.

NAKASHIMA, A.; PEIXER, G.; LOZANO, J.; BARBOSA, J. A lumped-element magnetic refrigerator model. *Applied Thermal Engineering*, v. 204, p. 117918, 2022.

PEIXER, G. F.; NAKASHIMA, A. T.; LOZANO, J. A.; BARBOSA, J. R. System-level multi-objective optimization of a magnetic air conditioner through coupling of artificial neural networks and genetic algorithms. *Applied Thermal Engineering*, v. 227, p. 120368, 2023.

PENONCELLO, S. G. *Thermal Energy Systems: Design and Analysis*. 2nd. ed. CRC Press, 2018. Disponível em: <https://doi.org/10.1201/b22141>.

PETERS, S.; SZEREMETA, J. F. *Cálculo numérico computacional*. [S.l.]: Editora da UFSC, 2019.

PHILLIPS, D. *Python 3 object oriented programming*. [S.l.]: Packt Publishing Ltd, 2010.

RICHARDSON, D. H. *An object oriented simulation framework for steady-state analysis of vapor compression refrigeration systems and components*. Tese (Doutorado) — University of Maryland, College Park, 2006.

RICHTER, C. C. *Proposal of new object-oriented equation-based model libraries for thermodynamic systems*. Tese (Doutorado) — Braunschweig, Techn. Univ., Diss., 2008, 2008.

SGANZERLA, Y. F.; BORDIGNON, J. C.; PEIXER, G. F.; LOZANO, J. A.; BARBOSA, J. R. Optimization of precooling cycles for hydrogen liquefaction systems by machine learning enhanced genetic algorithms. In: ABCM - BRAZILIAN SOCIETY OF MECHANICAL SCIENCES AND ENGINEERING. *Proceedings of the 20th Brazilian Congress of Thermal Sciences – ENCIT 2024*. Foz do Iguaçu, Brazil, 2024.

STOECKER, W. F. *Design of Thermal Systems*. 3rd. ed. New York: McGraw-Hill, 1989.

SYSTEMS, O. T. *Vapor Cycle – System Design Overview*. 2025. Acesso em: 20 jun. 2025. Disponível em: <https://optimizedthermalsystems.com/vapCyc-details/>.

TESPY. *Heat Pump Tutorial*. 2025. Disponível em: <https://tespy.readthedocs.io/en/main/basics/heat_pump.html>.

TESPY. *Rankine Cycle Tutorial*. 2025. Disponível em: <https://tespy.readthedocs.io/en/main/basics/rankine_cycle.html>.

TESS, T. E. S. S. *TRNSYS – Transient System Simulation Tool*. 2025. Disponível em: <https://www.trnsys.com/>.

VERING, C.; ENGELPRACHT, M.; GÖBEL, S.; HOSEINPOORI, S.; WÜLLHORST, F.; SCHWENZER, C.; RADEMACHER, M.; HINRICHS, S.; CHANDRA, F.; MEHRFELD, P. et al. Open-source vapor compression library (vclib): Heat pump modeling for education and research. *Computer Applications in Engineering Education*, Wiley Online Library, v. 30, n. 5, p. 1498–1509, 2022.

WITTE, F. *TESPy - Thermal Engineering Systems in Python*. 2025. <https://tespy.readthedocs.io/en/main/basics/intro.html>.

WITTE, F. *TESPy – Thermal Engineering Systems in Python*. 2025. Disponível em: <https://tespy.readthedocs.io/en/main/introduction.html>.

WITTE, F.; TUSCHY, I. TESPy: Thermal Engineering Systems in Python. *Journal of Open Source Software*, The Open Journal, v. 5, n. 49, p. 2178, 2020.

WITTE, F.; TUSCHY, I. TESPy: Thermal Engineering Systems in Python. *Journal of Open Source Software*, The Open Journal, v. 5, n. 49, p. 2178, 2020.

ZODER, M.; BALKE, J.; HOFMANN, M.; TSATSARONIS, G. Simulation and exergy analysis of energy conversion processes using a free and open-source framework—python-based object-oriented programming for gas-and steam turbine cycles. *Energies*, MDPI, v. 11, n. 10, p. 2609, 2018.