UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

ESCOLA DE ENGENHARIA

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

MATEUS KAHLER

DESENVOLVIMENTO DE SIMULADOR MODULAR DE CIRCUITOS ELÉTRICOS SOB O PARADIGMA DE ORIENTAÇÃO A OBJETOS

Porto Alegre

2018

MATEUS KAHLER

DESENVOLVIMENTO DE SIMULADOR MODULAR DE CIRCUITOS ELÉTRICOS SOB O PARADIGMA DE ORIENTAÇÃO A OBJETOS

Trabalho de Conclusão de curso apresentado ao Departamento de Engenharia Elétrica como requisito para a obtenção do título de Bacharel em Engenharia Elétrica.

Orientador: Prof. Dr. Bardo Ernst Josef Bodmann

Porto Alegre

2018

“It is my experience that proofs involving matrices can be shortened by 50% if one throws the matrices out. “

- Emil Artin

Agradecimentos

Minha mãe por me dar o mar,
meu pai por me dar o barco,
minha irmã por me dar tinta para colori-lo,

Ao Vinícius por navegar comigo,
a UFRGS pelos bons ventos,
ao prof. Bardo pela boa orientação,
e a você.

RESUMO

A evolução da engenharia contemporânea está atrelada a qualidade e disponibilidade de simuladores computacionais para os problemas que aborda. Na engenharia elétrica, a simulação de circuitos elétricos é ubíqua na indústria e meio acadêmico, como parte integral da concepção de produtos. Os principais simuladores disponíveis tem robustez estabelecida, mas são, em maioria, escritos em código imperativo sem abstração explícita de estruturas, ou de código indisponível. Este projeto estabelece as bases para desenvolvimento de um simulador de circuitos elétricos usando da orientação a objetos para abstrair componentes, conexões e comportamentos que constituem modelo de circuitos elétricos reais. O objetivo do projeto final é disponibilizar conjunto de códigos, de caráter modular e customizável, relativo às etapas requeridas para interpretação de circuitos, simulação destes e apresentação de resultados, visando usuários que pretendam adaptar módulos da simulação para problemas específicos sem requerimento que estes conheçam os detalhes do programa em sua totalidade, guiados pela documentação do projeto, podendo usar de conceitos da área da engenharia elétrica.

Palavras-chave: Simulação numérica. Circuitos elétricos. Orientação a objetos.

ABSTRACT

The evolution of contemporary engineering is linked to the quality and availability of computer simulators for the problems it addresses. In electrical engineering, electrical circuit simulation is ubiquitous in industry and academia, as an integral part of product design. The main available simulators have established robustness, but are mostly written in imperative code without explicit abstraction of structures, or of unavailable code. This project establishes the bases for the development of an electrical circuit simulator using object-orientation to abstract electric components, it`s connections and behaviors, that constitute the model of a real electric circuits. The objective of the final project is to provide a set of codes, with modular and customizable properties, related to the steps required for circuit interpretation, simulation of these and presentation of results, aiming at users who want to adapt simulation modules to specific problems without requiring them to know the program details in its entirety, guided by the documentation of the project, being able to use concepts from the field of electrical engineering.

Keywords: Numeric Simulation, Electric Circuits, Object Oriented Programming.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Este projeto consiste no desenvolvimento de um programa de computador que interprete um circuito elétrico fornecido como entrada e simule numericamente seu comportamento, permitindo seleção e tratamento de variáveis de interesse, desenvolvido sob o paradigma de orientação a objeto. O projeto foi motivado pela importância de simuladores na engenharia elétrica, escarça disponibilidade de simuladores de código aberto sob este paradigma, e interesse pessoal no estudo do tema.

1.1 A NECESSIDADE DE SIMULADORES

Durante etapas iniciais do projeto de circuitos elétricos, o engenheiro eletricista usa concepções simplificadas de componentes elétricos ou blocos funcionais que compõe o circuito inteiro, pois a resolução analítica completa de um circuito é de complexidade que cresce rapidamente relativa ao número de componentes e detalhamento da modelagem individual destes.

A simulação computacional de um circuito permite arbitrária aproximação de seu comportamento real e diminui o tempo requerido para projetos. A aproximação do comportamento real se dá pela escalabilidade dos simuladores: agregando subcircuitos em circuitos complexos é possível a consideração de efeitos de interconexão entre blocos funcionais ou efeitos da não idealidade de componentes individuais.

1.2 OBJETIVOS E CARACTERÍSTICAS DESEJADAS

O objetivo primário do projeto é a construção de um simulador capaz de realizar análise de ponto quiescente, disponibilizado com código-fonte que use da orientação a objetos para particionar a cadeia de operações requeridas para cumprir o objetivo em etapas funcionais.

Os objetivos secundários são desenvolvimento de interface, na forma de um leitor de arquivos *netlist* para utilizar do simulador sem lidar com código fonte, e realização de análise transiente.

O objetivo primário não visa desenvolver um simulador competitivo contra soluções existentes, mas sim pesquisa e desenvolvimento de um conjunto de

módulos requeridos para realização de simulações, que possam ser modificados e melhorados futuramente, sendo reutilizado para construir outros simuladores.

O objetivo de fornecer interface externa é demonstrar a eficácia do projeto para desenvolver produto final (na forma de programa executável compilado), e corresponde à adição de um interpretador de arquivos *netlist* ao conjunto de módulos desenvolvidos.

A realização de análise transiente pretende demonstrar uso da adaptabilidade proposta como objetivo primário, pois a simulação transiente pode ser realizada reaproveitando-se grande parte das etapas desenvolvidas para análise quiescente.

O código fonte é considerado parte do projeto, e decidiu-se que este deve ser aberto (no sentido de software livre) e em conformidade com o padrão da linguagem C++14, tornando-o independente de sistema operacional.

O simulador executável produzido é denominado MARA, um acrônimo para *Modular, Ajustável, Reutilizável e Adaptável*.

1.3 CADEIA DE FERRAMENTAS UTILIZADAS

A linguagem de programação escolhida é o C++, em padrão conhecido como C++14, dentro das especificações determinadas pela Organização Internacional de Normalização da linguagem, publicadas sob ISO/IEC 14882:2014. O compilador escolhido é o GCC, em versão superior ao 7.1, que define o C++14 como linguagem padrão para códigos de C++. O ambiente de desenvolvimento é um computador desktop com Windows 7 e Linux Ubuntu em versão superior a 14, dotado em ambos sistemas operacionais da IDE Codelite, versão superior a 10.0. No Windows, o porte do compilador utilizado é o projeto Mingw-w64. Para gerar gráficos de resultados, o projeto gera script para programa externo, o GNU Plot. O relatório final foi escrito no Microsoft Word 2010.

2 FUNDAMENTOS TEÓRICOS

Os fundamentos teóricos que justificam os passos adotados para atingir os objetivos propostos são apresentados nesta seção. A subseção 2.1 contextualiza os simuladores de circuitos elétricos no mercado atual, as subseções 2.2 a 2.4 ressaltam conceitos da Análise de Circuitos e da Álgebra que suportam os algoritmos elaborados, e as subseções 2.5 e 2.6 trazem conceitos da Ciência da Computação e da linguagem C++ usados como bases para estratégia de desenvolvimento.

2.1 CONTEXTO DOS SIMULADORES DE CIRCUITOS ELÉTRICOS

Um dos primeiros programas voltados à simulação de circuitos elétricos de notoriedade documentada é o *Spice*, lançado em 1973, sob as derivações do qual ainda são escritos a maioria dos simuladores disponíveis, comerciais ou não.

O problema da simulação de circuitos tem solução estabelecida e ferramentas robustas para o caso geral, mas o avanço da indústria e alterações no cenário mercadológico impõe adaptação das ferramentas existentes.

A indústria de alta tecnologia requer simuladores que considerem crescente número de efeitos secundários ou interdomínios, gerando mercado para desenvolvimento e venda de simuladores profissionais. Exemplo de simulador atual oferecido como produto para este tipo de aplicação é o *Spectre Circuit Simulator*[4] da empresa *Cadence*.

Controle de qualidade ou sigilo de etapas do processo produtivo também são necessidades que podem levar uma empresa a desenvolver simuladores de circuitos elétricos. Exemplo de simulador que surgiu como ferramenta interna de uma empresa e passou a ser oferecido como produto é uma variação do *NgSpice* mantido pela empresa europeia *Isotel*, denominada *d_process*, voltada a simulação de sinais mistos para circuitos embarcados.

Além do uso profissional, são desenvolvidos simuladores voltados para o público amador ou estudantes da eletrônica. Neste segmento é mais impactante o acesso a poder de processamento e avanços nas linguagens de programação, pois a simplicidade relativa destes simuladores em relação aos simuladores profissionais permite mais rápido desenvolvimento. Atualmente é possível acesso a simuladores

gratuitos e disponíveis online, com opção de processamento na nuvem, exemplificados pelo *PartSim*.

Análise do contexto mercadológico dos simuladores de circuitos elétricos leva o autor a crer que a realização deste projeto tem relevância para além de fins didáticos próprios, pois acesso a código fonte de algoritmos capazes de realizar simulações elétricas é por vezes requerido para desenvolvimento de produtos.

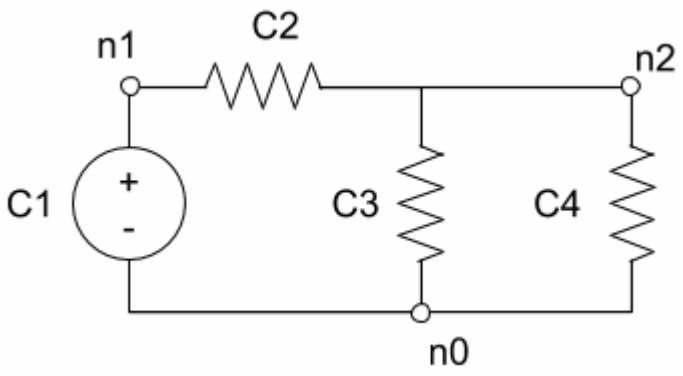
Exemplo de produto atual que tem características semelhantes à deste projeto é o *Qucs*, programa de código aberto para simulação de circuitos, escrito em C++, porém com foco na interface gráfica para o usuário.

2.2 ANÁLISE METODOLÓGICA DE CIRCUITOS ELÉTRICOS

A simulação computacional de circuitos elétricos requer definição de metodologia genérica para o tratamento do problema, que é apresentado na forma de um circuito elétrico de comportamento determinístico e grandezas elétricas na forma de incógnitas.

A abstração usual de um circuito elétrico é precedida pelo conceito de *componente* elétrico e *nodos* de conexão. Componentes são objetos com terminais expostos e que estabelecem comportamento definido entre as grandezas elétricas presentes nestes terminais e os nodos de um circuito elétrico são pontos de conexão entre terminais de componentes. Uma representação visual típica de um circuito elétrico é apresentada na Figura 1, onde componentes *c1..cK* conectam seus terminais a nodos *n0...nN*.

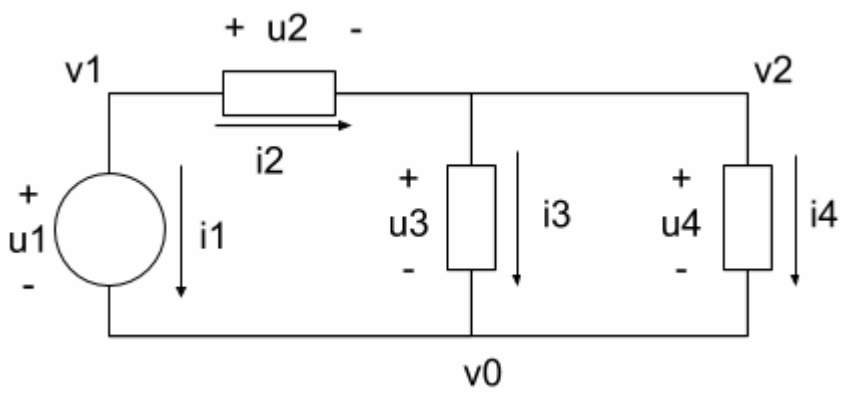
Figura 1. Representação de um circuito elétrico



Fonte: Autor

Sob o ponto de vista de análise de circuitos elétricos, o comportamento de um circuito pode ser completamente caracterizado se para cada um dos pontos de conexão, denominados nodos, for definida a tensão elétrica em relação a uma referência comum, juntamente à corrente elétrica entre dois terminais quaisquer de cada componente. A Figura 2 ilustra essas incógnitas como se dispõe no circuito da Figura 1, acrescidas da diferença de potencial entre os terminais de cada componente.

Figura 2. Representação de incógnitas na análise do circuito



Fonte: Autor

2.2.1 Leis de Kirchhoff para circuitos elétricos

Quando um circuito é representado por elementos concentrados, as Leis de Kirchhoff para circuitos elétricos podem ser utilizadas para gerar um sistema de equações que restringe as soluções para as incógnitas àquelas que representam comportamento realista. Estas leis baseiam-se na conservação de energia, manifestada no circuito como o fato de que um nodo não gera nem consome carga elétrica e que caminhos fechados do circuito não geram nem consomem potencial elétrico. Algebricamente, isto é representado pelas Equações 1 e 2, que expressam a Lei das Correntes de Kirchhoff e a Lei das Malhas de Kirchhoff, respectivamente.

Equação 1: Denominando Ω_n o conjunto de todos os ramos que se conectam ao nodo n, a soma das correntes i_k associadas a estes ramos é zero.

$$\sum_{k \in \Omega_n} i_k = 0\,, \qquad n = 1,2,3 \ldots N \tag{1}$$

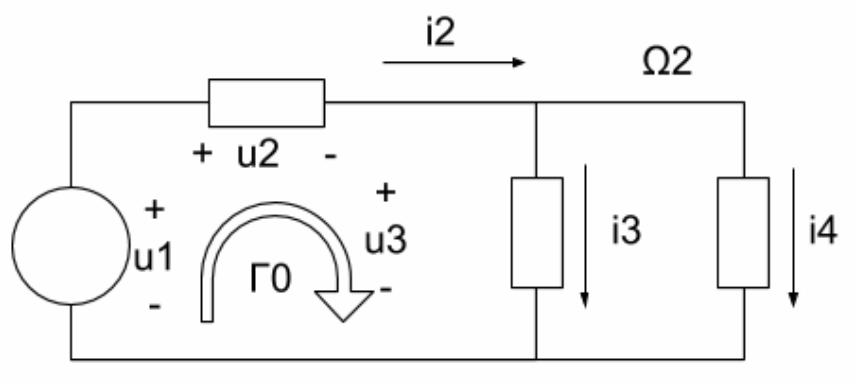
Equação 2: Denominando-se Γ um conjunto de ramos que constitui um caminho fechado no circuito, a soma das diferenças de potencial elétrico u_k entre os extremos destes ramos é zero.

$$\sum_{k\in\Gamma} u_k = 0\,, \qquad \forall\,\Gamma \tag{2}$$

Destaca-se que a diferença de potencial sobre um ramo pode ser inferida diretamente pelas tensões em seus terminais, pois para um ramo k, conectado entre os nodos a e b, $u_k = v_a - v_b$.

Exemplo das equações resultantes das Leis de Kirchhoff aplicadas ao nodo *n2* e um caminho fechado Γ_0 no circuito da Figura 2 é apresentado na Figura 3.

Figura 3. Exemplo de caminho fechado e tensões sobre este, e conjunto de correntes relacionadas a um nodo; Aplicação das Leis de Kirchhoff.



$$\Omega_2 \ni \{i_2, i_3, i_4\}$$

$$i_2 - i_3 - i_4 = 0$$

$$\Gamma_0 \ni \{u_1, u_2, u_3\}$$

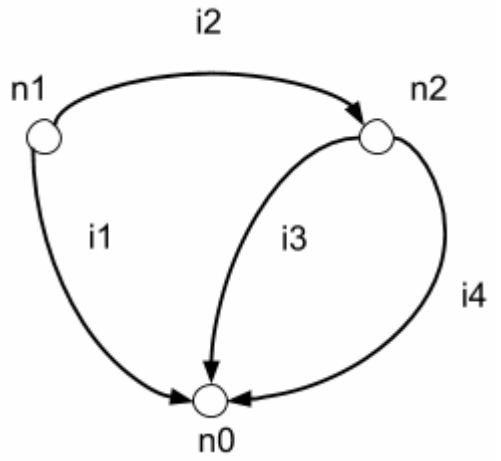
$$u_1 - u_2 - u_3 = 0$$

Fonte: Autor

2.2.2 Restrições topológicas

Usando de técnicas da Teoria dos Grafos, a topologia de um circuito pode ser representada como um grafo orientado, onde os nós representam os nodos do circuito e a orientação dos ramos representa sentido convencionado da corrente. As conexões de um grafo orientado podem ser representadas pela Matriz de Incidência, que por sua vez permite expressão matricial das Leis de Kirchhoff. A Matriz Incidência para o circuito da Figura 2 é apresentada na Figura 4, onde os elementos unitários (± 1) representam não o valor da corrente, mas se o sentido convencional desta adentra ou sai do respectivo nodo.

Figura 4. Representação do circuito da Figura 2 como grafo direcionado e Matriz de Incidência resultante, na abordagem adotada.



$$M_{y,x} = \begin{cases} +1 \ \ se\ corrente\ i_x\ sai\ de\ nodo\ n_y \\ -1 \ \ se\ corrente\ i_x\ entra\ em\ nodo\ n_y \\ 0\,, \quad\ c.c.\ \ (sem\ conexão) \end{cases}$$

$$m\ ramos \rightarrow$$

$$M = \begin{bmatrix} -1 & 0 & -1 & -1 \\ +1 & +1 & 0 & 0 \\ 0 & -1 & +1 & +1 \end{bmatrix} \downarrow n\ nodos$$

Fonte: Autor

Usando da Matriz de Incidência, e agrupando em vetores as incógnitas do circuito, as Leis de Kirchhoff podem ser expressas matricialmente conforme Equações 3 e 4; Nestas Equações, um vetor contendo a corrente sobre todos os ramos do circuito é denominado $\hat{\imath}$, um vetor contendo a queda de tensão sobre cada ramo é denominado $\hat{u}$, e um vetor contendo a tensão elétrica de cada nodo em relação a um nodo de referência é denominado $\hat{v}$. A matriz **A** é chamada Matriz de Incidência Reduzida, que difere da Matriz de Incidência pela ausência da linha representativa do nodo comum. A redução é realizada porque o conjunto de equações resultante dos equacionamentos pelas Leis de Kirchhoff não são independentes se a tensão do nodo de referência for mantida como incógnita (basta imaginar que, para um circuito qualquer, a adição de um valor de tensão elétrica idêntico a todos os nodos não altera o comportamento do circuito). Em geral, atribui-se o valor de zero Volt para o nodo comum, e esta é a abordagem adotada neste projeto.

$$A \times \hat{\imath} = 0 \tag{3}$$

$$A^T \times \hat{v} - \hat{u} = 0 \tag{4}$$

Destaca-se que os vetores $\hat{\imath}$, $\hat{u}$ e $\hat{v}$ são de incógnitas, e totalizam *2m+n-1* variáveis, e as Leis de Kirchhoff fornecem *m+n-1* equações.

Para que um circuito resulte em sistema definido de acordo com esta formulação, existem restrições topológicas que devem ser obedecidas pelo grafo equivalente deste. Estas restrições topológicas possuem analogias com restrições construtivas do circuito, e são referidas na segunda forma neste projeto. As restrições topológicas consideradas são:

a) o circuito não deve possuir ramos singulares: um ramo singular possui as duas extremidades conectadas ao mesmo nodo. Neste caso, a formulação matricial baseada no grafo gera equações dependentes;
b) o circuito não deve possuir subcircuitos desconexos: se um circuito possui partes completamente desconexas, não é possível relacionar as circulações de corrente conforme requisitado para que a matriz de incidência as mapeie para um sistema definido;
c) o circuito não deve possuir terminais livres: se o circuito possuir um caminho de circulação aberto (o equivalente a deixar, fisicamente, o terminal de algum componente desconectado do circuito), não é possível relacionar as circulações de queda de potencial conforme requisitado para que a matriz de incidência as mapeie para um sistema definido.

Além destas, existem restrições topológicas cuja verificação não foi implementada e são assumidas verdadeiras para todos os circuitos que são fornecidos ao simulador:

a) um circuito não deve possuir laços compostos unicamente por fonte de tensão: a soma das quedas de potencial em um laço fechado do circuito é usada como restrição pelas Leis de Kirchhoff. Quando um laço é formado exclusivamente por fontes de tensão, a união das imposições individuais das fontes com a equação de circulação gera subcircuitos independentes;
b) o grafo de um circuito não deve possuir um corte topológico cujas arestas sejam unicamente fontes de corrente: ao realizar-se um corte do circuito tem-se o equivalente a um supernodo para o qual, pelas Leis de Kirchhoff a soma das correntes nas arestas que o ligam ao circuito completo tem de ser nula. Caso seja possível gerar um super nodo conectado exclusivamente por fontes de corrente ao resto do circuito, não há liberdade para realizar o somatório imposto pela Lei de Kirchhoff, pois o somatório está limitado pelas imposições das fontes.

2.2.3 Ramos Fundamentais

O método considerado para atrelar um grafo a determinado circuito e gerar deste um sistema de equações equivalente faz uso de ramos simples, conectados a apenas dois nodos. Esta abstração é insuficiente para a proposta do projeto, pois existem componentes elétricos com mais de dois terminais.

O projeto assume que qualquer componente possa ser aproximado por um agregado de ramos básicos, conforme teoria da análise de circuito. Estes ramos estabelecem relação idealizada entre a diferença de potencial nos seus terminais e/ou a corrente que circula entre estes. Os ramos fundamentais podem ser:

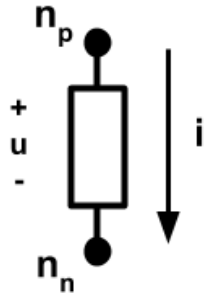
a) resistivos: permitem fluxo de corrente linearmente dependente da diferença de potencial elétrico entre seus terminais, através de uma quantidade representativa de sua resistência ohmica, expressa como R. A relação tem a forma $i = u/R$;
b) capacitivos: relacionam a corrente entre os terminais à derivada instantânea da diferença de potencial entre estes, através quantidade representativa de sua capacitância elétrica, expressa como C, na forma $i = C \cdot du/dt$;
c) indutivos: relacionam a diferença de potencial entre seus terminais à derivada instantânea da corrente sobre si, através de quantidade relativa à sua indutância, denominada L. A relação tem a forma $u = L \cdot di/dt$;
d) fontes de diferença de potencial: relacionam a diferença de potencial entre seus terminais à uma função específica. Quando a função em questão é um valor constante, por exemplo, é o ramo representativo de uma fonte de tensão fixa ideal.
e) fontes de corrente: atrelam a corrente que circula pelo ramo à uma função específica. Quando a função em questão é um valor constante, por exemplo, é o ramo representativo de uma fonte de corrente fixa ideal.

Os ramos resistivos, capacitivos e indutivos são ramos passivos, e as fontes de corrente e tensão são ramos ativos. Os ramos passivos capacitivos e indutivos são denominados ramos de primeira ordem.

Para todos os ramos fundamentais, são utilizadas as siglas n_p para referenciar-se ao nodo considerado positivo, n_n para referenciar-se ao nodo considerado negativo, u para referenciar-se a queda de potencial elétrico entre n_p e n_n, e i para corrente que flui pelo ramo, com sinal algébrico considerando fonte em n_p e

sumidouro em n_n, conforme Figura 5, onde a definição de corrente usa da notação $dQ(n_p, n_n)$ para denotar o fluxo de carga instantâneo do nodo n_p para o nodo n_n, através do ramo em questão, e $V(n_x)$ representa a diferença de potencial elétrico entre o nodo n_x e o nodo de referência n_0.

Figura 5: Definição de grandezas elétricas de um ramo em relação ao circuito

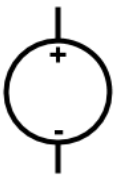
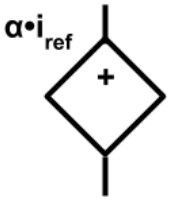
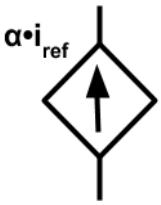
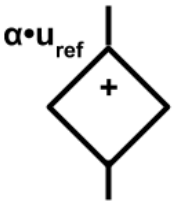
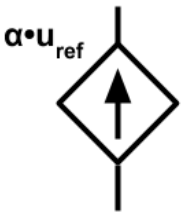


$$V(n_0) := 0$$
$$u = V(n_p) - V(n_n)$$
$$i = dQ(n_p, n_n)$$

Fonte: Autor

As funções de relação consideradas para os ramos de fonte são de três tipos: constantes, escalar de corrente em outro ramo, escalar de diferença de potencial entre dois nodos arbitrários do circuito. A Figura 6 representa ilustração de cada tipo de ramo, conforme serão ilustrados ao longo deste documento, onde i_{ref} representa determinada corrente de referência, u_{ref} determinada diferença de potencial de referência, e α determinado ganho para os ramos ativos controlados.

Figura 6. Ilustração dos Ramos fundamentais considerados

<table>
<tr><th colspan="3">Ramos Passivos</th><th colspan="3">Ramos Ativos</th></tr>
<tr><td>Resistivos</td><td colspan="2">Primeira Ordem</td><td colspan="2">Fixos</td><td colspan="2">Controlados</td></tr>
<tr><td>Resistivo</td><td>Capacitivo</td><td>Indutivo</td><td>diferença de potencial</td><td>corrente</td><td>diferença de potencial</td><td>corrente</td></tr>
<tr><td rowspan="2">  </td><td rowspan="2">  </td><td rowspan="2">  </td><td rowspan="2">  </td><td rowspan="2">  </td><td>controlado por corrente
  </td><td>controlado por corrente
  </td></tr>
<tr><td>controlado por tensão
  </td><td>controlado por tensão
  </td></tr>
</table>

Fonte: Autor

2.2.4 Equacionamento Nodal Modificado

É denominado Equacionamento Nodal Modificado um conjunto de métodos para gerar equações lineares nas quais a corrente dos ramos seja dependente das tensões nos nodos do circuito. De posse destas equações, a Lei das Correntes de Kirchhoff pode ser aplicada para gerar um sistema definido que possui as tensões em cada nodo como incógnita.

A principal motivação para usar da análise nodal e da Lei das Correntes ao invés da Lei das Malhas é que esta é genérica para qualquer topologia de circuito elétrico, inclusive para circuitos não-planares.

O equacionamento da corrente i_k, sobre um ramo resistivo de índice k, de resistência R_k, dependente das tensões nos nodos do circuito, onde v_{kp} é a incógnita associada ao valor de tensão elétrica no nodo positivo do ramo e v_{kn} é a incógnita associada ao valor de tensão no nodo negativo do ramo, é apresentado na Equação 5:

$$i_k = \frac{v_{kp}}{R_k} - \frac{v_{kn}}{R_k} \quad (5)$$

A Equação 5 pode ser substituída na Equação 1, Lei das Correntes de Kirchhoff, que é reproduzida aqui por conveniência:

$$\sum_{k \in \Omega_n} i_k = 0\,, \qquad n = 1,2,3 \ldots N$$

Realizando as N substituições possíveis são geradas N equações, conforme ilustrado na Equação 6, onde Ω_{n+} representa o conjunto de todos os ramos cujo terminal positivo conecta-se ao nodo n, e Ω_{n-} representa o conjunto de todos os ramos cujo terminal negativo conecta-se ao nodo n:

$$\left(\sum_{k \in \Omega_{n+}} \frac{v_{kp}}{R_k}\right) - \left(\sum_{k \in \Omega_{n-}} \frac{v_{kn}}{R_k}\right) = 0, \qquad n = 1,2,3 \ldots N \quad (6)$$

A construção destas equações é análoga a calcular a Matriz de Condutância de um circuito puramente resistivo.

Estendendo o sistema para contemplar ramos fontes de corrente fixa, a corrente destes ramos é dada pela Equação 7, onde I_x é uma constante, que pode ser adicionada diretamente ao resultado líquido da soma de correntes na Equação 6.

$$i_k = I_x \quad (7)$$

Ramos fontes de corrente controlada tem a corrente sobre si conforme Equação 8, onde α é o ganho característico do ramo, e ref_k é a referência de controle:

$$i_k = \alpha \cdot (ref_k) \quad (8)$$

Ramos fontes de diferença de potencial estabelecem relação entre duas incógnitas, ao invés de dependerem delas, e adicionam nova equação ao sistema. O método adotado para manter o sistema definido é a adição de uma nova incógnita, representativa da corrente sobre o ramo.

Para cada ramo fonte de tensão, de índice k, é definida uma incógnita i_{auxk} que representa o valor da corrente através da fonte, e o equacionamento da corrente para estes ramos é conforme Equação 9:

$$i_k = i_{auxk} \quad (9)$$

Um ramo fonte de tensão fixa adiciona Equação 10 ao sistema:

$$v_{kp} - v_{kn} = V_x \tag{10}$$

Ramos fonte de tensão controlada adicionam Equação 11 ao sistema, onde α é o ganho característico do ramo, e ref_k é a referência de controle:

$$v_{kp} - v_{kn} = \alpha \cdot (ref_k) \tag{11}$$

Quando a variável de referência de ramos controlados é a diferença de potencial entre dois nodos, esta quantidade está implícita nas incógnitas do sistema, conforme Equação 12:

$$ref_k = v_{refp} - v_{refn} \tag{12}$$

Para ramos controlados cuja variável de referência é a corrente sobre outro ramo, a equação de corrente do ramo correspondente é utilizada para determinar ref_k.

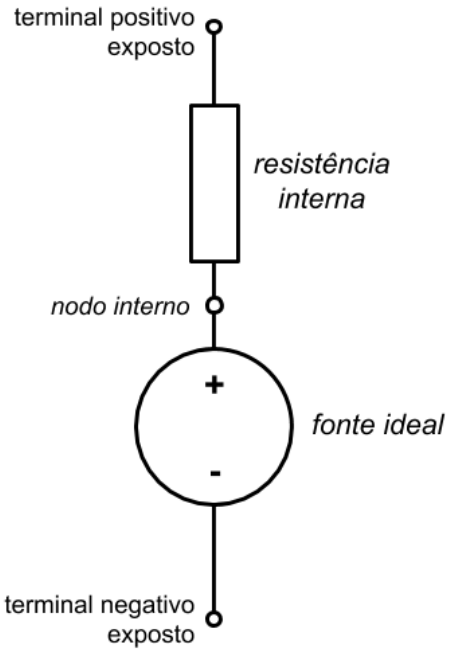
Ramos capacitivos ou indutivos não possuem equacionamento linear de suas correntes para comportamento ao longo do tempo, mas seu comportamento instantâneo pode ser aproximado por um ramo fonte. Para ramos capacitivos, o comportamento instantâneo é aproximado por um ramo fonte de tensão e, para ramos indutivos, o comportamento instantâneo é aproximado por um ramo fonte de corrente.

Através dos métodos apresentados, um circuito de N nodos, composto por M ramos fundamentais, pode ser usado para gerar $N + NV$ equações, onde NV é a soma da quantidade de ramos fonte de tensão presentes no circuito. Respeitadas as restrições topológicas apresentadas na subseção 2.2.2, a equação relativa ao nodo referência pode ser descartada, e as equações resultantes agregadas em sistema cujas primeiras $N - 1$ incógnitas representam potencial elétrico de cada nodo em relação ao referencial, e as NV incógnitas seguintes representam a corrente sobre cada ramo fonte de tensão.

2.3 MODELAGEM DE COMPONENTES

Os componentes elétricos, com suas não idealidades, podem ser aproximados por um conjunto de ramos básicos ideais. Um exemplo disso é uma bateria, cuja principal não idealidade pode ser aproximada por uma resistência em série com uma fonte ideal[15], conforme Figura 7.

Figura 7. Modelagem de fonte não ideal



Fonte: Autor

Destaca-se que o conjunto agora possui um nodo não exposto entre o ramo fonte de tensão e o resistor. Estes *nodos internos* são considerados na análise, mas não costumam ser considerados pela abstração que se tem do componente elétrico como um todo.

Para separar os níveis de abstração entre um componente não ideal e seus ramos constituintes, o projeto usa de *subcircuitos*, assim como faz, por exemplo, o simulador 5Spice. A aplicação dessa estratégia no simulador final foi possibilitar diferentes modelos para cada componente.

O modelo ideal foi implementado para todos os componentes, e modelos não ideais para alguns: fontes de tensão, fontes de corrente e diodo. As fontes com resistência interna foram escolhidas por serem consideras pelo autor como boas primeiras aproximações do comportamento real de um circuito idealizado, e por terem representação relacionada com os Teoremas de Thevenin - para fontes de

diferença de potencial – e com o teorema de Norton – para fontes de corrente. O diodo foi escolhido por ser considerado pelo autor componente fundamental na eletrônica, e possuir modelagem através da Equação de Shockley, que também serve como caso de prova para a modelagem de comportamento não linear proposto na subseção 2.6.

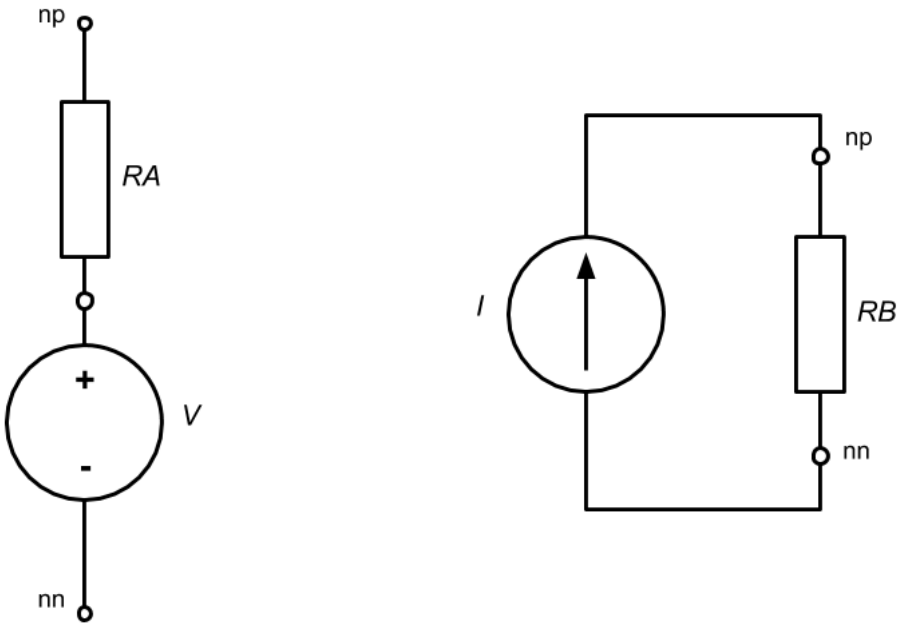
Uma fonte com resistência interna $R_{interna}$ é representada pela junção de um resistor em série ou em paralelo, de valor determinado pelas Equações 13 ou 14, para o caso de fontes de tensão ou corrente, respectivamente.

$$RA = R_{interna} \tag{13}$$

$$RB = 1/R_{interna} \tag{14}$$

O subcircuito resultante é ilustrado na Figura 8 para fontes fixas. Fontes controladas possuem configuração equivalente (com substituição do ramo de fonte fixa pelo ramo de fonte controlada).

Figura 8. Equivalente de Fontes com resistência interna



Fonte: Autor

Diodos ideias são representados por uma alta resistência quando seus terminais estão expostos a uma diferença de potencial abaixo da tensão de avalanche, e por uma resistência baixa quando expostos a uma diferença de potencial acima desta, de acordo com a Equação 15, onde R_{ALTA}, R_{BAIXA} e $V_{avalanche}$ são parâmetros utilizados para construir o modelo.

$$R_d = \begin{cases} R_{ALTA} \ , & u_k < V_{avalanche} \\ R_{BAIXA} \ , & u_k \geq V_{avalanche} \end{cases} \quad (15)$$

O modelo para diodos não ideais é realizado de maneira análoga – através de uma resistência variável – porém de forma a satisfazer equação no formato da Equação 16 – a Equação de Shockley - considerando que $R_d = u_k/i_k$, e u_k e i_k são a diferença de potencial e a corrente sobre o diodo, respectivamente. I_s é a corrente de saturação e V_T é uma constante térmica (efeitos térmicos não são considerados no projeto).

$$i_k = I_S \cdot (e^{u_k/(V_T \cdot n)} - 1) \quad (16)$$

Para condicionar um circuito completo ao equacionamento nodal modificado proposto na subseção 2.2.4, cada um dos componentes fornece o subcircuito representativo do modelo escolhido, esses modelos são agrupados de acordo com algoritmos apresentados na seção 3, *Execução do Projeto*, em específico na subseção 3.3.3, *Circuitos de Subcircuitos*.

2.4 CONCEITOS BASILARES DA ORIENTAÇÃO A OBJETOS E DO C++

Para desenvolver o programa de computador objetivado neste projeto, foi considerado o uso eficaz de recursos computacionais, que tem caráter quantitativo, e modularidade do código, avaliada de forma qualitativa pelo autor.

A quantificação dos recursos necessários para todas as etapas do programa estão fora do escopo do trabalho, mas algumas decisões baseadas no seu uso racional são apresentadas nesta seção, em principal a forma como a memória é organizada usando de classes na subseção 2.4.1, e as principais caraterísticas da linguagem escolhida quanto à manipulação de objetos, na subseção 2.4.2.

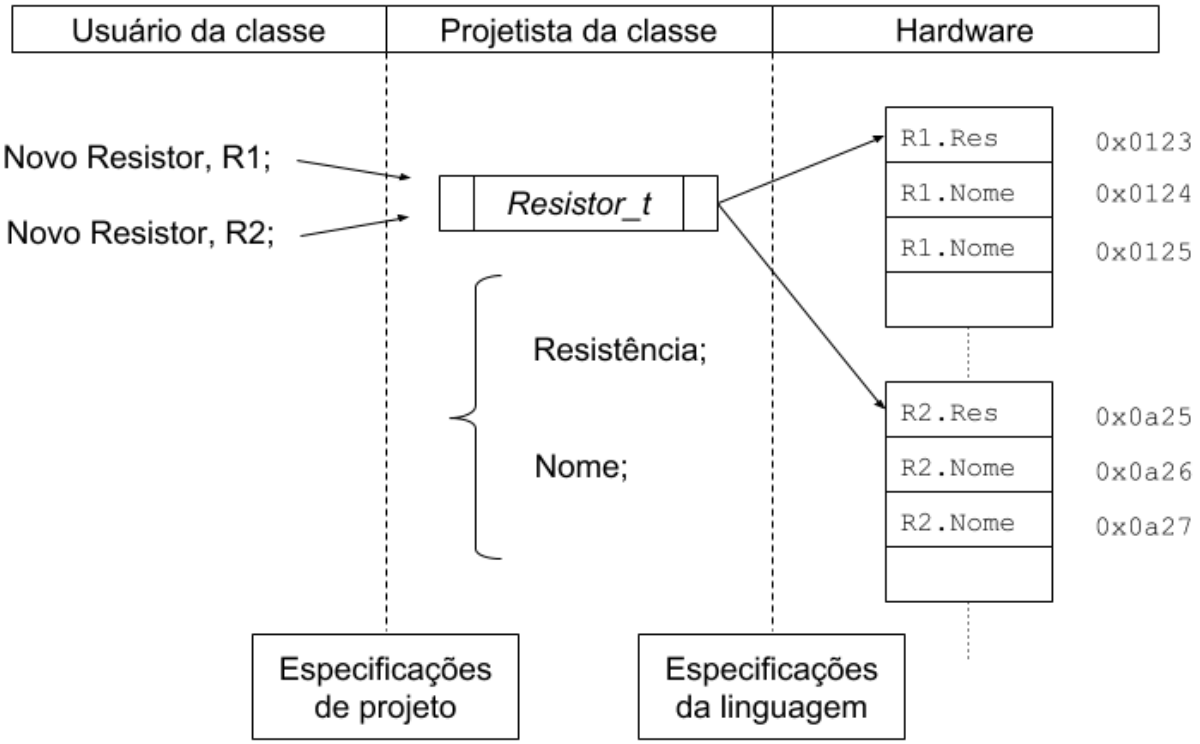
2.4.1 Objetos e Classes

A orientação a objetos introduz o conceito de *classes* para administrar o controle sobre a informação em áreas da memória. Ao definir uma classe para representar Resistores, denominada, por exemplo, *Resistor_t,* mesmo que este seja representado por *um número*, o programador não mais requisita explicitamente memória para armazenar *um número* quando requer um resistor em seu algoritmo mas, ao invés disto, requisita memória para um *Resistor_t*, que é traduzido pelo

compilador, usando da definição provida para classe *Resistor_t*, na quantidade de memória efetivamente requerida.

Classes são apenas definições, e não entidades em si dentro do programa. Quando uma variável de determinada classe é concretizada, tem-se um *objeto*. Cada variável utilizada em um programa escrito em linguagem orientada a objetos é uma instância de determinada classe. No contexto deste projeto, o termo *tipo* é usado de forma intercambiável com o termo *classe*, pois dada certa classe de objetos, de nome T, objetos pertencentes à classe T são ditos objetos do tipo T. Uma ilustração do conceito de objetos é apresentado na Figura 10, (onde os endereços em hardware apresentados na última coluna são fictícios para fins de ilustração).

Figura 9. Representação da definição de objetos



Fonte: Autor

Uma classe pode conter objetos e métodos, ditos *membros da classe,* e esses objetos e métodos podem ser expostos ou protegidos. A proteção de membros internos ao funcionamento da classe é útil no projeto de código modular, pois o projetista só precisa emitir garantias sobre métodos e objetos expostos, ficando livre para alterar o comportamento de métodos internos.

Uma classe também pode ser usada como base para classes ditas derivadas, que herdam variáveis e métodos da classe base. Classes ditas virtuais são classes

base que não contém representação completa para ser instanciada, mas servem de base para classes completas. Neste projeto, herança de classes é usada extensivamente para generalizar algoritmos. Todos os componentes elétricos, por exemplo, são derivados de uma mesma classe denominada *Componente*. Assim sendo, um Capacitor, por exemplo, pode ser implementado de forma independente de como é implementado um Resistor, desde que ambos cumpram com uma interface exposta pela classe base, e os algoritmos que manipulam componentes são os mesmos para ambos.

2.4.2 Características do C++

O C++ surgiu de um projeto para agregar o conceito de classes para a linguagem C, iniciado em 1979, padronizado em 1998. O C++ é um conjunto de especificações que devem ser atendidas por um compilador. Existe um comitê internacional que mantém e atualiza estas especificações, mas não existe um compilador oficial. Exemplos de compiladores são: Microsoft Visual C++, GCC, e Clang. O C++ é, hoje, umas das linguagens de programação mais importantes do mundo.

As características consideradas principais do C++ são:

a) Orientação a Objeto: Este é o paradigma basilar da linguagem. As variáveis de um programa contêm estado interno e métodos próprios associados de acordo com uma especificação abstrata;
b) *Tipagem* forte: As variáveis declaradas são de *tipo* específico, e a linguagem possui restrições semânticas estáticas (exemplo: uma função que opera sobre objetos do tipo *Capacitor* não pode ser evocada para operar sobre *Indutores*);
c) Compilada: O C++ supõe que o código fonte será convertido para código de máquina (ao invés de interpretado, como na linguagem Python, por exemplo);
d) Nível arbitrário de abstração: O C++ propicia acesso a características intrínsecas do hardware no qual o programa é executado (como endereçamento de memória e garantias quanto a quantidade de memória requerida por cada tipo de variável);
e) Biblioteca Padrão: Um vasto conjunto de algoritmos e classes é definido pela linguagem [22] e deve ser distribuído em conjunto com os compiladores. A biblioteca padrão do C++ (*C++ Standard Library*, ou apenas *std* library) fornece suporte para noções como organizar uma lista ou procurar um

elemento nesta, que são utilizadas como blocos fundamentais para elaboração de algoritmos mais complexos;

f) Multiparadigma: Característica decorrente da flexibilidade da linguagem, que torna possível emular outros paradigmas em C++. O paradigma funcional é usado extensivamente na biblioteca padrão pra fornecer conceitos como *comparar dois objetos,* usando dos chamados objetos funcionais (objetos desenvolvidos com o intuito de representar operações, e não estados). Neste projeto, o paradigma funcional foi utilizado para realização de componentes em um *Estoque de componentes* (em detalhes na subseção 3.3).

2.4.3 A Notação O-Grande

O custo computacional de um algoritmo é dependente da quantidade de objetos sobre os quais o algoritmo tem de operar, como a quantidade de nodos em um circuito, ou a quantidade de componentes que tem de ser ordenado em um segmento da memória. Como o principal interesse sobre o custo é como este se comporta com o crescimento da quantidade de dados de entrada, é usada a notação O-Grande, que representa o comportamento do custo conforme o número de entradas tende ao infinito.

A seguinte lista de exemplos visa elucidar a notação O-Grande sem que seja necessário recorrer a referências externas: Um algoritmo de custo $O(1)$ consome uma quantidade máxima constante de recursos, uma função de custo $O(n)$ consome uma quantidade máxima de recursos que é múltipla de uma função linear dependente do número de entradas, uma função de custo $O(n^2)$ consome uma quantidade de recursos máxima que é múltipla de uma função quadrática dependente do número de entradas, e assim sucessivamente.

Notar que o custo pela notação O-Grande não se refere a contagem de operações ou blocos de memória em nenhuma unidade específica, no sentido de que, atrelando a quantidade de operações requeridas ao tempo, e o custo de memória a bytes, o comportamento não se altera quando a medida de custo de tempo for feita em segundos ou horas, nem quando a medida de custo de memória for feita em bytes ou megabytes.

2.5 ESTRATÉGIAS DE PROGRAMAÇÃO

2.5.1 Contêineres de dados e iteradores

Contêineres de dados são estruturas que tem por objetivo armazenar e prover acesso a outros objetos. As principais diferenças entre tipos de contêineres são a forma como os itens contidos são armazenados na memória e como podem ser acessados. Foram escolhidos (da biblioteca padrão) e desenvolvidos (em específico para este projeto) contêineres que garantissem métodos de acesso desejado, com objetivo secundário de minimizar a complexidade computacional.

A maioria dos contêineres tem por objetivo organizar vários objetos, mas existem contêineres cujo objetivo é encapsular um agregado de informação em uma interface padrão, formando um único objeto. Os principais conceitos de contêineres utilizados através da biblioteca padrão no projeto são *Arrays*, *Vetores*, *Pares*, *Mapas*, *Conjunto de Únicos*, *Fila*, *String*, *Objeto Funcional*, e *Objetos de Exceção*.

Os principais contêineres desenvolvidos especialmente para o projeto foram as classes denominadas *Indexador* e *Conectado*. A classe *Indexador* tem função de manter índice fixo atrelado a elementos adicionados em seu vetor interno, e a classe *Conectado* tem função de atrelar um elemento a um vetor de índices que representa onde este conecta-se. Ambas as classes desenvolvidas são *gabaritos metaprogramados* (conforme definido na subseção 2.5.3).

2.5.2 Avaliação postergada

É denominada avaliação postergada - ou avaliação atrasada - uma técnica computacional na qual uma expressão não é efetivamente computada até que seu resultado seja explicitamente requerido. Neste projeto, a avaliação postergada é utilizada para reter o cálculo dos índices de nodos ou ramos utilizados por um circuito, através do contêiner *Indexador*.

2.5.3 Metaprogramação

Sendo uma linguagem fortemente tipada, o C++ exige que os algoritmos tenham bem definidos sobre que tipo de dados estão operando em cada comando. Isso pode ser contra produtivo no sentido de reutilização de código, pois, por exemplo, as operações para conectar um Capacitor e um Indutor compartilham da

mesma lógica operacional. Para atacar este problema, o C++ usa do conceito de *metaprogramação* para possibilitar escrita de funções e classes *gabarito*, que são traduzidos para código explícito quando requisitados para um conjunto específicos de tipos envolvidos. A metaprogramação recebe esse nome por ser uma forma de programar a geração do código fonte, em etapa anterior a compilação. Assim como uma função tem resultado dependente dos valores passados como argumento durante a execução de um programa, o código fonte gerado por uma classe gabarito é dependente dos tipos passados como argumento durante a etapa de compilação.

2.6 MÉTODO ITERATIVO PARA COMPORTAMENTOS NÃO-LINEARES

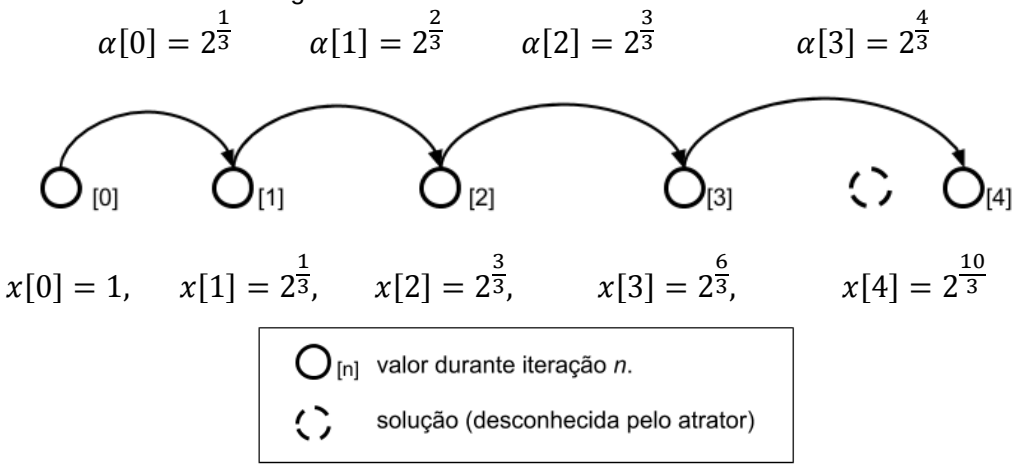
Para estabilizar o valor de componentes cujo comportamento não é linear, foi desenvolvida uma classe que representa um método atrator para um ponto estável.

Componentes com ramo não linear são requisitados a ajustarem suas propriedades de forma a aproximarem-se de um valor que resolve sua equação característica dentro de margem estipulada aceitável pela simulação.

A classe que manipula o atrator é denominada *Jogral*, e estipula um valor inicial para a propriedade não linear sendo simulada. Após cada iteração, é verificado se o valor da variável deve crescer ou decrescer e este valor é então multiplicado por uma constante. Para permitir aproximação arbitrária do ponto de solução, o valor pelo qual a propriedade está sendo iterada é multiplicado a cada iteração e é ajustado de acordo com uma avaliação do algoritmo se a solução está convergindo ou não.

Caso o valor esteja *subindo* (sendo multiplicado por uma constante maior que a unidade), e a equação a ser resolvida esteja convergindo, o atrator continua incrementando a propriedade e a taxa de crescimento desta, conforme Figura 11, onde o fator de incremento inicial é $\alpha = \sqrt[3]{2}$, que foi escolhido por ser número irracional maior que a unidade, e o valor inicial do algoritmo é $x_0 = 1$, escolhido por ser elementro neutro da multiplicação (permitindo usar o atrator como multiplicador de outros valores).

Figura 10. Atrator incrementando estimativa

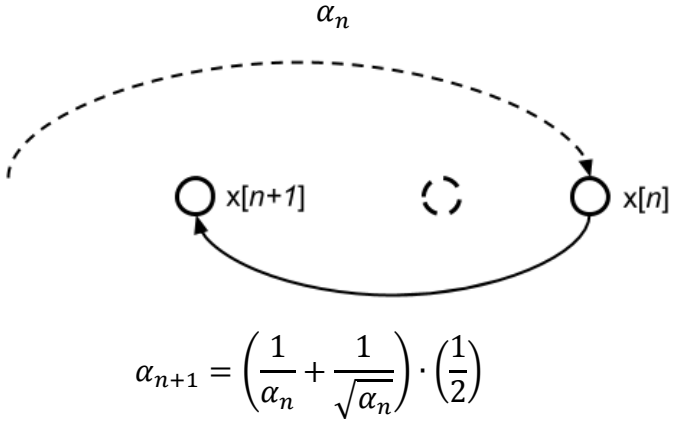


Fonte: Autor

Caso o valor esteja *subindo*, e a estimativa afastando-se da solução, um novo fator de multiplicação é calculado para decrementar a estimativa, conforme ilustrado na Figura 12. O valor não é simplesmente invertido, pois isto incorreria na repetição de estimativas prévias, mas calculado de acordo com a Equação 17:

$$\alpha_{n+1}(inversão) = \left(\frac{1}{\alpha_n} + \frac{1}{\sqrt{\alpha_n}}\right) \cdot \left(\frac{1}{2}\right) \tag{17}$$

Figura 11. Atrator decrementando estimativa após sucessivos incrementos



Fonte: Autor

O atrator funciona da mesma maneira quando o valor está decrescendo a cada iteração, também usando da fórmula (17) para inverter o coeficiente de multiplicação. Notar que:

$$\left(\frac{1}{\alpha_n}+\frac{1}{\sqrt{\alpha_n}}\right)\cdot\left(\frac{1}{2}\right)>1\ ,\qquad \forall\ \alpha_n\in(0,1)$$

$$\left(\frac{1}{\alpha_n}+\frac{1}{\sqrt{\alpha_n}}\right)\cdot\left(\frac{1}{2}\right)<1\ ,\qquad \forall\ \alpha_n\in(1,\infty)$$

2.7 MÉTODO DE INTEGRAÇÃO PARA COMPONENTES COM ESTADO

Como ramos capacitivos e indutivos são considerados instantaneamente fontes de valor fixo, a evolução do estado destes é obtida por integração numérica através do Método de Euler, de formulação representada na Equação 18, onde Δt é o passo da simulação, definido no projeto durante construção da classe de Análise Transiente.

$$E_{n+1}=E_n+\Delta\mathrm{t}\cdot\frac{\mathrm{dE}}{\mathrm{dt}} \qquad (18)$$

Para Capacitores, representados por uma fonte de diferença de potencial, o estado é o valor da fonte, e a derivada instantânea é dependente da corrente sobre este, pela relação $i=C\cdot dV/dt$. Para indutores, cujo estado é o valor instantâneo de uma fonte de corrente, a derivada é dependente da queda de potencial sobre este, $u=L\cdot dI/dt$.

3 EXECUÇÃO DO PROJETO

Esta seção detalha como cada módulo requerido para construção do simulador foi implementado, trazendo relações dos fundamentos teóricos da seção 2 com o código-fonte em si. A definição dos tipos fundamentais é apresentada na subseção 3.1, a definição dos algoritmos e classes utilizadas para manipulações algébricas é apresentada na seção 3.2, os componentes são apresentados na subseção 3.3, o leitor de arquivos *netlist* é apresentado na subseção 3.4, e o condicionamento dos componentes para simulação é apresentado na subseção 3.5. Trechos de código considerados mais relevantes são transliterados como excertos de código para as correspondentes subseções, e o código fonte completo foi disponibilizado em um repositório aberto no GitHub.

3.1 PREDEFINIÇÕES DO PROGRAMA

O código-fonte desenvolvido contém um arquivo considerado por todos os outros como basilar. Neste arquivo, denominado *Dogmas*, são definidos os tipos numéricos usados para representação de grandezas contínuas e o tipo numérico considerado para indexamento de elementos, que são:

a) Grandezas contínuas: Quando for referido um *número*, sem especificação adicional, é referida a representação de um número real. A manipulação dos componentes é programada para aceitar qualquer um dos tipos numéricos fornecidos pela linguagem, mas o simulador será construído usando de *double*s. Os *float*s conseguem representar cerca de sete dígitos decimais significativos, e os *double*s cerca de dezesseis. Assumindo que os últimos dígitos perdem significância durante a manipulação numérica, o uso de *floats* restringiria em demasia a faixa dinâmica das grandezas simuláveis. No código fonte, o tipo numérico é declarado como *NUM_T*, definido de forma análoga a:

$$número \triangleq double\ (64\ bits, IEEE\ 754);$$

b) Índice: Para manter relação entre um objeto e sua posição na memória, sem usar de ponteiros diretos ao hardware (que exige compromisso adicional entre as classes contêineres e os utilizadores destas), é adotada a estratégia de indexação. Independentemente de como um item for armazenado em algum contêiner desenvolvido pelo projeto, poderá ser acessado por um índice positivo. Apesar de positivos, operações entre índices podem resultar em resultado negativo, por tal foi escolhido um tipo *com sinal*. O tamanho escolhido para os índices foi de 32 bits. O uso de um inteiro de 32 bits com sinal fornece espaço para que os algoritmos operem sem se preocupar com *overflow* ou *underflow* para uma faixa ampla de complexidade de circuitos, pois com estes é possível indexar até 2^{31} itens distintos (como componentes ou nodos). No código fonte, o tipo de indexação é declarado como *ind_t*, definido de forma análoga a:

$$índice \triangleq std :: int32\_t\ (inteiro\ de\ 32\ bits\ com\ sinal\ );$$

Como é recorrente a necessidade de operar sobre um *Vetor* de *Índices* ou um *Vetor de Números* estes tipos foram abreviados como *inds_t* e *NUMs_t*, respectivamente.

3.1.1 Indexador e Vetor Indexado

Para desenvolver uma classe que aliasse o baixo custo de procura do *Vetor* com um mecanismo de indexação fixa (pois quando um elemento é removido de um *Vetor*, todos os elementos subsequentes são movidos para que não haja espaço não-utilizado na memória[26]), foi desenvolvida uma classe auxiliar denominada *Vetor Indexado*.

O *Vetor Indexado* usa de uma classe denominada *Indexador* para gerar índices únicos e imutáveis para objetos inseridos neste. O *Indexador* é definido conforme mostra Excerto 1.

Excerto de código 1: Declaração de Indexador

```
class Indexador
{
        ind_t contador;
        std::deque<ind_t> livres;
        inds_t mutable contiguo;
        bool   mutable contiguo_relaxado;
};
```

Fonte: Autor

O *Indexador* possui membros *mutáveis* para permitir avaliação atrasada de quais índices estão atualmente sendo utilizados. Quando um novo índice é requerido para indexar novo elemento, a classe primeiramente procura se existe algum índice livre na memória (que anteriormente referenciava algum elemento que foi removido) e o reutiliza. Apenas não havendo índices livres o vetor interno é expandido. Esta técnica permite que circuitos alterem sua topologia adicionando e removendo ramos sem que os índices utilizados tenham de ser recalculados imediatamente.

3.1.2 Ramos Fundamentais

Ramos fundamentais foram desenvolvidos através de uma classe denominada *Ramo Base*, que contém um parâmetro numérico atrelado à sua propriedade principal, uma etiqueta de que tipo de ramo representa, e índices relativos a elementos de controle, para o caso do ramo ser controlado.

Os ramos fundamentais foram desenvolvidos através de uma estrutura simples, sem uso de classes base e derivadas, conforme exposto no Excerto 2, onde *Ramo_Tipo_enum_t* é uma enumeração dos possíveis tipos de ramo.

Excerto de código 2: Declaração de Ramo Base e tipos deste

```
enum class Ramo_Tipo_enum_t : ind_t {
                Resistivo,
                DDP_Fixa,
                Corrente_Fixa,
                DDP_C_DDP,
                DDP_C_Corrente,
                Corrente_C_DDP,
                Corrente_C_Corrente,
                Capacitivo,
                Indutivo
        };
struct Ramo_Base
{
        Ramo_Tipo_enum_t tipo;
        NUM_T pr; //parâmetro
        ind_t  cp, cn; //índices de controle
        Ramo_Base() = default ; //construtor
};
```

Fonte: Autor

3.1.3 Circuito Ramificado

Para representar um circuito composto pelos ramos básicos apresentados na subseção anterior, foi desenvolvida uma classe denominada *Circuito Ramificado*, que contém um *Vetor Indexado* interno de ramos básicos, conforme Excerto 3. Esta classe também possui variáveis atreladas a avaliação atrasada.

Excerto de código 3: Definição do Circuito Ramificado

```
struct Circuito_Ramificado
{
        //ramos que compõe o circuito e nodos a que se conectam
        VetorIndexado<Conectado<Ramo_Base>> ramos;
        //sinalizador para avaliação postergada de nodos indexados
        bool   mutable nodo_relaxado;
        //conjunto ordenado de nodos utilizados
        inds_t mutable nodos_indexados;
};
```

Fonte: Autor

3.1.4 Circuito de Subcircuitos

Para cumprir com a abstração de subcircuitos proposta para o projeto, foi desenvolvida uma classe denominada *Circuito de Subcircuitos*, que mantém um mapa entre os nodos externos fornecidos como conexões dos componentes e os índices internos reservados para estes, conforme necessário. Uma estrutura auxiliar denominada *circs_refs_t* (uma corruptela de *tipo para referências a circuitos*) mantém indexação dos ramos e nodos referenciados por cada subcircuito. A estrutura de memória do Circuito de Subcircuitos é apresentada no Excerto 4.

Excerto de código 4: Definição do Circuito Ramificado

```
class Circuito_de_Subcircuitos
{
        protected:
        struct circ_refs_t
        {
                inds_t ramosi;
                inds_t nodose;
                inds_t nodosi;
        };
        std::map<ind_t, ind_t>    mapa_nodosE;
           //sinalizador para avaliação postergada de [nodos_indexados]
        Indexador                 nodos_index;
        VetorIndexado<circ_refs_t>   subcircs;
        Circuito_Ramificado          circuito
}
```

Fonte: Autor

3.1.5 Circuito de Componentes

Durante etapa intermediária entre leitura do circuito e manipulação algébrica deste, os objetos que representam componentes elétricos são mantidos conectados como um circuito em uma estrutura denominada *Circuito de Componentes*. Esta classe define um tipo auxiliar que é um ponteiro único para a classe base dos Componentes (descrita na subseção 3.3) e mantém Vetor Indexado destes, conforme Excerto 5. A escolha por ponteiros únicos é que, através de polimorfismo, um ponteiro para classe base pode ser utilizado para manipular componentes de maneira uniforme, mas cada classe derivada que representa um componente pode ser desenvolvida de maneira independente (cumprindo com um contrato de pré-requisitos definidos pelo programa, conforme definido na subseção 3.3). Além disso, ponteiros únicos automaticamente liberam recursos alocados para os objetos que

apontam, através do conceito de RAII (*Resource acquisition is initialization*, Aquisição de Recurso é Inicialização, em português), da linguagem C++.

Excerto de código 5: Definição do Circuito de Componentes

```
struct Circuito_de_Componentes
{
        typedef std::unique_ptr<Componente> comp_upt_t;
        typedef VetorIndexado<Conectado<comp_upt_t>>
   veci_comps_conectados_t;
protected:
        //vetor indexado de todos os componentes conectados
        veci_comps_conectados_t componentes_conectados;
}
```

Fonte: Autor

3.1.6 Circuito Esquemático

O nível mais alto de abstração dos circuitos dentro do programa é implementado através de uma classe denominada *Circuito Esquemático*, que tem por objetivo armazenar na memória um vetor de componentes e suas respectivas conexões, bem como possíveis condições iniciais destes.

Excerto de código 6: Definição do Circuito de Componentes

```
struct Circuito_Esquematico
{
        std::string Nome;
        std::vector<Componente_Esquematico> Componentes;
          //conexões externas seguidas de conexões de controle
        std::vector<std::vector<std::string>> Conexoes;
        std::vector<Condicao_Inicial_Esquematico> Condicoes_Iniciais;
}
```

Fonte: Autor

3.2 ABSTRAÇÕES ALGÉBRICAS

Para solucionar o sistema linear resultante do condicionamento do circuito a ser simulado, foram desenvolvidas classes específicas para manipulação de Matrizes e Sistemas Lineares, descritas nas subseções 3.3.1, e para manipulação de grafos, conforme descrito na subseção 3.3.2.

3.2.1 Matrizes e Sistemas Lineares

Matrizes são representadas por um vetor de números e índices que quantificam sua altura e largura. O vetor contíguo é acessado como uma matriz bidimensional conforme Equação 19, que define o deslocamento a ser aplicado ao início do vetor interno de uma Matriz para acessar elemento na coluna x e linha y. A declaração de uma matriz se dá conforme Excerto 7.

$$deslocamento(x, y) = Largura \cdot y + x \tag{19}$$

Excerto de código 7: Declaração de Matriz

```
struct Matriz
{
        ind_t A;
        ind_t L;
        NUMs_t mem;
        Matriz(ind_t A, ind_t L) : A(A), L(L), mem(A*L){;}
};
```

Fonte: Autor

Sistemas Lineares são definidos pela junção de uma Matriz e um Vetor de Números, que representam os coeficientes do sistema e o vetor de constantes, M e y, respectivamente, conforme Equação 20. O trecho de código referente à declaração do Sistema Linear é apresentado no Excerto 8.

$$M \times x = y \tag{20}$$

Excerto de código 8: Declaração de Sistema Linear

```
struct Sistema_Linear
{
        // Sistema do tipo A * x = y
        Matriz A;
        NUMs_t y;
        Sistema_Linear(ind_t const ordem)
                : A(ordem, ordem), y(ordem, static_cast<NUM_T>(0)){;};
        Sistema_Linear():A(0, 0){;};
        ~Sistema_Linear()=default;
};
```

Fonte: Autor

Para resolver sistemas apresentados no formato da Equação 20, foi utilizado o método de Eliminação Gaussiana com triangulação por pivoteamento completo.

3.3.2 Grafos

A definição de classes e operações referentes a grafos é implementada em *namespace* próprio, denominado *Topologia*. Neste espaço de definições, são declarados *Arestas* e *Vértices*, e com estes é definida uma estrutura denominada *Grafo*, conforme excerto 9.

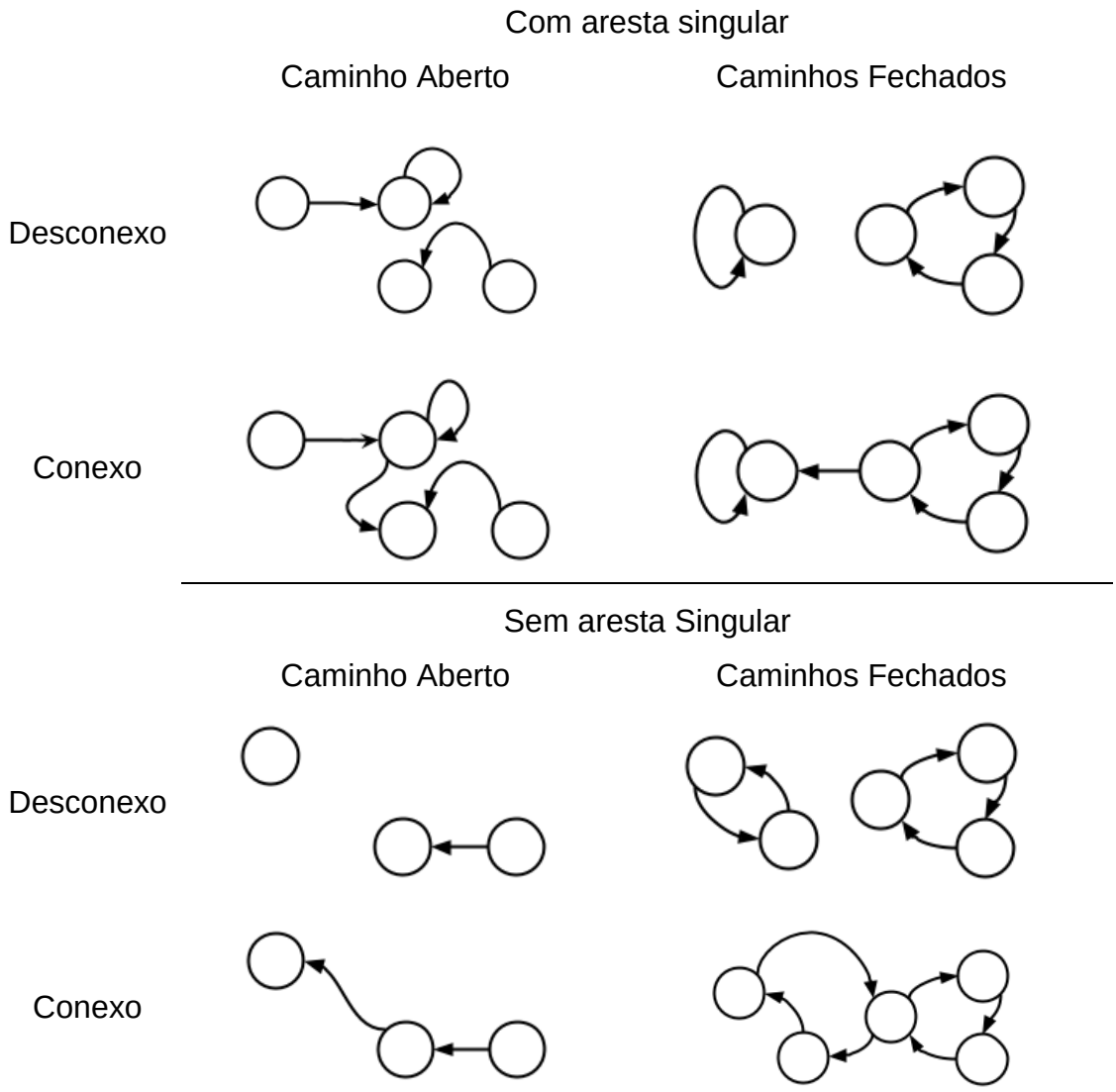
Excerto de código 9: Declaração de Grafo

```
struct Aresta
{
        ind_t v0;
        ind_t v1;
};

struct Vertice
{
        inds_t as;
};

struct Grafo
{
        std::vector<Vertice> verts;
        std::vector<Aresta> arests;
};
```

Fonte: Autor

Como os vértices da classe *Grafo* mantém lista de a que arestas conectam-se, verificar o grau de um vértice é operação de custo constante. A classe também implementa algoritmos para verificação de algumas restrições, conforme apresentadas na subseção 2.2.2 dos Fundamentos Teóricos, e ilustrados na Figura 23. Quando componentes são compilados em um circuito ramificado completo, apenas os cujo grafo atrelado é conexo, sem ramos singulares e de caminhos fechados seguem para simulação, os demais erguem erro de lógica e interrompem a execução do programa.

Figura 12. Exemplos de Grafos e validação topológica



Fonte: Autor

3.3.3 Análise Nodal Modificada

A classe que manipula um circuito e o condiciona para o método de resolução escolhido para o projeto é denominada *Sistema ANM*, o método implementado é uma modificação da Análise Nodal Modificada presente na literatura (NAJM, 2010) (HO, RUEHLI, BRENNAN, 1975) .

A estrutura contém um *Sistema Linear* no qual são estampadas as equações resultantes de um circuito completo. Para tal, são mantidos mapas entre as indexações internas e externas de nodos e componentes, além de um mapa entre índices referentes a fontes de diferença de potencial e a equação adicional que requerem no sistema, conforme Excerto 10.

Excerto de código 10: Declaração de memória para Sistema ANMM

```
struct Sistema_ANMM
{
        Algebra::Sistema_Linear sistema_linear;
protected:
        std::map<ind_t, ind_t> m_nodos_circ_graf;
        inds_t                 m_nodos_graf_circ;
        std::map<ind_t, ind_t> m_ramos_circ_graf;
        inds_t                 m_ramos_graf_circ;
        std::map<ind_t, ind_t> m_fonteVic_iaux;
        ind_t N_nodos;
};
```

Fonte: Autor

3.4 MODELAGEM DE COMPONENTES

A classe base para todos os componentes possui métodos puramente virtuais que são suficientes para que todos os componentes sejam tratados de forma homogênea e forneçam informações suficientes para realizar a análise. Cada tipo de componente deve declarar se seu modelo é ajustável, fornecer o circuito equivalente para o modelo selecionado e declarar se possui determinada propriedade (como “corrente” ou “ddp”), que pode ser requisitada como objetivo da análise. O Excerto 11 demonstra como essa interface foi implementada.

Excerto de código 11: Declaração classe virtual para componentes elétricos

```
struct Componente
{
//modelagem
        virtual bool modelo_ajustavel()const = 0;
        virtual Modelo_Ramificado
           modelo_inicial(Condicionamento::Contexto_Analise const) = 0;
        virtual Modelo_Ramificado modelo_condicional
            (Condicionamento::Estado_Subcircuito const &,
             Condicionamento::Contexto_Analise const &) = 0;
        virtual ind_t Conexoes_quantidade() const = 0;
//extração de propriedades/objetivos de simulação
        virtual bool  possui_propriedade(std::string) = 0;
        virtual NUM_T extrai_propriedade(std::string,
           Condicionamento::Estado_Subcircuito const &) = 0;
//informação do componente
        virtual std::string tipo_str()const = 0;
//destrutor
        virtual ~Componente() = default;
};
```

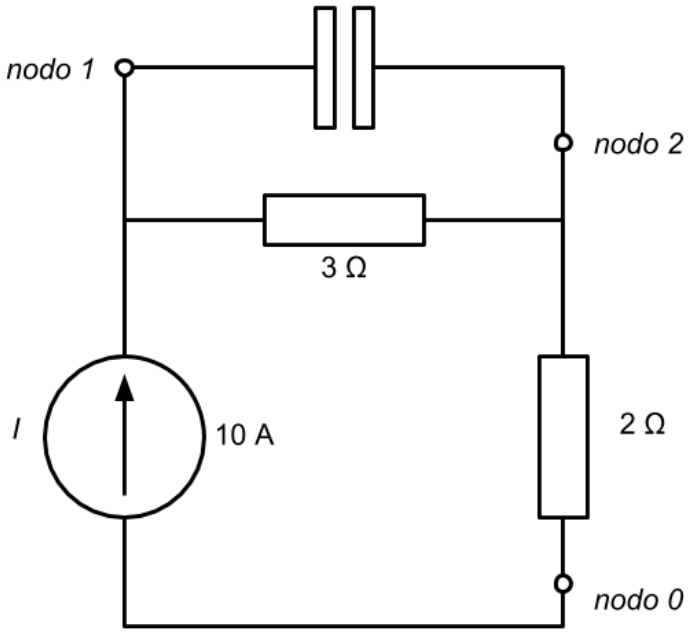
Fonte: Autor

3.5 INTERFACE PARA USUÁRIO EXTERNO

Cada módulo desenvolvido para cumprir o objetivo de montar um simulador completo possui métodos expostos que permitem programar um circuito usando diretamente de código fonte. O objetivo de expor essa funcionalidade é permitir que o código possa ser adaptado para simulações procedurais ou de fim específico. Para permitir uso do simulador sem lidar diretamente com código fonte, também foi desenvolvido um leitor de arquivos *netlist* denominado IVO, em um acrônimo para *Interpretador de Verbetes em Objeto*, explicado na subseção 3.5.1.

O circuito demonstrado na Figura 12 é montado diretamente no código-fonte conforme Excerto 12, usando modelos ideais. Resultados de simulações são expostos na Seção 4.

Figura 13. Circuito com divisor resistivo, fonte de corrente e capacitor



Fonte: Autor

Excerto de código 12: Declaração em código-fonte de circuito da Figura 12

```
Circuito_Esquematico Esquema;
Componente_Esquematico I1("FonteIfixa", "ideal", "I1", 10);
Componente_Esquematico R1("Resistor", "ideal", "R1", 3);
Componente_Esquematico R2("Resistor", "ideal", "R2", 2);
Componente_Esquematico C1("Capacitor", "ideal", "C1", 2);
Esquema.Adiciona(I1, "0", "n1");
Esquema.Adiciona(R1, "n1", "n2");
Esquema.Adiciona(R2, "n2", "0");
Esquema.Adiciona(C1, "n1", "n2");
```

Fonte: Autor

3.5.1 Leitura de Arquivos *netlist*

O leitor desenvolvido foi inspirado em sintaxe genérica para arquivos *netlist*, onde são declarados, em um arquivo de texto, componentes com seus parâmetros e conexões, bem como a análise e objetivos desejados. Cada linha do arquivo é interpretada como um objeto de classe base denominada *Verbete*, apresentada no Excerto 13. A classe *Verbete* mantém um índice representativo da linha que o originou e uma etiqueta de que tipo de informação a linha contém: Novo circuito, nova análise, novo componente, novo objetivo ou novo comando.

Excerto de código 13: Declaração de estrutura para Verbete de Arquivo *netlist*

```
enum class classe_verbete_t : ind_t
{
        circuito,
        analise,
        componente,
        objetivo,
        comando
};
struct Verbete_t
{
        ind_t linha_origem;
        classe_verbete_t tipo;
        Verbete_t (ind_t linha_n, classe_verbete_t Tipo)
         : linha_origem(linha_n), tipo(Tipo) {;}
};
```

Fonte: Autor

Definição completa da sintaxe desenvolvida é fornecida no repositório do projeto. Exemplo do circuito representado na Figura 12 descrito conforme a sintaxe desenvolvida é demonstrado a seguir, onde a diferença de potencial (*ddp*) sobre o Capacitor é requisitada como objetivo de análise transiente. O nodo denominado “0” (o numeral zero) é reservado para representar a referência de potencial nulo, usualmente denominado *terra* na engenharia elétrica.

Exemplo de netlist na sintaxe desenvolvida:

```
+ Circuito Exemplo

FonteIfixa ideal I1 10A entre 0 n1
Capacitor ideal C1 1F CI=0 entre n1 n2
Resistor ideal R1 3Ω entre n1 n2
Resistor ideal R2 2Ω entre n2 0

+ Análise Transiente 5s 1ms
C1 ddp
```

3.6 CONDICIONAMENTO PARA SIMULAÇÕES

A expansão da representação de um circuito para ambos os tipos de análise considerados possuem passos iniciais em comum, por isso foi criada uma classe base, denominada *Análise Base*, que mantém um vetor contendo os objetivos da análise, um mapa entre Componentes e seus nomes, um mapa entre nodos e seus nomes, um objeto representando o circuito de componentes carregados e o Circuito Ramificado equivalente completo. Nesta classe, também são definidos o erro relativo e absoluto aceitáveis pela simulação, conforme Excerto 14. O valor absoluto definido para os erros aceitáveis é baseado no tipo numérico escolhido (*double, 64 bits, IEEE 754*), conforme apresentado na subseção 3.1, o erro relativo foi definido como meia parte por mil.

Excerto de código 14: Declaração da Classe Base para análises

```
struct Analise_Base
{
        static constexpr NUM_T
          erro_relativo_padrao_estabilidade = 5e-4;
        static constexpr NUM_T
          erro_absoluto_padrao_estabilidade = 5e-15;
//typedefs
        typedef std::map<std::string, ind_t> mapa_nome_ind_t;
        typedef std::map<ind_t, ind_t>       mapa_ind_ind_t;
//mapeamento
        mapa_nome_ind_t m_comp_nome_i;
        mapa_nome_ind_t m_nodo_nome_i;
//memória
        Circuito_de_Componentes circuito_componentes;
        Circuito_Equivalente_t  circuito_equivalente;
        Tratamento::Objetivos_t objetivos;
};
```

Fonte: Autor

3.6.1 Particularidades da Análise Quiescente

A análise quiescente é dividida em três etapas:

a) Carrega Esquemático: O circuito esquemático é carregado para memória da classe de simulação. Esta etapa é dividida em quatro sub-passos: mapeamento dos nodos, montagem do circuito, coleta de condições iniciais e interpretação das conexões de controle.
b) Substituição de Ramos de primeira ordem: Os ramos de componentes reativos são substituídos pelo seu equivalente instantâneo, conforme explanado na subseção 2.2.4.
c) Estabilização de ponto de operação: O circuito completo é estabilizado através de sucessivas iterações de acordo com a declaração dos componentes se estes necessitam de ajuste considerando o resultado da iteração atual, até um máximo deliberado de mil iterações.

3.6.2 Particularidades da Análise Transiente

A análise transiente é implementada através de uma sucessão de análises quiescentes, com respectivo ajuste de componentes com estado entre cada passo. Neste projeto, o passo inicial é sempre considerado em $t = 0s$, e o número de passos executados é inferido pelo intervalo de tempo desejado entre cada passo e o instante final escolhido. O usuário deve fornecer – como argumento no código fonte, ou como parâmetro em arquivo netlist – o passo de tempo desejado.

4 RESULTADOS

Esta seção apresenta alguns testes que foram desenvolvidos para avaliar o desempenho do simulador. Cada teste foi desenvolvido na forma de código-fonte que avalia um dos pontos chave implementados.

4.1 RESOLUÇÃO ALGÉBRICA

Para testar o método de pivoteamento desenvolvido, foi escrito script que gera matriz de dimensão 100×100 e a preenche com coeficientes pseudoaleatórios usando de um gerador Mersenne-Twister de 64 bits para alimentar a mantissa e o expoente destes de acordo com uma distribuição linear uniforme para ambos, conforme Equação 21 e Excerto 15. Os limites para o expoente foram escolhidos para representarem valores considerados dentro dos objetivos para o projeto (componentes entre micro e mega, cujos valores das propriedades multiplicados podem gerar expoentes entre -12 e 12), e o vetor de constantes ajustado para que a solução correta seja um vetor contendo os inteiros $[1, 2, 3 \ldots 100]$.

$$M_{100\times100}, \qquad m_{i,j} = a^b, \qquad a \sim U(-1,1), \qquad b \sim U(-12,12) \tag{21}$$

Excerto de código 15: Programa para testar Resolução Algébrica

```
ind_t const N = 100;
vector<NUM_T> xs;
for(ind_t k=0; k<N; k++)
{
        xs.emplace_back(k+1);
}
Algebra::Matriz M(N,N);
std::uniform_real_distribution<NUM_T> ae_dist(-12, 12);
std::uniform_real_distribution<NUM_T> am_dist(-1, 1);
std::mt19937_64 RNG(42);
for(ind_t e=0; e<N; e++)
{
        for(ind_t k=0; k<N; k++)
            { M.acesso(e, k) = am_dist(RNG)*std::pow(10., ae_dist(RNG)); }
}
NUMs_t u;
for(ind_t e=0; e<N; e++)
{
        u.emplace_back(0.);
        for(ind_t k=0; k<N; k++)
        {
                u.back() += xs[k]*M.acesso(e,k);
        }
}
auto x = Algebra::Resolve(M, u);
```

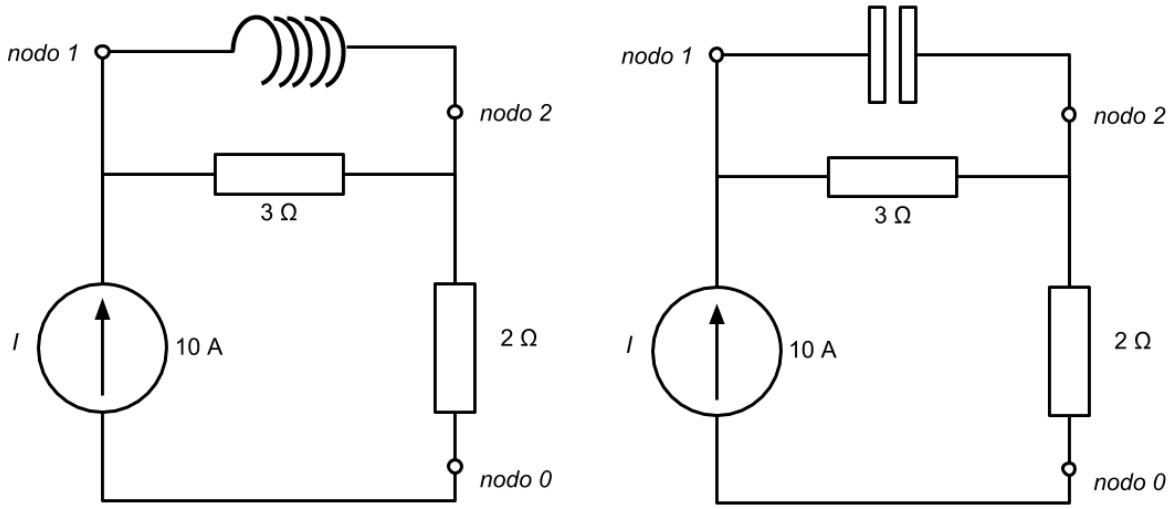
Fonte: Autor

Como os valores são todos sorteados, existe uma chance de ser gerado um sistema singular, caso no qual o teste não fornece resultado válido. A troca da semente do gerador aleatório é provavelmente suficiente para condicionar o sistema, dada a baixa possibilidade de ocorrência de matrizes singulares (esta situação nunca foi observada durante os testes). O código fonte para este teste está disponível no repositório sob o nome de *TesteAlgebra*.

4.2 TESTE DE INFERÊNCIA DE CONDIÇÕES INICIAIS

Para testar o método de inferência de condições iniciais proposto, no qual um capacitor é substituído por uma fonte de corrente nula e um indutor por uma fonte de tensão nula quando as condições iniciais destes não são fornecidas, foi gerado script semelhante ao apresentado na subseção 3.5, para os circuitos apresentados na Figura 13.

Figura 14. Circuitos de teste para inferência de Condições Iniciais



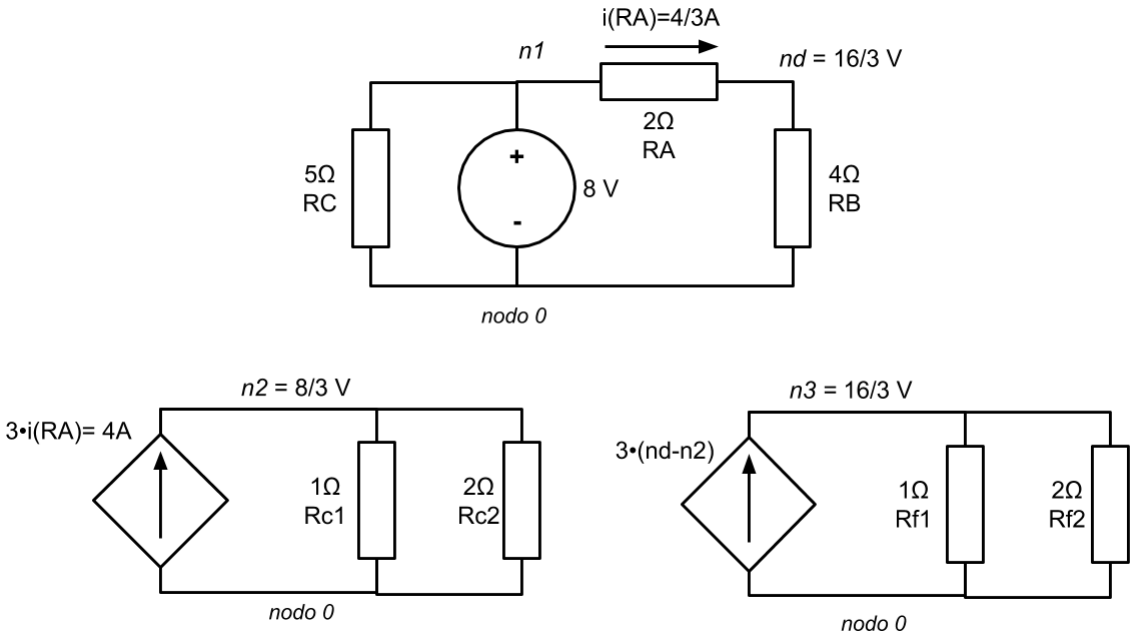
Fonte Autor

O código fonte para este teste está disponível no repositório sob o nome de *TesteAutoCI,* e o resultado é de $30V$ para o capacitor e $10A$ para o indutor.

4.3 TESTE DE FONTES CONTROLADAS

Para testar o método de equacionamento de fontes de corrente controladas, foi gerado circuito conforme ilustrado na Figura 14. O script para este teste está disponível no repositório sob o nome de *TesteIControlada*.

Figura 15. Circuitos de teste de fontes de corrente controladas



Fonte: Autor

Para testar o método de equacionamento de fontes de diferença de potencial controlado, também foi programado um circuito que utiliza destas, este está disponível no repositório com o nome *TesteVControlada*.

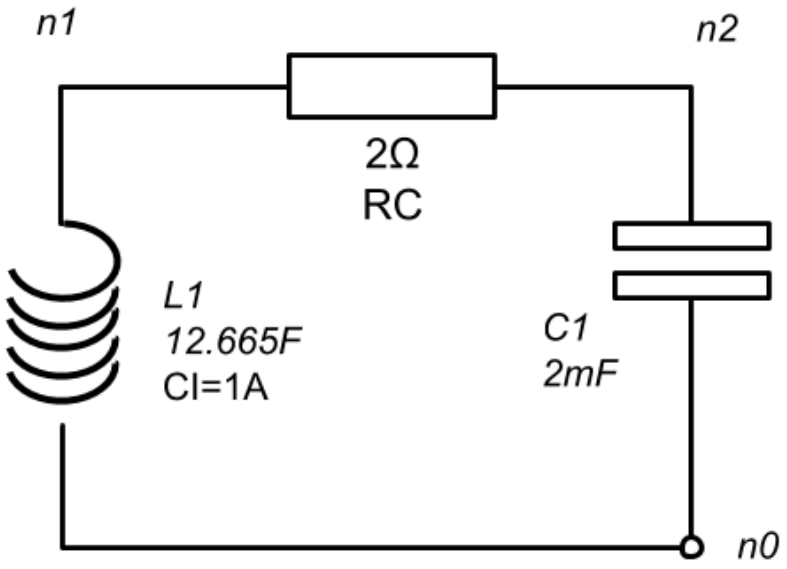
4.4 TESTE DE GRAFOS

Para testar todos os tipos de grafos apresentados na seção 3.3.2, foi elaborado script denominado *TesteGrafos*. Este arquivo está presente no repositório e identifica corretamente as oito variações apresentadas na Figura 11 da subseção 3.2.2.

4.5 TESTE DE LEITOR DE ARQUIVOS NETLIST

Para testar o leitor de arquivos *netlist*, foi gerado um arquivo denominado *TesteIVO*, disponível no repositório, que ao ser executado dentro da estrutura de pastas do repositório, simula o circuito da Figura 15, através do Excerto 16, gerando o gráfico da Figura 17. Notar que o título do gráfico, bem como a legenda deste e outros textos do grafo resultante podem ser alterados no código-fonte.

Figura 16. Circuito Representado pelo arquivo de teste para o leitor netlist



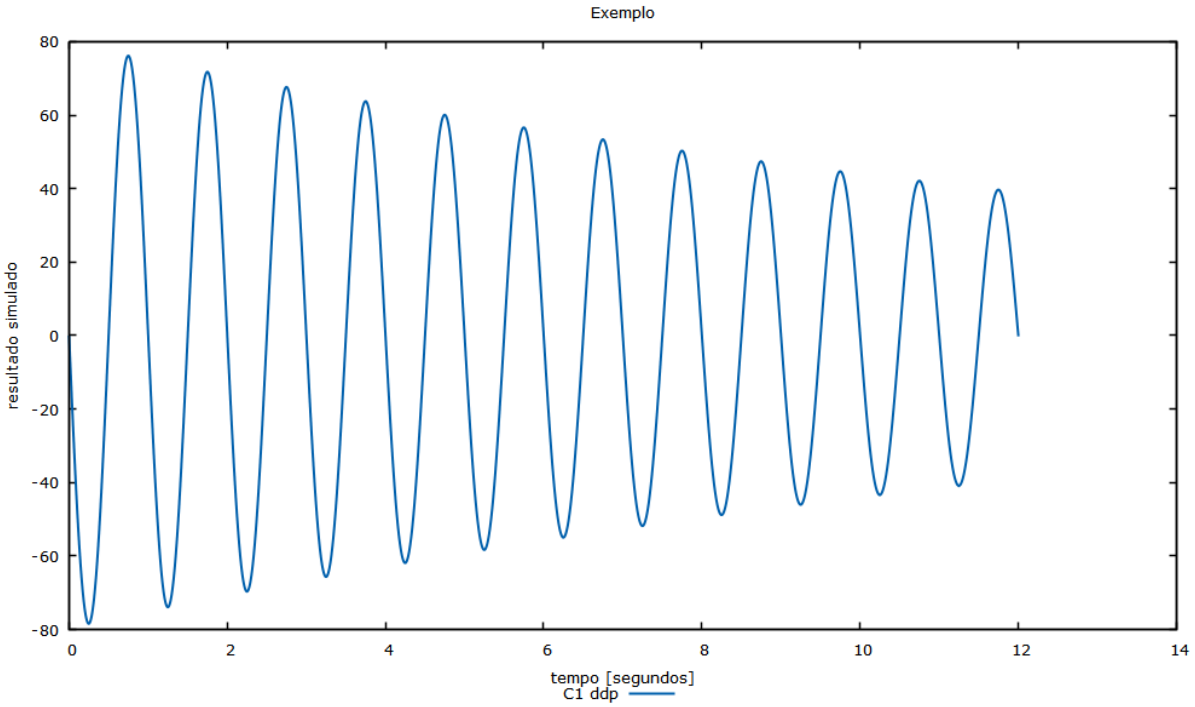
Fonte: Autor

Excerto de código 16: Arquivo netlist para circuito da Figura 15

```
+ Circuito Exemplo


Capacitor ideal C1 2mF CI=0 entre n1 0

Resistor ideal R2 2Ω entre n1 n2

Indutor ideal L1 12,665H CI=1 entre n2 0


+ Análise Transiente 12s 1ms
C1 ddp
```

Fonte: Autor

Figura 17. Resultado da simulação do Arquivo TesteIVO



Fonte: Autor

5 CONCLUSÕES

O objetivo primário foi bem-sucedido. Os módulos desenvolvidos foram utilizados para construir um simulador de ponto quiescente usando da Análise Nodal Modificada proposta, e a eficácia da modularidade foi verificada pela reutilização de grande parte do código para programar a Análise Transiente.

Os modelos de fonte de diferença de potencial e corrente não ideais implementados demonstram que o método proposto, de modelos baseados em subcircuitos, é eficaz e pode ser expandido para todos os componentes através da orientação a objeto.

O projeto teria sido melhor executado se reduzido o nível de customização do simulador e melhor documentadas e testadas as opções desenvolvidas.

REFERÊNCIAS

LIST of free electronics circuit Simulators. In: Wikipedia: A enciclopédia livre. [s.l.] : Wikimedia Foundation, 2018. Disponível em: <https://en.wikipedia.org/wiki/List_of_free_electronics_circuit_simulators>. Acesso em: 18 Dez. 2018.

ISO/IEC DIS 14882:2014 C++14 Information technology - Programming languages: C++

SUTTER, Herb. We Have C++14! [s.l.] : Standard C++ Foundation, 2014. Disponível em <https://isocpp.org/blog/2014/08/we-have-cpp14>. Acesso em: 18 Dez. 2018.

RABAEV, Jan M., The Spice Page. [s.l.]:[s.n.], [s.d.] Disponível em <http://bwrcs.eecs.berkeley.edu/Classes/IcBook/SPICE/>. Acesso em: 18 Dez. 2018.

SPECTRE Circuit Simulator. [s.l.] : Cadence Design Systems Inc., [s.d.] Disponível em <https://www.cadence.com/content/cadence-www/global/en_US/home/tools/custom-ic-analog-rf-design/circuit-simulation/spectre-circuit-simulator.html> Acesso em: 19 Dez. 2018.

MIXED Signal & Domain Simulation for Embedded Worlds. [s.l.] : Isotel, [s.d.] Disponível em: <http://www.isotel.eu/mixedsim/index.html> Acesso em: 18 Dez. 2018.

ONLINE Circuit Simulator with SPICE. Versão 2.2.7. [s.l.] : Partsim [s.d.]. Disponível em <https://www.partsim.com/simulator>. Acesso em: 18 Dez. 2018.

QUCS project: Quite Universal Circuit Simulator. [s.l.] : Qucs team, c2017. Disponível em <http://qucs.sourceforge.net>. Acesso em 19 Dez. 2018.

ALEXANDER, Charles K., SADIKU, Matthew N. O. Kirchhoff's Laws. In: _____. Fundamentals of Electric Circuits. 5ª Ed. Nova Iorque: McGraw-Hill, 2013 cap. 2.4, p. 37.

NAJM, Farid N. Unique Solvability. In: _____. Circuit Simulation. Nova Jérsei: Wiley-IEEE Press, 2010. cap. 2.4.3, p. 30.

NAJM, Farid N. Network Graphs. In: _____. Circuit Simulation. Nova Jérsei: Wiley-IEEE Press, 2010. cap. 2.2.1, p. 20.

NAJM, Farid N. Orthogonal Spaces. In: _____. Circuit Simulation. Nova Jérsei: Wiley-IEEE Press, 2010. cap. 2.3.3, p. 24.

ALEXANDER, Charles K., SADIKU, Matthew N. O. Circuit Elements. In: _____. Fundamentals of Electric Circuits. 5ª Ed. Nova Iorque: McGraw-Hill, 2013 cap. 1.6, p. 15.

ALEXANDER, Charles K., SADIKU, Matthew N. O. Applications. In: _____. Fundamentals of Electric Circuits. 5ª Ed. Nova Iorque: McGraw-Hill, 2013 cap. 4.10, p. 155.

ANDRESEN, Richard P. Creating a Spice subcircuit (how to). [s.l.] : [s.n], 2003 Disponível em < http://www.5spice.com/html/Subckts.html>. Acesso em 18 Dez. 2018.

ALEXANDER, Charles K., SADIKU, Matthew N. O. Thevenin's Theorem. In: _____. Fundamentals of Electric Circuits. 5ª Ed. Nova Iorque: McGraw-Hill, 2013 cap. 4.5, p. 139.

ALEXANDER, Charles K., SADIKU, Matthew N. O. Thevenin's Theorem. In: _____. Fundamentals of Electric Circuits. 5ª Ed. Nova Iorque: McGraw-Hill, 2013 cap. 4.6, p. 145.

NAJM, Farid N. Equivalent Circuit Model. In: _____. Circuit Simulation. Nova Jérsei: Wiley-IEEE, 2010 Press. cap. 2.4.3, p. 17.

BROKKEN, Frank B. Inheritance. In: _____. C++ Annotations Version 11.0.0. Groningen: Universidade de Groningen, 2018. cap 13. Disponível em: <http://www.icce.rug.nl/documents/cplusplus/cplusplus13.html>. Acesso em 19 Dez. 2018.

BROKKEN, Frank B. Virtual functions. In: _____. C++ Annotations Version 11.0.0. Groningen: Universidade de Groningen, 2018. cap 14.1. Disponível em: <http://www.icce.rug.nl/documents/cplusplus/cplusplus14.html>. Acesso em 19 Dez. 2018.

BROKKEN, Frank B. Pure virtual functions. In: _____. C++ Annotations Version 11.0.0. Groningen: Universidade de Groningen, 2018. cap 14.3. Disponível em: <http://www.icce.rug.nl/documents/cplusplus/cplusplus14.html>. Acesso em 19 Dez. 2018.

TIOBE Index for December 2018. [s.l] : TIOBE the software quality company. Disponível em <https://www.tiobe.com/tiobe-index>. Acesso em 19 Dez. 2018.

BROKKEN, Frank B. C++ advantages and claims. In: _____. C++ Annotations Version 11.0.0. Groningen: Universidade de Groningen, 2018. cap 2.3. Disponível em: <http://www.icce.rug.nl/documents/cplusplus/cplusplus02.html>. Acesso em 19 Dez. 2018.

CORMEN, Thomas. BALKCOM, Devin. Notação big-Θ. [s.l.] : Khan Academy. Disponível em <https://pt.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation> Acesso em: 19 Dez. 2018.

BROKKEN, Frank B. Abstract Containers. In: _____. C++ Annotations Version 11.0.0. Groningen: Universidade de Groningen, 2018. cap 12. Disponível em: <http://www.icce.rug.nl/documents/cplusplus/cplusplus12.html>. Acesso em 19 Dez. 2018.

BROKKEN, Frank B. Polymorphous wrappers for function objects. In: _____. C++ Annotations Version 11.0.0. Groningen: Universidade de Groningen, 2018. cap 21.12. Disponível em: <http://www.icce.rug.nl/documents/cplusplus/cplusplus21.html>. Acesso em 19 Dez. 2018.

BROKKEN, Frank B. Exceptions. In: _____. C++ Annotations Version 11.0.0. Groningen: Universidade de Groningen, 2018. cap 10. Disponível em: < http://www.icce.rug.nl/documents/cplusplus/cplusplus10.html>. Acesso em 19 Dez. 2018.

MILEWSKI, Bartosz. Getting Lazy with C++ [s.l] : [blog pessoal]. Disponível em <https://bartoszmilewski.com/2014/04/21/getting-lazy-with-c/> Acesso em 19 Dez. 2018.

BROKKEN, Frank B. Class Templates. In: _____. C++ Annotations Version 11.0.0. Groningen: Universidade de Groningen, 2018. cap 22. Disponível em: < http://www.icce.rug.nl/documents/cplusplus/cplusplus22.html>. Acesso em 19 Dez. 2018.

KAHLER, Mateus. MARA. [s.l.] : GitHub, 2018. Disponível em <https://github.com/mateuskahler/MARA> Acesso em 19 Dez. 2018.

BROKKEN, Frank B. The class ‘unique_ptr‘. In: _____. C++ Annotations Version 11.0.0. Groningen: Universidade de Groningen, 2018. cap 18.3. Disponível em: <http://www.icce.rug.nl/documents/cplusplus/cplusplus18.html#an2132>. Acesso em 19 Dez. 2018.

NAJM, Farid N. Equivalent Circuit Model. In: _____. Circuit Simulation. Nova Jérsei: Wiley-IEEE Press, 2010. cap. 2.4.3, p. 17.

C++ Programming/RAII. In: Wikibooks Open Books for an open world. [s.l] : Wikimedia Foundation, c2018. Disponível em <https://en.wikibooks.org/wiki/C++_Programming/RAII> Acesso em 19 Dez. 2018.

RAII. In: cppreference.com. [s.l] : [s.n], c2017 Disponível em < https://en.cppreference.com/w/cpp/language/raii>. Acesso em: 19 Dez. 2018.

NAJM, Farid N. Modified Nodal Analysis. In: _____. Circuit Simulation. Nova Jérsei: Wiley-IEEE Press, 2010. cap. 2.4.4, p. 32.

CHUNG-WEN Ho, ALBERT E. Ruehli, PIERCE A. Brennan. The Modified Nodal Approach to Network Analysis, [s.l.] : IEEE Transactions on Circuits and Systems, v. 22, n. 6, Jun. 1975. p504-509.

STD::MERSENNE_TWISTER_ENGINE. In: cppreference.com. [s.l.] : [s.n.], c2016. Disponível em <https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine >. Acesso em: 19 Dez. 2018.

STD::UNIFORM_REAL_DISTRIBUTION. In: cppreference.com. [s.l] : [s.n], c2018. Disponível em <https://en.cppreference.com/w/cpp/numeric/random/uniform_real_distribution> >. Acesso em: 19 Dez. 2018.