Análise Detalhada do Código do Simulador de Conversor Buck CC-CCEste código implementa uma interface gráfica para simulação de um conversor Buck (abaixador de tensão) CC-CC usando Python com as bibliotecas Tkinter e Matplotlib. Vou explicar cada parte do código em detalhes:

1. **Importações e Configurações Iniciais**

```
import tkinter as tk
from tkinter import ttk
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg,
NavigationToolbar2Tk
from matplotlib.figure import Figure
import matplotlib

# Configurar matplotlib para usar o backend TkAgg
matplotlib.use('TkAgg')
```

As seguintes bibliotecas foram importadas:

tkinter e ***ttk***: criação de interface gráfica

numpy: cálculos numéricos e simulação.

matplotlib: plotagem dos gráficos

FigureCanvasTkAgg e ***NavigationToolbar2Tk***: integram os gráficos do matplotlib com auxílio da biblioteca *Tkintermatplotlib.use('TkAgg')*

Tkintermatplotlib.use('TkAgg'): Configura o *matplotlib* para usar o *backend TkAgg*, que permite a integração com *Tkinter*

Em seguida, cria-se uma classe principal a partir da qual todas as funções referentes ao código estarão sujeitas a ela (classe raiz – *root*).

2. *Def __init__(self, root):*

```
def __init__(self, root):
    self.root = root
    self.root.title("Simulador de Conversor Buck CC-CC")
    self.root.geometry("1200x800")
    self.root.minsize(1000, 700)

    # Configurar estilo
    self.setup_style()
```

```
        # Variáveis do circuito
        self.setup_variables()

        # Criar interface
        self.create_widgets()

        # Simulação inicial
        self.run_simulation()
```

Essa função apenas inicializa e realiza a configuração e definição da janela principal com títulos e dimensões.

Chamada de métodos:

Configura o estilo visual (*setup_style*)

Criar variáveis do circuito (*setup_variables*)

Construir a interface (*create_widgets*)

Executar a primeira simulação (*run_simulation*)

3. def setup_style

```
def setup_style(self):
    self.style = ttk.Style()
    self.style.theme_use('clam')

    # Configurar cores
    self.bg_color = '#f0f3f5'
    self.frame_color = '#ffffff'
    self.accent_color = '#4e73df'
    self.text_color = '#2e2e2e'

    # Configurar estilos
    self.style.configure('TFrame', background=self.bg_color)
    self.style.configure('TLabel', background=self.bg_color,
foreground=self.text_color)
    # ... outros estilos ...
```

Define o tema visual da aplicação (*clam*)

Configura cores para fundo, frames, texto e elementos de destaque

Aplica estilos consistentes a todos os widgets

4. setup_variables

```
def setup_variables(self):
    # Parâmetros iniciais
```

```
        self.Vin = tk.DoubleVar(value=36.0)
        self.Vout = tk.DoubleVar(value=12.0)
        self.Iout = tk.DoubleVar(value=2.0)
        self.fsw = tk.DoubleVar(value=50000)
        self.L = tk.DoubleVar(value=220e-6)
        self.C = tk.DoubleVar(value=47e-6)
        self.R_esr = tk.DoubleVar(value=0.01)

        # Resultados
        self.results = {
            'Vavg': tk.StringVar(value='---'),
            'Vripple': tk.StringVar(value='---'),
            'Iripple': tk.StringVar(value='---'),
            'Duty': tk.StringVar(value='---')
        }
```

Cria variáveis Tkinter para armazenar:Parâmetros do circuito (tensão de entrada/saída, corrente, frequência, etc.) Os resultados da simulação tais como tensão média, *ripple*, *duty cycle* são vinculadas aos *widgets* da interface.

5. def creat_wigdets(self):

```
def create_widgets(self):
    # Frame principal
    main_frame = ttk.Frame(self.root)
    main_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

    # Painel esquerdo (controles)
    left_panel = ttk.Frame(main_frame, width=350)
    left_panel.pack(side=tk.LEFT, fill=tk.Y)
    left_panel.pack_propagate(False)

    # Painel direito (gráficos)
    right_panel = ttk.Frame(main_frame)
    right_panel.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True,
padx=(10, 0))

    # Criar seções
    self.create_parameter_section(left_panel)
    self.create_results_section(left_panel)
    self.create_graph_section(right_panel)
```

Divide a interface em dois painéis principais:

Esquerdo: controles e resultados (fixo em 350px de largura).

Direito: gráficos (expansível)

Essa função chama métodos para criar cada seção da interface.

6. def create_parameter_section

```
def create_parameter_section(self, parent):
    frame = ttk.LabelFrame(parent, text="PARÂMETROS DO
CIRCUITO", padding=(15, 10))
    frame.pack(fill=tk.X, pady=(0, 15))

    # Entradas de parâmetros
    params = [
        ("Tensão de Entrada (V)", self.Vin),
        ("Tensão de Saída (V)", self.Vout),
        # ... outros parâmetros ...
    ]

    for text, var in params:
        row = ttk.Frame(frame)
        row.pack(fill=tk.X, pady=5)

        ttk.Label(row, text=text, width=20,
anchor=tk.W).pack(side=tk.LEFT)
        entry = ttk.Entry(row, textvariable=var, width=10,
justify=tk.RIGHT)
        entry.pack(side=tk.RIGHT)
        entry.bind('<KeyRelease>', lambda e:
self.validate_entry(e.widget))

    # Botão de simulação
    ttk.Button(btn_frame, text="SIMULAR",
command=self.run_simulation,
               style='Accent.TButton').pack(fill=tk.X)
```

Cria um frame com os parâmetros ajustáveis do circuito. Para cada parâmetro, cria um rótulo e uma entrada (*Entry*) e vincula a validação de entrada ao evento *KeyRelease*. Adiciona botão "SIMULAR" que executa *run_simulation.*

7. def create_results_section

```
def create_results_section(self, parent):
    frame = ttk.LabelFrame(parent, text="RESULTADOS",
padding=(15, 10))
    frame.pack(fill=tk.BOTH, expand=True)

    results = [
```

```
            ("Tensão Média (V)", 'Vavg'),
            ("Ripple de Tensão (V)", 'Vripple'),
            # ... outros resultados ...
        ]

        for text, key in results:
            row = ttk.Frame(frame)
            row.pack(fill=tk.X, pady=5)

            ttk.Label(row, text=text, width=20,
anchor=tk.W).pack(side=tk.LEFT)
            ttk.Label(row, textvariable=self.results[key], width=10,
                      foreground='blue',
anchor=tk.E).pack(side=tk.RIGHT)
```

Cria um frame para exibir os resultados da simulação. Para cada resultado, mostra um rótulo descritivo e o valor, o qual é atualizado dinamicamente. Os valores são vinculados às variáveis *Tkinter* criadas em *setup_variables.*

8. def create_graph_section

```
def create_graph_section(self, parent):
    # Frame para os gráficos
    graph_frame = ttk.Frame(parent)
    graph_frame.pack(fill=tk.BOTH, expand=True)

    # Criar figura matplotlib
    self.fig = Figure(figsize=(8, 6), dpi=100,
facecolor=self.bg_color)
    self.canvas = FigureCanvasTkAgg(self.fig,
master=graph_frame)
    self.canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)

    # Barra de ferramentas
    toolbar = NavigationToolbar2Tk(self.canvas, graph_frame,
pack_toolbar=False)
    toolbar.update()
    toolbar.pack(fill=tk.X)

    # Configurar subplots
    self.ax1 = self.fig.add_subplot(311)  # Tensão na carga
    self.ax2 = self.fig.add_subplot(312)  # Tensão no indutor
    self.ax3 = self.fig.add_subplot(313)  # Tensão no capacitor
```

Cria a área de gráficos com:

- Uma figura *matplotlib* com 3 *subplots* (tensão na carga, indutor e capacitor).
- Um *canvas* para integrar a figura com *Tkinter*
- Barra de ferramentas do *matplotlib* para realizar as funções *zoom*, *pan/save.*

9. def validate_entry

```
def validate_entry(self, widget):
    try:
        float(widget.get())
        widget.config(foreground='black')
    except ValueError:
        widget.config(foreground='red')
```

Função de segurança para notificar o usuário de entradas válidas. Valida se a entrada do usuário é um número válido. Muda a cor do texto para vermelho se inválido, preto se válido.

10.def run_simulation

```
def run_simulation(self):
        try:
            # Obter parâmetros
            Vin = self.Vin.get()
            Vout = self.Vout.get()
            Iout = self.Iout.get()
            fsw = self.fsw.get()
            L = self.L.get()
            C = self.C.get()
            R_esr = self.R_esr.get()
            R_load = Vout / Iout

            # Verificar valores
            if Vin <= Vout:
                raise ValueError("A tensão de entrada deve ser
maior que a saída!")

            # Calcular duty cycle
            D = Vout / Vin

            # Tempo de simulação
            t_sim = 5e-3  # 5 ms
            dt = 1 / (fsw * 200)
            t = np.arange(0, t_sim, dt)

            # Inicializar variáveis
```

```
            Vout = np.zeros_like(t)
            V_L = np.zeros_like(t)
            V_C = np.zeros_like(t)
            I_L = np.zeros_like(t)
            I_C = np.zeros_like(t)

            # Condições iniciais
            V_C[0] = 0.0
            I_L[0] = 0.0

            # Simulação
            for i in range(1, len(t)):
                # Controle PWM
                if (t[i] * fsw) % 1.0 < D:
                    V_L[i] = Vin - V_C[i-1]  # MOSFET ligado
                else:
                    V_L[i] = -V_C[i-1]       # MOSFET desligado

                # Atualizar corrente no indutor
                I_L[i] = I_L[i-1] + (V_L[i] / L) * dt

                # Atualizar corrente no capacitor
                I_C[i] = I_L[i] - (V_C[i-1] / R_load)

                # Atualizar tensão no capacitor
                V_C[i] = V_C[i-1] + (I_C[i] / C) * dt

                # Tensão na carga (com ESR)
                Vout[i] = V_C[i] + (I_C[i] * R_esr)

            # Calcular resultados
            start_idx = int(0.9 * len(t))  # Ignorar transitório
            Vavg = np.mean(Vout[start_idx:])
            Vripple = np.max(Vout[start_idx:]) -
np.min(Vout[start_idx:])
            Iripple = np.max(I_L) - np.min(I_L)

            # Atualizar interface
            self.results['Vavg'].set(f"{Vavg:.3f}")
            self.results['Vripple'].set(f"{Vripple:.3f}")
            self.results['Iripple'].set(f"{Iripple:.3f}")
            self.results['Duty'].set(f"{D*100:.1f}")

            # Atualizar gráficos
            self.update_plots(t, Vout, V_L, V_C, Vavg)

        except Exception as e:
            messagebox.showerror("Erro", f"Falha na
simulação:\n{str(e)}")
```

Essa função é o cerne do código, onde se é calculado, a partir das entradas, todos os parâmetros de saída, formas de ondas, amortecimento da tensão e valor médio da tensão de saída. A fim de esclarecer o processo de operação da seguinte função, fora separados em pontos seu funcionamento:

1. Obtém os parâmetros da interface
2. Verifica se a tensão de entrada é maior que a saída.
3. Calcula o *duty cycle* (razão entre tensão de saída e entrada).
4. Configura o tempo de simulação (5ms com resolução adequada).
5. Simula o circuito iterativamente:
    5.1.1. Modela o chaveamento PWM.
    5.1.2. Calcula tensão e corrente no indutor.
    5.1.3. Calcula tensão e corrente no capacitor
    5.1.4. Considera o ESR (resistência série equivalente) do capacitor.
6. Calcula os resultados (média, *ripple*, dentre outros).
7. Atualiza a interface com os resultados.
8. Atualiza os gráficos.

11.def update_plots

```
def update_plots(self, t, Vout, V_L, V_C, Vavg):
    # Converter tempo para ms
    t_ms = t * 1000

    # Limpar e atualizar gráficos
    for ax in [self.ax1, self.ax2, self.ax3]:
        ax.clear()

    # Gráfico 1: Tensão na Carga
    self.ax1.plot(t_ms, Vout, 'b', label='Tensão na Carga')
    self.ax1.axhline(y=Vavg, color='r', linestyle='--',
label=f'Média: {Vavg:.2f}V')
    # ... configurações do gráfico ...

    # Gráfico 2: Tensão no Indutor
    self.ax2.plot(t_ms, V_L, 'g')
    # ... configurações do gráfico ...

    # Gráfico 3: Tensão no Capacitor
    self.ax3.plot(t_ms, V_C, 'm')
    # ... configurações do gráfico ...
```

```
        # Ajustar layout e redesenhar
        self.fig.tight_layout()
        self.canvas.draw()
```

Atualiza os três gráficos com os novos dados da simulação além de configurar os títulos, eixos, legendas e grades, converte o tempo para milissegundos para melhor legibilidade e redesenha o *canvas* com os novos gráficos.

12.Bloco Principal

```
if __name__ == "__main__":
    root = tk.Tk()
    app = BuckConverterApp(root)
    root.mainloop()
```

Atualiza os três gráficos com os novos dados da simulação além de configurar os títulos, eixos, legendas e grades, converte o tempo para milissegundos para melhor legibilidade e redesenha o *canvas* com os novos gráficos.