Análise Detalhada do Código do Simulador de Conversor Buck CC-CC

O código implementa uma interface gráfica para simulação de um conversor Buck (abaixador de tensão) CC-CC usando Python com as bibliotecas *Tkinter* e *Matplotlib*.

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* ).

1. *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*)

1. ***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)  
 self.style.configure('TButton', font=('Segoe UI', 10), padding=6)  
 self.style.configure('TEntry', padding=5)  
 self.style.configure('TLabelframe', background=self.bg\_color)  
 self.style.configure('TLabelframe.Label', background=self.bg\_color,   
 foreground=self.accent\_color, font=('Segoe UI', 10, 'bold'))  
 self.style.configure('Accent.TButton', background=self.accent\_color,   
 foreground='white', font=('Segoe UI', 10, 'bold'))

Define o tema visual da aplicação (*clam*).

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

Aplica estilos consistentes a todos os *widgets*.

1. *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.

1. 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.

1. **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.*

1. **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.*

1. **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.*

1. **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.

1. ***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, seu funcionamento foi separado em pontos.

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:
   * 1. Modela o chaveamento PWM.
     2. Calcula tensão e corrente no indutor.
     3. Calcula tensão e corrente no capacitor
     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.
9. **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.

1. **Bloco Principal**

if \_\_name\_\_ == "\_\_main\_\_":  
 root = tk.Tk()  
 app = BuckConverterApp(root)  
 root.mainloop()

Por fim, cria-se uma janela principal *Tkinter* e instancia a aplicação *BuckConverterAppInicia*, a qual é o loop principal de eventos.