Análise do código do conversor AC-CC *Step-Down*

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

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. **BuckConverterApp**

class BuckConverterApp:  
 def \_\_init\_\_(self, root):  
 # Configuração inicial da janela  
 self.root = root  
 self.root.title("Simulador de Conversor Buck CC-CC")  
 self.root.geometry("1200x800")  
 self.root.minsize(1000, 700)  
   
 # Inicialização dos componentes  
 self.setup\_style()  
 self.setup\_variables()  
 self.create\_widgets()  
 self.run\_simulation()

Primeiro, estabelece-se uma *class* a qual todas as funções estarão sujeitas. A primeira função subsequente à definição da *class* está dedicada a gerar uma janela principal a partir das especificações passadas, como por exemplo tamanho, geometria e resolução da janela.

1. **def setup\_style**

def setup\_style(self):  
 self.style = ttk.Style()  
 self.style.theme\_use('clam')  
   
 self.bg\_color = '#f0f3f5'  
 self.frame\_color = '#ffffff'  
 self.accent\_color = '#4e73df'  
 self.text\_color = '#2e2e2e'  
   
 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('Accent.TButton', background=self.accent\_color,   
 foreground='white', font=('Segoe UI', 10, 'bold'))

Define um esquema de cores e aplica estilos escrita a todos os *widgets* através do *tk.Style()* para escolha do tema *clam*, por exemplo.

1. ***def setup\_variables()***

def setup\_variables(self):  
 # Parâmetros iniciais  
 self.Vin = tk.DoubleVar(value=36.0) # Tensão de entrada  
 self.Vout = tk.DoubleVar(value=12.0) # Tensão de saída desejada  
 self.Iout = tk.DoubleVar(value=2.0) # Corrente de saída  
 self.fsw = tk.DoubleVar(value=50000) # Frequência de chaveamento (Hz)  
   
 # Resultados calculados  
 self.results = {  
 'Vavg': tk.StringVar(value='---'), # Tensão média na carga  
 'Vripple': tk.StringVar(value='---'), # Ripple de tensão  
 'Iripple': tk.StringVar(value='---'), # Ripple de corrente  
 'Duty': tk.StringVar(value='---') # Duty cycle (%)  
 }

Cria variáveis Tkinter para armazenar tanto os parâmetros ajustáveis quanto os resultados calculados. Assim, pode-se realizar a atualização automática da interface *a posteriori*.

1. ***def create\_widgets()***

def create\_widgets(self):  
 main\_frame = ttk.Frame(self.root)  
 main\_frame.pack(fill=tk.BOTH, expand=True)  
   
 # Painel esquerdo (controles)  
 left\_panel = ttk.Frame(main\_frame, width=350)  
 left\_panel.pack(side=tk.LEFT, fill=tk.Y)  
   
 # Painel direito (gráficos)  
 right\_panel = ttk.Frame(main\_frame)  
 right\_panel.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)  
   
 self.create\_parameter\_section(left\_panel)  
 self.create\_results\_section(left\_panel)  
 self.create\_graph\_section(right\_panel)

Essa função consiste na criação de widgets para a visualização dos resultados bem como sua organização e disposição na tela para o usuário.

1. ***def create\_parameter\_section()***

def create\_parameter\_section(self, parent):  
 frame = ttk.LabelFrame(parent, text="PARÂMETROS DO CIRCUITO")  
   
 params = [  
 ("Tensão de Entrada (V)", self.Vin),  
 ("Tensão de Saída (V)", self.Vout),  
 # ... outros parâmetros ...  
 ]  
   
 for text, var in params:  
 row = ttk.Frame(frame)  
 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>', self.validate\_entry)  
   
 ttk.Button(frame, text="SIMULAR", command=self.run\_simulation,  
 style='Accent.TButton').pack(fill=tk.X)

Cria um formulário com entradas para todos os parâmetros do circuito, vinculando cada entrada a uma variável *Tkinter* . O botão "SIMULAR" aciona a recálculo.

1. ***def run\_simulation()***

def run\_simulation(self):  
 try:  
 # Obter valores da interface  
 Vin = self.Vin.get()  
 Vout = self.Vout.get()  
 fsw = self.fsw.get()  
 L = self.L.get()  
 C = self.C.get()  
   
 # Verificar se Vin > Vout  
 if Vin <= Vout:  
 raise ValueError("A tensão de entrada deve ser maior que a saída!")  
   
 # Calcular duty cycle  
 D = Vout / Vin  
   
 # Configurar tempo de simulação  
 t\_sim = 5e-3 # 5 ms  
 dt = 1 / (fsw \* 200) # Passo de tempo  
 t = np.arange(0, t\_sim, dt)  
   
 # Inicializar arrays  
 Vout = np.zeros\_like(t)  
 V\_L = np.zeros\_like(t)  
 V\_C = np.zeros\_like(t)  
 I\_L = np.zeros\_like(t)  
   
 # Simulação por iteração  
 for i in range(1, len(t)):  
 # Controle PWM  
 if (t[i] \* fsw) % 1.0 < D:  
 V\_L[i] = Vin - V\_C[i-1] # Estado ligado  
 else:  
 V\_L[i] = -V\_C[i-1] # Estado desligado  
   
 # Atualizar corrente no indutor  
 I\_L[i] = I\_L[i-1] + (V\_L[i] / L) \* dt  
   
 # Atualizar tensão no capacitor  
 I\_C = I\_L[i] - (V\_C[i-1] / R\_load)  
 V\_C[i] = V\_C[i-1] + (I\_C / C) \* dt  
   
 # Tensão na carga com ESR  
 Vout[i] = V\_C[i] + (I\_C \* R\_esr)  
   
 # Processar resultados e atualizar interface  
 self.process\_results(t, Vout, I\_L, D)  
   
 except Exception as e:  
 messagebox.showerror("Erro", f"Falha na simulação:\n{str(e)}")

Aqui há a implementação lógica do funcionamento do circuito, na qual está presente o chaveamento PWM simulando a operação de um MOSFET e a dinâmica da carga e descarga dos componentes ativos do Buck, indutor e capacitor. Além disso, também foi adicionado a resistência ôhmica referente ao capacitor, apenas para tornar a simulação mais próxima possível do real.

1. ***def process\_results***

def process\_results(self, t, Vout, I\_L, D):  
 # Ignorar transitório inicial (últimos 10%)  
 start\_idx = int(0.9 \* len(t))  
   
 # Calcular métricas  
 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)

Apresenta os resultados ao usuário de modo já formatado através da formatação f”” presente de maneira nativa em Python. Os resultados são apresentados dentro da área delimitada nos *widgets*.

1. ***def create\_graph\_section() e def update\_plots()***

def create\_graph\_section(self, parent):  
 self.fig = Figure(figsize=(8, 6), facecolor=self.bg\_color)  
 self.canvas = FigureCanvasTkAgg(self.fig, master=parent)  
 self.canvas.get\_tk\_widget().pack(fill=tk.BOTH, expand=True)  
   
 # Criar 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  
   
 # Adicionar barra de ferramentas  
 toolbar = NavigationToolbar2Tk(self.canvas, parent)  
 toolbar.update()

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')  
 self.ax1.set\_title('Tensão na Carga')  
 self.ax1.set\_ylabel('Tensão (V)')  
 self.ax1.legend()  
 self.ax1.grid(True)  
   
 # Gráfico 2: Tensão no Indutor  
 self.ax2.plot(t\_ms, V\_L, 'g')  
 self.ax2.set\_title('Tensão no Indutor')  
 self.ax2.set\_ylabel('Tensão (V)')  
 self.ax2.grid(True)  
   
 # Gráfico 3: Tensão no Capacitor  
 self.ax3.plot(t\_ms, V\_C, 'm')  
 self.ax3.set\_title('Tensão no Capacitor')  
 self.ax3.set\_xlabel('Tempo (ms)')  
 self.ax3.set\_ylabel('Tensão (V)')  
 self.ax3.grid(True)  
   
 # Ajustar layout e redesenhar  
 self.fig.tight\_layout()  
 self.canvas.draw()

Aqui, esses dois segmentos do código constituem a parte de apresentação gráfica, sendo que para a visualização dos três resultados distintos se é utilizado de *subplots* oriundos da biblioteca *matplotlib*, os quais são definidos, em tamanho e outras características, pela primeira função. Assim, para todas as novas entradas realizadas a função realiza o update dos gráficos presentes nos *subplots* plotando-os novamente.

1. **Bloco Principal**

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

Por fim, a aplicação é chamada e as funções entram em operação através do “*app*” *BuckConverterApp*. Assim, cria-se a janela principal através da *Tkinter* e instancia-se a aplicação desse modo o loop principal de eventos se inicia até que o usuário deseje encerrar a aplicação.