―生成 AI 時代の回路解析―

ChatGPT で昇圧チョッパ解析

古橋　武

令和 7 年 1 月 31 日

目 次

1　はじめに

本稿は，ChatGPT o1 との対話を通じて行った昇圧チョッパ解析のログをまとめたものです．

「電流断続モードにおける出力電圧解析」では，非常に興味深い経験を得ました．筆者が ChatGPT o1 を用いて解析を進めたところ，わずか 2 回のやり取りでこの生成 AI は電流断続モードにおける正しい式を導き出したのです．「すごい時代がやってきた」と実感せざるを得ませんでした．

筆者はパワーエレクトロニクスのオンライン学習コースを主催していますが，その中で受講生に課している多くのレポート課題を ChatGPT がいとも簡単に解いてしまうことを確認しました．実験を除くレポート課題であれば，自分で考えなくても生成 AI が答えを提示してくれるのです．誰でも ChatGPT を使えば，高い能力を発揮できます．では，このような状況で私たちエンジニアは今後どのように学び，どのような力を身につけるべきでしょうか――まさに本質的な問いを真剣に考える時代に突入したといえます．

生成 AI が当たり前となった今，講義や課題設計の従来手法の価値は大きく低下してしまいました．講師は「ChatGPT が使えること」を前提に，コースや学習内容を根本から再設計しなければなりません．学習者も，単純作業的な解法を覚えるのではなく，問題の背景を理解し，導出過程を考察し，どこに応用できるかを見極める力をいっそう強化する必要があります．そして，ChatGPT をどう活用し，自分のどの能力を伸ばしていくべきかを常に考え続けることが求められています．

なお，令和 7 年 1 月中旬の時点で，ChatGPT 4o（20 ドル／月）では，答えに至るまでに誤りが多く，導出過程を細かくガイドしなければ正解にたどり着かない場合がありました．ChatGPT o1（200 ドル／月）も，ときどき誤った解答を返しました．ツールが日進月歩で進化する一方で，その使いこなし方や正誤を見極めるための基礎知識は依然として人間に求められます．どれほど生成 AI が高度化したとしても，最終的に製品の性能や安全性を保証する責任はエンジニアにあるからです．

このレポート記事では，ChatGPT o1 の能力の一端を紹介するとともに，今後のエンジニア教育のあり方や，学習者が何を学ぶべきかを考えるきっかけになればと期待しています．

2　昇圧チョッパの実験

2.1　実験回路

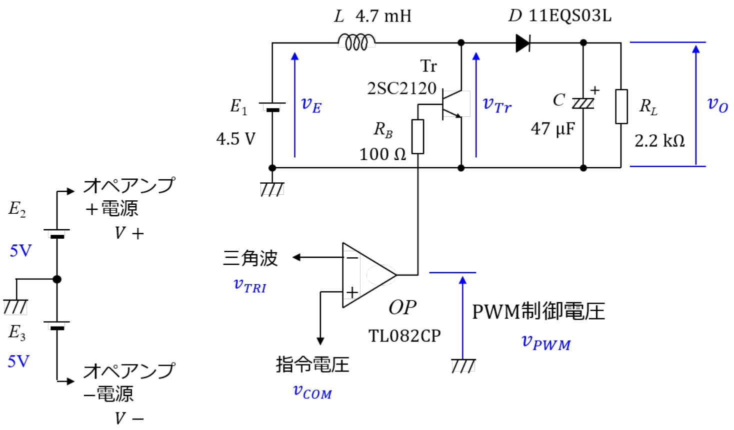


図 1: 昇圧チョッパの実験回路

図 1 は昇圧チョッパの実験回路です．三角波 v_{TRI} の繰り返し周期 $T_{TRI} = 100\,[\mu\text{s}]$（スイッチング周波数 $f_{SW} = 10\,[\text{kHz}]$），ピーク値 $V_{TP} = 2.5\,[\text{V}]$ に設定して，実験を行いました．実験課題は以下の 2 項目

でした．

1. 指令電圧 $v_{COM} = -2,\ 0,\ 2\,[\mathrm{V}]$ における v_{COM}，v_{TRI}，v_E，v_o の波形計測．
2. 指令電圧 $v_{COM} = 1.8\,[\mathrm{V}]$ から $2.2\,[\mathrm{V}]$ まで $0.2\,[\mathrm{V}]$ 刻みで変化させて，v_{PWM}，v_O の波形計測．

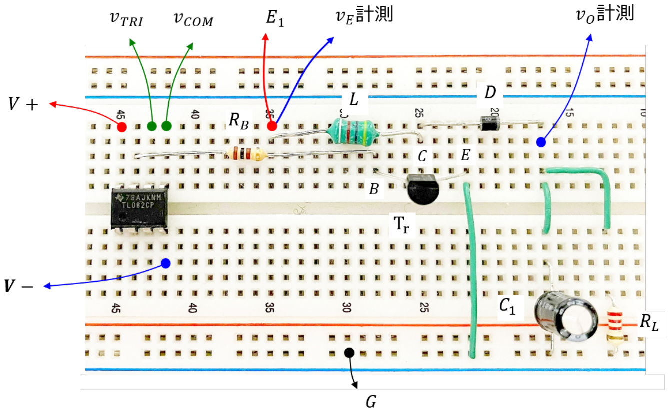
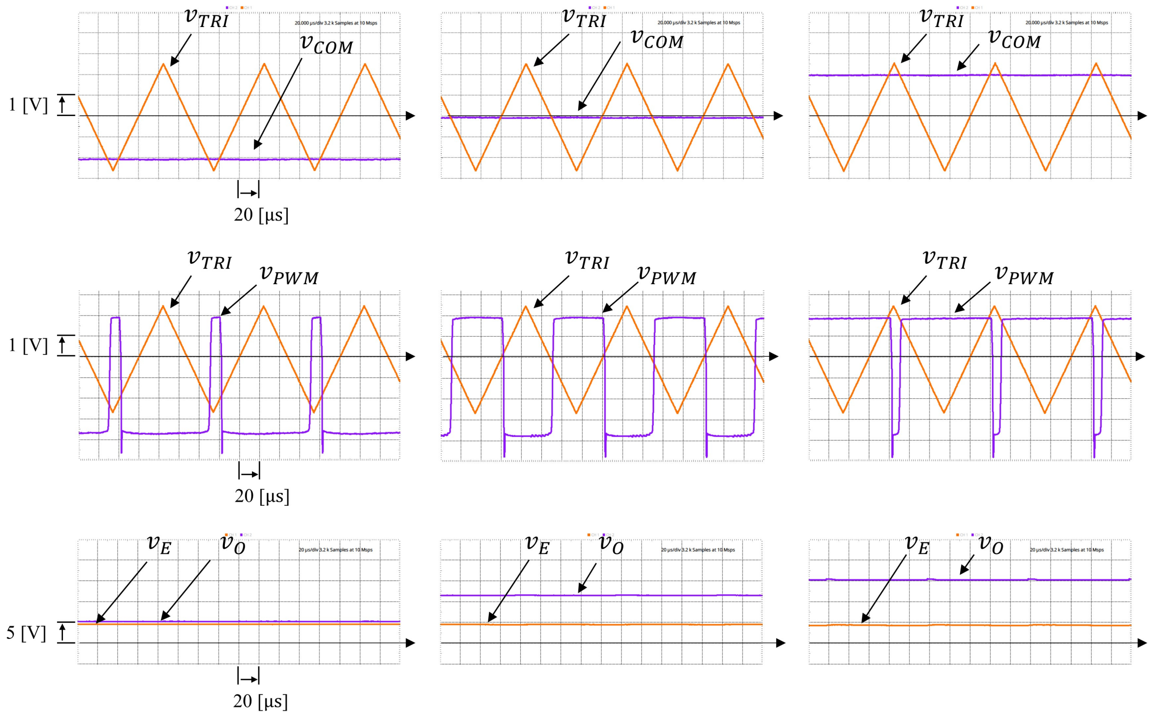


図 2: 昇圧チョッパの実験回路の配線例

図 2 は製作した実験回路の配線例です．ブレッドボード上に実験回路を組みました．

2.2 実験結果



(a) $v_{COM} = -2[\mathrm{V}]$ (b) $v_{COM} = 0[\mathrm{V}]$ (c) $v_{COM} = 2[\mathrm{V}]$

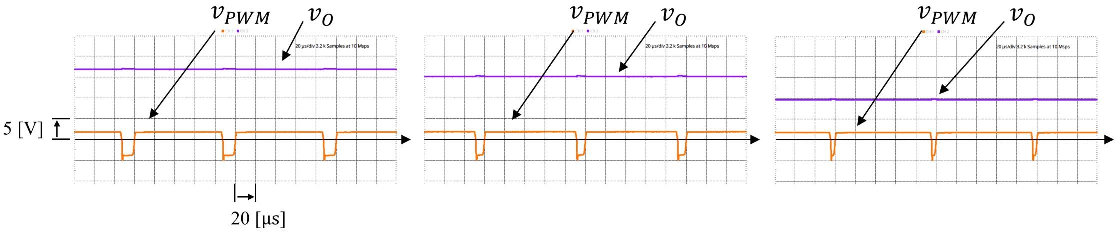
図 3: 昇圧チョッパの実験波形例（実験課題 1）

図 3 は，実験課題 1 の結果を示しています．指令電圧 v_{COM} を (a) -2 V，(b) 0 V，(c) 2 V に設定したときの波形です．

上段には，指令電圧 v_{COM} と三角波電圧 v_{TRI} の波形が示されています．

中段には，三角波電圧 v_{TRI} と PWM 制御電圧 v_{PWM} が示されており，v_{COM} が高くなるにつれて，v_{PWM} が “High” となる期間が長くなっていることが分かります．この “High” の期間中は，トランジスタ Tr がオンになります．

下段には，電源電圧 v_E と出力電圧 v_O の波形が示されています．指令電圧 $v_{COM} = -2$ V の場合，v_O は電源電圧 v_E よりわずかに高い程度の電圧ですが，$v_{COM} = 0$ V，2 V と上げるにつれて v_O もそれぞれ約 12 V，15 V と上昇し，昇圧が実現されていることが確認できます．



(a) v_{COM} = 1.8 [V]　(b) v_{COM} = 2 [V]　(c) v_{COM} = 2.2 [V]

図 4: 昇圧チョッパの実験波形例（実験課題 2）

図 4 は，実験課題 2 の結果を示しています．指令電圧 v_{COM} を (a) 1.8 V，(b) 2.0 V，(c) 2.2 V に設定した際の PWM 制御波形 v_{PWM} と出力電圧 v_O の波形です．

指令電圧が 2.0 V 付近の比較的高い領域になると，トランジスタがオンとなる時間が長くなる（言い換えれば，トランジスタがオフとなる時間が短くなる）につれて，v_O が低くなるという結果が得られました．

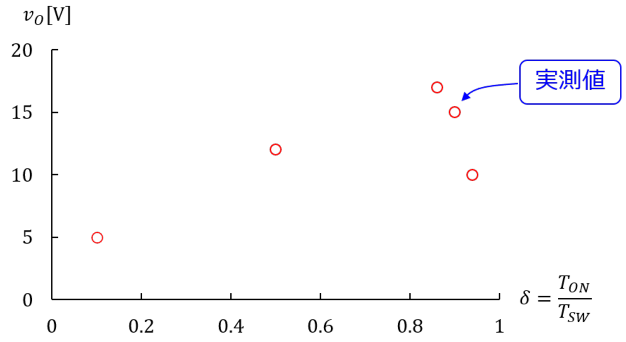


図 5: 昇圧チョッパの出力電圧 v_O 対通流率 δ の実験結果

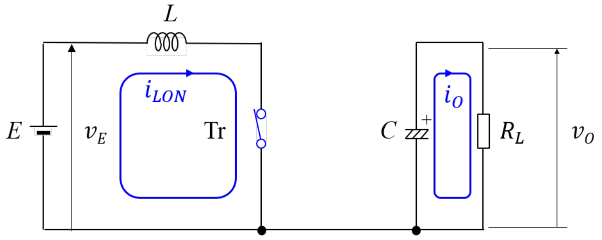
図 5 は昇圧チョッパの出力電圧 v_O −通流率 δ の実験結果です．

3 ChatGPT で回路解析

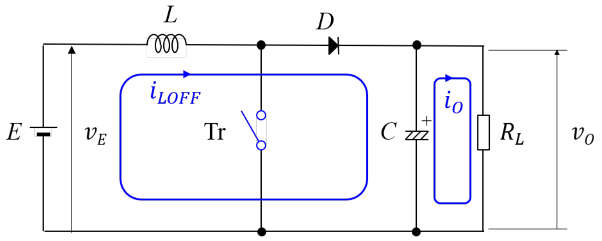
3.1 出力電圧 v_O の理論値（電流連続モード）

ここからは，筆者が ChatGPT o1 に昇圧チョッパの回路解析を依頼した対話ログを紹介します．

筆者の質問／依頼



(a) トランジスタ・オンモード　$T_{SWi} \le t \le T_{SWi} + T_{ON}$　$i = 0, 1, 2, \cdots$



(b) トランジスタ・オフモード $T_{SWi} + T_{ON} \le t \le T_{SWi+1}$　$i = 0, 1, 2, \cdots$

図 6: 昇圧チョッパの等価回路（R がない場合）

これは昇圧チョッパの等価回路です．(a) はトランジスタ・オンモード，(b) はオフモードの回路です．電流連続モードにおける出力電圧 vO と通流率 δ，電源電圧 vE の関係式を導出して．スイッチング周期は TSW とします．

ChatGPT o1 の回答

昇圧チョッパの電流連続モードにおける出力電圧と通流率の関係式

以下，スイッチング周期を T_{SW} とし，そのうちトランジスタがオンしている時間を δT_{SW} とする（δ は通流率＝デューティ比）．電流が連続する (CCM: Continuous Conduction Mode) 状態では，インダクタ電流 i_L が 1 周期ごとに同じ状態へ戻るため，インダクタ電圧の平均値は 1 周期内でゼロになる（ボルト秒バランス）．これにより以下の関係式を導く．

1. トランジスタ・オンモード（図 (a)）

トランジスタがオンしている区間 δT_{SW} では，インダクタには電源電圧 v_E がそのまま加わる．したがってオン時のインダクタ電圧 v_L は

$$v_L = v_E.$$

2. トランジスタ・オフモード（図 (b)）

トランジスタがオフしている区間 $(1-\delta)T_{\mathrm{SW}}$ では，ダイオードが順方向に導通し，インダクタの一端は電源 v_E，他端は出力電圧 v_O 側に接続される．よってオフ時のインダクタ電圧は

$$v_L = v_E - v_O.$$

3. ボルト秒バランス（平均インダクタ電圧 = 0）

連続モードの定常状態では，インダクタ電流の初期値と終端値が 1 周期ごとに等しく，インダクタ両端の平均電圧はゼロになる．よって

$$(\text{オン時の電圧}) \times (\text{オン時間}) \ + \ (\text{オフ時の電圧}) \times (\text{オフ時間}) = 0$$

を用いると，

$$v_E \cdot \big(\delta T_{\mathrm{SW}}\big) \ + \ \big(v_E - v_O\big)\big((1-\delta)T_{\mathrm{SW}}\big) = 0.$$

両辺を T_{SW} で割って整理すると，

$$v_E\,\delta \ + \ \big(v_E - v_O\big)\,(1-\delta) = 0,$$

これを展開すると，

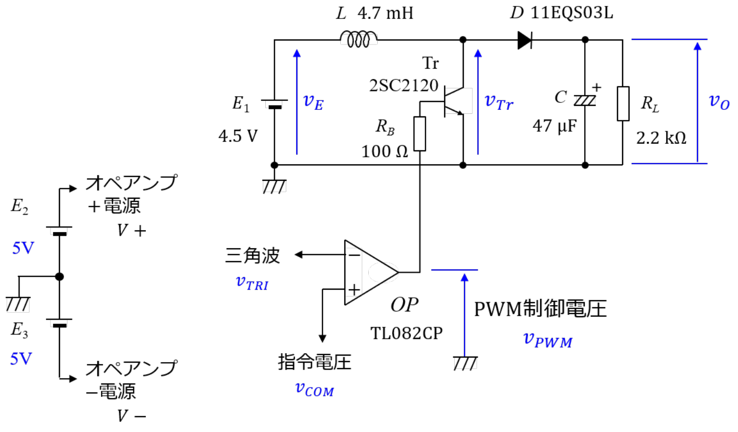
$$v_E\,\delta + v_E(1-\delta) - v_O(1-\delta) = 0,$$

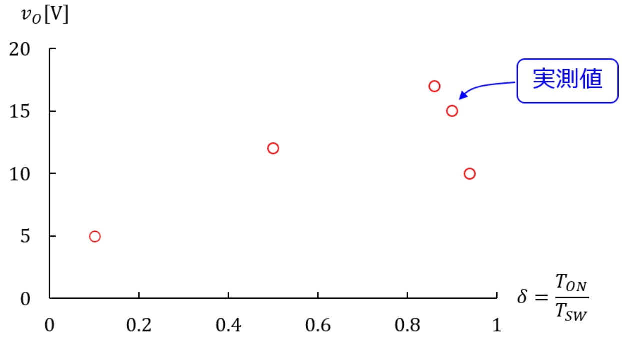
$$v_E - v_O(1-\delta) = 0,$$
$$v_O(1-\delta) = v_E.$$

したがって，昇圧チョッパ（ブーストコンバータ）の理想的な電流連続モードにおける出力電圧 v_O，通流率（デューティ比）δ，および電源電圧 v_E の関係は

$$\boxed{v_O = \frac{v_E}{1-\delta}}.$$

筆者の質問／依頼





これは昇圧チョッパの実験回路と vO-δ特性の実験結果です．この実験結果に上の vO の理論値を重ねて描く python のスクリプトを生成して．

ChatGPT o1 の回答

ステップアップチョッパ v_O–δ 特性の実験結果と理論曲線を重ね描画する Python スクリプト

以下は，実測されたデータをプロットし，理論式

$$v_O = \frac{v_E}{1-\delta}$$

による曲線を重ねて描画する Python スクリプト例です．実測値は `delta_data` と `vO_data` にご自身の実験データを入れてください．

```
import numpy as np
import matplotlib.pyplot as plt

# ==== 実験で得られたデータをここに入れる（例）====
delta_data = np.array([0.2, 0.3, 0.4, 0.5, 0.6])  # デューティ比の実測値
```

```
v0_data    = np.array([6.2, 7.5, 9.0, 11.0, 14.0]) # 出力電圧の実測値

# ==== 理論式のパラメータ ====
vE = 4.5  # 入力電圧（V）

# ==== 理論値の計算関数 ====
def v0_theoretical(delta, vE):
    """ v0 = vE / (1 - delta) """
    return vE / (1 - delta)

# ==== 理論曲線用のデータ生成 ====
# 0 < δ < 1 で描画するための点を作成
delta_range = np.linspace(0.01, 0.95, 200)
v0_range    = v0_theoretical(delta_range, vE)

# ==== グラフ描画 ====
plt.figure(figsize=(6,4))

# 実験値の散布図
plt.plot(delta_data, v0_data, 'ro', label='Measured (実測値)')

# 理論曲線
plt.plot(delta_range, v0_range, 'k-', label='Theory: $v_{0} = \\frac{v_{E}}{1-\\delta}$')

# 軸ラベルやタイトル等の設定
plt.xlabel('$\\delta$ (Duty ratio)')
plt.ylabel('$v_{0}$ (V)')
plt.title('Step-up Chopper: $v_{0}$ vs. $\\delta$')
plt.grid(True)
plt.legend()

plt.tight_layout()
plt.show()
```

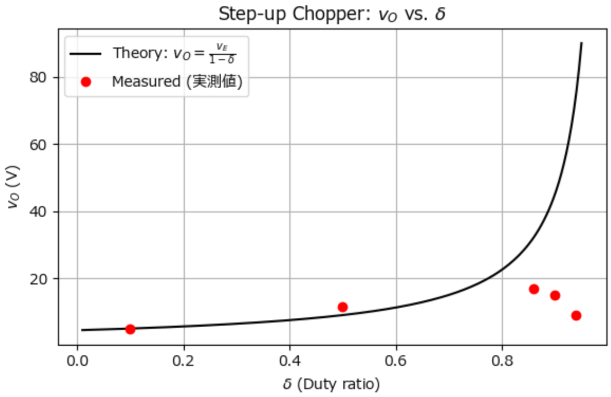
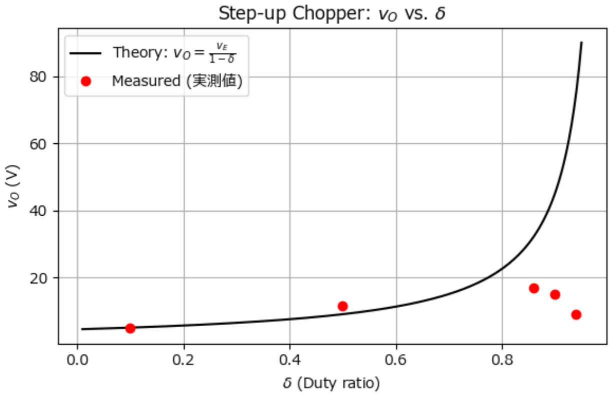


図 7: 昇圧チョッパの $v_O - \delta$ 特性（理論値 (電流連続モード) と実験値の比較）

図 7 は，delta_data と vO_data の値を以下のように実験値に置き換えて，コードを実行した結果です．

```
delta_data = np.array([0.1, 0.5, 0.86, 0.9, 0.94])  # デューティ比の実測値
vO_data    = np.array([5, 12, 17, 15, 10]) # 出力電圧の実測値
```

筆者の質問／依頼



これはコードの実行結果です．実験値と電流連続モードの理論値は大きく異なりました． δ ＝ 0.5 では，実験値の方が高く， δ =0.9 あたりでは実験値の方が大幅に低いです．この原因を考察して．

ChatGPT o1 の回答

実験値と理論値の大きな差異の考察

ステップアップチョッパの理想式

$$v_O = \frac{v_E}{1-\delta}$$

は以下のような前提を置いた理想解析にもとづいている．

1. スイッチ（トランジスタ）のオン抵抗やスイッチング損失が無い．
2. ダイオードの順方向降下が無視できる．

3. インダクタやコンデンサの内部抵抗が無視できる．
4. 電流が完全に連続している（入力出力ともにリプルや飽和などの影響を受けない）．

実際の回路では，これらの前提が満たされないため，下記のような理由で実験値と理論解析値の間に大きな差が生じることがある．

(1) 電流連続モードではない可能性

負荷が軽い場合やインダクタが小さい場合など，設定したデューティ比や負荷条件によってはインダクタ電流が途切れてしまい，電流不連続モード（DCM）になることがある．不連続モードでは

$$v_O = \frac{v_E}{1-\delta}$$

はそのまま適用できず，実際の出力電圧は理想値より小さくなる傾向がある．

(2) スイッチ・ダイオード・インダクタ等の非理想要因

実際のトランジスタにはオン抵抗（あるいはコレクタ・エミッタ間飽和電圧）やスイッチング損失が存在し，オン状態のときにも入力電源からインダクタへ流れる電流が一部損失する．さらにダイオードには順方向電圧降下があり，インダクタ電流が出力に流れ込む際に数 0.7 V（シリコンダイオード），あるいは数 0.3 V（ショットキーダイオード）のロスが生じる．またインダクタやコンデンサに内部抵抗（ESR）があると電力損失が増え，理想的な昇圧比を実現できなくなる．

低デューティ比領域 ($\delta \approx 0.5$) で実験値が理論値より高くなる場合　インダクタ電流のリプルや回路上の過渡的なオーバーシュート，測定誤差などにより，一時的に出力電圧が高く測定される可能性がある．

高デューティ比領域 ($\delta \approx 0.9$) で実験値が理論値を大きく下回る場合　トランジスタが長時間オンし続けることで，オン抵抗や飽和電圧，スイッチング損失が相対的に無視できなくなり，結果として理想値ほど出力電圧が上昇しない．またインダクタ電流が過度に大きくなると，部品の発熱や飽和が生じ，さらに出力電圧を押し下げる要因にもなる．

(3) 実際のデューティ比と制御のずれ

実験ではデューティ比を $\delta = 0.5$ と設定していても，ゲートドライバ回路やトランジスタの特性（飽和電圧，遅延時間など）により，理想どおりのオン時間が得られない場合がある．特に高いデューティ比（$\delta \to 1$）ではドライバのデッドタイムやスイッチング遅延が相対的に大きく影響し，実効デューティ比が減少したりロスが増大したりして出力電圧が理論値を下回る傾向が強まる．

(4) 負荷条件やリプルによる影響

負荷抵抗 R_L が実験中に変動したり，測定時のリプル電圧が大きい場合には，瞬間的な値にばらつきが生じて理想式と大きく乖離した値が得られることがある．高デューティ比では出力コンデンサに流れ込む脈動電流が増加し，平均としての出力電圧が理想値より低下するケースも多い．

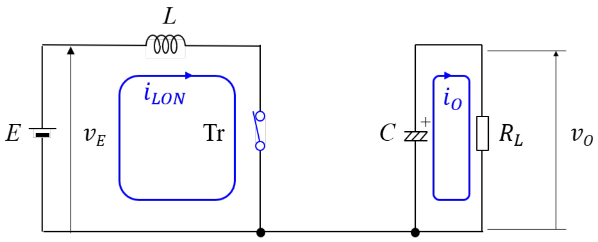
まとめ

- 実験値が理想値より高くなる原因：インダクタやスイッチング動作による瞬間的なオーバーシュートや測定誤差など．
- 実験値が理想値を大きく下回る原因：電流不連続モード，トランジスタやダイオード等の非理想要因（オン抵抗，順方向降下，スイッチング損失），実効デューティ比のずれ，インダクタの飽和や回路部品の発熱・抵抗による損失などで高い昇圧比を達成しにくくなっている．

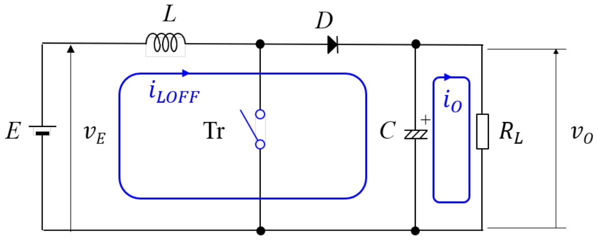
要するに，実回路では部品の損失や制御特性の詳細，あるいはモード遷移などが加わり，単純な CCM 理想式どおりにはならないため，観測結果がグラフのように大きくずれることは珍しくない．

3.2　昇圧チョッパの状態方程式

筆者の質問／依頼



(a) トランジスタ・オンモード　$T_{SWi} \le t \le T_{SWi} + T_{ON}$　$i = 0, 1, 2, \cdots$



(b) トランジスタ・オフモード $T_{SWi} + T_{ON} \le t \le T_{SWi+1}$　$i = 0, 1, 2, \cdots$

図 8: 昇圧チョッパの等価回路

これは昇圧チョッパの等価回路です．(a) はトランジスタ・オンモード，(b) はオフモードの回路です．それぞれで，iLON と vO, iLOFF と vO の微分方程式を立てて，状態方程式にまとめて．

ChatGPT o1 の回答

昇圧チョッパの状態方程式

以下では，インダクタ電流 i_L と出力電圧 v_O を状態変数とし，トランジスタがオンしている区間とオフしている区間それぞれの微分方程式を導き，最終的に状態方程式の形にまとめる．

1. トランジスタ・オンモード (図 (a))

トランジスタがオンしている間は，インダクタは電源電圧 v_E と直結され，ダイオードは逆方向となる．よってインダクタ電流 i_L と出力電圧 v_O は以下となる．

(1) インダクタ電流

$$L\frac{di_{LON}}{dt} = v_E \implies \frac{di_{LON}}{dt} = \frac{v_E}{L}.$$

(2) 出力電圧　オン時にはインダクタからは電流が出力へ供給されないため，出力側はコンデンサが負荷抵抗 R_L に放電する．コンデンサ電流は $i_C = -i_O = -\frac{v_O}{R_L}$ となるため

$$C\frac{dv_O}{dt} = -\frac{v_O}{R_L} \implies \frac{dv_O}{dt} = -\frac{1}{R_LC}v_O.$$

よってオン時の微分方程式は

$$\begin{cases} \dfrac{di_{LON}}{dt} = \dfrac{v_E}{L}, \\ \dfrac{dv_O}{dt} = -\dfrac{1}{R_LC}v_O. \end{cases}$$

2. トランジスタ・オフモード (図 (b))

トランジスタがオフしている間はダイオードが順方向導通し，インダクタ電流は出力側へ流れ込む．インダクタの一端が電源 v_E，他端が出力電圧 v_O となるので，

(1) インダクタ電流

$$L\frac{di_{LOFF}}{dt} = v_E - v_O \implies \frac{di_{LOFF}}{dt} = \frac{v_E - v_O}{L}.$$

(2) 出力電圧　コンデンサ電流はインダクタ電流 i_{LOFF} の一部を受け取り，残りが負荷抵抗へ流れる．すなわち

$$i_C = i_{LOFF} - i_O = i_{LOFF} - \frac{v_O}{R_L}.$$

よって

$$C\frac{dv_O}{dt} = i_{LOFF} - \frac{v_O}{R_L} \implies \frac{dv_O}{dt} = \frac{i_{LOFF}}{C} - \frac{v_O}{R_L\,C}.$$

よってオフ時の微分方程式は

$$\begin{cases} \dfrac{di_{LOFF}}{dt} = \dfrac{v_E - v_O}{L}, \\ \dfrac{dv_O}{dt} = \dfrac{i_{LOFF}}{C} - \dfrac{v_O}{R_LC}. \end{cases}$$

3. 状態方程式としての表現

インダクタ電流 i_L と出力電圧 v_O を

$$x_1 = i_L, \quad x_2 = v_O$$

と定義し，入力として電源電圧 v_E を用いると，スイッチのオン／オフに応じて以下のような行列方程式の形で表せる．

(1) トランジスタ・オン時

$$\frac{d}{dt}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & 0 \\ 0 & -\dfrac{1}{R_L C} \end{pmatrix}}_{A_{\text{on}}} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \underbrace{\begin{pmatrix} \dfrac{1}{L} \\ 0 \end{pmatrix}}_{B_{\text{on}}} v_E.$$

(2) トランジスタ・オフ時

$$\frac{d}{dt}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & -\dfrac{1}{L} \\ \dfrac{1}{C} & -\dfrac{1}{R_L C} \end{pmatrix}}_{A_{\text{off}}} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \underbrace{\begin{pmatrix} \dfrac{1}{L} \\ 0 \end{pmatrix}}_{B_{\text{off}}} v_E.$$

このようにオン時とオフ時で行列 A および B が切り替わるスイッチト系（切り替え系）として昇圧チョッパを表現することができる．

3.3 昇圧チョッパのシミュレーション

● 状態方程式のコーディング

筆者の質問／依頼

これらの状態方程式をルンゲクッタ法で解くシミュレーション・コードを python で書いて．ただし，E = 4.5V, L = 4.7mH, C=47 μ F, RL=2.2k Ω，TSW=100 μ s, TON = 50 μ s として．コードはオブジェクト指向形式にして．

ChatGPT o1 の回答

昇圧チョッパをルンゲクッタ法で解く Python コード（オブジェクト指向形式）

以下のコードは，昇圧チョッパの状態方程式をトランジスタ ON/OFF で切り替えながら 4 次のルンゲクッタ法で解くシミュレーションの例です．与えられたパラメータ

$$E = 4.5\,\mathrm{V}, \quad L = 4.7\,\mathrm{mH}, \quad C = 47\,\mu\mathrm{F}, \quad R_L = 2.2\,\mathrm{k}\Omega, \quad T_{\mathrm{SW}} = 100\,\mu\mathrm{s}, \quad T_{\mathrm{ON}} = 50\,\mu\mathrm{s}$$

を用いて数値解析を行います．

```
import numpy as np
import matplotlib.pyplot as plt

class BoostConverterSim:
    def __init__(self,
                 E=4.5,
                 L=4.7e-3,
                 C=47e-6,
                 RL=2.2e3,
                 Tsw=100e-6,
                 Ton=50e-6,
                 x0=None):
        """
        昇圧チョッパ (ブーストコンバータ) のパラメータと初期状態を設定.

        Parameters
        ----------
        E    : float
            電源電圧 [V]
        L    : float
            インダクタンス [H]
        C    : float
            キャパシタンス [F]
        RL   : float
            負荷抵抗 [Ω]
        Tsw  : float
            スイッチング周期 [s]
        Ton  : float
            トランジスタ ON 時間 [s]
        x0   : (float, float), optional
            [ i_L(0), v_O(0) ] の初期値. None なら [0.0, 0.0] とする.
        """
        self.E = E
```

```
        self.L = L
        self.C = C
        self.RL = RL
        self.Tsw = Tsw
        self.Ton = Ton

        # 状態変数の初期値 [ i_L(0), v_O(0) ]
        if x0 is None:
            x0 = np.array([0.0, 0.0])
        self.x0 = np.array(x0, dtype=float)

    def _f_on(self, t, x):
        """
        オンモード（トランジスタ ON）時の状態方程式:
          dx1/dt = E/L
          dx2/dt = - (1/(R_L*C)) vO
        x1 = i_L, x2 = v_O
        """
        i_L, v_O = x
        diL = self.E / self.L
        dvO = - (v_O / (self.RL * self.C))
        return np.array([diL, dvO])

    def _f_off(self, t, x):
        """
        オフモード（トランジスタ OFF）時の状態方程式:
          dx1/dt = (E - vO)/L
          dx2/dt = (i_L / C) - (vO/(R_L*C))
        x1 = i_L, x2 = v_O
        """
        i_L, v_O = x
        diL = (self.E - v_O) / self.L
        dvO = (i_L / self.C) - (v_O / (self.RL * self.C))
        return np.array([diL, dvO])

    def _rk4_step(self, f, t, x, dt):
        """
        4次のルンゲクッタ法で x(t) を1ステップ dt 進める.

        Parameters
        ----------
        f  : callable
            dx/dt = f(t, x) を返す関数
```

```
        t  : float
            現在時刻
        x  : ndarray
            現在の状態ベクトル
        dt : float
            ステップ幅

        Returns
        -------
        x_next : ndarray
            次の時刻 t + dt における状態ベクトル
        """
        k1 = f(t, x)
        k2 = f(t + dt/2, x + (dt/2)*k1)
        k3 = f(t + dt/2, x + (dt/2)*k2)
        k4 = f(t + dt,   x + dt*k3)
        x_next = x + (dt/6)*(k1 + 2*k2 + 2*k3 + k4)
        return x_next

    def simulate(self, t_end=5e-3, dt=1e-6):
        """
        t=0 から t=t_end までシミュレーションを実行する.

        Parameters
        ----------
        t_end : float
            シミュレーション終了時刻 [s]
        dt    : float
            数値積分の刻み幅 [s]

        Returns
        -------
        t_array : ndarray
            時刻列
        x_array : ndarray
            各時刻における状態ベクトル [i_L, v_O]
            shape = (len(t_array), 2)
        """
        n_steps = int(np.floor(t_end/dt))

        # 結果格納用
        t_array = np.zeros(n_steps+1)
        x_array = np.zeros((n_steps+1, 2))
```

```
        # 初期条件
        t_array[0] = 0.0
        x_array[0] = self.x0

        for i in range(n_steps):
            t = t_array[i]
            x = x_array[i]

            # (t % Tsw) < Ton ならトランジスタON, そうでなければOFF
            phase = (t % self.Tsw)
            if phase < self.Ton:
                f = self._f_on
            else:
                f = self._f_off

            # ルンゲクッタ1ステップ
            x_next = self._rk4_step(f, t, x, dt)

            t_array[i+1] = t + dt
            x_array[i+1] = x_next

        return t_array, x_array


# ==== メイン部(例) ====
if __name__ == "__main__":
    # シミュレーションオブジェクト生成
    sim = BoostConverterSim(
        E=4.5,
        L=4.7e-3,
        C=47e-6,
        RL=2.2e3,
        Tsw=100e-6,
        Ton=50e-6,
        x0=[0.0, 0.0]  # 初期値 (iL(0)=0, vO(0)=0)
    )

    # シミュレーション実行
    t_end = 5e-3  # 5msまで実行
    dt = 1e-6     # 刻み幅 1us
    t_array, x_array = sim.simulate(t_end=t_end, dt=dt)
```

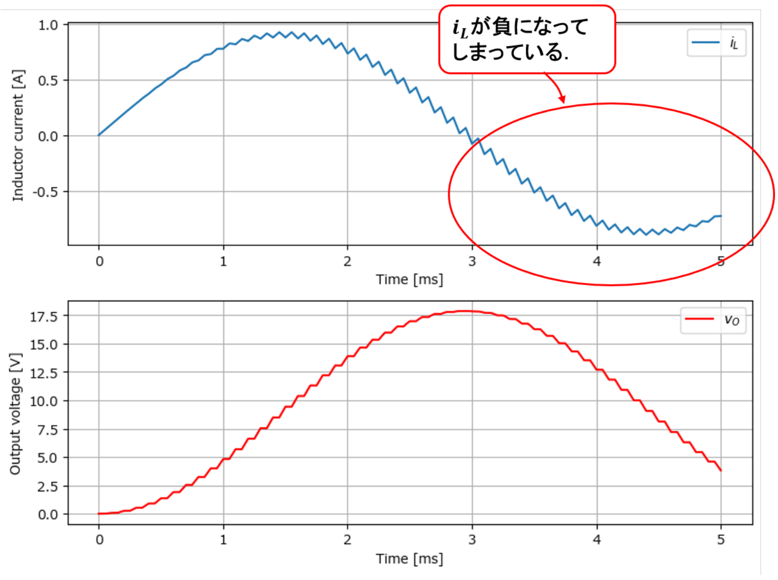
```
# 結果の取り出し
iL_history = x_array[:,0]
vO_history = x_array[:,1]

# グラフ描画
plt.figure(figsize=(8,6))

plt.subplot(2,1,1)
plt.plot(t_array*1e3, iL_history, label='$i_L$')
plt.xlabel('Time [ms]')
plt.ylabel('Inductor current [A]')
plt.grid(True)
plt.legend()

plt.subplot(2,1,2)
plt.plot(t_array*1e3, vO_history, 'r', label='$v_O$')
plt.xlabel('Time [ms]')
plt.ylabel('Output voltage [V]')
plt.grid(True)
plt.legend()

plt.tight_layout()
plt.show()
```



i_Lが負になってしまっている．

図 9: コードの実行結果

図 9 は，コードの実行結果を示しています．横軸は時間（単位：ms）で，縦軸は上段がインダクタ電流 i_L，下段が出力電圧 v_O です．トランジスタを 100μs の周期でオン・オフすることにより，電流波形はのこ

ぎり波状に，電圧波形は階段状になります．しかし，i_L が負となり，負荷から電源へ電流が逆流する誤った動作が確認されました．

● $i_L \geq 0$ へと改良

筆者の質問／依頼

iL が負になったら，強制的に 0 で置き換えるようにして．

ChatGPT o1 の回答

インダクタ電流が負になったら 0 にクランプする昇圧チョッパの Python コード

以下のコードは，昇圧チョッパの状態方程式をトランジスタの ON/OFF に応じて切り替えつつ 4 次のルンゲクッタ法で解き，計算結果でインダクタ電流 i_L が負になった場合に強制的に 0 へクランプする処理を加えたものである．

```
import numpy as np
import matplotlib.pyplot as plt

class BoostConverterSim:
    def __init__(self,
                 E=4.5,
                 L=4.7e-3,
                 C=47e-6,
                 RL=2.2e3,
                 Tsw=100e-6,
                 Ton=50e-6,
                 x0=None):
        """
        昇圧チョッパ (ブーストコンバータ) のパラメータと初期状態を設定.

        Parameters
        ----------
        E   : float
            電源電圧 [V]
        L   : float
            インダクタンス [H]
        C   : float
            キャパシタンス [F]
        RL  : float
            負荷抵抗 [Ω]
        Tsw : float
            スイッチング周期 [s]
```

```
        Ton : float
            トランジスタ ON 時間 [s]
        x0  : (float, float), optional
            [ i_L(0), v_O(0) ] の初期値. None なら [0.0, 0.0] とする.
        """
        self.E = E
        self.L = L
        self.C = C
        self.RL = RL
        self.Tsw = Tsw
        self.Ton = Ton

        # 状態変数の初期値 [ i_L(0), v_O(0) ]
        if x0 is None:
            x0 = np.array([0.0, 0.0])
        self.x0 = np.array(x0, dtype=float)

    def _f_on(self, t, x):
        """
        オンモード (トランジスタ ON) 時の状態方程式:
          dx1/dt = E/L
          dx2/dt = - (1/(R_L*C)) * vO
        x1 = i_L, x2 = v_O
        """
        i_L, v_O = x
        diL = self.E / self.L
        dvO = - (v_O / (self.RL * self.C))
        return np.array([diL, dvO])

    def _f_off(self, t, x):
        """
        オフモード (トランジスタ OFF) 時の状態方程式:
          dx1/dt = (E - vO)/L
          dx2/dt = (i_L / C) - (vO/(R_L*C))
        x1 = i_L, x2 = v_O
        """
        i_L, v_O = x
        diL = (self.E - v_O) / self.L
        dvO = (i_L / self.C) - (v_O / (self.RL * self.C))
        return np.array([diL, dvO])

    def _rk4_step(self, f, t, x, dt):
        """
```

```
        4 次のルンゲクッタ法で x(t) を 1 ステップ dt 進める.

        Parameters
        ----------
        f  : callable
            dx/dt = f(t, x) を返す関数
        t  : float
            現在時刻
        x  : ndarray
            現在の状態ベクトル
        dt : float
            ステップ幅

        Returns
        -------
        x_next : ndarray
            次の時刻 t + dt における状態ベクトル
        """
        k1 = f(t, x)
        k2 = f(t + dt/2, x + (dt/2)*k1)
        k3 = f(t + dt/2, x + (dt/2)*k2)
        k4 = f(t + dt,   x + dt*k3)
        x_next = x + (dt/6)*(k1 + 2*k2 + 2*k3 + k4)
        return x_next

    def simulate(self, t_end=5e-3, dt=1e-6):
        """
        t=0 から t=t_end までシミュレーションを実行する.

        Parameters
        ----------
        t_end : float
            シミュレーション終了時刻 [s]
        dt    : float
            数値積分の刻み幅 [s]

        Returns
        -------
        t_array : ndarray
            時刻列
        x_array : ndarray
            各時刻における状態ベクトル [i_L, v_0]
            shape = (len(t_array), 2)
```

```
        """
        n_steps = int(np.floor(t_end/dt))

        # 結果格納用
        t_array = np.zeros(n_steps+1)
        x_array = np.zeros((n_steps+1, 2))

        # 初期条件
        t_array[0] = 0.0
        x_array[0] = self.x0

        for i in range(n_steps):
            t = t_array[i]
            x = x_array[i]

            # (t % Tsw) < Ton ならトランジスタON，そうでなければOFF
            phase = (t % self.Tsw)
            if phase < self.Ton:
                f = self._f_on
            else:
                f = self._f_off

            # ルンゲクッタ1ステップ
            x_next = self._rk4_step(f, t, x, dt)

            # iL が負になったら0にクランプ
            if x_next[0] < 0.0:
                 x_next[0] = 0.0

            t_array[i+1] = t + dt
            x_array[i+1] = x_next

        return t_array, x_array


# ==== メイン部（例） ====
if __name__ == "__main__":
    # シミュレーションオブジェクト生成
    sim = BoostConverterSim(
        E=4.5,
        L=4.7e-3,
        C=47e-6,
        RL=2.2e3,
```

```
        Tsw=100e-6,
        Ton=50e-6,
        x0=[0.0, 0.0]  # 初期値 (iL(0)=0, v0(0)=0)
    )

    # シミュレーション実行
    t_end = 5e-3  # 5ms まで実行
    dt = 1e-6     # 刻み幅 1us
    t_array, x_array = sim.simulate(t_end=t_end, dt=dt)

    # 結果の取り出し
    iL_history = x_array[:,0]
    vO_history = x_array[:,1]

    # グラフ描画
    plt.figure(figsize=(8,6))

    plt.subplot(2,1,1)
    plt.plot(t_array*1e3, iL_history, label='$i_L$')
    plt.xlabel('Time [ms]')
    plt.ylabel('Inductor current [A]')
    plt.grid(True)
    plt.legend()

    plt.subplot(2,1,2)
    plt.plot(t_array*1e3, vO_history, 'r', label='$v_O$')
    plt.xlabel('Time [ms]')
    plt.ylabel('Output voltage [V]')
    plt.grid(True)
    plt.legend()

    plt.tight_layout()
    plt.show()
```

追加説明

オフモードなどで本来インダクタ電流がブロックされる経路をモデル化する上で,「電流が負にならない」という制約をシミュレーションの後処理で簡易的に実装するため，ルンゲクッタ 1 ステップごとに $i_L < 0$ の判定があった際は 0 にクランプしている.

より厳密には,「途中で電流が 0 となったタイミングで切り替え」というイベント検出を行う方法もあるが，実装が複雑になるため，ここではシンプルに 1 ステップごとにクランプする実装としている.

これにより，インダクタ電流がマイナス方向には流れないという近似を含むシミュレーションが可能となる．

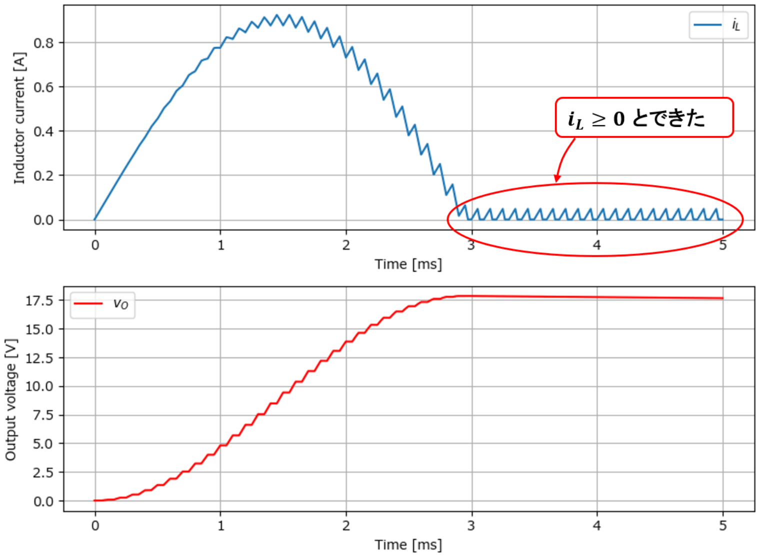


図 10: コード（i_L の負値を強制的に 0 とする修正）の実行結果

● 初期値とシミュレーション時間設定

筆者の質問／依頼

iL と vO の初期値をメインブロック内で指定できるようにして．vO の初期値を 12V にして．t_end を 0.5sec にして

ChatGPT o1 の回答

Python コード：iL(0)=0, vO(0)=12V, t_end=0.5s の昇圧チョッパシミュレーション

以下のコードは，メイン部で初期値（$i_L(0) = 0, v_O(0) = 12\,\mathrm{V}$）を指定し，シミュレーション終了時刻を 0.5 秒に設定した例である．また，インダクタ電流が負になった場合には 0 でクランプする処理を残してある．

```
import numpy as np
import matplotlib.pyplot as plt

class BoostConverterSim:
    def __init__(self,
                 E=4.5,
                 L=4.7e-3,
```

```
                 C=47e-6,
                 RL=2.2e3,
                 Tsw=100e-6,
                 Ton=50e-6,
                 x0=None):
        """
        昇圧チョッパ（ブーストコンバータ）のパラメータと初期状態を設定.

        Parameters
        ----------
        E   : float
            電源電圧 [V]
        L   : float
            インダクタンス [H]
        C   : float
            キャパシタンス [F]
        RL  : float
            負荷抵抗 [Ω]
        Tsw : float
            スイッチング周期 [s]
        Ton : float
            トランジスタ ON 時間 [s]
        x0  : (float, float), optional
            [ i_L(0), v_O(0) ] の初期値.
            省略（None）なら [0.0, 0.0] とする.
        """
        self.E = E
        self.L = L
        self.C = C
        self.RL = RL
        self.Tsw = Tsw
        self.Ton = Ton

        # 状態変数の初期値 [ i_L(0), v_O(0) ]
        if x0 is None:
            x0 = np.array([0.0, 0.0])
        self.x0 = np.array(x0, dtype=float)

    def _f_on(self, t, x):
        """
        オンモード (トランジスタ ON) 時の状態方程式:
          dx1/dt = E/L
          dx2/dt = - (1/(R_L*C)) * vO
```

```
        x1 = i_L, x2 = v_O
        """
        i_L, v_O = x
        diL = self.E / self.L
        dvO = - (v_O / (self.RL * self.C))
        return np.array([diL, dvO])

    def _f_off(self, t, x):
        """
        オフモード (トランジスタ OFF) 時の状態方程式:
          dx1/dt = (E - vO)/L
          dx2/dt = (i_L / C) - (vO/(R_L*C))
        x1 = i_L, x2 = v_O
        """
        i_L, v_O = x
        diL = (self.E - v_O) / self.L
        dvO = (i_L / self.C) - (v_O / (self.RL * self.C))
        return np.array([diL, dvO])

    def _rk4_step(self, f, t, x, dt):
        """
        4 次のルンゲクッタ法で x(t) を 1 ステップ dt 進める.

        Parameters
        ----------
        f  : callable
            dx/dt = f(t, x) を返す関数
        t  : float
            現在時刻
        x  : ndarray
            現在の状態ベクトル
        dt : float
            ステップ幅

        Returns
        -------
        x_next : ndarray
            次の時刻 t + dt における状態ベクトル
        """
        k1 = f(t, x)
        k2 = f(t + dt/2, x + (dt/2)*k1)
        k3 = f(t + dt/2, x + (dt/2)*k2)
        k4 = f(t + dt,   x + dt*k3)
```

```
        x_next = x + (dt/6)*(k1 + 2*k2 + 2*k3 + k4)
        return x_next

    def simulate(self, t_end=0.5, dt=1e-6):
        """
        t=0 から t=t_end までシミュレーションを実行する.

        Parameters
        ----------
        t_end : float
            シミュレーション終了時刻 [s]
        dt    : float
            数値積分の刻み幅 [s]

        Returns
        -------
        t_array : ndarray
            時刻列
        x_array : ndarray
            各時刻における状態ベクトル [i_L, v_O]
            shape = (len(t_array), 2)
        """
        n_steps = int(np.floor(t_end/dt))

        # 結果格納用
        t_array = np.zeros(n_steps+1)
        x_array = np.zeros((n_steps+1, 2))

        # 初期条件
        t_array[0] = 0.0
        x_array[0] = self.x0

        for i in range(n_steps):
            t = t_array[i]
            x = x_array[i]

            # (t % Tsw) < Ton ならトランジスタ ON, そうでなければ OFF
            phase = (t % self.Tsw)
            if phase < self.Ton:
                f = self._f_on
            else:
                f = self._f_off
```

```
            # ルンゲクッタ 1 ステップ
            x_next = self._rk4_step(f, t, x, dt)

            # iL が負になったら 0 にクランプ
            if x_next[0] < 0.0:
                x_next[0] = 0.0

            t_array[i+1] = t + dt
            x_array[i+1] = x_next

        return t_array, x_array


# ==== メイン部 (例) ====
if __name__ == "__main__":
    # 初期値指定: [iL(0), vO(0)] = [0.0, 12.0]
    x0_init = [0.0, 12.0]

    # シミュレーションオブジェクト生成
    sim = BoostConverterSim(
        E=4.5,
        L=4.7e-3,
        C=47e-6,
        RL=2.2e3,
        Tsw=100e-6,
        Ton=50e-6,
        x0=x0_init  # ここで初期値を指定
    )

    # シミュレーション実行
    t_end = 0.5   # 0.5 秒まで実行
    dt = 1e-6     # 刻み幅 1us
    t_array, x_array = sim.simulate(t_end=t_end, dt=dt)

    # 結果の取り出し
    iL_history = x_array[:,0]
    vO_history = x_array[:,1]

    # グラフ描画
    plt.figure(figsize=(8,6))

    plt.subplot(2,1,1)
    plt.plot(t_array, iL_history, label='$i_L$')
```

```
    plt.xlabel('Time [s]')
    plt.ylabel('Inductor current [A]')
    plt.grid(True)
    plt.legend()

    plt.subplot(2,1,2)
    plt.plot(t_array, vO_history, 'r', label='$v_O$')
    plt.xlabel('Time [s]')
    plt.ylabel('Output voltage [V]')
    plt.grid(True)
    plt.legend()

    plt.tight_layout()
    plt.show()
```

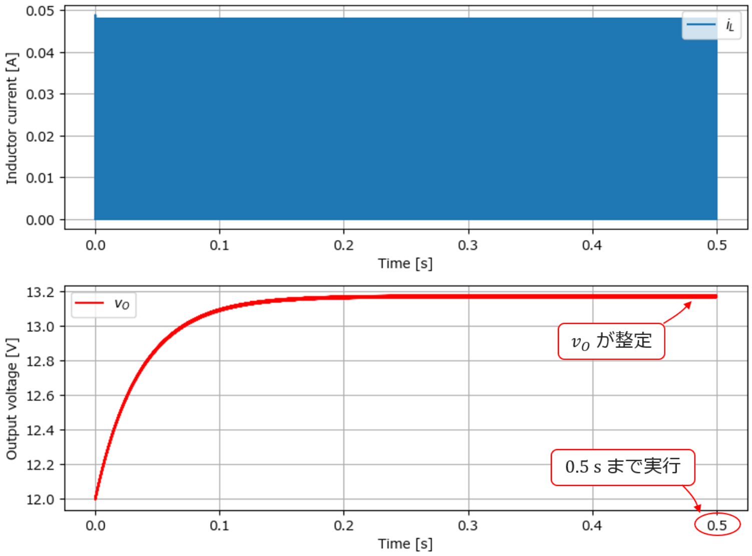


図 11: コード（i_L の負値を強制的に 0 とする修正）の実行結果

図 11 は，メインブロックで i_L と v_O の初期値を指定できるようにし，$v_O = 12$[V] を初期値としてシミュレーション終了時刻 (t_end) を 0.5 s に延長して実行した結果を示しています．グラフでは i_L の波形が塗りつぶされてしまい判別しづらいものの，v_O が整定していることが確認できます．

● 最後の 3 サイクルの波形描画

筆者の質問／依頼

グラフ表示において，iL, vO の下限値は 0, 上限値は最大値の 1.2 倍にして．グラフは最後の 3*TSW の期間を描くようにして．

ChatGPT o1 の回答

グラフ表示の範囲と軸設定を変更した昇圧チョッパの Python コード

以下では，昇圧チョッパのシミュレーション結果をプロットする際，

- 最後の $3 \times T_{\mathrm{SW}}$ の期間だけを表示する，
- 縦軸の下限値を 0，　上限値を区間内の最大値 ×1.2 にする，

という処理を加えたサンプルコードを示す．

```
import numpy as np
import matplotlib.pyplot as plt

class BoostConverterSim:
    def __init__(self,
                 E=4.5,
                 L=4.7e-3,
                 C=47e-6,
                 RL=2.2e3,
                 Tsw=100e-6,
                 Ton=50e-6,
                 x0=None):
        """
        昇圧チョッパ (ブーストコンバータ) のパラメータと初期状態を設定.

        Parameters
        ----------
        E   : float
            電源電圧 [V]
        L   : float
            インダクタンス [H]
        C   : float
            キャパシタンス [F]
        RL  : float
            負荷抵抗 [Ω]
        Tsw : float
            スイッチング周期 [s]
        Ton : float
```

```
            トランジスタ ON 時間 [s]
        x0 : (float, float), optional
            [ i_L(0), v_O(0) ] の初期値.
            省略 (None) なら [0.0, 0.0] とする.
        """
        self.E = E
        self.L = L
        self.C = C
        self.RL = RL
        self.Tsw = Tsw
        self.Ton = Ton

        # 状態変数の初期値 [ i_L(0), v_O(0) ]
        if x0 is None:
            x0 = np.array([0.0, 0.0])
        self.x0 = np.array(x0, dtype=float)

    def _f_on(self, t, x):
        """
        オンモード (トランジスタ ON) 時の状態方程式:
          dx1/dt = E/L
          dx2/dt = - (1/(R_L*C)) * vO
        x1 = i_L, x2 = v_O
        """
        i_L, v_O = x
        diL = self.E / self.L
        dvO = - (v_O / (self.RL * self.C))
        return np.array([diL, dvO])

    def _f_off(self, t, x):
        """
        オフモード (トランジスタ OFF) 時の状態方程式:
          dx1/dt = (E - vO)/L
          dx2/dt = (i_L / C) - (vO/(R_L*C))
        x1 = i_L, x2 = v_O
        """
        i_L, v_O = x
        diL = (self.E - v_O) / self.L
        dvO = (i_L / self.C) - (v_O / (self.RL * self.C))
        return np.array([diL, dvO])

    def _rk4_step(self, f, t, x, dt):
        """
```

```
        4次のルンゲクッタ法で x(t) を1ステップ dt 進める.

        Parameters
        ----------
        f   : callable
            dx/dt = f(t, x) を返す関数
        t   : float
            現在時刻
        x   : ndarray
            現在の状態ベクトル
        dt : float
            ステップ幅

        Returns
        -------
        x_next : ndarray
            次の時刻 t + dt における状態ベクトル
        """
        k1 = f(t, x)
        k2 = f(t + dt/2, x + (dt/2)*k1)
        k3 = f(t + dt/2, x + (dt/2)*k2)
        k4 = f(t + dt,   x + dt*k3)
        x_next = x + (dt/6)*(k1 + 2*k2 + 2*k3 + k4)
        return x_next

    def simulate(self, t_end=0.5, dt=1e-6):
        """
        t=0 から t=t_end までシミュレーションを実行する.

        Parameters
        ----------
        t_end : float
            シミュレーション終了時刻 [s]
        dt    : float
            数値積分の刻み幅 [s]

        Returns
        -------
        t_array : ndarray
            時刻列
        x_array : ndarray
            各時刻における状態ベクトル [i_L, v_0]
            shape = (len(t_array), 2)
```

```
        """
        n_steps = int(np.floor(t_end/dt))

        # 結果格納用
        t_array = np.zeros(n_steps+1)
        x_array = np.zeros((n_steps+1, 2))

        # 初期条件
        t_array[0] = 0.0
        x_array[0] = self.x0

        for i in range(n_steps):
            t = t_array[i]
            x = x_array[i]

            # オン／オフ切り替え
            phase = (t % self.Tsw)
            if phase < self.Ton:
                f = self._f_on
            else:
                f = self._f_off

            # ルンゲクッタ 1 ステップ
            x_next = self._rk4_step(f, t, x, dt)

            # iL が負になったら 0 にクランプ
            if x_next[0] < 0.0:
                x_next[0] = 0.0

            t_array[i+1] = t + dt
            x_array[i+1] = x_next

        return t_array, x_array


# ==== メイン部（例） ====
if __name__ == "__main__":
    # パラメータ設定（例）
    x0_init = [0.0, 12.0]  # 初期値: iL(0)=0, vO(0)=12V
    T_END = 0.5            # シミュレーション終了時刻 [s]
    DT = 1e-6              # 数値積分の刻み幅 [s]

    # シミュレーションオブジェクト生成
```

```
sim = BoostConverterSim(
    E=4.5,
    L=4.7e-3,
    C=47e-6,
    RL=2.2e3,
    Tsw=100e-6,
    Ton=50e-6,
    x0=x0_init
)

# シミュレーション実行
t_array, x_array = sim.simulate(t_end=T_END, dt=DT)

iL_history = x_array[:,0]
vO_history = x_array[:,1]

# ---- グラフ表示の設定 ----
# 1) 最後の 3*Tsw 秒間のみ描画
T_window = 3 * sim.Tsw                # 3倍のスイッチング周期
t_start_zoom = T_END - T_window       # ズーム開始時刻
# t_array から, この時刻以上の部分だけ抽出
idx_zoom = np.searchsorted(t_array, t_start_zoom)
t_zoom = t_array[idx_zoom:]
iL_zoom = iL_history[idx_zoom:]
vO_zoom = vO_history[idx_zoom:]

# 2) それぞれの最大値を求め, 上限を1.2倍に
iL_max = np.max(iL_zoom)
vO_max = np.max(vO_zoom)
# 下限は 0, 上限は最大値 * 1.2
iL_ylim = 1.2 * iL_max
vO_ylim = 1.2 * vO_max

# ---- グラフ描画開始 ----
plt.figure(figsize=(8,6))

# Inductor current
plt.subplot(2,1,1)
plt.plot(t_zoom, iL_zoom, label='$i_L$')
plt.xlabel('Time [s]')
plt.ylabel('Inductor current [A]')
plt.grid(True)
plt.legend()
```

```
plt.ylim([0, iL_ylim])
plt.xlim([t_zoom[0], t_zoom[-1]])

# Output voltage
plt.subplot(2,1,2)
plt.plot(t_zoom, vO_zoom, 'r', label='$v_O$')
plt.xlabel('Time [s]')
plt.ylabel('Output voltage [V]')
plt.grid(True)
plt.legend()
plt.ylim([0, vO_ylim])
plt.xlim([t_zoom[0], t_zoom[-1]])

plt.tight_layout()
plt.show()
```

変更点の要約

1. シミュレーション結果 `t_array` と `x_array` から，`T_END - 3*Tsw` 以上のインデックスのみ抽出し，最後の $3 \times T_{\mathrm{SW}}$ 区間だけを描画している．
2. 縦軸を

   $$0 \quad \text{から} \quad (\text{区間最大値}) \times 1.2$$

   に設定している．

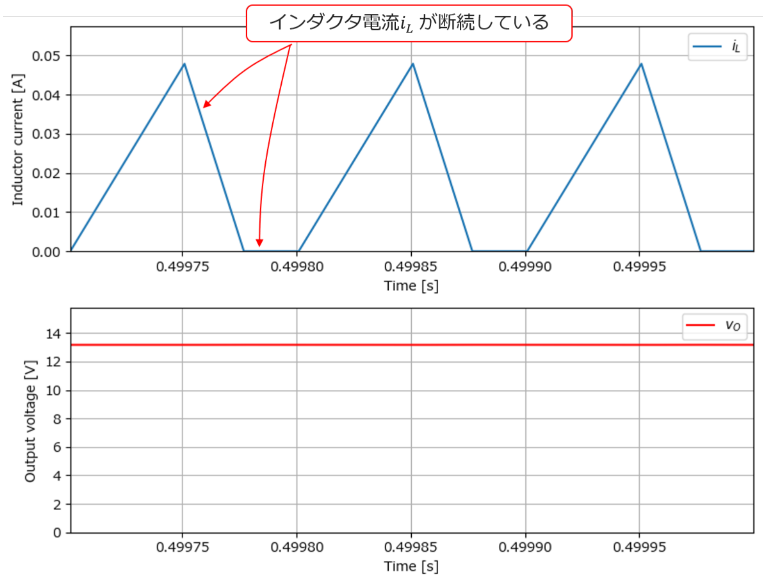


図 12: コード（i_L と v_O の下限値を 0，上限値をそれぞれの最大値の 1.2 倍に設定し，グラフ描画をシミュレーション結果の最後の $3 \times T_{\mathrm{SW}}$ の期間に限定）の実行結果

図 12 は，i_L と v_O の下限値を 0，上限値をそれぞれの最大値の 1.2 倍に設定し，グラフ描画をシミュレーション結果の最後の $3 \times T_{\mathrm{SW}}$ の期間に限定して実行した結果です．スイッチングの最後の 3 周期の波形が描画され，i_L の波形がはっきりと判るようになりました．この結果では，通流率 $\delta = 0.5$ のときにはインダクタ電流 i_L は毎スイッチングサイクルでゼロとなり，断続しています．

● v_O, i_L 特性のシミュレーション値，理論値，実験値の対比

筆者の質問／依頼

通流率 δ＝TON/TSW を 0.1, 0.5, 0.86, 0.9, 0.94 と変えてシミュレーションを実行して，最後の 3*TSW の期間の vO, iL の平均値を求めて．縦軸をこの平均値と vO の実験値（5, 12, 17, 15, 10V)，横軸を δ とする散布図を描き，この散布図に vO の理論値（電流連続モード）を重ねて描画するように python のコードを書きかえて．

ChatGPT o1 の回答

昇圧チョッパ：デューティ比を変えてシミュレーションし，最後の 3 周期平均と実験値，理論値を比較表示する Python コード

以下のコードでは，デューティ比 $\delta = 0.1, 0.5, 0.86, 0.9, 0.94$ ごとにシミュレーションを行い，各シミュレーション結果のうち「最後の $3 \times T_{\mathrm{SW}}$ 区間」での v_O, i_L の平均値を取得する．さらに，実験値と理論式

$$v_O = \frac{E}{1 - \delta}$$

を同一グラフに重ねて描画する．

```
import numpy as np
import matplotlib.pyplot as plt

class BoostConverterSim:
    def __init__(self,
                 E=4.5,
                 L=4.7e-3,
                 C=47e-6,
                 RL=2.2e3,
                 Tsw=100e-6,
                 Ton=50e-6,
                 x0=None):
        """
        昇圧チョッパ (ブーストコンバータ) のパラメータと初期状態を設定.

        Parameters
        ----------
        E   : float
            電源電圧 [V]
        L   : float
            インダクタンス [H]
        C   : float
            キャパシタンス [F]
        RL  : float
            負荷抵抗 [Ω]
        Tsw : float
            スイッチング周期 [s]
        Ton : float
            トランジスタ ON 時間 [s]
        x0  : (float, float), optional
            [ i_L(0), v_O(0) ] の初期値.
            省略 (None) なら [0.0, 0.0] とする.
        """
        self.E = E
        self.L = L
        self.C = C
        self.RL = RL
        self.Tsw = Tsw
        self.Ton = Ton

        # 状態変数の初期値 [ i_L(0), v_O(0) ]
        if x0 is None:
            x0 = np.array([0.0, 0.0])
```

```
        self.x0 = np.array(x0, dtype=float)

    def _f_on(self, t, x):
        """
        オンモード (トランジスタ ON) 時の状態方程式:
          dx1/dt = E / L
          dx2/dt = - (v0 / (R_L*C))
        x1 = i_L, x2 = v_0
        """
        i_L, v_0 = x
        diL = self.E / self.L
        dv0 = -(v_0 / (self.RL * self.C))
        return np.array([diL, dv0])

    def _f_off(self, t, x):
        """
        オフモード (トランジスタ OFF) 時の状態方程式:
          dx1/dt = (E - v0) / L
          dx2/dt = (i_L / C) - (v0 / (R_L*C))
        x1 = i_L, x2 = v_0
        """
        i_L, v_0 = x
        diL = (self.E - v_0) / self.L
        dv0 = (i_L / self.C) - (v_0 / (self.RL * self.C))
        return np.array([diL, dv0])

    def _rk4_step(self, f, t, x, dt):
        """
        4次のルンゲクッタ法で x(t) を1ステップ dt 進める.
        """
        k1 = f(t, x)
        k2 = f(t + dt/2, x + (dt/2)*k1)
        k3 = f(t + dt/2, x + (dt/2)*k2)
        k4 = f(t + dt,   x + dt*k3)
        x_next = x + (dt/6)*(k1 + 2*k2 + 2*k3 + k4)
        return x_next

    def simulate(self, t_end=5e-3, dt=1e-6):
        """
        t=0 から t=t_end までシミュレーションを実行する.

        Parameters
        ----------
```

```
        t_end : float
            シミュレーション終了時刻 [s]
        dt    : float
            数値積分の刻み幅 [s]

        Returns
        -------
        t_array : ndarray
            時刻列 (shape = (N+1,))
        x_array : ndarray
            各時刻における状態ベクトル [i_L, v_O] (shape = (N+1, 2))
        """
        n_steps = int(np.floor(t_end/dt))

        t_array = np.zeros(n_steps+1)
        x_array = np.zeros((n_steps+1, 2))

        t_array[0] = 0.0
        x_array[0] = self.x0

        for i in range(n_steps):
            t = t_array[i]
            x = x_array[i]

            # デューティ比に応じてオン／オフ判定
            phase = (t % self.Tsw)
            if phase < self.Ton:
                f = self._f_on
            else:
                f = self._f_off

            # ルンゲクッタ1ステップ
            x_next = self._rk4_step(f, t, x, dt)

            # iLが負になったら0にクランプ
            if x_next[0] < 0.0:
                x_next[0] = 0.0

            t_array[i+1] = t + dt
            x_array[i+1] = x_next

        return t_array, x_array
```

```
# -------------------------------------------------
# メイン処理
# -------------------------------------------------
if __name__ == "__main__":
    # A. パラメータ設定
    E = 4.5
    L = 4.7e-3
    C = 47e-6
    RL = 2.2e3
    Tsw = 100e-6
    T_END = 0.5      # シミュレーション時間
    dt = 1e-6         # 時間刻み幅

    # 複数のデューティ比
    duty_list = [0.1, 0.5, 0.86, 0.9, 0.94]

    # 実験値 (v0)
    v0_exp_data = [5, 12, 17, 15, 10]

    # シミュレーションから得た平均値を格納するリスト
    v0_sim_mean_list = []
    iL_sim_mean_list = []

    # B. デューティ比ごとにシミュレーションを実行し，最後の 3*Tsw 区間の平均を取る
    for delta in duty_list:
        Ton = delta * Tsw

        # シミュレーションオブジェクト生成
        sim = BoostConverterSim(E=E, L=L, C=C, RL=RL, Tsw=Tsw, Ton=Ton, x0=[0.0, 0.0])
        t_array, x_array = sim.simulate(t_end=T_END, dt=dt)

        iL_history = x_array[:,0]
        v0_history = x_array[:,1]

        # 最後の 3*Tsw 区間の平均を計算
        T_window = 3.0 * Tsw
        t_start_zoom = T_END - T_window
        idx_zoom = np.searchsorted(t_array, t_start_zoom)

        iL_zoom = iL_history[idx_zoom:]
        v0_zoom = v0_history[idx_zoom:]
```

```
    iL_mean = np.mean(iL_zoom)
    v0_mean = np.mean(v0_zoom)

    iL_sim_mean_list.append(iL_mean)
    v0_sim_mean_list.append(v0_mean)

# C. グラフ描画（横軸：δ，縦軸：v0）
#  1) シミュレーション平均値
#  2) 実験値
#  3) 理論値 CCM: v0 = E / (1 - δ)
delta_range = np.linspace(0.0, 0.99, 200)
v0_theory   = E / (1.0 - delta_range)

plt.figure(figsize=(7,5))

# シミュレーション平均値（青丸）
plt.scatter(duty_list, v0_sim_mean_list, color='blue', marker='o',
            label='Simulation (avg)')

# 実験値（赤×）
plt.scatter(duty_list, v0_exp_data, color='red', marker='x',
            label='Experiment')

# 理論値（緑色の曲線）
plt.plot(delta_range, v0_theory, color='green',

         label=r'Theory: $v_{0}=\frac{E}{1-\delta}$')

plt.xlabel(r'Duty ratio $\delta$')
plt.ylabel(r'Output voltage $v_{0}$ [V]')
plt.title('Boost Converter: Simulation vs. Experiment vs. Theory')
plt.grid(True)
plt.legend()

# x 軸は 0〜1
plt.xlim([0,1])

# y 軸の上限をシミュレーション値，実験値のうち最も大きい値の 1.2 倍に
y_max_all = max(max(v0_sim_mean_list), max(v0_exp_data))
plt.ylim([0, y_max_all*1.2])

plt.tight_layout()
plt.show()
```

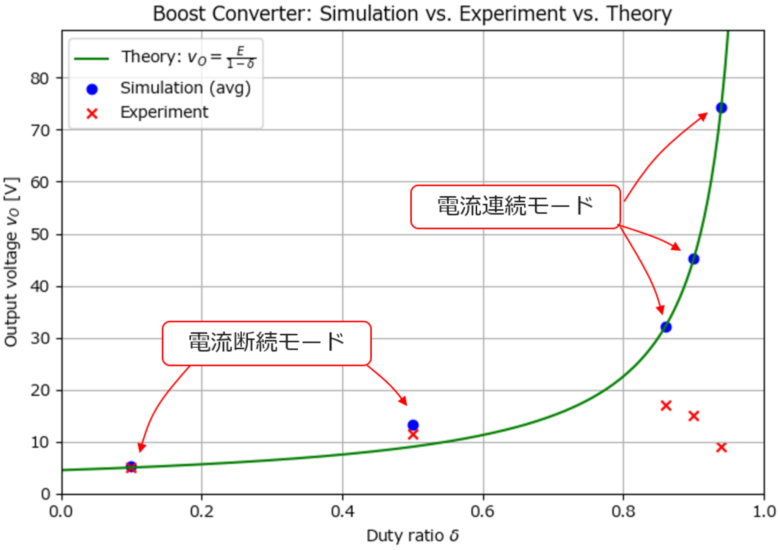


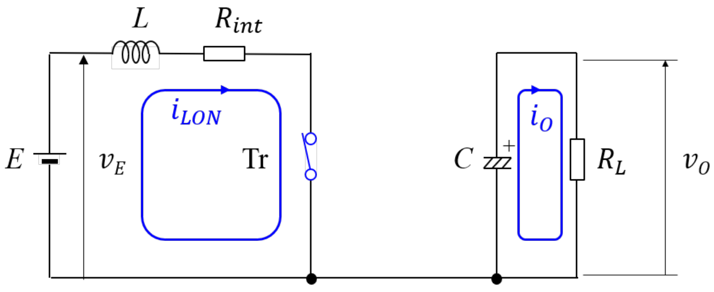
図 13: 通流率 δ を変えて，v_O のシミュレーション値，理論値，実験値の比較

図 13 は，コードの実行結果を示しています．ここでは，通流率 δ を変化させつつ，出力電圧 v_O のシミュレーション値（青丸）を求め，実験値（赤バツ）および理論値（緑線）と比較しています．

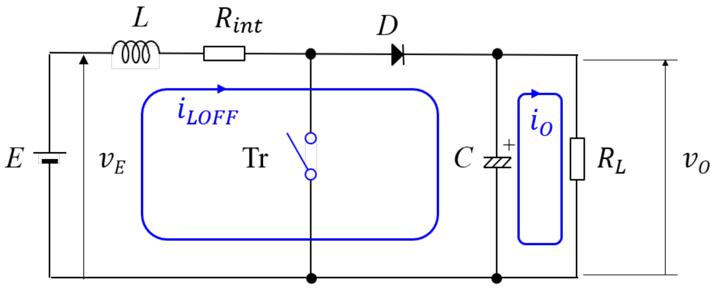
通流率 $\delta = 0.5$ の場合，シミュレーションによる v_O は実験値よりわずかに高い値となりました．一方，実験値が電流連続を前提とした理論値よりも高くなった理由としては，この通流率では実際に電流が断続していたことが考えられます．

3.4　インダクタの内部抵抗を考慮した状態方程式

筆者の質問／依頼



(a) トランジスタ・オンモード　$T_{SWi} \le t \le T_{SWi} + T_{ON}$　$i = 0, 1, 2, \cdots$



(b) トランジスタ・オフモード $T_{SWi} + T_{ON} \le t \le T_{SWi+1}$　$i = 0, 1, 2, \cdots$

図 14: インダクタ L の内部抵抗 R_{int} を含む場合の昇圧チョッパの等価回路

この昇圧チョッパの等価回路にはインダクタ L の内部抵抗 Rint が追加されています．トランジスタ・オンモードとオフモードそれぞれの状態方程式を書きかえて

ChatGPT o1 の回答

インダクタ内部抵抗 R_{int} を考慮した昇圧チョッパの状態方程式

以下では，インダクタに直列の内部抵抗 R_{int} を考慮し，状態変数をインダクタ電流 i_L と出力電圧 v_O として，トランジスタがオンの区間とオフの区間でそれぞれの微分方程式を示す．

1. トランジスタ・オンモード（図 (a)）

(1) インダクタ電流　トランジスタがオンしている間，インダクタには電源電圧 v_E が加わるが，内部抵抗 R_{int} で電圧降下 $R_{int}\, i_L$ が生じる．よってインダクタ両端電圧は $v_E - R_{int}\, i_L$ となり，

$$L\,\frac{di_L}{dt} = v_E - R_{int}\, i_L \implies \frac{di_L}{dt} = \frac{v_E - R_{int}\, i_L}{L}.$$

(2) 出力電圧　オン時にはダイオードが逆方向となり，インダクタからは出力側へ電流が供給されない．よってコンデンサは負荷抵抗 R_L に放電するだけであり，

$$i_C = -\frac{v_O}{R_L} \implies C\frac{dv_O}{dt} = -\frac{v_O}{R_L} \implies \frac{dv_O}{dt} = -\frac{v_O}{R_L\,C}.$$

$$\text{オンモード:}\begin{cases} \dfrac{di_L}{dt} = \dfrac{v_E - R_{int}\,i_L}{L}, \\ \dfrac{dv_O}{dt} = -\dfrac{v_O}{R_L\,C}. \end{cases}$$

2. トランジスタ・オフモード (図 (b))

(1) インダクタ電流　トランジスタがオフしている間，インダクタ電流はダイオードを介して出力側に流れ込む．このときインダクタ電圧は $v_E - R_{int}\,i_L - v_O$ となるので，

$$L\frac{di_L}{dt} = v_E - R_{int}\,i_L - v_O \implies \frac{di_L}{dt} = \frac{v_E - R_{int}\,i_L - v_O}{L}.$$

(2) 出力電圧　オフモード中はインダクタ電流 i_L の一部がコンデンサに流れ込み，残りが負荷抵抗へ流れる．

$$i_C = i_L - \frac{v_O}{R_L} \implies C\frac{dv_O}{dt} = i_L - \frac{v_O}{R_L} \implies \frac{dv_O}{dt} = \frac{i_L}{C} - \frac{v_O}{R_L\,C}.$$

$$\text{オフモード:}\begin{cases} \dfrac{di_L}{dt} = \dfrac{v_E - R_{int}\,i_L - v_O}{L}, \\ \dfrac{dv_O}{dt} = \dfrac{i_L}{C} - \dfrac{v_O}{R_L\,C}. \end{cases}$$

まとめ

- オンモードでは

$$L\frac{di_L}{dt} = v_E - R_{int}\,i_L, \qquad C\frac{dv_O}{dt} = -\frac{v_O}{R_L}.$$

- オフモードでは

$$L\frac{di_L}{dt} = v_E - R_{int}\,i_L - v_O, \qquad C\frac{dv_O}{dt} = i_L - \frac{v_O}{R_L}.$$

インダクタ内部抵抗 R_{int} が加わることで，インダクタ電流の変化において抵抗電圧降下 $R_{int}\,i_L$ が差し引かれる形となり，理想モデルに比べると実際の昇圧効果は低下することになる．

3.5　インダクタの内部抵抗を考慮した場合のシミュレーション

● 状態方程式のコーディング

筆者の質問／依頼

(最後の 3 サイクルの波形を描画するコードを呈示して)
このコードに，インダクタ L の内部抵抗 Rint = 30 Ωを取り入れて書きかえて

ChatGPT o1 の回答

インダクタ内部抵抗 $R_{\mathrm{int}} = 30\,\Omega$ を考慮した昇圧チョッパの Python コード

以下のコード例では，インダクタ内部抵抗 R_{int} を組み込むために，オンモードおよびオフモードの状態方程式に $-R_{\mathrm{int}}\,i_L$ の項を追加している．初期値やシミュレーション条件は一例である．

```
import numpy as np
import matplotlib.pyplot as plt

class BoostConverterSim:
    def __init__(self,
                 E=4.5,
                 L=4.7e-3,
                 Rint=30.0,    # ← インダクタ内部抵抗 [Ω]
                 C=47e-6,
                 RL=2.2e3,
                 Tsw=100e-6,
                 Ton=50e-6,
                 x0=None):
        """
        昇圧チョッパ (ブーストコンバータ) のパラメータと初期状態を設定.

        Parameters
        ----------
        E   : float
            電源電圧 [V]
        L   : float
            インダクタンス [H]
        Rint: float
            インダクタ内部抵抗 [Ω]
        C   : float
            キャパシタンス [F]
        RL  : float
            負荷抵抗 [Ω]
        Tsw : float
```

```
            スイッチング周期 [s]
        Ton : float
            トランジスタ ON 時間 [s]
        x0  : (float, float), optional
            [ i_L(0), v_O(0) ] の初期値.
            None なら [0.0, 0.0] とする.
        """
        self.E = E
        self.L = L
        self.Rint = Rint  # インダクタ内部抵抗
        self.C = C
        self.RL = RL
        self.Tsw = Tsw
        self.Ton = Ton

        # 状態変数の初期値 [ i_L(0), v_O(0) ]
        if x0 is None:
            x0 = np.array([0.0, 0.0])
        self.x0 = np.array(x0, dtype=float)

    def _f_on(self, t, x):
        """
        オンモード (トランジスタ ON) 時の状態方程式:
          dx1/dt = ( E - Rint * i_L ) / L
          dx2/dt = - vO / (RL * C)

        x1 = i_L, x2 = v_O
        """
        i_L, v_O = x
        diL = (self.E - self.Rint * i_L) / self.L
        dvO = - (v_O / (self.RL * self.C))
        return np.array([diL, dvO])

    def _f_off(self, t, x):
        """
        オフモード (トランジスタ OFF) 時の状態方程式:
          dx1/dt = ( E - Rint*i_L - vO ) / L
          dx2/dt = ( i_L / C ) - ( vO / (RL*C) )

        x1 = i_L, x2 = v_O
        """
        i_L, v_O = x
        diL = (self.E - self.Rint * i_L - v_O) / self.L
```

```
        dv0 = (i_L / self.C) - (v_0 / (self.RL * self.C))
        return np.array([diL, dv0])

    def _rk4_step(self, f, t, x, dt):
        """
        4 次のルンゲクッタ法で x(t) を 1 ステップ dt 進める.
        """
        k1 = f(t, x)
        k2 = f(t + dt/2, x + (dt/2)*k1)
        k3 = f(t + dt/2, x + (dt/2)*k2)
        k4 = f(t + dt,   x + dt*k3)
        x_next = x + (dt/6)*(k1 + 2*k2 + 2*k3 + k4)
        return x_next

    def simulate(self, t_end=0.5, dt=1e-6):
        """
        t=0 から t=t_end までシミュレーションを実行する.

        Parameters
        ----------
        t_end : float
            シミュレーション終了時刻 [s]
        dt    : float
            数値積分の刻み幅 [s]

        Returns
        -------
        t_array : ndarray
            時刻列
        x_array : ndarray
            各時刻における状態ベクトル [i_L, v_0]
            shape = (len(t_array), 2)
        """
        n_steps = int(np.floor(t_end/dt))

        # 結果格納用
        t_array = np.zeros(n_steps+1)
        x_array = np.zeros((n_steps+1, 2))

        # 初期条件
        t_array[0] = 0.0
        x_array[0] = self.x0
```

```
        for i in range(n_steps):
            t = t_array[i]
            x = x_array[i]

            # オン／オフ切り替え
            phase = (t % self.Tsw)
            if phase < self.Ton:
                f = self._f_on
            else:
                f = self._f_off

            # ルンゲクッタ1ステップ
            x_next = self._rk4_step(f, t, x, dt)

            # iLが負になったら0にクランプ
            if x_next[0] < 0.0:
                x_next[0] = 0.0

            t_array[i+1] = t + dt
            x_array[i+1] = x_next

        return t_array, x_array


# ==== メイン部（例） ====
if __name__ == "__main__":
    # パラメータ設定（例）
    x0_init = [0.0, 12.0]  # 初期値: iL(0)=0, v0(0)=12V
    T_END = 0.5            # シミュレーション終了時刻 [s]
    DT = 1e-6              # 数値積分の刻み幅 [s]

    # シミュレーションオブジェクト生成
    # Rint=30.0 で内部抵抗を設定
    sim = BoostConverterSim(
        E=4.5,
        L=4.7e-3,
        Rint=30.0,
        C=47e-6,
        RL=2.2e3,
        Tsw=100e-6,
        Ton=50e-6,
        x0=x0_init
    )
```

```
# シミュレーション実行
t_array, x_array = sim.simulate(t_end=T_END, dt=DT)

iL_history = x_array[:,0]
vO_history = x_array[:,1]

# ---- グラフ表示の設定 ----
# 1） 最後の 3*Tsw 秒間のみ描画
T_window = 3 * sim.Tsw
t_start_zoom = T_END - T_window
idx_zoom = np.searchsorted(t_array, t_start_zoom)
t_zoom = t_array[idx_zoom:]
iL_zoom = iL_history[idx_zoom:]
vO_zoom = vO_history[idx_zoom:]

# 2） それぞれの最大値を求め，上限を 1.2 倍に
iL_max = np.max(iL_zoom)
vO_max = np.max(vO_zoom)
iL_ylim = 1.2 * iL_max
vO_ylim = 1.2 * vO_max

# ---- グラフ描画 ----
plt.figure(figsize=(8,6))

plt.subplot(2,1,1)
plt.plot(t_zoom, iL_zoom, label='$i_L$')
plt.xlabel('Time [s]')
plt.ylabel('Inductor current [A]')
plt.grid(True)
plt.legend()
plt.ylim([0, iL_ylim])
plt.xlim([t_zoom[0], t_zoom[-1]])

plt.subplot(2,1,2)
plt.plot(t_zoom, vO_zoom, 'r', label='$v_O$')
plt.xlabel('Time [s]')
plt.ylabel('Output voltage [V]')
plt.grid(True)
plt.legend()
plt.ylim([0, vO_ylim])
plt.xlim([t_zoom[0], t_zoom[-1]])
```

```
    plt.tight_layout()
    plt.show()
```

修正点の概要

- コンストラクタに R_{int} （インダクタの内部抵抗）を追加し，既定値を 30 Ω に設定.
- オンモードの微分方程式:

$$\frac{di_L}{dt} = \frac{E - R_{\mathrm{int}}\, i_L}{L}, \qquad \frac{dv_O}{dt} = -\frac{v_O}{R_L\, C}.$$

- オフモードの微分方程式:

$$\frac{di_L}{dt} = \frac{E - R_{\mathrm{int}}\, i_L - v_O}{L}, \qquad \frac{dv_O}{dt} = \frac{i_L}{C} - \frac{v_O}{R_L\, C}.$$

このように，オン／オフいずれの状態方程式でも，インダクタ電流 i_L に対する内部抵抗の電圧降下 $R_{\mathrm{int}}\, i_L$ を差し引いているのがポイントである.

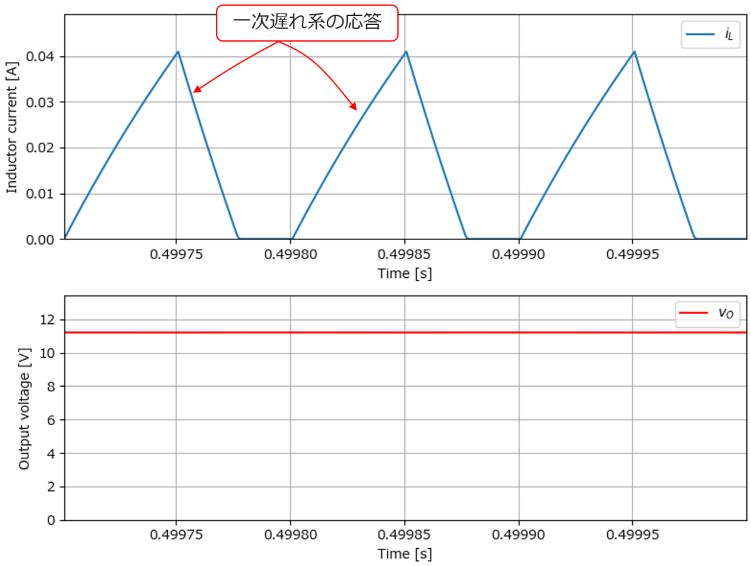


図 15: インダクタ L の内部抵抗 R_{int} を含む昇圧チョッパ等価回路のシミュレーション波形例（$\delta = 0.5$）

図 15 は，インダクタ L の内部抵抗 R_{int} を考慮した昇圧チョッパ等価回路のシミュレーション波形例です．インダクタ電流 i_L は R_{int} の影響で一次遅れ系の応答を示しています.

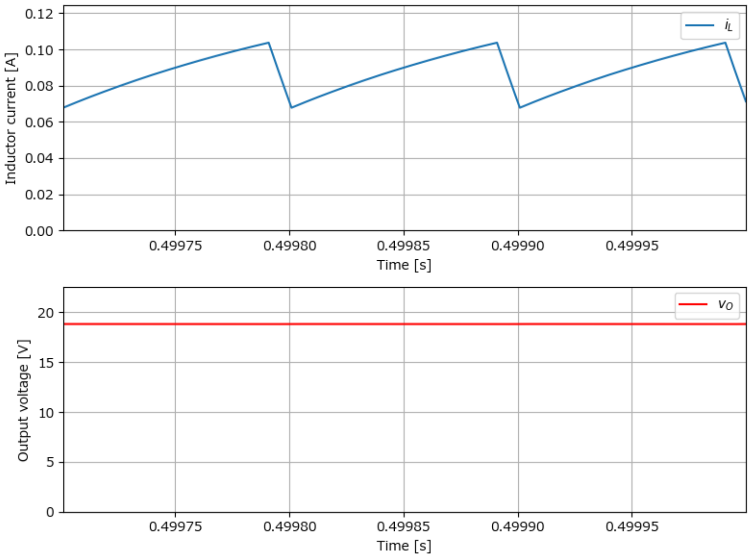


図 16: インダクタ L の内部抵抗 R_{int} を含む昇圧チョッパ等価回路のシミュレーション波形例（$\delta = 0.9$）

図 16 は，通流率 $\delta = 0.9$ としたときの i_L と v_O の波形例です．δ を大きくしたことで，一次遅れ系の応答が顕著に現れています．i_L は連続となりました．

● v_O, i_L 特性のシミュレーション値，理論値，実験値の対比

筆者の質問／依頼

（通流率 δ を変えて v_O のシミュレーション値を求めて，理論値，実験値と比較するグラフを描くコードを呈示して）
このコードに，インダクタ L の内部抵抗 Rint を取り入れるように書きかえて．

ChatGPT o1 の回答

インダクタ内部抵抗 R_{int} を考慮した昇圧チョッパの Python コード

以下は，インダクタ内部抵抗 R_{int} を導入し，オン／オフそれぞれの状態方程式に $-R_{\mathrm{int}}\, i_L$ の項を加えたシミュレーションコード例である．

```
import numpy as np
import matplotlib.pyplot as plt

class BoostConverterSim:
    def __init__(self,
                 E=4.5,
                 L=4.7e-3,
                 Rint=30.0,   # ← インダクタの内部抵抗 [Ω] を追加
                 C=47e-6,
                 RL=2.2e3,
```

```
                 Tsw=100e-6,
                 Ton=50e-6,
                 x0=None):
        """
        昇圧チョッパ (ブーストコンバータ) のパラメータと初期状態を設定.

        Parameters
        ----------
        E    : float
            電源電圧 [V]
        L    : float
            インダクタンス [H]
        Rint: float
            インダクタ内部抵抗 [Ω] (追加)
        C    : float
            キャパシタンス [F]
        RL   : float
            負荷抵抗 [Ω]
        Tsw  : float
            スイッチング周期 [s]
        Ton  : float
            トランジスタ ON 時間 [s]
        x0   : (float, float), optional
            [ i_L(0), v_O(0) ] の初期値.
            省略 (None) なら [0.0, 0.0] とする.
        """
        self.E = E
        self.L = L
        self.Rint = Rint  # 追加
        self.C = C
        self.RL = RL
        self.Tsw = Tsw
        self.Ton = Ton

        # 状態変数の初期値 [ i_L(0), v_O(0) ]
        if x0 is None:
            x0 = np.array([0.0, 0.0])
        self.x0 = np.array(x0, dtype=float)

    def _f_on(self, t, x):
        """
        オンモード (トランジスタ ON) 時の状態方程式:
          dx1/dt = ( E - Rint * i_L ) / L
```

```
          dx2/dt = - ( v0 / ( R_L*C ) )
        x1 = i_L, x2 = v_O
        """
        i_L, v_O = x
        diL = (self.E - self.Rint * i_L) / self.L
        dv0 = -(v_O / (self.RL * self.C))
        return np.array([diL, dv0])

    def _f_off(self, t, x):
        """
        オフモード（トランジスタ OFF）時の状態方程式：
          dx1/dt = ( E - Rint * i_L - v0 ) / L
          dx2/dt = ( i_L / C ) - ( v0 / ( R_L*C ) )
        x1 = i_L, x2 = v_O
        """
        i_L, v_O = x
        diL = (self.E - self.Rint * i_L - v_O) / self.L
        dv0 = (i_L / self.C) - (v_O / (self.RL * self.C))
        return np.array([diL, dv0])

    def _rk4_step(self, f, t, x, dt):
        """
        4 次のルンゲクッタ法で x(t) を 1 ステップ dt 進める．
        """
        k1 = f(t, x)
        k2 = f(t + dt/2, x + (dt/2)*k1)
        k3 = f(t + dt/2, x + (dt/2)*k2)
        k4 = f(t + dt,   x + dt*k3)
        x_next = x + (dt/6)*(k1 + 2*k2 + 2*k3 + k4)
        return x_next

    def simulate(self, t_end=5e-3, dt=1e-6):
        """
        t=0 から t=t_end までシミュレーションを実行する．

        Parameters
        ----------
        t_end : float
            シミュレーション終了時刻 [s]
        dt    : float
            数値積分の刻み幅 [s]

        Returns
```

```
        -------
        t_array : ndarray
            時刻列 (shape = (N+1,))
        x_array : ndarray
            各時刻における状態ベクトル [i_L, v_0] (shape = (N+1, 2))
        """
        n_steps = int(np.floor(t_end/dt))

        t_array = np.zeros(n_steps+1)
        x_array = np.zeros((n_steps+1, 2))

        t_array[0] = 0.0
        x_array[0] = self.x0

        for i in range(n_steps):
            t = t_array[i]
            x = x_array[i]

            # デューティ比に応じてオン／オフ判定
            phase = (t % self.Tsw)
            if phase < self.Ton:
                f = self._f_on
            else:
                f = self._f_off

            # ルンゲクッタ1ステップ
            x_next = self._rk4_step(f, t, x, dt)

            # iL が負になったら0にクランプ
            if x_next[0] < 0.0:
                x_next[0] = 0.0

            t_array[i+1] = t + dt
            x_array[i+1] = x_next

        return t_array, x_array


# ===================================================
# メイン処理
# ===================================================
if __name__ == "__main__":
    E = 4.5
```

```
L = 4.7e-3
Rint = 30.0  # インダクタ内部抵抗 [Ω]
C = 47e-6
RL = 2.2e3
Tsw = 100e-6
T_END = 5e-3

# 複数のデューティ比
duty_list = [0.1, 0.5, 0.86, 0.9, 0.94]

# 実験値 (v0)
v0_exp_data = [5, 12, 17, 15, 10]

v0_sim_mean_list = []
iL_sim_mean_list = []

# シミュレーションをデューティ比ごとに実行
for delta in duty_list:
    Ton = delta * Tsw
    sim = BoostConverterSim(E=E, L=L, Rint=Rint, C=C, RL=RL,
                            Tsw=Tsw, Ton=Ton, x0=[0.0, 0.0])

    dt = 1e-6
    t_array, x_array = sim.simulate(t_end=T_END, dt=dt)

    iL_history = x_array[:,0]
    v0_history = x_array[:,1]

    # 最後の 3*Tsw 区間を取り出して平均化
    T_window = 3.0 * Tsw
    t_start_zoom = T_END - T_window
    idx_zoom = np.searchsorted(t_array, t_start_zoom)

    iL_zoom = iL_history[idx_zoom:]
    v0_zoom = v0_history[idx_zoom:]

    iL_mean = np.mean(iL_zoom)
    v0_mean = np.mean(v0_zoom)

    iL_sim_mean_list.append(iL_mean)
    v0_sim_mean_list.append(v0_mean)

# グラフ描画
```

```
delta_range = np.linspace(0.0, 0.99, 200)
vO_theory   = E / (1.0 - delta_range)

plt.figure(figsize=(7,5))

plt.scatter(duty_list, vO_sim_mean_list,
            color='blue', marker='o', label='Simulation (avg)')
plt.scatter(duty_list, vO_exp_data,
            color='red', marker='x', label='Experiment')
plt.plot(delta_range, vO_theory,

         color='green', label=r'Theory: $v_{O}=\frac{E}{1-\delta}$')

plt.xlabel(r'Duty ratio $\delta$')
plt.ylabel(r'Output voltage $v_{O}$ [V]')
plt.title('Boost Converter with $R_{\mathrm{int}}=30\Omega$')
plt.grid(True)
plt.legend()
plt.xlim([0,1])
plt.ylim([0, max(max(vO_sim_mean_list), max(vO_exp_data))*1.2])
plt.tight_layout()
plt.show()
```

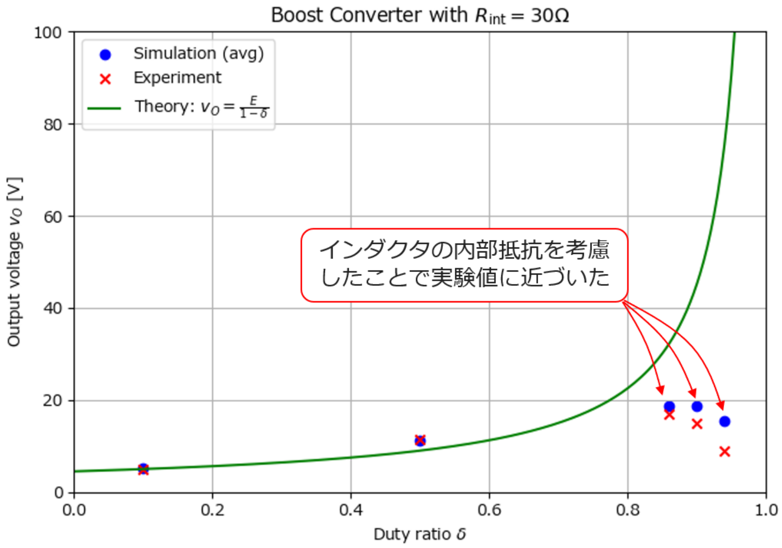
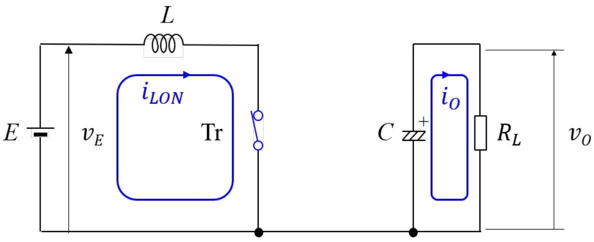


図 17: 通流率 δ を変えて，v_O のシミュレーション値，理論値，実験値の比較（インダクタ L の内部抵抗 $R_{int}=30\Omega$ を考慮）

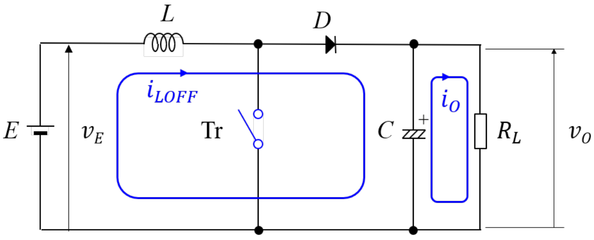
図 17 は，インダクタ L の内部抵抗 $R_{int} = 30\Omega$ を考慮したコードの実行結果です．通流率 δ を変えて，出力電圧 v_O のシミュレーション値（$R_{int} = 30\Omega$）（青丸）を求め，実験値（赤バツ）および理論値（緑線）と比較しています．インダクタの内部抵抗を状態方程式に組み入れたことで，通流率 $\delta = 0.8$ を超えた領域で，シミュレーション値が実験値と同様に減小する結果が得られました．

3.6 出力電圧 v_O の理論値（電流断続モード）

筆者の質問／依頼



(a) トランジスタ・オンモード　$T_{swi} \le t \le T_{swi} + T_{ON}$　$i = 0, 1, 2, \cdots$



(b) トランジスタ・オフモード $T_{swi} + T_{ON} \le t \le T_{swi+1}$　$i = 0, 1, 2, \cdots$

図 18: 昇圧チョッパの等価回路

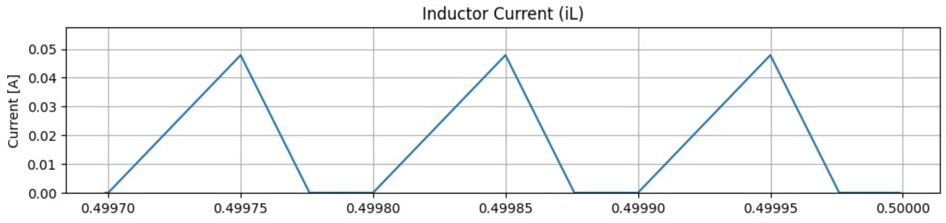


図 19: インダクタ電流波形例（インダクタ抵抗 $R = 0$, 通流率 $\delta = 0.5(\mathrm{TON} = 50\mu s)$

これは昇圧チョッパのトランジスタ・オン時とオフ時の等価回路と通流率 δ =0.5 のときのインダクタ電流 iL の波形です．iL の特徴は毎スイッチングサイクルで断続している点です．この電流断続時における，昇圧チョッパ vO と電源電圧 VE, 通流率 δ の関係式を求めて．

ChatGPT o1 の回答

以下に示すのは，不連続伝導（DCM）時の昇圧チョッパ出力電圧の代表的な導出例です．

まず，連続モード (CCM) では

$$v_O = \frac{v_E}{1-\delta} \quad (1)$$

のようになりますが，不連続モード (DCM) では，負荷抵抗 R，インダクタ L，スイッチング周波数 f_s などにも依存します．

以下，教科書などでよく示される形として，DCM での平均出力電圧の一例を示します．理想化や近似の程度によって異なる形になることもありますが，代表的には次のように書かれます．

$$v_O = \frac{v_E}{1-\delta}\left(1+\frac{\delta^2}{2\,(1-\delta)}\,\frac{R}{L\,f_s}\right). \quad (2)$$

あるいは電圧比として

$$\frac{v_O}{v_E} = \frac{1}{1-\delta}\left(1+\frac{\delta^2}{2\,(1-\delta)}\,\frac{R}{L\,f_s}\right). \quad (3)$$

この導出は，以下のような流れで得られます．

1. **オン期間** $T_{\text{on}} = \delta\,T_s$
   スイッチ (トランジスタ) がオンしている間，インダクタには
   $$v_L = v_E \quad (4)$$
   (理想スイッチ・ダイオード逆バイアス) が加わり，インダクタ電流 i_L が線形的に上昇します．オン終了時点のピーク電流は
   $$I_{\text{peak}} = \frac{v_E}{L}\,T_{\text{on}} = \frac{v_E\,\delta\,T_s}{L}. \quad (5)$$
2. **オフ期間中の導通区間** T'_{off}
   スイッチがオフになるとダイオードが導通し，インダクタ電圧は
   $$v_L = v_E - v_O. \quad (6)$$
   インダクタ電流はピーク I_{peak} から 0 に向かって減衰し，0 になるまでの時間を T'_{off} とすると
   $$T'_{\text{off}} = \frac{I_{\text{peak}}\,L}{v_O - v_E} = \frac{\frac{v_E\,\delta\,T_s}{L}\,L}{v_O - v_E} = \frac{v_E\,\delta\,T_s}{v_O - v_E}. \quad (7)$$
3. **負荷電流 (平均出力電流) と電荷量の一致**
   負荷が抵抗 R のみで理想とすると，
   $$I_O = \frac{v_O}{R}. \quad (8)$$

1 周期中にインダクタが送り出す電荷量 Q は三角波形の面積なので

$$Q = \frac{1}{2} I_{\mathrm{peak}} T'_{\mathrm{off}}. \tag{9}$$

これを周期 T_s で割ると平均出力電流が得られるため，

$$I_O = \frac{Q}{T_s} = \frac{1}{T_s}\frac{1}{2} I_{\mathrm{peak}} T'_{\mathrm{off}}. \tag{10}$$

これを $I_O = \frac{v_O}{R}$ に等置して，I_{peak}, T'_{off} を代入・整理すると

$$v_O = \frac{v_E}{1-\delta}\left(1 + \frac{\delta^2}{2\,(1-\delta)}\,\frac{R}{L\,f_s}\right). \tag{11}$$

要するに，不連続モードでは負荷抵抗やインダクタンス，スイッチング周波数が出力電圧に影響するため，連続モード (CCM) の単純な

$$v_O = \frac{v_E}{1-\delta} \tag{12}$$

に比べて，補正項が現れる，という点が大きな違いです．

以上が典型的な不連続モード (DCM) 下のブースト・コンバータの電圧式となります．

筆者の質問／依頼

最後の vO の式が間違っています．導出の過程で計算間違いをしているようです．

ChatGPT o1 の回答

以下では,「トランジスタ ON 時にインダクタ電流が 0 から立ち上がり，OFF 時に再び 0 に落ちる」不連続伝導モード (DCM) のブースト・コンバータを改めて厳密導出します．

(1) 動作の前提：
スイッチ (トランジスタ) がオンしているあいだ

$$v_L = v_E, \tag{13}$$

オフ時は

$$v_L = v_E - v_O. \tag{14}$$

電流が 0 から立ち上がり，オフ期間中に再度 0 に戻るのが DCM です．

(2) オン期間中のピーク電流：
オン期間 $\delta\,T_s$ 中でのインダクタ電流は線形上昇し，ピーク電流は

$$I_{\mathrm{peak}} = \frac{v_E}{L}\,\delta\,T_s. \tag{15}$$

(3) オフ期間中に電流が 0 になるまでの時間 T'_{off}：
オフ直後は

$$v_L \quad = \quad v_E - v_O. \tag{16}$$

よってピーク I_{peak} から 0 になるのに要する時間は

$$T'_{\mathrm{off}} \quad = \quad \frac{I_{\mathrm{peak}}\,L}{v_O - v_E} \quad = \quad \frac{v_E\,\delta\,T_s}{v_O - v_E}. \tag{17}$$

(4) 出力側へ移される電荷量と平均出力電流：
1 周期でインダクタが出力に送る電荷 Q は三角波面積より

$$Q \quad = \quad \frac{1}{2}\,I_{\mathrm{peak}}\,T'_{\mathrm{off}}. \tag{18}$$

それを周期 T_s で割ったものが平均出力電流 $I_O = \frac{v_O}{R}$ になるので，

$$\frac{v_O}{R} \quad = \quad \frac{Q}{T_s} = \frac{1}{T_s}\,\frac{1}{2}\,I_{\mathrm{peak}}\,T'_{\mathrm{off}}. \tag{19}$$

$I_{\mathrm{peak}}, T'_{\mathrm{off}}$ を代入すると

$$\frac{v_O}{R} \quad = \quad \frac{1}{T_s}\,\frac{1}{2}\left(\frac{v_E\,\delta\,T_s}{L}\right)\left(\frac{v_E\,\delta\,T_s}{v_O - v_E}\right) = \frac{v_E^2\,\delta^2\,T_s}{2\,L\,(v_O - v_E)}. \tag{20}$$

よって

$$v_O\,(v_O - v_E) \quad = \quad \frac{v_E^2\,\delta^2\,T_s\,R}{2\,L}, \tag{21}$$

あるいは $T_s = 1/f_s$ を代入して

$$v_O^2 - v_E\,v_O - \frac{v_E^2\,\delta^2\,R}{2\,L\,f_s} \quad = \quad 0. \tag{22}$$

ここから v_O は

$$v_O \quad = \quad \frac{v_E}{2}\left[1 + \sqrt{1 + 2\,\frac{\delta^2\,R}{L\,f_s}}\right]. \tag{23}$$

したがって，最終的には **2 次方程式**に基づくこの解が，不連続モード (DCM) での厳密な平均出力電圧式となります．

(5) まとめ：

- $v_O^2 \ - \ v_E\,v_O \ - \ \dfrac{v_E^2\,\delta^2\,R}{2\,L\,f_s} \ = \ 0$ が正しい形．
- 教科書等で $\frac{v_E}{1-\delta}(1+\dots)$ の形が載っている場合もありますが，導出途中の近似や定義の違いがあることが多い．

筆者の質問／依頼

教科書等によっては (1 − δ) が明示的に出る式が載っている場合があるとのことですが，載っている教科書を教えて

ChatGPT o1 の回答

(1) 例として挙げられる海外の教科書:

- N. Mohan, T. M. Undeland, W. P. Robbins, *Power Electronics: Converters, Applications, and Design*, John Wiley & Sons.
- R. W. Erickson, D. Maksimovic, *Fundamentals of Power Electronics*, Springer.
- M. H. Rashid, *Power Electronics: Circuits, Devices, and Applications*, Pearson.

これらにはブーストコンバータの連続 (CCM)/不連続 (DCM) の導出が掲載されていますが，導出過程の中間式や簡易近似として $(1-D)$（ここで $D=\delta$）が分母に現れる式が書かれている版があります．

(2) 日本語の文献例:

- 伊藤健一 他,『パワーエレクトロニクス』(森北出版)
- 国沢正典 他,『電力エレクトロニクス入門』(オーム社)
- 猪狩潔 他,『スイッチング電源の設計』 など

上記も同様に，CCM の単純な式 $\frac{v_E}{1-\delta}$ との対比で DCM を導入する際や，導出過程で $(1-\delta)$ が登場する書き方をしているものがあります．

(3) なぜ形が違うか:

1. デューティ比 δ の定義 (オン時間なのかオフ時間なのか) や記号 $D=1-\delta$ の使い方．
2. オン期間→オフ期間におけるインダクタ電圧極性・電流の切れ方をどう扱うか．
3. 完全な厳密導出 (三角波面積から 2 次方程式へ) と，途中式/簡易式の違い．

最終的には

$$v_O^2 - v_E\,v_O - \frac{v_E^2\,\delta^2\,R}{2\,L\,f_s} \quad = \quad 0 \tag{24}$$

の形に帰着するのが理想部品前提の厳密解ですが，教科書では CCM の式との対比を示すために $(1-\delta)$ を含む形式の途中式を載せていることがある，というわけです．

筆者の質問／依頼

連続モードの式と断続モードの式を用いて，横軸を $\delta : 0 \sim 0.95$，縦軸を v_O としてグラフを描いて．$v_E = 4.5\,\mathrm{V}$，$L = 4.7\,\mathrm{mH}$，$R = 2200\,\Omega$，$f_s = 10\,\mathrm{kHz}$ です．

ChatGPT o1 の回答

連続モード (CCM) の式は

$$v_{O,\mathrm{CCM}}(\delta) \quad = \quad \frac{v_E}{1-\delta}, \tag{25}$$

断続モード (DCM) の式（厳密解）は

$$v_{O,\mathrm{DCM}}(\delta) \quad = \quad \frac{v_E}{2}\Bigl[1+\sqrt{1+2\,\frac{\delta^2\,R}{L\,f_s}}\Bigr]. \tag{26}$$

これらを $v_E = 4.5\,\mathrm{V}$, $L = 4.7\,\mathrm{mH}$, $R = 2200\,\Omega$, $f_s = 10\,\mathrm{kHz}$ に代入し，$\delta = 0$ から 0.95 まで動かして数値を表にすると，...（以下略）

筆者の質問／依頼

グラフを描く python のコードを書いて．

ChatGPT o1 の回答

以下の Python コードで，同じパラメータを用いて CCM と DCM の出力電圧をプロットできます．

```
import numpy as np
import matplotlib.pyplot as plt

#--- パラメータ設定 ---
vE = 4.5        # [V]
L  = 4.7e-3     # [H]
R  = 2200       # [Ω]
f_s = 1.0e4     # [Hz]

delta_vals = np.linspace(0, 0.95, 200)

# CCM
vO_CCM = vE / (1 - delta_vals)

# DCM
vO_DCM = 0.5 * vE * (
    1.0 + np.sqrt(1.0 + 2.0 * (delta_vals**2) * R / (L * f_s))
```

```
)

plt.figure(figsize=(8,6))
plt.plot(delta_vals, vO_CCM, label='CCM', color='red')
plt.plot(delta_vals, vO_DCM, label='DCM', color='blue')
plt.xlabel('Duty Ratio δ')
plt.ylabel('Output Voltage vO [V]')
plt.title('Boost Converter: CCM vs. DCM')
plt.grid(True)
plt.legend()
plt.ylim([0, 100])
plt.xlim([0, 0.95])
plt.show()
```

実行すると，CCM と DCM それぞれの v_O が同一グラフ上に描かれ，δ を変化させたときの振る舞いを比較できます．

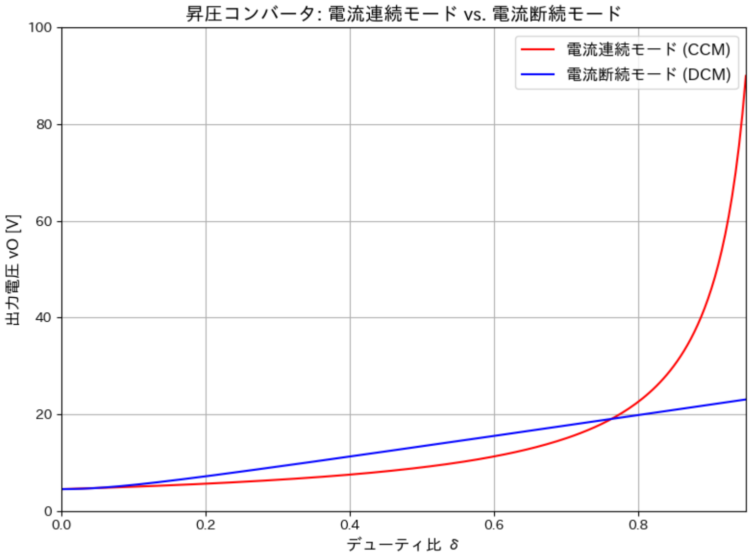


図 20: V_O の理論値（電流連続モードと電流断続モードの比較）