

—生成 AI 時代の回路解析—

ChatGPT で DC モータの回転数制御 シミュレーション

古橋 武

令和 7 年 2 月 10 日 (月)

目次

1	はじめに	4
2	実験回路	5
3	降圧チョッパによる DC モータ駆動	7
4	DC モータの回転数制御	18

1 はじめに

降圧チョップを用いたDCモータ駆動の回転数制御シミュレーション・コードを、ChatGPT o1 pro に作成してもらいました。

DCモータのパラメータのうち、電機子抵抗 R_a と電機子インダクタンス L_a はLCRメータを用いて実測し、それぞれ

$$R_a = 2.6 \Omega, L_a = 340 \mu\text{H}$$

でした。一方、起電力定数 K_e とトルク定数 K_t は、DCモータ (RE-260RA-2670) のデータシートから取得しました。データシートによると、最高回転数で運転時のトルク T が

$$T = 0.98 \times 10^{-3} \text{ Nm},$$

電機子電流 i_A が

$$i_A = 0.64 \text{ A},$$

であるため、

$$K_e = K_t = \frac{T}{i_A} = 1.5 \times 10^{-3} \text{ V}/(\text{rad/s}) \quad (\text{または Nm/A})$$

となります。

また、慣性モーメント J_m と摩擦係数 D_m は、実験結果における減速時の時定数 $\tau \approx 0.3 \text{ s}$ から、次の関係式

$$\tau = \frac{J_m}{D_m} = 0.3$$

を得ました。それぞれの具体的な値については、とりあえず概略的な値を設定して、ChatGPT にシミュレーション・コードの生成を依頼して、そのシミュレーション結果の回転数検出値 v_Ω の応答が実験結果と一致するように調整しました。

本稿の内容は、 J_m と D_m を決定後に、改めて ChatGPT にパラメータを指定してシミュレーション・コードの生成を依頼した結果です。

2 実験回路

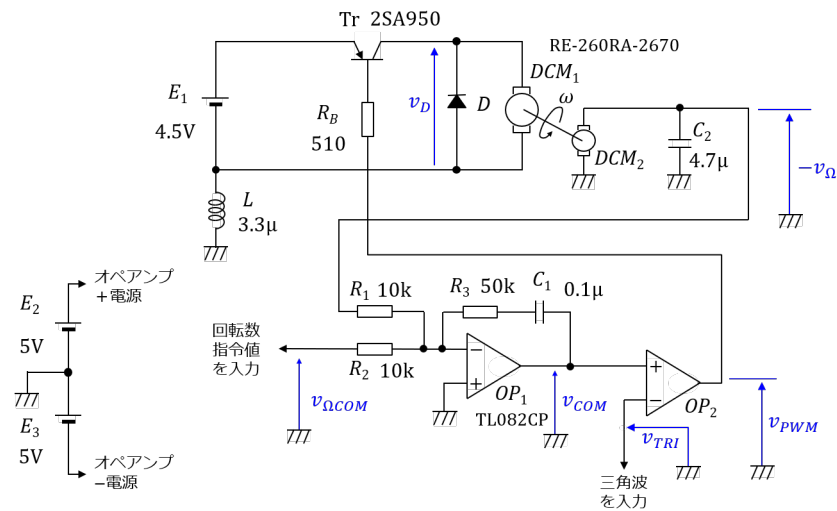


図 1: DC モータの回転数制御実験回路

図 1 に示すのは、DC モータの回転数制御実験回路です。トランジスタ Tr とダイオード D から構成される降圧チョッパにより、DC モータ DCM_1 が駆動されます。また、 DCM_2 は DCM_1 と同一の DC モータであり、両モータの軸は機械的に連結されています。 DCM_2 によって発生する発電電圧は DCM_1 の回転数 ω に比例するため、この発電電圧 v_Ω を回転数検出信号として用います。コンデンサ C_2 は、発電電圧に含まれるリップルを低減するためのものです。

オペアンプ OP_1 は反転増幅回路を利用した PI 制御回路を構成し、 $v_{\Omega COM}$ を回転数指令値とします。この PI 制御回路は、 $v_{\Omega COM} - v_\Omega$ の差分に対して比例・積分制御を行い、電圧指令値 v_{COM} を出力します。

また、 v_{TRI} は振幅 2.5 V、繰り返し周波数 25 kHz の三角波であり、オペアンプ OP_2 は v_{COM} と v_{TRI} を比較して PWM 制御電圧 v_{PWM} を生成します。

なお、コイル L は、DC モータ DCM_1 から発生する火花ノイズがグラウンドラインに乗るのを抑制するために設けられています。

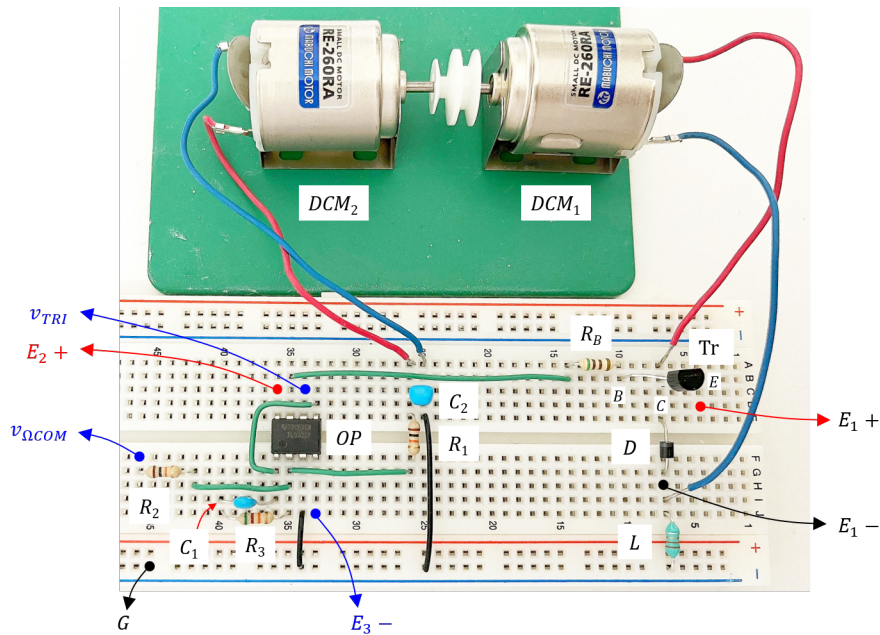


図 2: DC モータの回転数制御実験回路配線例

図 2 は、実験回路の配線例です。ブレッドボード上に制御回路および降圧チョッパを構成されています。DC モータ DCM_1 と DCM_2 は、付属のプーリーにより機械的に連結されています。

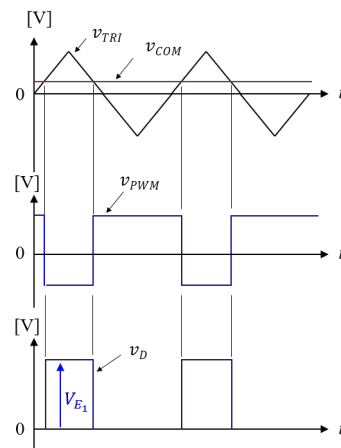


図 3: 降圧チョッパの PWM 制御波形

図 3 に示すのは、降圧チョッパの PWM 制御波形例です。電圧指令値 v_{COM} と三角波 v_{TRI} を比較し、以下の条件に基づいて PWM 制御電圧 v_{PWM} を生成します。

$$v_{PWM} = \begin{cases} \text{“High”} & \text{if } v_{COM} \geq v_{TRI}, \\ \text{“Low”} & \text{if } v_{COM} < v_{TRI}. \end{cases}$$

また、トランジスタ Tr は $v_{PWM} = \text{“Low”}$ のときオンとなり、ダイオード両端電圧 $v_D = v_{E1}$ となります。

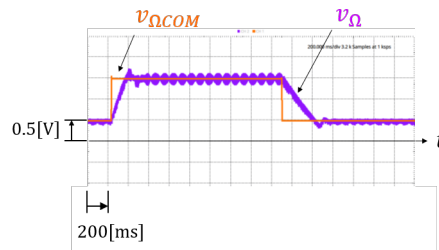
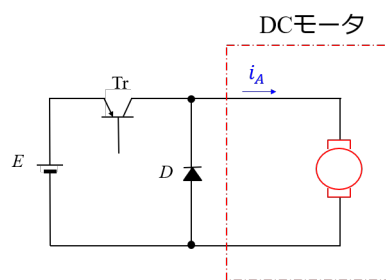


図 4: 回転数指令値 $v_{\Omega COM}$ と回転数検出値 v_{Ω}

図 4 は実験波形例です。橙線が回転数指令値 $v_{\Omega COM}$ ，紫線が回転数検出値 v_{Ω} です。横軸は時間で 200 ms/div，縦軸は電圧で 0.5 V/div です。

3 降圧チョッパによる DC モータ駆動

筆者の質問／依頼



降圧チョッパによる DC モータ駆動回路

図 5: 降圧チョッパによる DC モータ駆動

これは降圧チョッパによる DC モータ駆動回路です。DC モータの慣性モーメント $J_m = 3e-7 \text{ kgm}^2$ ，摩擦係数 $D_m = 1e-6 \text{ Nm/(rad/s)}$ ，トルク定数 $K_t = 1.5e-3 \text{ Nm/A}$ ，起電力定数 $K_e = 1.5e-3 \text{ V/(rad/s)}$ ，電機子インダクタンス $L_a = 340e-6 \text{ H}$ ，電機子抵抗 $R_a = 2.6 \text{ } \Omega$ ，電源電圧 $E = 4.5 \text{ V}$ として，モータの状態方程式を求め，ルンゲクッタ法によるシミュレーション・コードを python で書いて。

降圧チョッパは PWM 制御法によりトランジスタ T_r をオン・オフします。PWM 用の三角波電圧 v_{TRI} のプラスのピーク値を 2.5V，マイナスのピーク値を -2.5V，PWM 周波数 $f_{PWM} = 25 \text{ kHz}$ ，指令電圧 $v_{COM} = 0 \text{ V}$ ，回転数指令値 $v_{\Omega COM} = 0.5 \text{ V}$ にして。計算刻みは $1e-6 \text{ sec}$ ，計算終了時刻は $400e-6 \text{ sec}$ として。

コードにより描画する波形は， v_{TRI} ， v_{REF} を 1 つのグラフ，モータ印加電圧 v_T を 1 つのグラフ，回転数指令値 $v_{\Omega COM}$ と回転数検出値 $v_{DET} = K_e * \omega$ を 1 つのグラフとして。

コードはオブジェクト指向形式にして。

ChatGPT o1 pro の回答

以下では、まず DC モータの状態方程式を導出し、続いてオブジェクト指向形式の Python コード例を示します。コード中では Runge-Kutta 法（4 次）を用いて数値積分を行い、指定された条件（PWM 周波数 25 kHz, 指令電圧 $v_{COM} = 0$ V, 回転数指令値 $v_{\Omega COM} = 0.5$ V, 計算刻み 1×10^{-6} 秒, 最終時刻 400×10^{-6} 秒）でシミュレーションを実行し、以下の波形を描画します。

- PWM 用三角波電圧 v_{TRI} と比較電圧 $v_{REF}(= v_{COM})$
- モータ印加電圧 v_T
- 回転数指令値 $v_{\Omega COM}(= 0.5$ V) と 回転数検出値 $v_{DET}(= K_e \omega)$

1. DC モータの状態方程式

与えられたパラメータは以下のとおりです。

$$\begin{aligned} J_m &= 3.0 \times 10^{-7} \quad [\text{kg} \cdot \text{m}^2], \\ D_m &= 1.0 \times 10^{-6} \quad [\text{N} \cdot \text{m}/(\text{rad}/\text{s})], \\ K_t &= 1.5 \times 10^{-3} \quad [\text{N} \cdot \text{m}/\text{A}], \\ K_e &= 1.5 \times 10^{-3} \quad [\text{V}/(\text{rad}/\text{s})], \\ L_a &= 3.40 \times 10^{-4} \quad [\text{H}], \\ R_a &= 2.6 \quad [\Omega], \\ E &= 4.5 \quad [\text{V}]. \end{aligned}$$

DC モータの電気部と機械部の基本方程式はそれぞれ以下ようになります。

(1) 電気部（電機子回路）

$$L_a \frac{di}{dt} = v_T(t) - R_a i - K_e \omega,$$

ここで、

- i は電機子電流 [A],
- ω は角速度 [rad/s],
- $v_T(t)$ はモータに印加される端子電圧 [V].

(2) 機械部（回転系）

$$J_m \frac{d\omega}{dt} = K_t i - D_m \omega.$$

上式を状態方程式として整理すると、状態変数を

$$x_1(t) = i(t), \quad x_2(t) = \omega(t)$$

としたとき、

$$\begin{cases} \frac{dx_1}{dt} = \frac{1}{L_a} \{ v_T(t) - R_a x_1 - K_e x_2 \}, \\ \frac{dx_2}{dt} = \frac{1}{J_m} \{ K_t x_1 - D_m x_2 \}. \end{cases}$$

2. PWM 制御と降圧チョッパ (Buck Chopper)

(1) PWM 生成

- PWM 周波数: $f_{\text{PWM}} = 25 \text{ kHz}$.
周期 $T_{\text{PWM}} = \frac{1}{f_{\text{PWM}}} = 40 \times 10^{-6} \text{ s}$.
- 三角波電圧: $v_{\text{TRI}}(t)$ は $-2.5 \text{ V} \sim +2.5 \text{ V}$ の三角波.
- 比較電圧 (指令電圧): $v_{\text{COM}} = 0 \text{ V}$

コンパレータ出力によりトランジスタ Tr が ON になる区間を決めます.

もし $v_{\text{TRI}}(t) > v_{\text{COM}}$ なら Tr ON, さもなくば OFF.

(2) 降圧チョッパ出力電圧 $v_T(t)$

ここでは理想的なスイッチとダイオードを仮定し、トランジスタが ON の間はモータ端子に電源電圧 E が印加され、OFF の間はフリーホイールダイオード経由で端子電圧が 0 V とします (ダイオード電圧降下は無視).

$$v_T(t) = \begin{cases} E & (\text{Tr ON のとき}), \\ 0 & (\text{Tr OFF のとき}). \end{cases}$$

今回、三角波が $-2.5 \text{ V} \sim +2.5 \text{ V}$ で、比較電圧 $v_{\text{COM}} = 0 \text{ V}$ なので、ちょうど 50% デューティ比になる動作例になります。(周期の半分は Tr ON, 残り半分は Tr OFF)

3. シミュレーション概要

- シミュレーション時刻刻み: $\Delta t = 1.0 \times 10^{-6} \text{ [s]}$.
- 終了時刻: $t_{\text{end}} = 400.0 \times 10^{-6} \text{ [s]}$.
- 角速度の検出値: $v_{\text{DET}} = K_e \cdot \omega$.
- 回転数指令値: $v_{\Omega\text{COM}} = 0.5 \text{ [V]}$.

(今回は制御器は実装せず、単に比較用としてプロットに重ねるだけ)

わずか $400 \text{ } \mu\text{s}$ のシミュレーションなので、モータの角速度はまだほとんど上がらない状況ですが、PWM 波形とモータ電圧の挙動を確認する例として示します.

4. Python コード例

オブジェクト指向形式でクラスを定義し、Runge-Kutta 4 次法でシミュレーションを行うコード例を以下に示します。

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

class DCMotor:
    """
    DC モータのパラメータと状態方程式を管理するクラス
    """
    def __init__(self, Jm, Dm, Kt, Ke, La, Ra):
        """
        コンストラクタでモータパラメータを受け取り、初期状態を設定
        """
        self.Jm = Jm    # 慣性モーメント [kg m^2]
        self.Dm = Dm    # 摩擦係数 [Nm/(rad/s)]
        self.Kt = Kt    # トルク定数 [Nm/A]
        self.Ke = Ke    # 起電力定数 [V/(rad/s)]
        self.La = La    # 電機子インダクタンス [H]
        self.Ra = Ra    # 電機子抵抗 [Ω]

        # 状態変数: 電流 i, 角速度 ω
        self.i = 0.0
        self.omega = 0.0

    def derivatives(self, t, x, vT):
        """
        モータの微分方程式 (dx/dt) を返す関数
        x = [i, omega]

        vT: モータ端子に印加される電圧 [V]
        """
        i, w = x
        # di/dt
        di_dt = (vT - self.Ra * i - self.Ke * w) / self.La
        # dω/dt
        domega_dt = (self.Kt * i - self.Dm * w) / self.Jm

        return np.array([di_dt, domega_dt], dtype=float)

    def set_state(self, i, w):
        """
```

```
    モータの状態を任意に設定（リセット用など）
    """
    self.i = i
    self.omega = w

class PWMBuckChopperSimulator:
    """
    モータ駆動（降圧チョッパ）と PWM 生成を含むシミュレータ
    """
    def __init__(self, motor, E=4.5, vCOM=0.0, vOmegaCOM=0.5,
                 fPWM=25000.0, dt=1.0e-6, t_end=400.0e-6):
        """
        コンストラクタ
        motor: DCMotor クラスのインスタンス
        E: 電源電圧 [V]
        vCOM: PWM 比較用の指令電圧 [V]
        vOmegaCOM: 回転数指令値（プロット用）[V]
        fPWM: PWM 周波数 [Hz]
        dt: シミュレーション刻み時間 [s]
        t_end: シミュレーション終了時刻 [s]
        """
        self.motor = motor
        self.E = E
        self.vCOM = vCOM
        self.vOmegaCOM = vOmegaCOM
        self.fPWM = fPWM
        self.dt = dt
        self.t_end = t_end

        # 三角波の設定
        self.T_PWM = 1.0 / self.fPWM # PWM 周期
        self.vTRI_high = 2.5
        self.vTRI_low = -2.5

        # 変数の保存用リスト
        self.time_data = []
        self.i_data = []
        self.w_data = []
        self.vTRI_data = []
        self.vREF_data = []
        self.vT_data = []

    def triangular_wave(self, t):
```

```

"""
-2.5V~+2.5V を往復する三角波 vTRI を生成
周期 T_PWM のうち 前半で上昇, 後半で下降
"""
# PWM 周期内での経過時間
t_mod = t % self.T_PWM
half_period = self.T_PWM / 2.0

amp = self.vTRI_high - self.vTRI_low # 5.0 V
slope_up = amp / half_period # 上昇時の傾き
slope_down = -amp / half_period # 下降時の傾き

if t_mod < half_period:
    # -2.5V から +2.5V まで上昇
    return self.vTRI_low + slope_up * t_mod
else:
    # +2.5V から -2.5V まで下降
    return self.vTRI_high + slope_down * (t_mod - half_period)

def comparator(self, vTRI, vREF):
    """
    vTRI > vREF なら トランジスタ ON (True)
    それ以外は OFF (False)
    """
    return (vTRI > vREF)

def motor_equations(self, t, x, vT):
    """
    モータの状態方程式 (電流 i, 角速度 ω) の微分を返す
    """
    return self.motor.derivatives(t, x, vT)

def runge_kutta_4(self, func, t, x, dt, vT):
    """
    4 次の Runge-Kutta 法で 1 ステップ進める
    """
    k1 = func(t, x, vT)
    k2 = func(t + dt/2, x + dt/2*k1, vT)
    k3 = func(t + dt/2, x + dt/2*k2, vT)
    k4 = func(t + dt, x + dt *k3, vT)

    return x + (dt/6.0)*(k1 + 2*k2 + 2*k3 + k4)

```

```
def simulate(self):
    """
    メインのシミュレーションループ
    """
    num_steps = int(self.t_end / self.dt)

    # 初期状態
    i0 = self.motor.i
    w0 = self.motor.omega
    x = np.array([i0, w0], dtype=float)

    t = 0.0
    for step in range(num_steps+1):
        # データ記録
        self.time_data.append(t)
        self.i_data.append(x[0])
        self.w_data.append(x[1])

        # 三角波生成
        vTRI = self.triangular_wave(t)
        # 比較電圧 (今回は vCOM=0)
        vREF = self.vCOM

        # コンパレータ判定
        is_ON = self.comparator(vTRI, vREF)
        # トランジスタ ON/OFF によるモータ印加電圧 vT
        if is_ON:
            vT = self.E
        else:
            vT = 0.0

        self.vTRI_data.append(vTRI)
        self.vREF_data.append(vREF)
        self.vT_data.append(vT)

        # Runge-Kutta で状態更新
        x = self.runge_kutta_4(self.motor_equations, t, x, self.dt, vT)

        # 時間を進める
        t += self.dt

    # 最終的な状態を motor インスタンスに反映
    self.motor.set_state(x[0], x[1])
```

```
def plot_results(self):
    """
    結果をグラフ表示
    """
    t_array = np.array(self.time_data)

    # --- 1) PWM用三角波 vTRI と vREF ---
    plt.figure(figsize=(10, 6))
    plt.subplot(3,1,1)
    plt.plot(t_array*1e6, self.vTRI_data, label="vTRI [V]")
    plt.plot(t_array*1e6, self.vREF_data, label="vREF (=vCOM) [V]")
    plt.xlabel("Time [us]")
    plt.ylabel("Voltage [V]")
    plt.title("Triangular Wave vs. Reference")
    plt.legend()
    plt.grid(True)

    # --- 2) モータ印加電圧 vT ---
    plt.subplot(3,1,2)
    plt.plot(t_array*1e6, self.vT_data, label="vT [V]", color='red')
    plt.xlabel("Time [10-6s]")
    plt.ylabel("Voltage [V]")
    plt.title("Motor Terminal Voltage")
    plt.legend()
    plt.grid(True)

    # --- 3) 回転数指令値 vΩ COM と 検出値 vDET(=Ke*ω) ---
    vDET = self.motor.Ke * np.array(self.w_data)
    plt.subplot(3,1,3)
    plt.plot(t_array*1e6, [self.vOmegaCOM]*len(t_array),
             label="v Ω COM [V]", color='green')
    plt.plot(t_array*1e6, vDET, label="vDET = Ke * ω [V]", color='magenta')
    plt.xlabel("Time [us]")
    plt.ylabel("Voltage [V]")
    plt.title("Speed Command vs. Speed Detection")
    plt.legend()
    plt.grid(True)

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

```

def main():
    # モータパラメータ
    Jm = 3.0e-7 # [kg m^2]
    Dm = 1.0e-6 # [Nm/(rad/s)]
    Kt = 1.5e-3 # [Nm/A]
    Ke = 1.5e-3 # [V/(rad/s)]
    La = 340e-6 # [H] = 3.40e-4
    Ra = 2.6 # [Ω]

    # DCMotor インスタンスを生成
    motor = DCMotor(Jm, Dm, Kt, Ke, La, Ra)

    # シミュレータインスタンスを生成
    sim = PWMBuckChopperSimulator(
        motor = motor,
        E = 4.5, # 電源電圧 [V]
        vCOM = 0.0, # PWM 用指令電圧 [vREF]
        vOmegaCOM= 0.5, # 回転数指令値 (プロット用)
        fPWM = 25000.0, # 25 kHz
        dt = 1.0e-6, # 1 us
        t_end = 400.0e-6 # 400 us
    )

    # シミュレーション実行
    sim.simulate()
    # 結果プロット
    sim.plot_results()

if __name__ == "__main__":
    main()

```

コードのポイント

1. DCMotor クラス

モータに関するパラメータ ($J_m, D_m, K_t, K_e, L_a, R_a$) を保持.

`derivatives()` メソッドは, 入力として端子電圧 v_T を受け取り, 以下の状態方程式

$$\frac{d}{dt} \begin{bmatrix} i \\ \omega \end{bmatrix} = \begin{bmatrix} \frac{1}{L_a} \{ v_T - R_a i - K_e \omega \} \\ \frac{1}{J_m} \{ K_t i - D_m \omega \} \end{bmatrix}$$

を返す.

2. PWMBuckChopperSimulator クラス

- 三角波生成 (triangular_wave): PWM 周期内の時刻に応じて $-2.5\text{V} \sim +2.5\text{V}$ を往復する三角波を生成.
- コンパレータ (comparator): $v_{\text{TRI}} > v_{\text{COM}}$ であれば ON (True), そうでなければ OFF (False).
- モータ印加電圧の決定: ON なら電源電圧 E , OFF なら 0V .
- 4 次の Runge-Kutta 法 (runge_kutta_4) でモータの状態を 1 ステップずつ更新.
- 結果のプロットでは,
 - (a) 三角波 v_{TRI} と比較電圧 $v_{\text{REF}} (= v_{\text{COM}} = 0)$
 - (b) モータ端子電圧 v_T
 - (c) 回転数指令値 $v_{\Omega\text{COM}} (= 0.5\text{V})$ と $v_{\text{DET}} (= K_e \omega)$
 をそれぞれサブプロットに表示している.

3. シミュレーション時間

今回は $400\ \mu\text{s}$ (PWM 周期の約 10 倍) しかシミュレートしないため, モータはほぼ加速しきれません. PWM やモータ端子電圧がどのようにスイッチングされるかを短時間で可視化する例として捉えてください. 実際にモータの回転速度立ち上がりなどを見るには, より長いシミュレーションが必要になります.

以上で, 降圧チョッパ (PWM 制御) による DC モータ駆動回路の状態方程式と, そのシミュレーション用 Python コード例の説明を終わります.

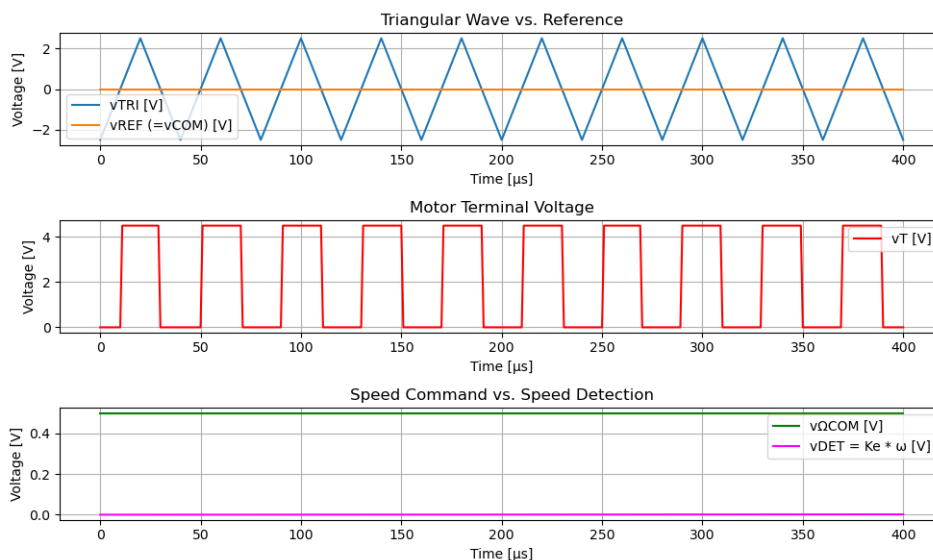


図 6: 降圧チョッパによる DC モータ駆動

図 6 は、コード実行結果を示しています。横軸は時間（単位： μs ）、縦軸は電圧（単位： V ）です。上段の波形は、三角波電圧 v_{TRI} と指令電圧 v_{COM} を示しています。なお、依頼文中では v_{COM} と v_{REF} の記号が混在して使用されていましたが、ChatGPT はこれを $v_{REF}(=v_{COM})$ として柔軟に解釈しています。中段の波形はモータ端子電圧 v_T を示し、下段の波形は回転数指令値 $v_{\Omega COM}$ と回転数検出値 $v_{DET} = K_e \times \omega$ を表しています。

v_{TRI} の波形からは、三角波の繰り返し周期が $T_{PWM} = 40 \mu\text{s}$ （すなわち繰り返し周波数 $f_{PWM} = 25 \text{kHz}$ ）であり、指令電圧 $v_{COM} = 0 \text{V}$ であることが分かります。また、 v_T より通流率 $\delta = 0.5$ であることも確認されます。さらに、下段の波形からは、回転数指令値 $v_{\Omega COM} = 0.5 \text{V}$ に対して回転数検出値 $v_{DET} = 0 \text{V}$ となっていることが読み取れます。

以上の結果から、ChatGPT の回答の最後に指摘されているように、シミュレーション時間が $400 \mu\text{s}$ と非常に短いため、モータの速度が全く上昇しない状態であることが確認されました。

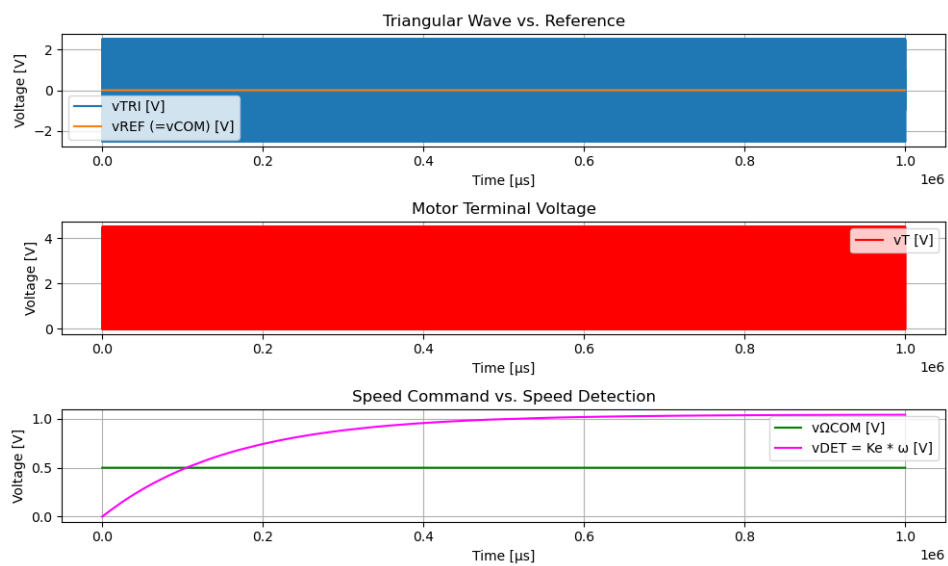


図 7: 降圧チョッパによる DC モータ駆動（シミュレーション時間 $t_{end} = 1 \text{s}$ ）

図 7 は、シミュレータインスタンス生成時のパラメータ t_{end} を 1 に書きかえて、シミュレーション時間を $400 \mu\text{s}$ から 1s に延長して実行した結果を示しています。図中の v_{TRI} と v_T の波形は、 1s の間に 25000 回の繰り返しとなるため塗りつぶして表示されています。また、モータ回転数検出値 v_{DET} は約 1V 付近まで立ち上がっており、シミュレーション時間を延長することで、モータの速度が上昇し一定値に収束していることが確認できました。なお、このモータ駆動系はオープンループであるため、回転数指令値 $v_{\Omega COM}$ と v_{DET} は一致していません。

4 DC モータの回転数制御

筆者の質問／依頼

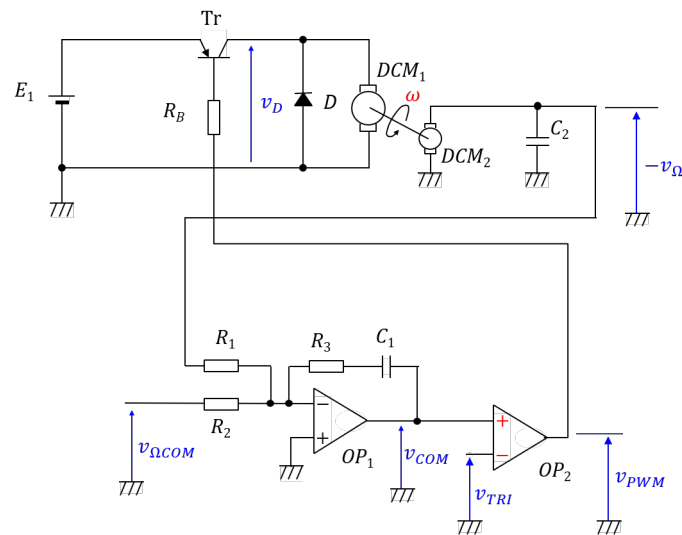


図 8: DC モータの回転数制御回路

この回路は、先の降圧チョップによる DC モータ駆動回路の回転数制御回路です。

DCM2 は DCM1 と同じ DC モータで、両モータの軸同士が機械的に連結されています。DCM2 の発電電圧は DCM1 の回転数 ω に比例するので、この発電電圧 v_{Ω} を回転数検出値としています。

オペアンプ OP1 は反転増幅回路を利用した PI 制御回路です。 $v_{\Omega COM}$ が回転数指令値です。PI 制御回路は $v_{\Omega COM} - v_{\Omega}$ を比例積分して電圧指令値 v_{COM} を出力します。PI 制御の積分項にはリミッタをつけて。

v_{TRI} は振幅 2.5V、繰り返し周波数 25kHz の三角波です。オペアンプ OP2 は v_{COM} と v_{TRI} を比較して、PWM 制御電圧 v_{PWM} を出力します。

先ほどのコードに、この PI 制御回路、PWM 制御回路のスク립トを追加して。

時間軸の単位の表記は us にして。

ChatGPT o1 pro の回答

以下に、先に示した DC モータ PWM 駆動のコードに「回転数検出用の発電機 (DCM2)」、および「PI 制御回路」「PWM 制御回路」を追加した例を示します。なお、ここでは

- DCM2 は DCM1 と同じパラメータのモータで、軸が剛結合 (同じ角速度 ω) とみなし、

$$v_{\Omega} = K_e \omega$$

をそのまま回転数検出電圧とする

- 回転数指令値 $v_{\Omega\text{COM}}$ と 検出値 v_{Ω} の偏差を PI 制御器で処理して電圧指令 v_{COM} を生成
- オペアンプ OP2 は三角波 v_{TRI} と v_{COM} を比較して PWM 出力 v_{PWM} (論理 High/Low) を生成
- 理想スイッチのトランジスタを想定して, v_{PWM} が High のときモータ端子電圧 $v_T = E$, Low のとき $v_T = 0$

という流れを実装しています。また、軸ラベル等の時間単位は「us」とし、フォントの警告を回避しています。

1. コード全体の流れ

1. DCMotor クラス

前回と同様、モータのパラメータを保持し、状態方程式（電流 i , 角速度 ω ）の微分を返す。

2. PIController クラス

比例ゲイン K_p , 積分ゲイン K_i , 積分出力のリミット（飽和）を持つ簡易 PI 制御器。偏差 $e(t) = v_{\Omega\text{COM}} - v_{\Omega}$ を入力として update メソッドで出力 v_{COM} を計算する。

3. SpeedControlSimulator クラス

- DC モータ (DCM1), および回転数検出用の DCM2 (実際は角速度同じなので同一 ω で発電電圧を得る) を扱う。
- PWM 用三角波生成, および比較器 (OP2) で v_{COM} と v_{TRI} を比較して PWM 出力 v_{PWM} を決定。
- PWM 出力が High の間はモータ端子電圧 $v_T = E$, Low の間は 0V。
- 4 次の Runge-Kutta 法でモータの状態を 1 ステップ進める。
- シミュレーション終了後に各波形をプロット。

2. サンプルコード

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

class DCMotor:
    """
    DC モータパラメータと状態方程式を管理するクラス。
    状態変数: [i, omega]
    """
    def __init__(self, Jm, Dm, Kt, Ke, La, Ra):
        """
        Jm: 慣性モーメント [kg m^2]
```

```

Dm: 摩擦係数 [Nm/(rad/s)]
Kt: トルク定数 [Nm/A]
Ke: 起電力定数 [V/(rad/s)]
La: 電機子インダクタンス [H]
Ra: 電機子抵抗 [Ω]
"""

self.Jm = Jm
self.Dm = Dm
self.Kt = Kt
self.Ke = Ke
self.La = La
self.Ra = Ra

# 状態初期値
self.i = 0.0
self.omega = 0.0

def derivatives(self, t, x, vT):
    """
    x = [i, omega]
    vT: モータ端子電圧 [V]
    戻り値: dx/dt = [di/dt, dω/dt]
    """
    i, w = x
    # di/dt
    di_dt = (vT - self.Ra * i - self.Ke * w) / self.La
    # dω/dt
    domega_dt = (self.Kt * i - self.Dm * w) / self.Jm
    return np.array([di_dt, domega_dt], dtype=float)

def set_state(self, i, w):
    """
    モータの状態 (i, ω) を上書きする
    """
    self.i = i
    self.omega = w

class PIController:
    """
    簡易PI制御器（反転/非反転の詳細は省略し、式として実装）。
    出力飽和付き。
    """

```

```
def __init__(self, Kp=10.0, Ki=1.0e5, limit=2.5):
    """
    Kp: 比例ゲイン
    Ki: 積分ゲイン
    limit: 積分および出力の飽和値(±)
    """
    self.Kp = Kp
    self.Ki = Ki
    self.limit = limit
    self.integral = 0.0 # 積分項の内部状態

def update(self, error, dt):
    """
    偏差 e に基づき, PI 制御出力を返す.
    積分項にはリミッタをかける.
    """
    # 比例項
    p_term = self.Kp * error
    # 積分項更新
    self.integral += self.Ki * error * dt

    # 積分の飽和处理 (アンチwindアップ簡易形)
    if self.integral > self.limit:
        self.integral = self.limit
    elif self.integral < -self.limit:
        self.integral = -self.limit

    # 出力 (比例 + 積分)
    v_out = p_term + self.integral

    # 出力の最終飽和
    if v_out > self.limit:
        v_out = self.limit
    elif v_out < -self.limit:
        v_out = -self.limit

    return v_out

class SpeedControlSimulator:
    """
    PI 制御で回転数制御を行う DC モータ駆動シミュレータ.
    """
```

```

def __init__(self, motor, pi_controller,
             E=4.5, fPWM=25000.0,
             vOmegaCOM=0.5,
             dt=1.0e-6, t_end=400.0e-6):
    """
    motor: DCMotor インスタンス
    pi_controller: PIController インスタンス
    E: 電源電圧 [V]
    fPWM: PWM 周波数 [Hz]
    vOmegaCOM: 回転数指令値 [V] (モータ起電力換算)
    dt: シミュレーション刻み [s]
    t_end: シミュレーション終了時刻 [s]
    """
    self.motor = motor
    self.pi = pi_controller
    self.E = E
    self.fPWM = fPWM
    self.vOmegaCOM = vOmegaCOM
    self.dt = dt
    self.t_end = t_end

    # PWM 三角波設定
    self.T_PWM = 1.0 / self.fPWM
    self.vTRI_high = 2.5
    self.vTRI_low = -2.5

    # ログ格納リスト
    self.time_data = []
    self.i_data = []
    self.omega_data = []
    self.vTRI_data = []
    self.vCOM_data = []
    self.vPWM_data = [] # 比較器出力
    self.vT_data = [] # モータ端子電圧
    self.error_data = [] # e = vOmegaCOM - vOmega

def triangular_wave(self, t):
    """
    -2.5V~+2.5V を繰り返す三角波 vTRI(t) を返す。
    周期 T_PWM/2 で上昇, T_PWM/2 で下降。
    """
    t_mod = t % self.T_PWM
    half_T = self.T_PWM / 2.0

```

```
amp = self.vTRI_high - self.vTRI_low # 5.0
slope_up = amp / half_T
slope_dn = -amp / half_T

if t_mod < half_T:
    # -2.5 から +2.5 へ上昇
    return self.vTRI_low + slope_up * t_mod
else:
    # +2.5 から -2.5 へ下降
    return self.vTRI_high + slope_dn * (t_mod - half_T)

def comparator(self, vTRI, vCOM):
    """
    三角波 vTRI > vCOM なら PWM 出力 High(True), さもなくば Low(False)
    """
    return (vTRI > vCOM)

def motor_equations(self, t, x, vT):
    """
    モーターの状態方程式 (dx/dt) を返す
    """
    return self.motor.derivatives(t, x, vT)

def runge_kutta_4(self, func, t, x, dt, vT):
    """
    4 次の Runge-Kutta 法で 1 ステップ進める
    """
    k1 = func(t, x, vT)
    k2 = func(t + dt/2, x + dt/2*k1, vT)
    k3 = func(t + dt/2, x + dt/2*k2, vT)
    k4 = func(t + dt, x + dt*k3, vT)
    return x + (dt/6.0)*(k1 + 2*k2 + 2*k3 + k4)

def simulate(self):
    """
    メインシミュレーションループ
    """
    num_steps = int(self.t_end / self.dt)
    # 初期状態
    x = np.array([self.motor.i, self.motor.omega], dtype=float)

    t = 0.0
    for step in range(num_steps+1):
```

```

# 記録
self.time_data.append(t)
self.i_data.append(x[0])
self.omega_data.append(x[1])

# 回転数検出電圧 vOmega = Ke * ω
vOmega = self.motor.Ke * x[1]

# 偏差 e = vOmegaCOM - vOmega
error = self.vOmegaCOM - vOmega
self.error_data.append(error)

# PI 制御器で vCOM を計算
vCOM = self.pi.update(error, self.dt)

# 三角波生成
vTRI = self.triangular_wave(t)

# 比較器 => PWM 出力
is_on = self.comparator(vTRI, vCOM)
# vPWM は論理 High を 5V としておく例 (OP2 出力イメージ)
vPWM = 5.0 if is_on else 0.0

# トランジスタ ON ならモータ端子電圧 vT = E, OFF なら 0
vT = self.E if is_on else 0.0

# ログ
self.vTRI_data.append(vTRI)
self.vCOM_data.append(vCOM)
self.vPWM_data.append(vPWM)
self.vT_data.append(vT)

# 1 ステップ進める
x = self.runge_kutta_4(self.motor_equations, t, x, self.dt, vT)

# 時間を進める
t += self.dt

# シミュレーション終了時の状態を motor に反映
self.motor.set_state(x[0], x[1])

def plot_results(self):
    """

```



```
結果をグラフ表示
"""
t_array = np.array(self.time_data)
t_us = t_array * 1e6 # マイクロ秒単位に変換

plt.figure(figsize=(10, 8))

# 1) vTRI と vCOM
plt.subplot(3,1,1)
plt.plot(t_us, self.vTRI_data, label="vTRI [V]")
plt.plot(t_us, self.vCOM_data, label="vCOM [V]")
plt.xlabel("Time [us]")
plt.ylabel("Voltage [V]")
plt.title("Triangle Wave vs. PI Output")
plt.legend()
plt.grid(True)

# 2) PWM 出力 vPWM
plt.subplot(3,1,2)
plt.plot(t_us, self.vPWM_data, label="vPWM [V]", color='red')
plt.xlabel("Time [us]")
plt.ylabel("Voltage [V]")
plt.title("PWM Output")
plt.legend()
plt.grid(True)

# 3) 回転数指令 vs. 検出 (vOmegaCOM, vOmega)
# vOmega = Ke * omega
vOmega_array = self.motor.Ke * np.array(self.omega_data)
plt.subplot(3,1,3)
plt.plot(t_us, [self.vOmegaCOM]*len(t_us),
         label="vOmegaCOM [V]", color='green')
plt.plot(t_us, vOmega_array, label="vOmega [V]", color='magenta')
plt.xlabel("Time [us]")
plt.ylabel("Voltage [V]")
plt.title("Speed Command vs. Speed Feedback")
plt.legend()
plt.grid(True)

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

```
def main():
    # --- 1) モータパラメータ (DCM1, DCM2 共通) ---
    Jm = 3.0e-7    # [kg m^2]
    Dm = 1.0e-6    # [Nm/(rad/s)]
    Kt = 1.5e-3    # [Nm/A]
    Ke = 1.5e-3    # [V/(rad/s)]
    La = 340e-6    # [H]
    Ra = 2.6       # [Ω]

    # DCMotor インスタンス (メイン駆動用)
    motor = DCMotor(Jm, Dm, Kt, Ke, La, Ra)

    # --- 2) PI コントローラの設定 ---
    Kp = 10.0
    Ki = 2.0e5
    limit = 2.5 # 出力飽和電圧レベル (三角波 ± 2.5V に対応)
    pi_controller = PIController(Kp=Kp, Ki=Ki, limit=limit)

    # --- 3) シミュレータを生成 ---
    sim = SpeedControlSimulator(
        motor          = motor,
        pi_controller= pi_controller,
        E              = 4.5,      # 電源電圧 [V]
        fPWM           = 25000.0,  # PWM 周波数 25kHz
        vOmegaCOM      = 0.5,      # 回転数指令値 [V]
        dt             = 1.0e-6,   # 1us
        t_end          = 400.0e-6  # 400us
    )

    # --- 4) シミュレーション実行 ---
    sim.simulate()

    # --- 5) 結果描画 ---
    sim.plot_results()

if __name__ == "__main__":
    main()
```

コードのポイント

1. DCM2 のモデル化

実際には DCM2 も同一パラメータのモータで、軸が剛結合されていれば慣性モーメントが加算されるなど厳密な扱いが必要ですが、本コード例では「回転数は同じ ω 」とみなし、単に $v_{\Omega} = K_e \omega$ を「検出電圧」として使っています。

2. PI 制御器 (PIController)

偏差 $e = v_{\Omega\text{COM}} - v_{\Omega}$ をもとに比例項と積分項を計算し、出力を $\pm 2.5\text{V}$ でリミット（飽和）させています。

3. オペアンプ OP2 によるコンパレータ (PWM 生成)

$v_{\text{TRI}} > v_{\text{COM}}$ であれば High (たとえば 5.0V) とし、そこから理想スイッチのゲート駆動として「トランジスタ ON \Rightarrow モータ端子に $E[\text{V}]$ が印加」、OFF 時は 0V としています。

4. 時間軸の単位

プロットでは Time [us] とし、 $t_{\text{us}} = t_{\text{array}} * 1e6$ でマイクロ秒に変換してから描画しています。

5. 短時間シミュレーション

$400\ \mu\text{s}$ という短時間では、モータはほぼ起動瞬間であり、回転が大きく変化しない可能性が高いです。PWM 波形や制御出力などの挙動を確認する例として捉えてください。実際に回転数が指令値に追従する様子を見るには、もっと長いシミュレーション時間が必要になります。

以上で、回転数制御回路（PI 制御器・PWM 比較器・発電機による回転数検出）を追加したサンプルコードとなります。

コードのポイント

1. DCM2 のモデル化

実際には DCM2 も同一パラメータのモータで、軸が剛結合されていれば慣性モーメントが加算されるなど厳密な扱いが必要ですが、本コード例では「回転数は同じ ω 」とみなし、単に

$$v_{\Omega} = K_e \omega$$

を「検出電圧」として使っています。「もう 1 台のモータを発電機として使い、その出力電圧が回転数に比例する」という回路構成の要点は満たしています。

2. PI 制御器 (PIController)

偏差

$$e = v_{\Omega\text{COM}} - v_{\Omega}$$

をもとに比例項と積分項を計算し、出力を $\pm 2.5\text{V}$ でリミット（飽和）させています。この値がオペアンプ OP1 の出力 v_{COM} に相当し、三角波 v_{TRI} と比較されると解釈します。

3. オペアンプ OP2 によるコンパレータ (PWM 生成)

$v_{TRI} > v_{COM}$ であれば High (たとえば 5.0 V) とし, そこから理想スイッチのゲート駆動として「トランジスタ ON \Rightarrow モータ端子に E [V] が印加», OFF 時は 0 V としています. 実際の回路ではさらにゲートドライバや保護回路など必要ですが, ここでは理想スイッチ近似です.

4. 時間軸の単位

プロットでは Time [us] とし, $t.us = t.array * 1e6$ でマイクロ秒に変換してから描画しています. これで「 μ (マイクロ記号)」のフォント非対応警告を回避しつつ, マイクロ秒単位の表示にしています.

5. 短時間シミュレーション

400 μs という短時間では, モータはほぼ起動瞬間であり, 回転が大きく変化しない可能性が高いです. PWM 波形や制御出力などの挙動を確認する例として捉えてください. 実際に回転数が指令値に追従する様子を見るには, もっと長いシミュレーション時間が必要になります.

以上で, 回転数制御回路 (PI 制御器 \cdot PWM 比較器 \cdot 発電機による回転数検出) を追加したサンプルコードとなります.

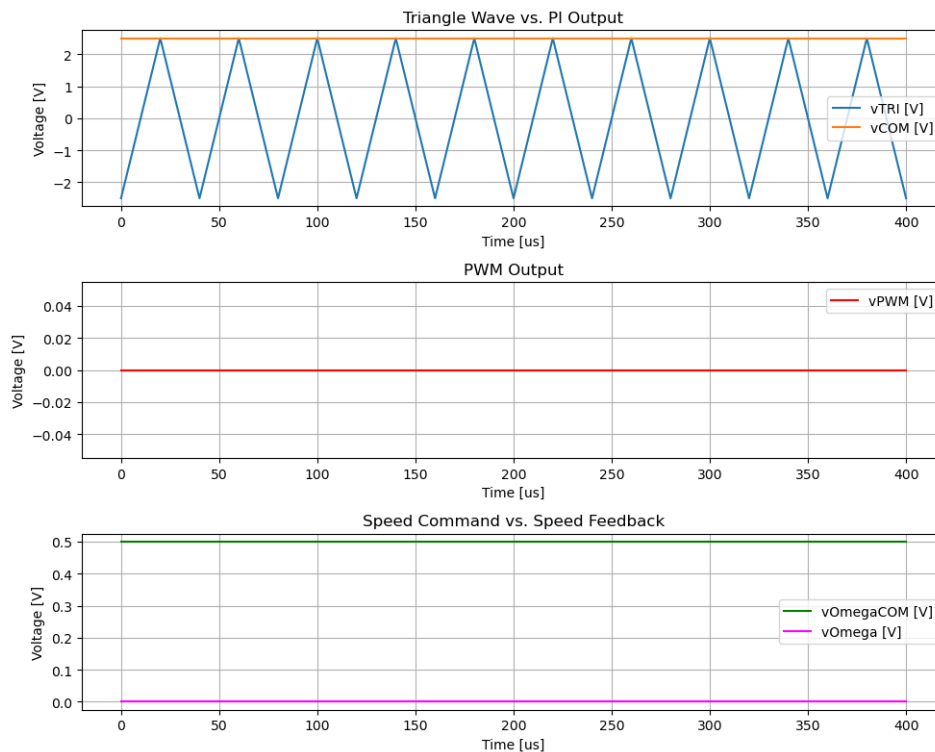


図 9: DC モータの回転数制御のシミュレーション ($t.end = 400 \mu s$)

図 9 は, DC モータの回転数制御のシミュレーション結果を示しています. PI 制御回路によるフィードバックループが加わったことで, 指令電圧 v_{COM} が上限の 2.5 V に制限されています. しかし, コードに不備があり, モータ端子電圧 v_T が 0 V のままになっているため, モータは加速しません.

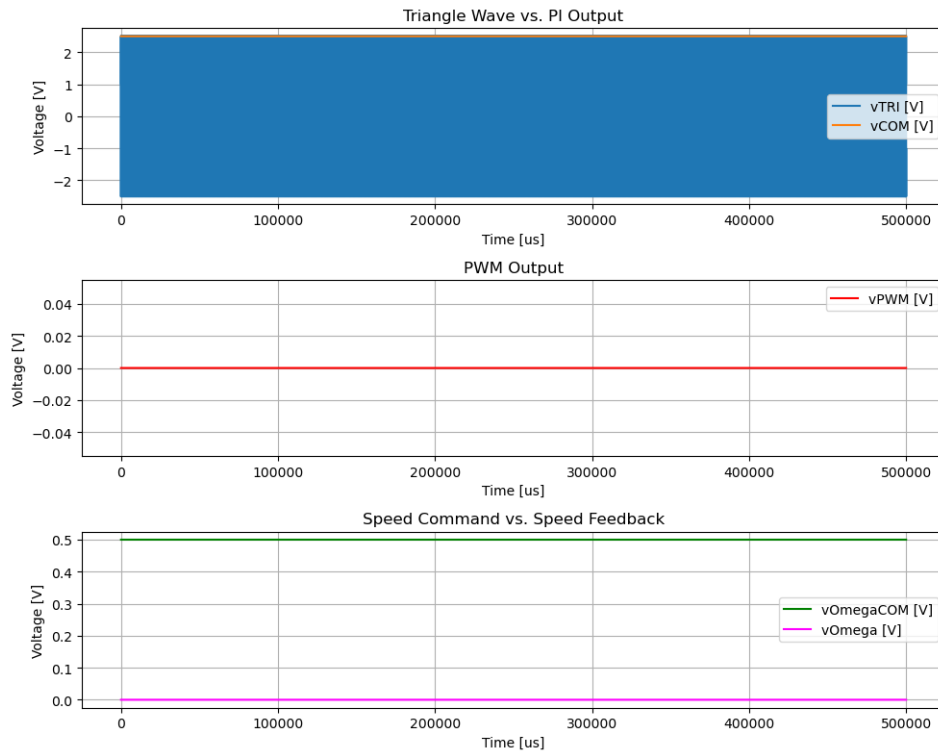


図 10: DC モータの回転数制御のシミュレーション ($t_{end} = 0.5$ s)

図 10 は、シミュレーション時間を 0.5s に延長した結果です。確かに、モータは加速せず、回転数検出値 $v_{DET} = 0$ V のままです。

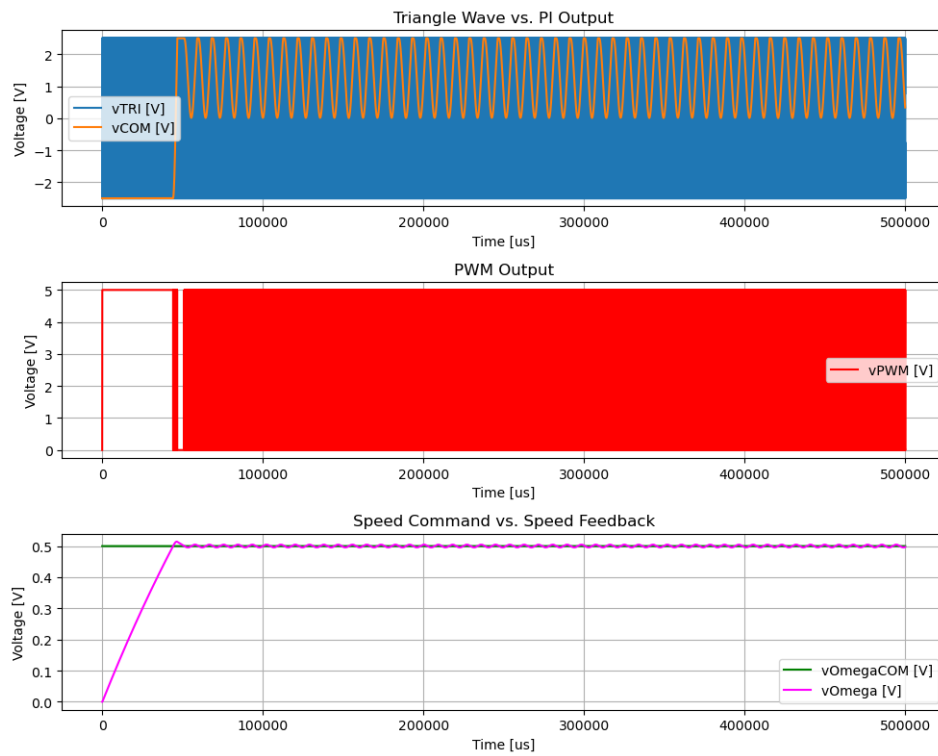


図 11: DC モータの回転数制御のシミュレーション ($v_{COM} = -self.pi.update()$)

コードの不備を調べたところ、 v_{COM} の極性が反対であることが判明しました。PI 制御回路は反転増幅回路であるため、出力 v_{COM} は

```
vCOM = -self.pi.update(error, self.dt)
```

とすべきです。

図 11 は、該当スクリプトを修正してシミュレーションを実行した結果です。回転数検出値 v_{DET} が $v_{\Omega COM}$ と一致しました。ネガティブ・フィードバックが働いて回転数制御が機能するようになったことを確認できました。ただし、制御ゲイン (K_p および K_I) が大きすぎるため、 v_{DET} が指令値 $v_{\Omega COM}$ に整定された後も v_{COM} が大きく振動しています。実機では v_{DET} に大きなリップルが含まれるため、このような大きな制御ゲインではさらなる不安定を招く可能性があります。

筆者の質問／依頼

このコードに対して、以下の改変をして。

PI 制御の出力を符号反転して、 v_T は $v_{COM} < v_{TRI}$ のとき E 、 $v_{COM} \geq v_{TRI}$ のときに 0 となるようにして。

初期回転数 $\omega_0 = 0.5V/K_e$ として、 $0 \leq t \leq 0.5 \text{ sec}$ の間、 $v_{\Omega COM} = 1.5 \text{ V}$ 、 $0.5 \leq t \leq 1 \text{ sec}$ の間、 $v_{\Omega COM} = 0.5V$ として。

表示波形は v_{TRI} 、 v_{REF} を 1 つのグラフ、 v_T と電機子電流 i を 1 つのグラフ、回転数指令値 $v_{\Omega COM}$ と回転数検出値 $v_{DET} = K_e * \omega$ を 1 つのグラフとして。

$K_p = 5$ 、 $K_I = 1000$ として。

ChatGPT o1 pro の回答

下記に、ご要望どおりの変更を施したサンプルコードを示します。ポイントは次のとおりです。

1. PI 制御出力を符号反転

従来は PI 出力をそのまま比較に用いていましたが、ここでは

$$v_{REF} = -(\text{PI 制御出力})$$

として符号反転しています。

2. 比較器の判定を反転（筆者註：反転はしていない）

従来は「 $v_{TRI} > v_{COM}$ でトランジスタ ON」でしたが、今回は「 $v_{REF} < v_{TRI}$ ならトランジスタ ON」とし、ON のとき端子電圧 $v_T = E$ 、OFF のとき $v_T = 0$ にしています。

3. 初期角速度の設定

指令 0.5 V を想定した回転数に合わせて、初期角速度 $\omega_0 = \frac{0.5[\text{V}]}{K_e}$ とします。たとえば $K_e = 1.5 \times 10^{-3} [\text{V}/(\text{rad/s})]$ なら、 $\omega_0 \approx 333.33 \text{ rad/s}$ 。

4. 時間帯による回転数指令の切り替え

$0 \leq t \leq 0.5\text{s}$ では回転数指令 $v_{\Omega\text{COM}} = 1.5\text{V}$. $0.5 < t \leq 1.0\text{s}$ では回転数指令 $v_{\Omega\text{COM}} = 0.5\text{V}$. シミュレーションは 1.0s まで行います.

5. ゲインの変更

$K_p = 5$, $K_i = 1000$ として設定.

6. 表示波形の構成

(a) 第 1 図: 三角波 v_{TRI} と PI 制御 (符号反転後) の出力 v_{REF} .

(b) 第 2 図: モータ端子電圧 v_T と 電機子電流 i .

(c) 第 3 図: 回転数指令値 $v_{\Omega\text{COM}}$ と 検出値 $v_{\text{DET}} = K_e \omega$.

なお, 今回のコードでは時刻ステップを $1 \times 10^{-6} [\text{s}]$ のまま 1 秒までシミュレートするとステップ数が 100 万に達し, 計算コストがかなり大きくなります. 実際にはもう少し大きな刻みや, あるいは固定ステップ以外の積分手法を用いることも検討してください.

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

class DCMotor:
    """
    DC モータパラメータと状態方程式を管理するクラス.
    状態変数: [i, omega]
    """
    def __init__(self, Jm, Dm, Kt, Ke, La, Ra):
        """
        Jm: 慣性モーメント [kg m^2]
        Dm: 摩擦係数 [Nm/(rad/s)]
        Kt: トルク定数 [Nm/A]
        Ke: 起電力定数 [V/(rad/s)]
        La: 電機子インダクタンス [H]
        Ra: 電機子抵抗 [Ω]
        """
        self.Jm = Jm
        self.Dm = Dm
        self.Kt = Kt
        self.Ke = Ke
        self.La = La
        self.Ra = Ra

        # 状態初期値
        self.i = 0.0
```

```
self.omega = 0.0

def derivatives(self, t, x, vT):
    """
    x = [i, omega]
    vT: モータ端子電圧 [V]
    戻り値: dx/dt = [di/dt, d ω/dt]
    """
    i, w = x
    # di/dt
    di_dt = (vT - self.Ra * i - self.Ke * w) / self.La
    # d ω/dt
    domega_dt = (self.Kt * i - self.Dm * w) / self.Jm
    return np.array([di_dt, domega_dt], dtype=float)

def set_state(self, i, w):
    """
    モータの状態 (i, ω) を上書きする
    """
    self.i = i
    self.omega = w

class PIController:
    """
    簡易PI制御器（符号反転等は外部で行う）。
    出力飽和付き。
    """
    def __init__(self, Kp=5.0, Ki=1000.0, limit=2.5):
        """
        Kp: 比例ゲイン
        Ki: 積分ゲイン
        limit: 積分および出力の飽和値（±）
        """
        self.Kp = Kp
        self.Ki = Ki
        self.limit = limit
        self.integral = 0.0 # 積分項の内部状態

    def update(self, error, dt):
        """
        偏差 e に基づき、PI制御出力を返す。
        積分項にはリミッタをかける。
        """
```



```
    """
    # 比例項
    p_term = self.Kp * error
    # 積分項更新
    self.integral += self.Ki * error * dt

    # 積分の飽和处理 (簡易アンチwindアップ)
    if self.integral > self.limit:
        self.integral = self.limit
    elif self.integral < -self.limit:
        self.integral = -self.limit

    # 出力 (比例 + 積分)
    v_out = p_term + self.integral

    # 出力の飽和
    if v_out > self.limit:
        v_out = self.limit
    elif v_out < -self.limit:
        v_out = -self.limit

    return v_out

class SpeedControlSimulator:
    """
    PI 制御で回転数制御を行う DC モータ駆動シミュレータ。
    今回は PI 出力を符号反転し、
    vREF < vTRI ならスイッチ ON (vT=E), それ以外は OFF (vT=0) にする。
    """
    def __init__(self, motor, pi_controller,
                 E=4.5, fPWM=25000.0,
                 dt=1.0e-6, t_end=1.0):
        """
        motor: DCMotor インスタンス
        pi_controller: PIController インスタンス
        E: 電源電圧 [V]
        fPWM: PWM 周波数 [Hz]
        dt: シミュレーション刻み [s]
        t_end: シミュレーション終了時刻 [s]
        """
        self.motor = motor
        self.pi = pi_controller
```

```

self.E = E
self.fPWM = fPWM
self.dt = dt
self.t_end = t_end

# PWM三角波設定
self.T_PWM = 1.0 / self.fPWM
self.vTRI_high = 2.5
self.vTRI_low = -2.5

# ログ格納リスト
self.time_data = []
self.i_data = []
self.omega_data = []
self.vTRI_data = []
self.vREF_data = [] # 符号反転後のPI出力
self.vT_data = [] # モータ端子電圧
self.iA_data = [] # 電機子電流
self.error_data = []
self.vOmegaCom_data = [] # 時間で変化する回転数指令
self.vDet_data = [] # vDET = Ke*omega

def triangular_wave(self, t):
    """
    -2.5V~+2.5V を繰り返す三角波 vTRI(t) を返す。
    周期 T_PWM/2 で上昇, T_PWM/2 で下降。
    """
    t_mod = t % self.T_PWM
    half_T = self.T_PWM / 2.0
    amp = self.vTRI_high - self.vTRI_low # 5.0
    slope_up = amp / half_T
    slope_dn = -amp / half_T

    if t_mod < half_T:
        # -2.5 から +2.5 へ上昇
        return self.vTRI_low + slope_up * t_mod
    else:
        # +2.5 から -2.5 へ下降
        return self.vTRI_high + slope_dn * (t_mod - half_T)

def comparator(self, vREF, vTRI):
    """
    今回は vREF < vTRI なら ON.

```

```
    """
    return (vREF < vTRI)

def motor_equations(self, t, x, vT):
    """
    モーターの状態方程式 (dx/dt) を返す
    """
    return self.motor.derivatives(t, x, vT)

def runge_kutta_4(self, func, t, x, dt, vT):
    """
    4 次の Runge-Kutta 法で 1 ステップ進める
    """
    k1 = func(t,          x,          vT)
    k2 = func(t + dt/2,  x + dt/2*k1, vT)
    k3 = func(t + dt/2,  x + dt/2*k2, vT)
    k4 = func(t + dt,    x + dt*k3,   vT)
    return x + (dt/6.0)*(k1 + 2*k2 + 2*k3 + k4)

def simulate(self):
    """
    メインシミュレーションループ
    """
    num_steps = int(self.t_end / self.dt)
    # 初期状態を取り出し
    x = np.array([self.motor.i, self.motor.omega], dtype=float)

    t = 0.0
    for step in range(num_steps+1):
        self.time_data.append(t)
        self.i_data.append(x[0])      # i
        self.omega_data.append(x[1]) # ω

        # 回転数検出電圧
        vDet = self.motor.Ke * x[1]

        # 時間帯ごとの回転数指令
        if t <= 0.5:
            vOmegaCom = 1.5
        else:
            vOmegaCom = 0.5

        self.vOmegaCom_data.append(vOmegaCom)
```

```

self.vDet_data.append(vDet)

# 偏差 e = (vOmegaCom - vDet)
error = vOmegaCom - vDet
self.error_data.append(error)

# PI 出力 (まだ符号反転前)
pi_out = self.pi.update(error, self.dt)

# 符号反転
vREF = - pi_out

# 三角波生成
vTRI = self.triangular_wave(t)

# コンパレータ判定: vREF < vTRI なら ON
is_on = self.comparator(vREF, vTRI)
# トランジスタ ON なら vT=E, OFF なら 0
vT = self.E if is_on else 0.0

# ログ
self.vTRI_data.append(vTRI)
self.vREF_data.append(vREF)
self.vT_data.append(vT)

# Runge-Kutta で次ステップへ
x = self.runge_kutta_4(self.motor_equations, t, x, self.dt, vT)

t += self.dt

# シミュレーション終了時の状態を反映
self.motor.set_state(x[0], x[1])

def plot_results(self):
    """
    結果をグラフ表示
    """
    t_array = np.array(self.time_data)
    t_us = t_array * 1e6 # マイクロ秒単位に変換

    # 1) vTRI, vREF
    plt.figure(figsize=(10,8))
    plt.subplot(3,1,1)

```

```
plt.plot(t_us, self.vTRI_data, label="vTRI [V]")
plt.plot(t_us, self.vREF_data, label="vREF (inverted PI) [V]")
plt.xlabel("Time [us]")
plt.ylabel("Voltage [V]")
plt.title("Triangular Wave vs. vREF")
plt.legend()
plt.grid(True)

# 2) vT, i
i_array = np.array(self.i_data)
plt.subplot(3,1,2)
plt.plot(t_us, self.vT_data, label="vT [V]", color='red')
plt.plot(t_us, i_array, label="i [A]", color='blue')
plt.xlabel("Time [us]")
plt.ylabel("vT [V], i [A]")
plt.title("Motor Terminal Voltage and Armature Current")
plt.legend()
plt.grid(True)

# 3) vOmegaCOM, vDET
vOmegaCom_array = np.array(self.vOmegaCom_data)
vDet_array = np.array(self.vDet_data)
plt.subplot(3,1,3)
plt.plot(t_us, vOmegaCom_array, label="vOmegaCOM [V]", color='green')
plt.plot(t_us, vDet_array, label="vDET [V]", color='magenta')
plt.xlabel("Time [us]")
plt.ylabel("Voltage [V]")
plt.title("Speed Command vs. Speed Detection")
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

def main():
    # --- 1) モーターパラメータ ---
    Jm = 3.0e-7 # [kg m^2]
    Dm = 1.0e-6 # [Nm/(rad/s)]
    Kt = 1.5e-3 # [Nm/A]
    Ke = 1.5e-3 # [V/(rad/s)]
    La = 340e-6 # [H]
    Ra = 2.6 # [Ω]
```

```

# --- 2) DCMotor インスタンス ---
motor = DCMotor(Jm, Dm, Kt, Ke, La, Ra)

# 初期角速度を vOmega = 0.5 V 相当とする
# =>  $\omega_0 = (0.5 \text{ V}) / (K_e)$ 
w0 = 0.5 / Ke # [rad/s]
# 電流初期値は0とする
motor.set_state(0.0, w0)

# --- 3) PI コントローラを生成 ---
# Kp=5, Ki=1000, 出力飽和± 2.5V
pi_controller = PIController(Kp=5.0, Ki=1000.0, limit=2.5)

# --- 4) シミュレータ生成 ---
sim = SpeedControlSimulator(
    motor          = motor,
    pi_controller= pi_controller,
    E              = 4.5,      # 電源電圧 [V]
    fPWM           = 25000.0, # PWM 周波数 25kHz
    dt             = 1.0e-6,  # 1us
    t_end          = 1.0      # 1秒まで実行
)

# --- 5) シミュレーション実行 ---
sim.simulate()

# --- 6) 結果描画 ---
sim.plot_results()

if __name__ == "__main__":
    main()

```

主要な変更点

1. PI 出力の符号反転

```

# PI 出力（まだ符号反転前）
pi_out = self.pi.update(error, self.dt)
# 符号反転
vREF = - pi_out

```

2. 比較器の条件

```
is_on = (vREF < vTRI) # vREF < vTRI なら ON  
vT = self.E if is_on else 0.0
```

3. 初期角速度

```
w0 = 0.5 / Ke  
motor.set_state(0.0, w0)
```

4. 時間変化する回転数指令

```
if t <= 0.5:  
    vOmegaCom = 1.5  
else:  
    vOmegaCom = 0.5
```

5. ゲイン設定

$K_p = 5$, $K_i = 1000$

6. グラフ構成

- (a) v_{TRI} と v_{REF}
- (b) v_T と 電機子電流 i
- (c) $v_{\Omega COM}$ と $v_{DET} = K_e \omega$

以上で、ご要望の内容を反映したコード例となります。長時間のシミュレーションでは計算コストが大きい点にご注意ください。

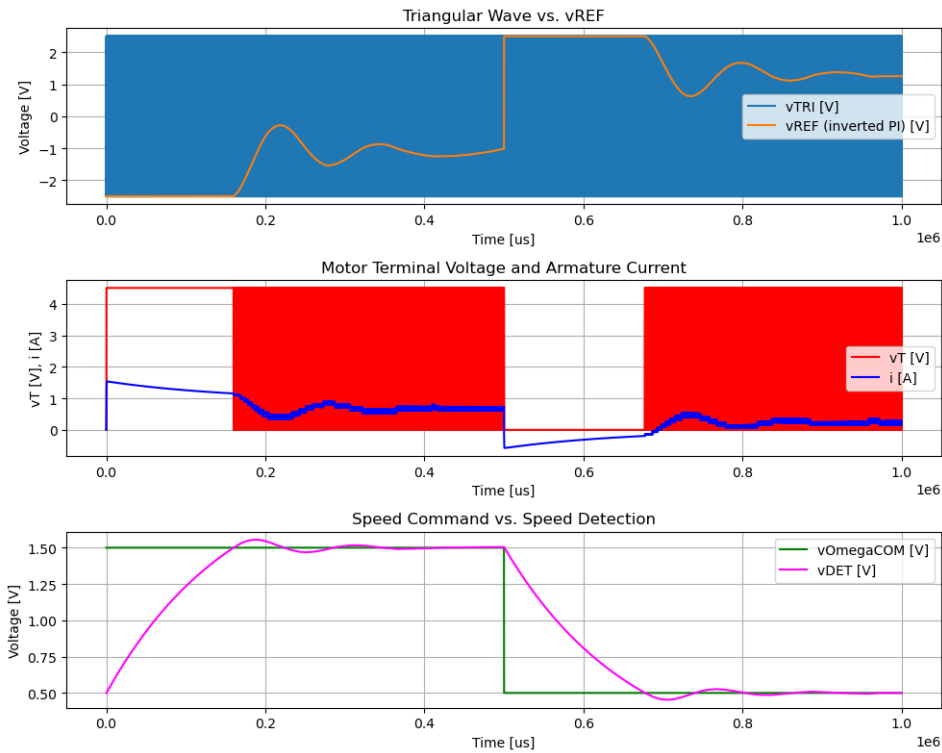


図 12: DC モータの回転数制御シミュレーション (ほぼ達成版 (電機子電流 $i < 0$ を許容))

図 12 は、本コードの実行結果を示しています。回転数指令値の初期値を

$$v_{\Omega COM} = 0.5 \text{ V}$$

とし、以下のように設定しました。

$$0 \leq t \leq 0.5 \text{ s} : v_{\Omega COM} = 1.5 \text{ V},$$

$$0.5 \text{ s} \leq t \leq 1 \text{ s} : v_{\Omega COM} = 0.5 \text{ V}.$$

また、制御ゲインを $K_p = 5$, $K_I = 1000$ としたことで、 v_{REF} ($= v_{COM}$) の振動は収まりました。

しかし、 $t = 0.5 \text{ s}$ 以降約 200 ms の間、電機子電流 i が $i < 0$ となってしまっています。これは図 8 の降圧チョッパでは発生し得ない現象です。そこで、以下のようにコードを修正しました。

Runge-Kutta で次ステップへ

```
x = self.runge_kutta_4(self.motor_equations, t, x, self.dt, vT)
```

上記の次の行に、以下を追加して $i < 0$ の場合は強制的に $i = 0$ とするようにしました。

```
if x[0] < 0:
    x[0] = 0
```

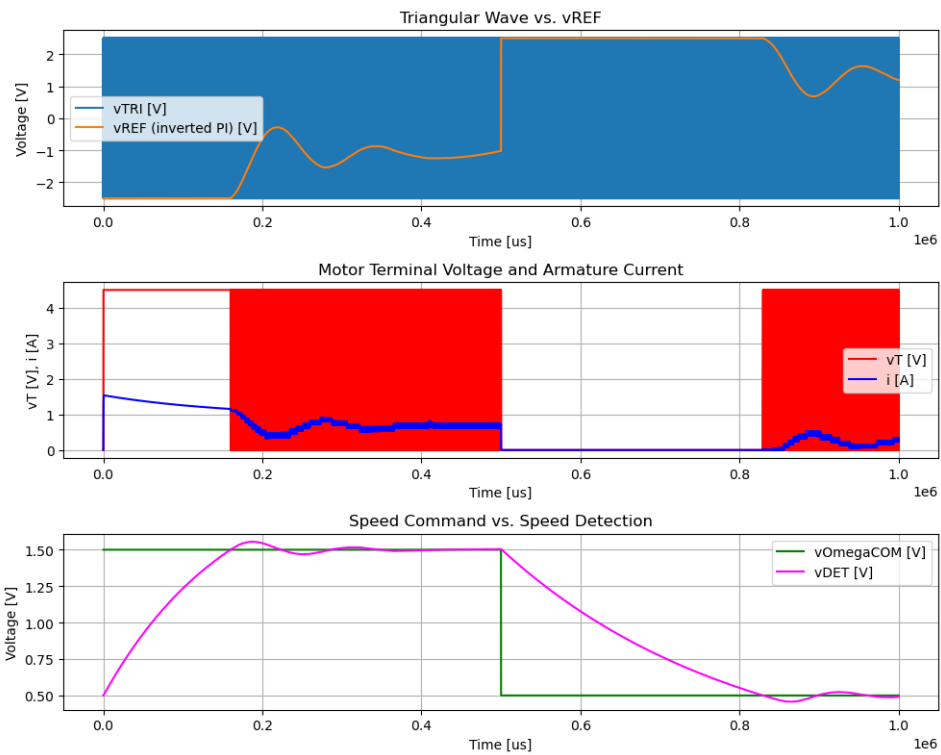



図 13: DC モータの回転数制御シミュレーション (達成版 ($i < 0$ を強制的 $i = 0$ に設定))

図 13 は、 $i < 0$ を強制的に $i = 0$ に設定した場合のシミュレーション結果です。 $t = 0.5\text{s}$ 以降しばらくの間、電機子電流 $i = 0$ となっていることが確認できました。 これにより、モータには電流によるブレーキがかからず、減速はモータの摩擦のみで行われるため、減速が緩やかになっています。