Simulação e Controle de um Motor Síncrono de Ímãs Permanentes usando Controle Orientado a Campo

1st Felipe Lenschow
Programa de pós graduação em engenharia elétrica
Universidade do Estado de Santa Catarina
Joinville, Santa Catarina, Brasil
felipe.lenschow@edu.udesc.br

***Abstract*—Este artigo apresenta a modelagem e simulação de um sistema de acionamento de Motor Síncrono de Ímãs Permanentes (PMSM) utilizando Controle Orientado a Campo (FOC). O modelo matemático do PMSM no referencial dq é derivado, e uma estratégia de controle empregando controladores Proporcional-Integral (PI) para regulação de velocidade e corrente é implementada. A simulação é desenvolvida em Python, permitindo uma análise modular e flexível do comportamento dinâmico do motor sob condições variadas de carga e velocidade. Os resultados demonstram a eficácia da estratégia FOC em manter um controle preciso de velocidade e geração eficiente de torque.**

***Index Terms*—PMSM, Controle Orientado a Campo, Simulação, Python, Acionamento de Motor**

I. Introdução

Motores Síncronos de Ímãs Permanentes (PMSMs) são amplamente utilizados em aplicações industriais, veículos elétricos e robótica devido à sua alta eficiência, alta densidade de potência e excelente desempenho dinâmico. Para alcançar um controle de alto desempenho, o Controle Orientado a Campo (FOC) é comumente empregado. O FOC permite o controle independente de fluxo e torque transformando as correntes trifásicas do estator para um referencial girante (referencial dq) alinhado com o fluxo do rotor [1].

Este artigo detalha o desenvolvimento de um ambiente de simulação para um sistema de acionamento PMSM. A simulação inclui a física do motor, o inversor de fonte de tensão e o algoritmo FOC. O objetivo é fornecer uma compreensão clara da dinâmica do sistema e validar a estratégia de controle através de simulação numérica.

II. Modelo do Sistema

A. Modelo Matemático do PMSM

O modelo dinâmico do PMSM é estabelecido no referencial girante síncrono dq. As equações elétricas são dadas por:

$$V_d = R_s I_d + L_d \frac{dI_d}{dt} - \omega_e L_q I_q \tag{1}$$

$$V_q = R_s I_q + L_q \frac{dI_q}{dt} + \omega_e L_d I_d + \omega_e \lambda_m \tag{2}$$

onde V_d, V_q são as tensões do estator, I_d, I_q são as correntes do estator, R_s é a resistência do estator, L_d, L_q são as indutâncias dos eixos d e q, ω_e é a velocidade angular elétrica, e λ_m é o fluxo concatenado do ímã permanente.

O torque eletromagnético T_e produzido pelo motor é:

$$T_e = \frac{3}{2} P \left[\lambda_m I_q + (L_d - L_q) I_d I_q\right] \tag{3}$$

onde P é o número de pares de polos. Para um PMSM de montagem superficial (SPMSM), $L_d \approx L_q$, e a equação de torque simplifica para $T_e = \frac{3}{2} P \lambda_m I_q$.

A dinâmica mecânica é descrita por:

$$J \frac{d\omega_m}{dt} = T_e - T_L - B\omega_m - T_c \tag{4}$$

onde J é o momento de inércia, ω_m é a velocidade mecânica, T_L é o torque de carga, B é o coeficiente de atrito viscoso, e T_c é o torque de atrito de Coulomb.

B. Modelo do Inversor

O Inversor de Fonte de Tensão (VSI) trifásico é modelado idealmente, assumindo que as tensões de referência geradas pelo controlador são aplicadas com precisão aos terminais do motor, limitadas apenas pela tensão do barramento CC V_{bus}. Os limites da Modulação por Largura de Pulso Vetorial Espacial (SVPWM) são considerados saturando a magnitude do vetor de tensão para $V_{bus}/\sqrt{3}$.

III. Estratégia de Controle

A estratégia FOC é implementada com uma estrutura de controle em cascata.

A. Malha de Controle de Corrente

Dois controladores PI internos regulam as correntes I_d e I_q. A referência de I_d é definida como zero ($I_d^* = 0$) para maximizar o torque por ampère para o modelo SPMSM. A referência de I_q é fornecida pela malha externa de velocidade. Termos de desacoplamento são calculados para compensar os efeitos de acoplamento cruzado entre os eixos d e q mostrados em (1) e (2).

B. *Malha de Controle de Velocidade*

Um controlador PI externo regula a velocidade do motor. O erro entre a velocidade de referência ω_{ref} e a velocidade medida ω_m aciona o controlador PI para gerar a corrente de torque de referência I_q^*. A saída do controlador de velocidade é saturada para limitar a corrente máxima e proteger o motor e o inversor.

IV. Resultados da Simulação

A simulação foi realizada utilizando Python. Os parâmetros do motor utilizados são: $P = 21$, $R_s = 4.485\Omega$, $L_d = L_q = 54.8$ mH, $\lambda_m = 0.201$ Wb, $J = 0.1444$ kg·m^2, $B = 0.0057$ Nms/rad.

O perfil de simulação consiste em:

- $t = 0.0s$: Início em 40 RPM.
- $t = 0.2s$: Degrau de torque de carga de 20 Nm aplicado.
- $t = 0.4s$: Degrau de referência de velocidade para 80 RPM.
- $t = 0.6s$: Degrau de referência de velocidade de volta para 40 RPM.
- $t = 0.8s$: Torque de carga removido.

A Fig. 1 mostra a resposta do sistema.

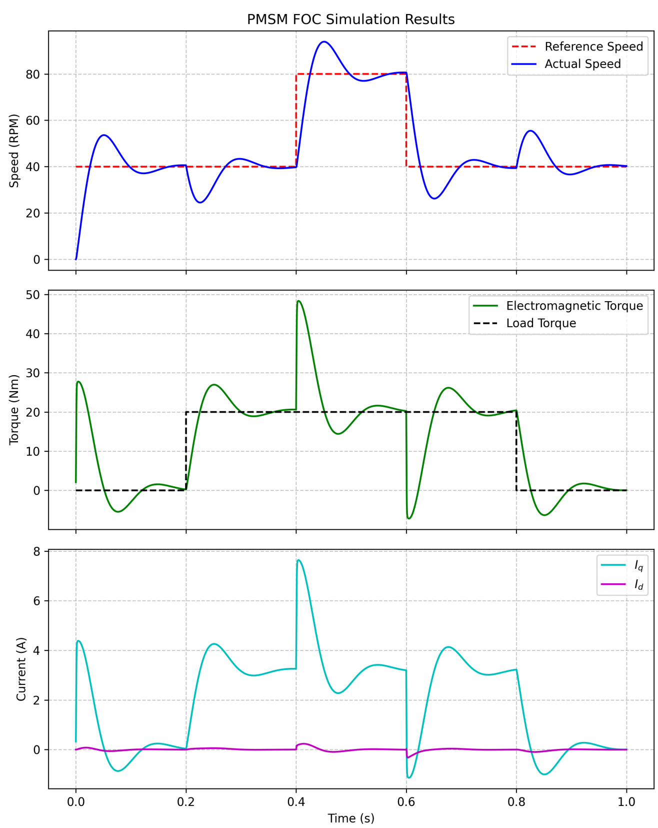

Fig. 1. Resultados da simulação mostrando Velocidade, Torque e Correntes (I_d, I_q).

O controlador de velocidade rastreia a referência de RPM com precisão e mínimo sobressinal. Quando o torque de carga é aplicado em $t = 0.2s$, observa-se uma pequena queda de velocidade, que é rapidamente rejeitada pelo controlador à medida que I_q aumenta para gerar o torque eletromagnético necessário. A corrente I_d é mantida em zero, garantindo uma operação eficiente.

V. Conclusão

Uma simulação completa de um acionamento PMSM usando FOC foi apresentada. A implementação modular em Python permite testes fáceis de diferentes parâmetros de controle e características do motor. Os resultados confirmam a robustez do esquema FOC em lidar com distúrbios de carga e rastrear referências de velocidade.

Appendix A
Código da Simulação

Os códigos fonte da simulação desenvolvida em Python são apresentados a seguir.

A. *PMSMMotor.py*

```
import math
import numpy as np

class PMSMMotor:
    def __init__(self, Ts):
        # --- Motor Parameters ---
        self.Ts = Ts            # Simulation step size
        self.Npp = 21.0         # Pole pairs
        self.Rs = 4.485         # Stator Resistance
        self.Ld = 0.0548        # D-axis Inductance
        self.Lq = 0.0548        # Q-axis Inductance
        self.Lambda_m = 0.201 # Magnet Flux
        self.Bn = 0.0057        # Friction
        self.J = 0.1444         # Inertia
        self.Tc = 0.3006        # Coulomb Torque

        # --- State Variables ---
        self.Id = 0.0
        self.Iq = 0.0
        self.Wr = 0.0           # Mechanical Speed (
            rad/s)
        self.theta = 0.0        # Mechanical Position
            (rad)
        self.theta_e = 0.0      # Electrical Position
            (rad)

    def physics_step(self, Va, Vb, Vc, Tload):
        #
            --------------------------------------------------

        # MOTOR PHYSICS MODEL (The "Plant")
        #
            --------------------------------------------------


        # Calculate Electrical Variables
        We = self.Npp * self.Wr        # Electrical
            Speed
        self.theta_e = self.Npp * self.theta
        # Wrap theta_e to 0-2pi
        self.theta_e = self.theta_e % (2 * math.pi)

        # --- Park Transform (Vabc -> Vdq) ---
        cos_t = math.cos(self.theta_e)
        sin_t = math.sin(self.theta_e)
        cos_t_m = math.cos(self.theta_e - 2*math.pi
            /3)
        sin_t_m = math.sin(self.theta_e - 2*math.pi
            /3)
```

```python
        cos_t_p = math.cos(self.theta_e + 2*math.pi/3)
        sin_t_p = math.sin(self.theta_e + 2*math.pi/3)

        Vd_ref = (2.0/3.0) * (Va * cos_t + Vb * cos_t_m + Vc * cos_t_p)
        Vq_ref = (2.0/3.0) * (-Va * sin_t - Vb * sin_t_m - Vc * sin_t_p)

        # Constants that depend on speed (We)
        g11 = 1 - (self.Ts * (self.Rs / self.Ld))
        g12 = (We * self.Lq * self.Ts) / self.Ld
        g21 = -We * self.Ld * self.Ts / self.Lq
        g22 = 1 - self.Rs * self.Ts / self.Lq
        h11 = self.Ts / self.Ld
        h22 = self.Ts / self.Lq
        i2 = -We * self.Lambda_m * self.Ts / self.Lq

        # Calculate next current states based on Applied Voltages
        Id_next = g11 * self.Id + g12 * self.Iq + h11 * Vd_ref
        Iq_next = g21 * self.Id + g22 * self.Iq + h22 * Vq_ref + i2

        self.Id = Id_next
        self.Iq = Iq_next

        # Torque Calculation
        Te = 1.5 * self.Npp * self.Iq * (self.Lambda_m + (self.Ld - self.Lq) * self.Id)

        # Mechanical Dynamics (Euler Integration)
        # Handle Coulomb friction direction
        Tc_dir = self.Tc if self.Wr > 0 else (-self.Tc if self.Wr < 0 else 0)

        accel = (Te - Tload - (self.Bn * self.Wr) - Tc_dir) / self.J
        self.Wr += accel * self.Ts

        # Position Integration
        self.theta += self.Wr * self.Ts
        self.theta = self.theta % (2*math.pi) # Wrap mechanical angle

        return Te
```

B. FOCController.py

```python
import math

class FOCController:
    def __init__(self, Ts, Imax=8.0):
        self.Ts = Ts
        self.Imax = Imax

        # --- Controller Parameters ---
        self.Kps = 1            # Speed P
        self.Kis = 55.0         # Speed I
        self.KpId = 119.0       # Id P
        self.KiId = 4015.0      # Id I
        self.KpIq = 119.0       # Iq P
        self.KiIq = 4015.0      # Iq I

        # --- Integrator States for PI Controllers ---
        self.Ui_s = 0.0
        self.Ui_Id = 0.0
        self.Ui_Iq = 0.0

    def control_step(self, RPMref, Wr, Ia, Ib, Ic, theta_e, Vbus):
        # ------------------------------------------------------------
        # FIELD ORIENTED CONTROL (FOC)
        # ------------------------------------------------------------

        # --- Clarke Transform (abc -> alpha, beta) ---
        # I_alpha = Ia
        # I_beta = (1/sqrt(3)) * (Ia + 2*Ib)  <-- Standard Clarke
        # Or: I_beta = (Ia + 2*Ib) / sqrt(3)

        I_alpha = Ia
        I_beta = (Ia + 2.0*Ib) / math.sqrt(3.0)

        # --- Park Transform (alpha, beta -> dq) ---
        cos_t = math.cos(theta_e)
        sin_t = math.sin(theta_e)

        Id = I_alpha * cos_t + I_beta * sin_t
        Iq = -I_alpha * sin_t + I_beta * cos_t

        # --- Speed Controller (Outer Loop) ---
        error_speed = (RPMref * 2 * math.pi / 60.0) - Wr

        # PI Calc
        Up_s = self.Kps * error_speed
        # Integral with Anti-windup clamping (Output limited to Imax)
        Ui_s_next = self.Ui_s + (self.Kis * self.Ts * error_speed)

        Iq_ref_unlimited = Up_s + Ui_s_next

        # Saturation / Clamp
        Iq_ref = max(-self.Imax, min(self.Imax, Iq_ref_unlimited))

        # Back-calculation / Anti-windup decision
        if Iq_ref == Iq_ref_unlimited:
            self.Ui_s = Ui_s_next
        else:
            # If saturated, do not update integral (simple anti-windup)
            pass

        Id_ref = 0.0 # MTPA would go here, 0 for surface PMSM

        # --- Current Controllers (Inner Loops) ---

        # Iq Loop
        err_Iq = Iq_ref - Iq
        Up_Iq = self.KpIq * err_Iq
        Ui_Iq_next = self.Ui_Iq + (self.KiIq * self.Ts * err_Iq)
        Vq_ref = Up_Iq + Ui_Iq_next # + Decoupling (omitted as it was 0)
        self.Ui_Iq = Ui_Iq_next # Simplified update

        # Id Loop
        err_Id = Id_ref - Id
        Up_Id = self.KpId * err_Id
        Ui_Id_next = self.Ui_Id + (self.KiId * self.Ts * err_Id)
        Vd_ref = Up_Id + Ui_Id_next # - Decoupling (omitted as it was 0)
        self.Ui_Id = Ui_Id_next
```

```
        # ---------------------------------------------------------------
        # INVERSE PARK & SVPWM
        # ---------------------------------------------------------------

        cos_t = math.cos(theta_e)
        sin_t = math.sin(theta_e)

        # Inverse Park
        Va_ref = cos_t * Vd_ref - sin_t * Vq_ref
        Vb_ref = math.cos(theta_e - 2*math.pi/3) * Vd_ref - math.sin(theta_e - 2*math.pi/3) * Vq_ref
        Vc_ref = math.cos(theta_e + 2*math.pi/3) * Vd_ref - math.sin(theta_e + 2*math.pi/3) * Vq_ref

        # SVPWM Min-Max Injection (from C code logic)
        # Note: In the original simulation, this was calculated but not explicitly used for Vd/Vq modification
        # except for the Vbus limitation below.

        # Voltage Saturation based on Vbus (Limit circle)
        V_mag = math.sqrt(Vd_ref**2 + Vq_ref**2)
        max_V = Vbus / math.sqrt(3) # Max phase voltage with SVPWM

        if V_mag > max_V:
            ratio = max_V / V_mag
            Vd_ref *= ratio
            Vq_ref *= ratio

        return Va_ref, Vb_ref, Vc_ref
```

C. Inverter.py

```
import math

class Inverter:
    def __init__(self):
        pass

    def step(self, Va_ref, Vb_ref, Vc_ref, Vbus):
        # ---------------------------------------------------------------
        # INVERTER MODEL
        # ---------------------------------------------------------------

        # Simple voltage source inverter model.
        # Limits the output phase voltages based on Vbus.
        # In a real inverter, this would involve PWM duty cycles.
        # Here we assume average voltage injection with saturation.

        # Max phase voltage (linear modulation limit for SVPWM)
        max_V = Vbus / math.sqrt(3)

        # Calculate magnitude of the requested voltage vector
        # (Assuming balanced 3-phase, we can estimate magnitude)
        # A more robust check is to convert back to alpha-beta or dq to check magnitude,
        # but here we can check individual phase limits or the vector sum.

        # Let's use the same logic as before: limit the vector magnitude.
        # To do this without dq, we can look at the peak.
        # Or simpler: just clamp individual phases to +/- Vbus/2 (DC link midpoint ref)
        # BUT, the previous logic used a circular limit on Vdq.

        # Let's stick to the previous logic's intent:
        # If we receive Vabc, we assume they are already appropriate.
        # However, we should enforce the physical limit of the bus.

        # For this simulation, let's assume the controller handles the circular limit (SVPWM),
        # and the inverter just hard-clamps if something goes wrong, or models the PWM effect.

        # Since the controller was doing the limiting, we can just pass through
        # or add a hard clamp for safety.

        # Let's add a simple clamp to +/- Vbus/2 for each phase relative to neutral point
        # (assuming ideal DC link utilization).

        limit = Vbus / 2.0

        Va = max(-limit, min(limit, Va_ref))
        Vb = max(-limit, min(limit, Vb_ref))
        Vc = max(-limit, min(limit, Vc_ref))

        return Va, Vb, Vc
```

D. Sensors.py

```
import math

class Sensors:
    def __init__(self):
        pass

    def measure(self, motor, RPMref):
        # ---------------------------------------------------------------
        # SENSOR MODEL
        # ---------------------------------------------------------------

        # 1. Measure Currents (Ia, Ib, Ic)
        # Calculate Ia, Ib, Ic from motor states (Id, Iq, theta_e)

        # Inverse Park Transform (dq -> alpha, beta)
        cos_t = math.cos(motor.theta_e)
        sin_t = math.sin(motor.theta_e)

        I_alpha = motor.Id * cos_t - motor.Iq * sin_t
        I_beta  = motor.Id * sin_t + motor.Iq * cos_t
```

```
        # Inverse Clarke (alpha, beta -> abc)
        Ia = I_alpha
        Ib = -0.5 * I_alpha + (math.sqrt(3)/2.0) * I_beta
        Ic = -0.5 * I_alpha - (math.sqrt(3)/2.0) * I_beta

        # 2. Measure Position (theta_e)
        # Assuming perfect position sensor for FOC
        theta_e = motor.theta_e

        # 3. Measure Speed
        # Reverting to actual speed measurement to fix feedback loop
        Wr_meas = motor.Wr

        return Ia, Ib, Ic, theta_e, Wr_meas
```

E. Simulate.py

```
import numpy as np
import matplotlib.pyplot as plt
import math
from Sim.PMSMMotor import PMSMMotor
from Sim.FOCController import FOCController
from Sim.Inverter import Inverter
from Sim.Sensors import Sensors

# --- Main Execution ---
if __name__ == "__main__":
    # Simulation Parameters
    Ts = 1e-4
    t_end = 1.0

    # Initialize Modules
    motor = PMSMMotor(Ts)
    controller = FOCController(Ts)
    inverter = Inverter()
    sensors = Sensors()

    # Time settings
    num_steps = int(t_end / Ts)

    # Storage for plotting
    history = {
        'time': np.zeros(num_steps),
        'rpm_ref': np.zeros(num_steps),
        'rpm_act': np.zeros(num_steps),
        'Iq': np.zeros(num_steps),
        'Id': np.zeros(num_steps),
        'Te': np.zeros(num_steps),
        'Tload': np.zeros(num_steps),
        'Vbus': np.zeros(num_steps)
    }

    print("Starting Simulation...")

    t = 0.0
    for k in range(num_steps):
        # Update time
        t = k * Ts
        history['time'][k] = t

        # --------------------------------------------------------
        # 1. INPUTS & PROFILE
        # --------------------------------------------------------
        RPMref = 40.0
        Tload = 0.0

        if t > 0.2:
            Tload = 20.0
        if t > 0.4:
            RPMref = 80.0
        if t > 0.6:
            RPMref = 40.0
        if t > 0.8:
            Tload = 0.0

        V_bus = 311.0

        # --------------------------------------------------------
        # 2. SENSORS STEP
        # --------------------------------------------------------
        Ia, Ib, Ic, theta_e, Wr_meas = sensors.measure(motor, RPMref)

        # --------------------------------------------------------
        # 3. CONTROL STEP
        # --------------------------------------------------------
        Va_ref, Vb_ref, Vc_ref = controller.control_step(
            RPMref, Wr_meas, Ia, Ib, Ic, theta_e, V_bus
        )

        # --------------------------------------------------------
        # 4. INVERTER STEP
        # --------------------------------------------------------
        Va, Vb, Vc = inverter.step(Va_ref, Vb_ref, Vc_ref, V_bus)

        # --------------------------------------------------------
        # 5. MOTOR PHYSICS STEP
        # --------------------------------------------------------
        Te = motor.physics_step(Va, Vb, Vc, Tload)

        # --------------------------------------------------------
        # 6. DATA LOGGING
        # --------------------------------------------------------
        history['rpm_ref'][k] = RPMref
        history['rpm_act'][k] = motor.Wr * 60 / (2*math.pi)
        history['Iq'][k] = motor.Iq
        history['Id'][k] = motor.Id
        history['Te'][k] = Te
        history['Tload'][k] = Tload
        history['Vbus'][k] = V_bus

    data = history

    # --- Plotting ---
```

```
    fig, (ax1, ax2, ax3) = plt.subplots(3, 1,
        figsize=(10, 12), sharex=True)

    # Plot 1: Speed
    ax1.plot(data['time'], data['rpm_ref'], 'r--',
        label='RPM Ref')
    ax1.plot(data['time'], data['rpm_act'], 'b-',
        label='RPM Actual')
    ax1.set_ylabel('Speed (RPM)')
    ax1.set_title('PMSM FOC Simulation')
    ax1.legend()
    ax1.grid(True)

    # Plot 2: Torque
    ax2.plot(data['time'], data['Te'], 'g-', label='
        Electromagnetic Torque')
    ax2.plot(data['time'], data['Tload'], 'k--',
        label='Load Torque')
    ax2.set_ylabel('Torque (Nm)')
    ax2.legend()
    ax2.grid(True)

    # Plot 3: Currents and Voltage Input
    ax3.plot(data['time'], data['Iq'], 'c-', label='
        Iq (A)')
    ax3.plot(data['time'], data['Id'], 'm-', label='
        Id (A)')
    # Scaling Vbus to fit on plot for visualization
    ax3.plot(data['time'], data['Vbus']/10, 'y-',
        alpha=0.3, label='Vbus Input / 10 (V)')
    ax3.set_ylabel('Current (A)')
    ax3.set_xlabel('Time (s)')
    ax3.legend()
    ax3.grid(True)

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

References

[1] F. Blaschke, "The principle of field orientation as applied to the new TRANSVECTOR closed-loop control system for rotating-field machines," Siemens Review, vol. 39, no. 5, pp. 217-220, 1972.