A PYTHON-BASED SIMULATION AND EMBEDDED IMPLEMENTATION OF SPACEVECTOR PULSE WIDTH MODULATION ON STM32 CONTROLLERS

Nguyen Tien Hung
TNU - University of Technology

ARTICLE INFO

Received: 24/6/2025
Revised: 26/11/2025
Published: 26/11/2025

KEYWORDS

Space-vector pulse width modulation
Three-phase voltage inverter
Python programming
Python simulation
STM32G431 microcontroller

ABSTRACT

This paper presents a detailed approach to developing and validating the space-vector pulse width modulation technique using the Python programming environment. The proposed simulation framework is implemented entirely in Python scripts and models a two-level, three-phase voltage-source inverter. It includes essential functions such as Clarke and Park transformations, sector determination, and duty cycle computation based on vector time decomposition. Unlike traditional simulation tools that rely on graphical user interfaces, the script-based method offers greater transparency and control over algorithmic implementation, enabling users to better understand each computational step involved in pulse generation. Although this approach may be less familiar to users accustomed to commercial block-diagram environments, it enhances comprehension of modulation principles and inverter behavior. Additionally, the Python script is designed with portability in mind, allowing straightforward conversion into embedded C code for real-time execution on STM32G431 microcontrollers. Experimental results confirm that the proposed method offers fast simulation times and reduced code complexity compared to conventional simulation tool chains.

MÔ PHỎNG PYTHON VÀ THỰC HIỆN ĐIỀU CHẾ VECTƠ KHÔNG GIAN TRÊN BỘ ĐIỀU KHIỂN STM32

Nguyễn Tiến Hưng
Trường Đại học Kỹ thuật Công nghiệp – ĐH Thái Nguyên

THÔNG TIN BÀI BÁO

Ngày nhận bài: 24/6/2025
Ngày hoàn thiện: 26/11/2025
Ngày đăng: 26/11/2025

TỪ KHÓA

Điều chế không gian vector
Bộ biến đổi nguồn áp ba pha
Lập trình Python
Mô phỏng Python
Vi điều khiển STM32G431

TÓM TẮT

Bài báo trình bày một phương pháp tiếp cận để phát triển và kiểm chứng kỹ thuật điều chế vectơ không gian trong môi trường lập trình Python. Phần mô phỏng được triển khai hoàn toàn bằng tập lệnh Python để mô hình hóa một bộ biến đổi nguồn áp ba pha, hai cấp. Trình mô phỏng bao gồm các chức năng cơ bản như phép biến đổi Clarke và Park, xác định các sector và tính các chu kỳ nhiệm vụ dựa trên phân tích các vectơ theo thời gian. Không giống như các công cụ mô phỏng truyền thống dựa trên giao diện đồ họa, phương pháp dựa trên tập lệnh cung cấp tính minh bạch và khả năng kiểm soát tốt hơn đối với việc triển khai thuật toán, cho phép người dùng hiểu rõ hơn từng bước tính toán liên quan đến việc tạo xung. Mặc dù phương pháp tiếp cận này có thể ít quen thuộc hơn đối với người dùng đã quen với các phần mềm thương mại sử dụng các sơ đồ khối, nhưng nó giúp tăng cường khả năng hiểu rõ các nguyên tắc điều chế và hành vi của bộ biến tần. Ngoài ra, tập lệnh Python được thiết kế có tính đến tính tương thích, cho phép chuyển đổi trực tiếp thành mã C nhúng để thực thi trên vi điều khiển STM32G431 trong thời gian thực. Kết quả thử nghiệm xác nhận rằng phương pháp được đề xuất cung cấp thời gian mô phỏng nhanh và giảm độ phức tạp của mã so với chuỗi công cụ mô phỏng thông thường.

DOI: https://doi.org/10.34238/tnu-jst.13117

Email: h.nguyentien@tnut.edu.vn

1. Introduction

Three-phase inverters are widely used in power electronics to convert direct current (DC) into alternating current (AC) with controllable amplitude and frequency, making them essential in motor drives, renewable energy systems, and industrial automation, and power converters [1], [2]. The quality of the output waveform generated by an inverter is determined by the switching states of its six semiconductor switches, which directly control the synthesis of the AC voltage. Several modulation techniques have been developed, including the six-step inverter method, hysteresis current control, sinusoidal pulse width modulation (SPWM), and space vector pulse width modulation (SVPWM), each offering specific trade-offs in complexity, efficiency, and waveform quality [3] – [5].

SVPWM is a fundamental control strategy widely employed in three-phase voltage-source inverters for industrial drives, renewable energy systems, and embedded power applications. It provides efficient and continuous modulation of output voltage magnitude, frequency, and phase angle, resulting in superior performance compared to traditional sinusoidal pulse-width modulation (PWM) techniques. While multilevel SVPWM methods have been developed to improve output waveform quality and reduce total harmonic distortion, the two-level, three-phase (2L3P) inverter remains the pedagogical and practical baseline due to its hardware simplicity, robust performance, and mathematically tractable control algorithm [6] – [8]. In practical applications, three major performance criteria must be addressed: harmonic distortion, switching frequency, and the effective utilization of the DC link voltage. Minimizing harmonic content in the output waveform is crucial, as lower harmonics improve the efficiency and dynamic performance of motor drive systems and reduce the need for bulky filtering components. Increasing the switching frequency typically enhances the smoothness of the output current and improves dynamic response. However, it also results in higher switching losses and thermal stress, and is limited by the physical switching speed and dead-time constraints of the semiconductor devices [9] – [11].

Despite the extensive theoretical and simulation-based literature on SVPWM, many published works offer limited discussion on the link between simulation and practical implementation details, particularly for real-time embedded systems. As a result, this paper presents a comprehensive approach that transitions from well-established simulation environments to a performance-validated implementation of SVPWM, suitable for both academic use and real-world embedded control systems. The SVPWM will be implemented on a 32-bit ARM Cortex-M4-based microcontroller. This work serves as a practical reference for researchers and educators seeking to bridge the gap between SVPWM theory and its deployment on modern microcontroller platforms.

The rest of this paper is organized as follows. Section 2 presents the three-phase voltage source inverter and the SVPWM technique, including its principle and a Python-based simulation of the SVPWM strategy. The simulation and experimental results from the implementation of SVPWM on STM32 controllers are addressed in Section 3. Finally, the conclusions and future directions are provided in Section 4.

2. Three-phase voltage source inverter and SVPWM technique

2.1. Principle of space vector modulation

Let the three-phase sinusoidal voltage component be $V_u = V_m \sin \omega_s t$, $V_v = V_m \sin\left(\omega_s t - \frac{2\pi}{3}\right)$, $V_w = V_m \sin\left(\omega_s t - \frac{4\pi}{3}\right)$, where ω_s is the stator angular frequency, V_m is the amplitude of the supply voltage. The SVPWM technique is aimed at swiftly estimating the reference voltage vector v_{ref} in the $\alpha\beta$ coordinate by combining the switching states representing the fundamental space vectors. Vectors v_{100}, v_{110}, v_{010}, v_{011}, v_{001}, and v_{101} represent varied non-zero vectors. The two

vectors, v_{000} and v_{111}, are known as null (or zero) vectors since they are derived when all windings are on the positive or negative terminals of the DC bus. The eight base vectors separated by a 60-degree phase difference from each other are formed in a hexagonal diagram in stator-fixed $\alpha\beta$ coordinates. The six base vectors shape the hexagon with six sectors $S_1 \cdots S_6$ as shown in Figure 1a. The two specific vectors, v_{000} and v_{111}, are plotted at the origin of the hexagon.

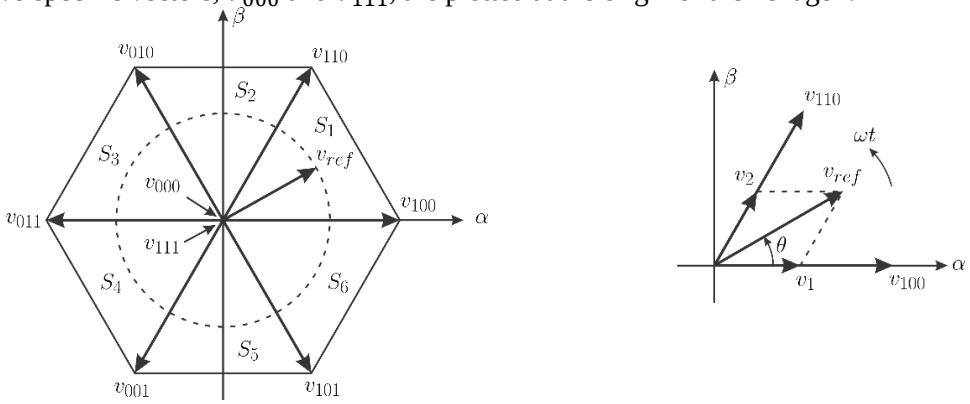


Figure 1. *The standard voltage vectors (a) and the realization of an arbitrary voltage vector from two boundary vectors (b)*

Now, let us assume that the vector to be realized, v_{ref}, is located in the sector S_1, the area between the standard vectors v_{100} and v_{110} as shown in Figure 1b. The reference vector v_{ref} can be implemented by applying vector v_1 for a specified time T_1 and vector v_2 for a specified time T_2. During the remaining time of T_s, the applied voltage should be imposed to zero by applying the null vector v_{000} (or v_{111}). The time for the null vector is denoted by T_0 (or T_7). The necessary times T_1, T_2, and T_0 (or T_7) can be calculated as follows [12], [13].

$$T_1 = T_s \frac{|v_1|}{|v_{ref}|},\ T_2 = T_s \frac{|v_2|}{|v_{ref}|},\ T_0 = T_7 = T_s - T_1 - T_2 \text{ -} \quad (1)$$

The realization of voltage vector v_{ref} can now be represented as follows:

$$v_{ref} = \frac{T_1}{T_s} v_{100} + \frac{T_2}{T_s} v_{110} + \frac{T_s - (T_1 + T_2)}{T_s} v_{000}\ (or\ v_{111}) \quad (2)$$

Switching durations T_1, and T_2 at any sector ϑ_n= 1 to 6 can be expressed as [8-9]

$$T_1 = T_s \sqrt{3} \frac{|v_{ref}|}{V_{dc}} \sin\left(\vartheta_n \frac{\pi}{3} - \theta\right) = T_s m_\gamma \sin\left(\vartheta_n \frac{\pi}{3} - \theta\right) \quad (3)$$

$$T_2 = T_s \sqrt{3} \frac{|v_{ref}|}{V_{dc}} \sin\left(\frac{\pi}{3} - \theta_r\right) = T_s m_\gamma \sin\left(\frac{\pi}{3} - \theta_r\right) \quad (4)$$

$$T_0 = T_s - (T_1 + T_2) \quad (5)$$

where $m_\gamma = \sqrt{3} \frac{|v_{ref}|}{V_{dc}}$ is called the magnitude modulation index, and $\theta_r = \vartheta_n \frac{\pi}{3} - \theta$.

2.2. A Python-based simulation of the SVPWM strategy

To initiate the simulation and numerical computation process, the open-source Python libraries NumPy and SciPy are imported. The NumPy library provides essential tools for efficient numerical operations, including array manipulation, linear algebra, and mathematical functions, which are fundamental for vector and matrix computations in control algorithms. The SciPy library complements this by offering a wide range of scientific computing capabilities, such as signal processing, optimization, and integration routines. These libraries form the computational backbone of the simulation environment and are imported as shown below:

```
import numpy as np
from scipy.signal import lfilter
```

Next, we declare necessary constants and parameters as follows:

```
# Constants
PI = np.pi; PI_2 = PI/2; PI_3 = PI/3; _SQRT3 = np.sqrt(3); Vdc = 24.0
# User-defined parameters
f_pwm = 10e3            # PWM frequency in Hz
T_sample = 1e-6         # Sampling time (s)
electrical_freq = 50    # Electrical frequency (Hz)
dead_time = 2e-6        # Dead-time in seconds
fs = 1e6              # Simulation frequency (1 MHz)
# Derived values
Ts = 1/f_pwm
omega_el = 2 * PI * electrical_freq
total_time = 2/electrical_freq
time_array = np.arange(0, total_time, T_sample)
```

In addition to the core space vector modulation logic, the implementation includes several auxiliary functions that support accurate and efficient signal generation. Two of the principal supporting functions are detailed as follows:

- The normalize_angle() function ensures that the electrical angle remains bounded within a fundamental 2π interval. This normalization is critical for maintaining the correct orientation of the rotating reference frame (the dq frame) with respect to the stationary $\alpha\beta$ coordinate system.
- The insert_dead_time()function is designed to introduce a non-overlapping delay, known as dead-time, between the switching events of complementary PWM signals. The function inserts a programmable delay at both the rising and falling edges of the main PWM signal, ensuring that one switch is fully turned off before its complementary switch is turned on.

```
def normalize_angle(angle):
    a = np.fmod(angle, 2 * PI)
    return a if a >= 0 else (a + 2 * PI)
def insert_dead_time(main_pwm, T_sample, dead_time):
    pwm_len = len(main_pwm); dead_samples = int(np.ceil(dead_time / T_sample))
    complement_pwm = 1 - main_pwm; i = 1; pwm_nt = complement_pwm.copy()
    for k in range(1, pwm_len):
        if k>=i:
            i = k; if pwm_nt[i] == 0 and pwm_nt[i - 1] == 1:
                complement_pwm[i:i+dead_samples] = 1; i = i+dead_samples
            elif pwm_nt[i] == 1 and pwm_nt[i - 1] == 0:  # Rising edge
                complement_pwm[i-dead_samples:i] = 1
    return complement_pwm
```

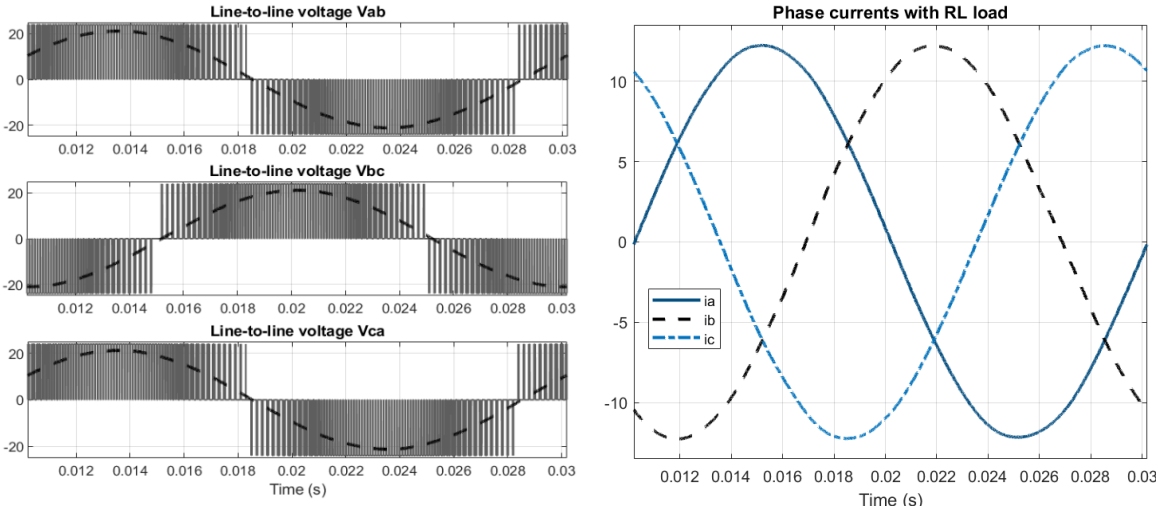
The switching durations derived in Equations (3), (4), and (5), corresponding respectively to the active vector durations T_1, T_2, and the zero vector duration T_0, are directly applied in the implementation of the compute_svpwm() function. Within this function, these durations are used to determine the precise time intervals during which each inverter switch should be activated within one PWM cycle. This implementation ensures that the reference voltage vector is synthesized accurately by modulating the duty cycles of the three-phase output signals. The correct computation and application of T_1, T_2, and T_0 are essential for generating the desired output voltage while maintaining the advantages of SVPWM. The compute_svpwm() function is written as follows:

```
def compute_svpwm(Vref, angle_el):
    angle_el = normalize_angle(angle_el + PI_2)
    sector = int(np.floor(angle_el / PI_3)) + 1
    T1 = Ts * _SQRT3 * Vref/Vdc * np.sin(sector * PI_3 - angle_el)
    T2 = Ts * _SQRT3 * Vref/Vdc * np.sin(angle_el - (sector - 1) * PI_3)
    T0 = max(0.0, Ts - T1 - T2)
    if sector == 1: Ta = T1 + T2 + T0/2; Tb = T2 + T0/2; Tc = T0/2
    elif sector == 2: Ta = T1 + T0/2; Tb = T1 + T2 + T0/2; Tc = T0/2
    elif sector == 3: Ta = T0/2; Tb = T1 + T2 + T0/2; Tc = T2 + T0/2
    elif sector == 4: Ta = T0/2; Tb = T1 + T0/2; Tc = T1 + T2 + T0/2
```

```
elif sector == 5: Ta = T2 + T0/2; Tb = T0/2; Tc = T1 + T2 + T0/2
elif sector == 6: Ta = T1 + T2 + T0/2; Tb = T0/2; Tc = T1 + T0/2
else: Ta = Tb = Tc = 0
return T1, T2, T0, Ta, Tb, Tc
```

3. Simulation results and implementation

3.1. Simulation results



(a) (b)

Figure 2. *Line-to-line voltages* $\boldsymbol{V_{ab}}$, $\boldsymbol{V_{bc}}$, $\boldsymbol{V_{ca}}$ *and phase currents* $\boldsymbol{i_a}$, $\boldsymbol{i_b}$, $\boldsymbol{i_c}$ *with RL load*

The waveforms of the line-to-line voltages V_{ab}, V_{bc}, and V_{ca}, along with their corresponding filtered signals, are illustrated in Figure 2a. As observed from the figure, the unfiltered voltages exhibit the characteristic stepped pattern of SVPWM, resulting from the discrete switching states of the inverter. After applying a low-pass filter to these signals, the fundamental components emerge clearly, revealing sinusoidal waveforms with minimal distortion. Figure 2b presents the measured waveforms of the three-phase currents i_a, i_b, and i_c under a balanced load consisting of a 0.5Ω resistor and a 1 mH inductor per phase. As illustrated in the figure, the phase currents exhibit smooth, nearly sinusoidal waveforms with minimal distortion, indicating that the SVPWM strategy effectively minimizes switching harmonics and delivers high-quality output.

3.2. The implementation of SVPWM on STM32 controllers

To demonstrate the implementation of the SVPWM on the STM32G431 microcontroller, the corresponding Python code was translated into C language. The resulting program operates in a bare-metal environment, meaning it does not rely on the Hardware Abstraction Layer (HAL), Low-Layer (LL), or CMSIS libraries provided by STMicroelectronics. Instead, all peripheral control is performed through direct manipulation of memory-mapped registers. This low-level approach enables precise control over timing, system performance, and hardware behavior.

In this paper, we use Timer 1 working in center-aligned mode with dead-time insertion to implement the SVPWM algorithm. Suppose the clock frequency supplied to Timer 1 is $f_{TCK} =$ 170 MHz and the PWM frequency is $f_{PWM} = 5$ kHz. Then, the ARR (auto-reload value) in the center-aligne mode is calculated as [14], [15]:

$$\text{ARR} = \frac{\text{f}_{TCK}}{2 \times \text{f}_{PWM} \times (\text{PSC} + 1)} = \frac{170 \times 10^6}{2 \times 10 \times 5^3} = 17{,}000 \quad (6)$$

where the prescaler (PSC) value of the timer is set to 0. Furthermore, the dead-time is chosen of 2 μs. First, we define the following constants

```
#define VDC         24.0f
#define POLE_PAIRS  4                 // Number of pole pairs
#define TSAMP       0.0001f           // Sampling time 100~us step
#define F_CPU       170000000UL
#define PWM_FREQ    5000
#define PWM_PERIOD  (F_CPU/PWM_FREQ/2)  // Center-aligned PWM
#define TS          (1/PWM_FREQ)
```

Similar to the Python implementation, the switching durations are directly utilized in the C-based version of the compute_svpwm() function. The C version of the function is implemented as follows:

```
void compute_svpwm(float Vref, float theta) {
 theta = normalize_angle(theta + PI_2); sector = theta/PI_3 + 1;
 float T1 = _SQRT3 * Vref/VDC * sinf(sector * PI_3 - theta);
 float T2 = _SQRT3 * Vref/VDC * sinf(theta - (sector - 1) * PI_3);
 float T0 = (1 - T1 - T2 > 0.0f) ? (1 - T1 - T2) : 0.0f; float Ta, Tb, Tc;
 switch (sector) {
  case 1: Ta = T1 + T2 + T0/2; Tb = T2 + T0/2; Tc = T0/2; break;
  case 2: Ta = T1 + T0/2; Tb = T1 + T2 + T0/2; Tc = T0/2; break;
  case 3: Ta = T0/2; Tb = T1 + T2 + T0/2; Tc = T2 + T0/2; break;
  case 4: Ta = T0/2; Tb = T1 + T0/2; Tc = T1 + T2 + T0/2; break;
  case 5: Ta = T2 + T0/2; Tb = T0/2; Tc = T1 + T2 + T0/2; break;
  case 6: Ta = T1 + T2 + T0/2; Tb = T0/2; Tc = T1 + T0/2; break;
  default: Ta = Tb = Tc = 0;
 }
 set_pwm_duty(Ta, Tb, Tc);
 }
```

To configure the PWM output duty cycles according to the computed vector durations, the set_pwm_duty() function is employed. This function translates the modulation times T_1, T_2, and T_0 into corresponding pulse widths for each leg of the three-phase inverter.

```
float normalize_angle(float angle) {
  angle = fmodf(angle, _2PI);        // Use fmodf for float type
  if (angle < 0.0f) angle += _2PI;
  return angle;
}
void set_pwm_duty(float Ta, float Tb, float Tc) {
  uint16_t arr = TIM1->ARR; TIM1->CCR1 = (uint16_t)(Ta * arr);
  TIM1->CCR2 = (uint16_t)(Tb * arr); TIM1->CCR3 = (uint16_t)(Tc * arr);
}
```

Within the main() function, the target mechanical speed specified in revolutions per minute (RPM) is first converted into its corresponding electrical angular velocity, expressed in radians per second. The electrical angular speed ω_e is given by $\omega_e = \omega_m \cdot p$, where ω_m is the mechanical angular velocity and p denotes the number of pole pairs. Subsequently, the electrical angle increment per sampling period is computed based on this angular velocity. Inside the infinite while(1) loop, this angle increment is repeatedly accumulated to update the electrical angle used for generating the SVPWM waveforms. The main() function is written as follows:

```
float motor_speed; float Vref = 12.0;
volatile float theta = 0.0f; float omega_m,omega_e, dtheta; int sector=1;
int main(void) {
  GPIO_Init(); Timer1_Init(); motor_speed = 1500.0; omega_m = motor_speed/60.0;
  omega_e = omega_m * POLE_PAIRS; dtheta = 2.0 * PI * omega_e * TSAMP;
  while (1) {
    theta = normalize_angle(theta + dtheta); compute_svpwm(Vref, theta);
    delay_us(100);
  }
}
```

3.3. Experimental results

The SVPWM algorithm is experimentally validated using the hardware setup illustrated in Figure 3. The system consists of a low-voltage brushless DC motor rated for 24 VDC operation, which is powered through a dedicated DC supply. The motor is driven by a two-level, three-phase voltage source inverter constructed using six IGBTs. Gate control signals for the IGBT switches are generated using PWM outputs from the STM32G431 microcontroller, operating in bare-metal mode to ensure precise timing and low-latency signal.

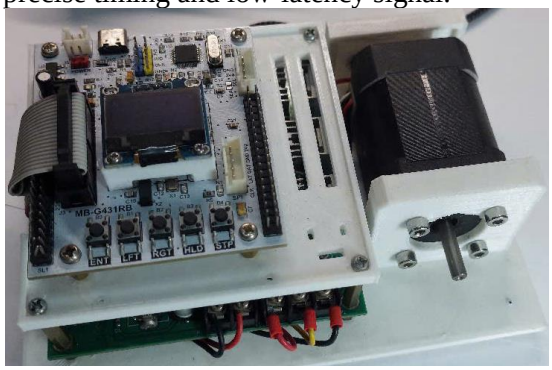
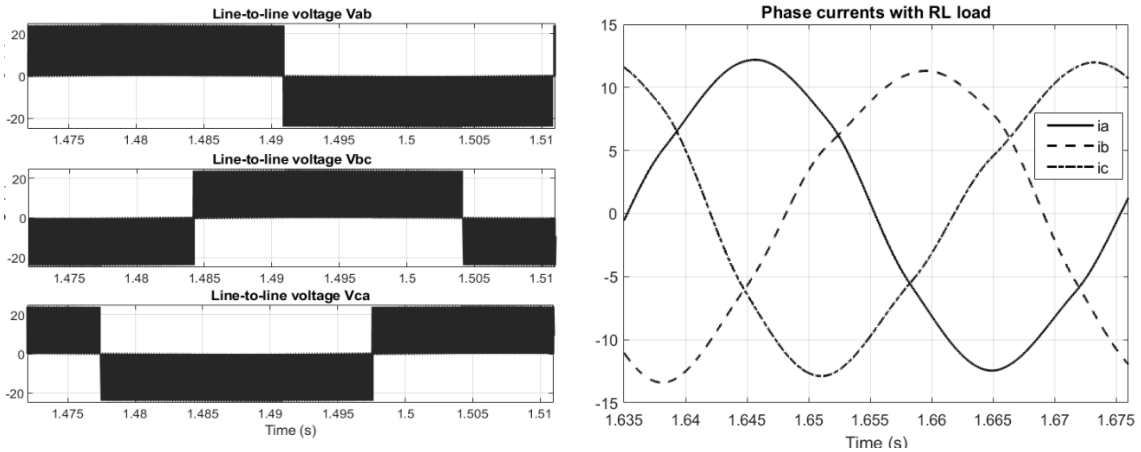


Figure 3. *The experimental setup*

The measured line-to-line voltages and phase currents are presented in Figure 4. As observed from the figure, the experimental waveforms closely resemble those obtained from the Python-based simulation, particularly under the same RL load conditions of 0.5 Ω resistance and 1 mH inductance. However, a slight deviation is noted in the shape of the phase current waveforms, which exhibit less sinusoidal characteristics in the experimental data.



(a) (b)

Figure 4. *The real line-to-line voltages (a) and phase currents (b)*

4. Conclusions and recommendations

This study presents a complete implementation of SVPWM using Python simulation and bare-metal C programming on the STM32G431. The Python framework provides a clear and flexible way to understand SVPWM principles, while waveform analysis confirms accurate sinusoidal voltage generation. The C code, directly translated from Python, achieves precise PWM control without HAL or CMSIS. Experimental results under load closely match simulations, validating the method's accuracy. Minor deviations in current waveforms are due to hardware non-idealities not captured in the simulation.

Future work may focus on extending this framework to support multilevel inverters, incorporate advanced features such as overmodulation and fault handling, and explore real-time tuning of SVPWM parameters. Additionally, integrating feedback mechanisms and developing closed-loop control strategies using sensor data would further enhance the applicability of this system in precision motor control applications.

REFERENCES

[1] G. Ala, N. Campagna, M. Caruso, V. Castiglia, A. O. D. Tommaso, R. Miceli, C. Nevoloso, G. Schettino, F. Viola, and M. Nguyen, "Stability of Microgrids: An Application of Virtual Synchronous Generator," *In Proc. International Conference on Engineering Research and Applications '602*, 2022, pp. 873-880.

[2] H. T. Do, T.D. Vu, K. N. Nguyen, E. Semail, and M. T. Nguyen, "High Quality Torque for Five-Phase Open-End Winding Non-sinusoidal PMSM Drives," In *Proc. International Conference on Engineering Research and Applications '944*, 2024, pp. 9-19.

[3] B. Wu, *High-power converters and AC drives*. Wiley-IEEE Press, 2006.

[4] M. H. Rashid, *Power electronics: devices, circuits, and applications*. Pearson, 2014.

[5] H. Abu-Rub, A. Iqbal, and J. Guzinski, *High performance control of AC drives with Matlab/Simulink models*. Wiley, 2012.

[6] M. M. Gaballah, "Design and implementation of space vector PWM inverter based on a low cost microcontroller," *Arabian Journal for Science and Engineering*, vol. 38, pp. 3059–3070, 2013.

[7] H. Zhang, Y. Meng, L. Ning, Y. Zou, X. Wang, and X. Wang, "Fast and simple space vector modulationmethod for multilevel converters," *IET Power Electronics*, vol. 13, pp. 14–22, 2020.

[8] A. Khaliq, S. A. R. Kashif, F. Ahmad, M. Anwar, Q. Shaheen, R. Akhtar, M. A. Shah, and A. Abdelmaboud, "Indirect vector control of linear induction motors using space vector pulse width modulation," *Computers, Materials and Continua*, vol. 74, pp. 6263–6287, 2022.

[9] L. Tiitinen, M. Hinkkanen, and L. Harnefors, "Design framework for sensorless control of synchronous machine drives," *IEEE Transactions on Industrial Electronics*, vol. 72, pp. 1379–1390, 2025.

[10] Z. Liu, W. Zhang, C. Li, X. Wang, and H. Qin, "Improved virtual SVPWM algorithm for CMV reduction and NPV oscillationelimination in three-level NPC inverter," *International Journal of Electrical Power and Energy Systems*, vol. 155, Part A, January 2024, Art. no. 109533, doi: 10.1016/j.ijepes.2023.109533.

[11] L. Tiitinen, M. Hinkkanen, and L. Harnefors, "Sensorless flux-vector control framework: An extension forinduction machines," *IEEE Transactions on Industrial Electronics*, vol. 99, pp. 1-6, 2025, doi: 10.1109/TIE.2025.3559958.

[12] V. Kumar, R. K. Behera, D. Joshi, and R. Bansal, *Power electronics, drives, and advanced applications.* CRC Press, 2020.

[13] N. P. Quang and J.-A. Dittrich, *Vector control of three-phase AC machines: System development in the practice*. Springer Berlin Heidelberg, 2008.

[14] STMicroelectronics, "STM32G4 series advanced ARM-based 32-bit MCUs," 2020. [Online]. Available: https://www.farnell.com/datasheets/3182254.pdf. [Accessed Jun. 10, 2025].

[15] STMicroelectronics, "STM32G431x6 STM32G431x8 STM32G431xB Datasheets," 2019. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32g431c6.pdf. [Accessed Jun. 10, 2025].