Implementation of an Advanced AC Brushless Motor Controller for Use in High Reliability Applications

Michael E. Aiello

April 2, 2016

Keywords: Networked AC Drives, IPM's, EMI reduction, FPGA, Soft processing cores, SERDES, Custom Simulator, Space-Vector PWM

Abstract

There has been many papers devoted to the analysis and implementation of high performance AC Brushless motor control algorithms and control techniques. Most emphasize a state-space approach using a combination of feedforward and feedback linearization along with transformations, to reduce the plant to a set of simple first order differential equations. Within this new basis, the control reduces to a simple set of linear time-invariant equations. One such approach, elegant in it's method of plant reduction, is presented here. More importantly, this paper suggests an approach to the design and implementation of a networked drive and control system using a custom simulation environment that allows not only the motor and control algorithms to be modeled, but the characteristics of the drive electronics as well. The input to this simulator is in the form of a behavioral model of the controller and at in it's final stage the actual C Language code that would be placed in the various processing units of the control hardware. During all stages of development, the plant (motor, switching amplifier, and feedback transducers) is presented as a near exact mathematical model of the physical system. For the physical design of the drive and associated control hardware, the stress here is to choose components that minimize the size and cost of the drive and at the same time maximize it's performance and reliability. This document also presents information on the additional benefits of 3-Level (neutral point clamp) modulation compared to traditional 2-Level modulation.

1 Introduction

The importance of an exact (or near exact) mathematical model of the system to be controlled can never be over emphasized. However, in most cases the analysis of motor control systems seems to be biased more towards the validity

of the the motor model and control algorithms and less on the impact of the physical implementation of the controller. Most simulations are done using a simplified analog model of the servo amplifier with unbounded sources. While in the majority of todays application, PWM servo amplifiers using digital control loops, finite voltage buses and fixed frequency sampling are the norm.

For instance, finite PWM switching with transitional *deadtime* along with zero-order hold sampling and control, contributes to most of the deviation in performance between a *idealized* simulated environment, and an actual controller.

The realities are that medium and high power switching amplifiers are today, still limited to a 20 kHz switching frequency with finite on/off times that require the introduction of *deadtime* into the control circuit. At these frequencies, output power stage filtering is not an option if high performance is to be maintained.

Electro-Magnetic radiation produced by switching power stage devices is a on-going problem. The introduction of the *Intelligent Power Module* (IPM), has significantly reduced the amount of EMI caused by parasitic interaction between the drives IGBT's and associated control components. However, because of the monolithic nature of the IPM, module manufacturers have now been able to greatly increase the switching times of these devices. Because of this, the problem has now shifted to that of high EM radiation at the output of the drive.

Furthermore, standard techniques used for measuring phase currents in the motor are now loosing the ability to reject ever higher *dv/dt*, causing significant error to be introduced in the control algorithm.

In addition the optocoupler[1], which is the predominant technology used for isolating the drive signals between the IPM and control circuitry in low to medium voltage drives, is also becoming the victim of high switching *dv/dt*.

EMI issues are not limited just to the power and control stage interface. Effective and reliable communications between the drives and the host controller using *copper* medium is also falling victim to power stage output interference. Protocols ranging from simple data/strobe to 4B/5B decoding over industrial ethernet are all susceptible to this noise. One technology poised to become an effective communication medium in networked drive systems is *Glass Optical Fiber* (GOF) a technology that is becoming cost effective due to the integration of SERDES circuitry into FPGA's.

On another front, the *Digital Signal Processor* (DSP) has become the standard for the processing of control algorithms on most digital servo drives. Processors utilizing *Very Long Instruction Word* (VLIW) technology having multiple execution units running in parallel are common place. Most of these devices have enough internal memory to allow the entire control algorithm to be placed on the chip itself, greatly increasing execution through put. However, these devices are usually very inefficient when it comes to accessing external memory

[1]High speed/high common mode rejection optocoupler made there debut with the MOSFET in the early 80's. They are predominantly used on drives operating at 230 VAC but have recently been gaining acceptance in the 480 VAC market.

or peripherals such as *Field Programmable Gate Arrays* (FPGA) because the access method usually involves the use of a DMA scheduler[2]. The most effective DSP's in regards to price and performance, do not have the type of built in peripherals (analog and digital), that are necessary to implement the interface for even a low performance digital drive. Thus, one could find a very efficient algorithm for controlling motor torque running in internal memory at say 50 uSec, while using up to ten percent of the time accessing external circuits such as current transducers and FPGA logic.

2 A Minimalist Approach to Networked Drive Design

A reliable system design is based on the reliability of the subcomponents used and method in which they are used. In the case of modern servo design the de facto standard for power stage design is the Intelligent Power Module (IPM) as shown in Figure 1.

[2]In modern DSP implementations, even accesses made explicitly by code execution involves DMA scheduling.

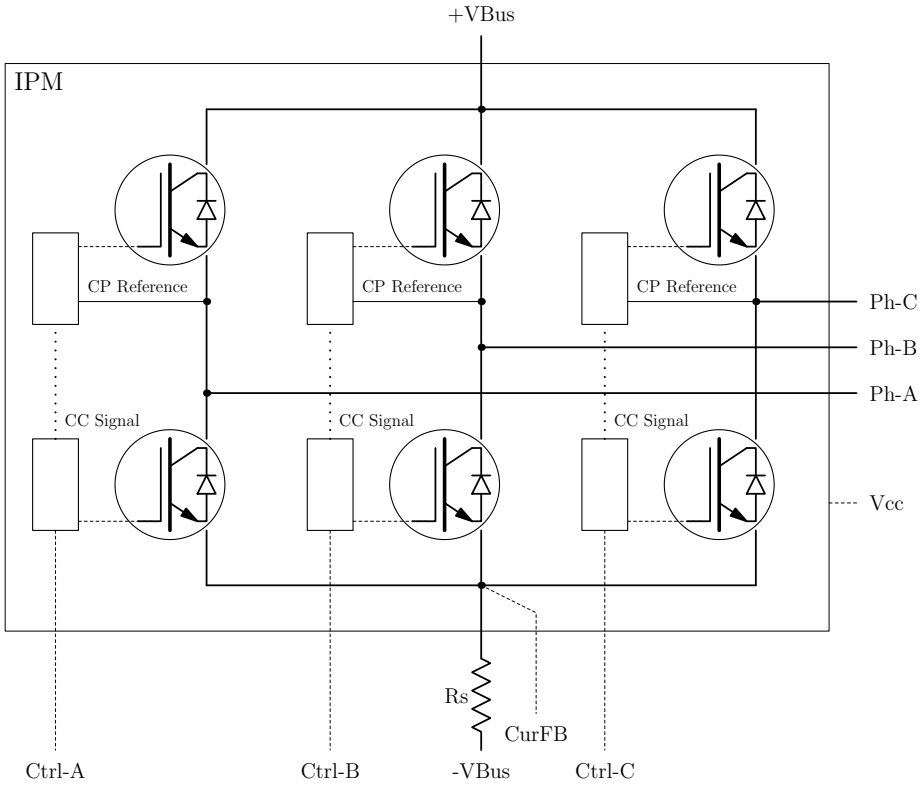

Figure 1: Simplified diagram of an Intelligent Power Module (IPM)

IPM's are used in very high quantities in *smart appliances*. The method in which these components are manufactured produces very low failure rates when used within there design constraints.

The module shown in Figure 1 is designed such that only one control power supply needs to be applied between *Ref* and *Vcc*[3]. The *Ref* connection is what is termed in industry, as the *Kelvin Tap* reference. This is point of reference relative to the IPM for which all control input and output connections *see* the least amount of common-mode signal interference.

The IPM'S use what is know as *HVIC level shifting* to control the high side IGBT's relative to *Ref* through control input signals *Ctrl-A, Ctrl-B and Ctrl-C*. This allows voltages as high as 340 and 720 VDC (depending on the part) to be applied between *+VBus and -VBus*.

In a basic design a shunt resistor *Rs* is usually attached to *-VBus* to sense high current flowing in the bus. Again, sensing is done relative to *Ref*. The circuit is completed by connecting a three phase AC motor to *Ph-A, Ph-B and*

[3]What is missing from this diagram is the *charge pump* diodes and capacitors that supply power to the high side IGBT's.

Ph-C.

In typical high performance servo amplifier designs, additional current transducers are also attached to *Ph-A and Ph-B*[4]. Two types of sensors are used, *hall-effect or sigma-delta optocouplers*. *Hall-effect* transducers are generally more sensitive to magnetic interference while *sigma-delta* are more sensitive to electric interference. These effects become more pronounced with increased *dv/dt's* introduced by the IPM.

One would ask if it possible to use the shunt resistor *Rs* as a sole sensing element for current in the motor, ridding the current feedback of *dv/dt* interaction caused by transducers referenced to *Ph-A* and *Ph-B*. Actually, this has been done for years in low performance AC drives, usually by restricting current flow in only two of the three motor phases at any given time, in a step wise manner. What is really needed however, is method to control current in all three motor phases sinusoidally, at all operating points using a single sense resistor.

Such a configuration is shown in Figure 2, a proposed *minimal part count*, high performance AC servo drive. In this figure, only the important components are outlined. For example, representative blocks such as the low voltage DC step-down converters used to power the FPGA's and buffer circuits for the encoder interface are not represented.

The IPM outlined in Figure 1 above, is now combined with representations of other circuitry to illustrate a single drive component of a networked drive system. Let us first describe each of the major components represented in Figure 2.

[4]In a balanced, unconnected neutral configurations, the sensing of Ph-C current is made redundant

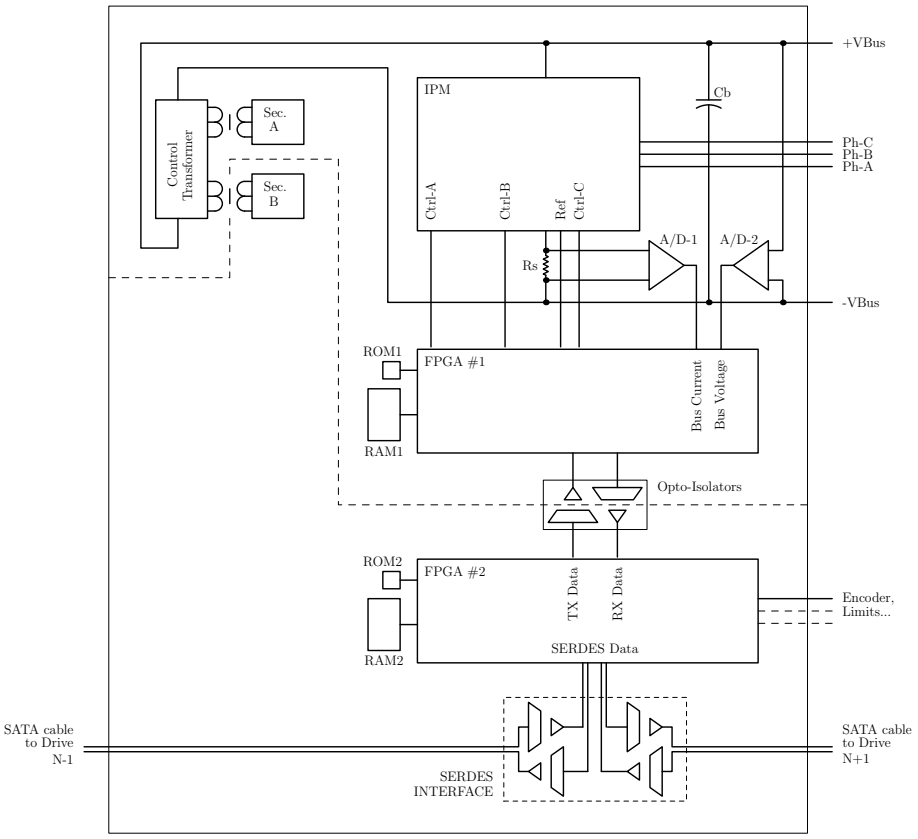

Figure 2: Simplified diagram of a high performance digital drive using minimal components.

IPM The *intelligent power module* which was describe in detail above. The bus voltage is supplied through *+VBus* and *-VBus*. The three phase servo motor is connected through *Ph-A*, *Ph-B* and *Ph-C*.

FPGA #1 A*Field Programmable Gate Array* referenced to the power side of the drive. In other words, referenced directly to the *Kelvin tap* of the IPM as described above such that all input and output signals to this device require **no** intermediate isolating components. Thus, the bus current sense resistor *Rs* and the bus voltage *+VBus* are referenced directly without isolation. As will be described in more detail later, this FPGA is responsible for running algorithms associated with current regulation and motor commutation. Power to this device is supplied indirectly by *Sec. A* described below.

FPGA #2 A*Field Programmable Gate Array* referenced to the control side of the drive. This reference is the *signal common* of the system which is tied indirectly to earth ground. As will be described later, this FPGA is responsible for running algorithms associated with velocity and position control of the motor. Here

encoder and limits can be connect directly[5] to the FPGA. Power to the this device is supplied by *Sec. B* described below.

ROM1/2 As mentioned above, each FPGA operates with multiple *soft* processing cores and associated state machines. In order to get this firmware *booted* into the FPGA through the SERDES connection, a start-up core must be loaded into each FPGA at power-up. These small serial ROM's provide the storage for these *boot* cores.

RAM1/2 This component is optional. Once the system is booted (through the SERDES interface) with the soft cores and hardware interface, additional area for code and data storage can be made availible. In the newer FPGA's there is usually enough internal memory within the gate array making this option unnecessary.

OPTO ISOLATION To insure a high level of galvanic isolation between the power side and control side of the drive, two opposing *TOSLINK* devices[6] are configured to operate in full duplex. Manchester clock-recovery encoding is used on both devices to limit the signal paths to one optical lane in each direction. The manchester encoding/decoding is done entirely in the hardware fabric of each FPGA.

SERDES Drive-to-drive and controller-to-drive communications is accomplished using standard SATA cables. The FPGA *(FPGA #2)* uses a custom protocol to communicate similar to EtherCat® .

A/D-1, A/D-2 Since both *FPGA #1* and the *IPM* are referenced to -*VBus*, only simple non-isolated A/D interfaces are needed to monitor current flow in *Rb* and measure the instantaneous bus voltage at +*VBus*. This data is then processed by the *soft* cores running in FPGA #1. In the newer Altera® and Xilinx® FPGA's, these A/D's are integrated into the chip.

CONTROL TRANSFORMER, SEC. A, SEC. B The drive input power is derived from a single DC source connected to +*VBus* and -*VBus*. Since the control circuitry for the drive are referenced at two potentials (-VBus and earth ground), two control power supply secondaries are required. These supplies are represented by *Sec. A* and *Sec. B*[7].

Capacitor Cb A nominal voltage of 320 DC is supplied through +*VBus* and -*VBus*. In a networked configuration, the proximity of the drives to the DC power supply depends on the length of the cabling. In applications such as a multi-axis gantry system used in machining cell, usually only a single DC bus power supply is needed. This supply may or may not be regulated and may or may not be capable of reverse power *regeneration* back in the the line source. However, the requirement is that it's terminal impedance must maintained at a minimum especially at high frequencies. With this in mind, only a relatively small value of bus capacitance on the drive is required (typically 2-10 uF at 100 kHz). This

[5] Again, sub-components such as line buffers, comparators and alike, are not shown in this figure. In addition, analog sin/cos encoder multiplication A/D converters maybe included as described below.

[6] A TOSLINK device is similar to a photocoupler except that a section of optical cable is used to link the emitter to the detector. The Toshiba TOTX1400(F) and TORX1400(F) are examples of such a device.

[7] The power supply is a simple isolated flyback design using a custom ferrite transformer and Power Integrations® TOPSwitch® .

requirement is important to keep the overall size of the drive as small as possible. The reason for the requirement of the drive having a small *footprint* will become evident below.

Before we analyze the drive system as a whole, it would be prudent to briefly discuss a controller that would be responsible for coordinating the control of the drives on the network. Such a unit is proposed and represented below.

One of the ideas here is to create a design of a central control board that can be used either *stand-alone*, or as a co-processor in PC Workstation environment. Both implementation cost and performance is an issue here especially when used in a stand-alone environment. Let us discuss briefly some key components of one such proposed design illustrated in Figure 3.

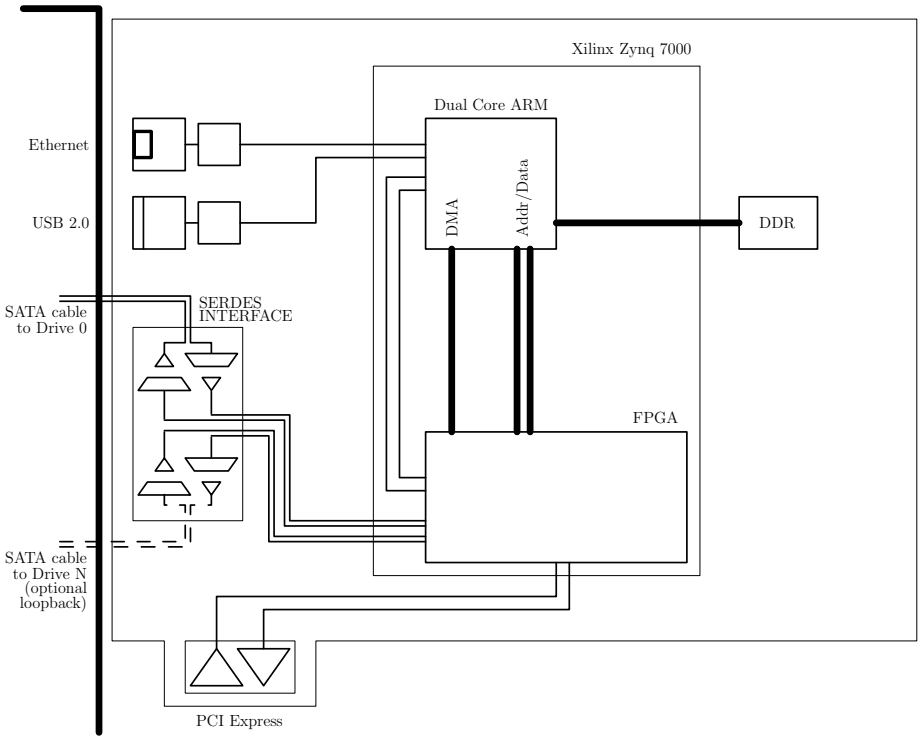

Figure 3: Simplified diagram of Control Board.

Xilinx Zynq 7000 Because of the requirement for *stand-alone* operation, this control board should contain a robust central processing unit capable of generating coordinated motion [8] for up to thirty two *drives* of the type described in Figure

[8] Trajectory generation should be generated using NURBS (paths and patches). One very interesting extension to this approach is vector velocity control using a technique known as Pythagorean-hodograph curves. Also, the availability of low cost, open-architecture CAD-CAM systems such as Rhino-3D® make this approach very attractive.

2 above. This processing unit should also provide means of running a full multitasking operating system such as Linux. The new Xilinx Zynq 7000® combines two ARM9 Cortex-A9® cores with integrated Ethernet and USB controllers seems a perfect choice. The Zynq chip has an integrated FPGA. Like the FPGA devices on the digital drives, the FPGA here should also be capable of running at least one *soft* processing core to coordinate the communications between the ARM processors and the SERDES interface. Since one of the requirements of this design is have the ability to operate in a PC environment, a PCI connection (PCI Express) is a requirement. The ideal place to coordinate PCI communication routing is within the FPGA since as shown in this design, the FPGA manages the SERDES communications and has direct port connections to the ARM modules. Also, logic can be implemented in the FPGA to allow the PCI Express to act as a bus master within the PC architecture. This would allow the direct processing of DMA descriptors such that a real time thread (or kernel) running on the PC could transmit and receive data packets to and from the SERDES bus directly. The PC would be responsible for all motion related tasks freeing the ARM processor for other non-motion related processes.

PCI Express A *single lane*, high speed serial communications port between the PC and the FPGA as described above.

Ethernet and USB When the control board is used used in a *stand-alone* enviroment, communications to the outside world can be achieved using standard Ethernet and USB coordinated by the Linux kernel running on one of the ARM portion of the cores within the Zynq.

SERDES INTERFACE In an identical manner to that described for the digital drive in Figure 2 above, SATA ports provides communications to the network drive system. Notice that only one port connects to the array of digital drives. The unused port is reserved.

Putting together the Control board and multiple Digital Drive boards described above yields a networked drive system illustrated in Figure 4 below.

Let us describe the components that make up this drive system.

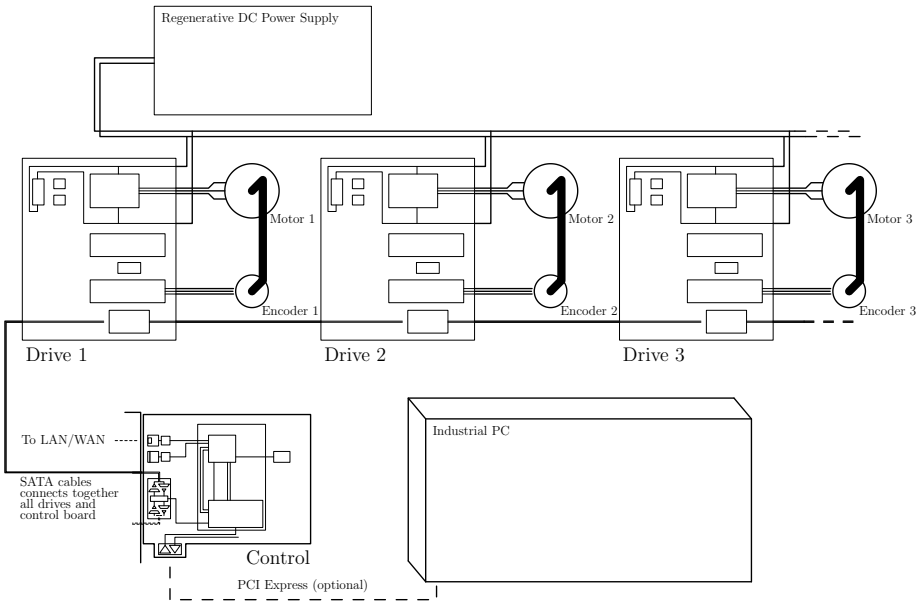

Figure 4: A networked drive system.

Drive 1,2,3... This is a chain of digital drives as illustrated in Figure 2 above which are connected using the SATA cables. For greater seperation between the control and drives, the SATA cable can be replaced with Glass Fiber Optical Cable with now change in communication protocol.

Control The control board as illustrated in Figure 3 above. The board can be operated in an embedded environment with connection to a LAN/WAN [9] or though a PCIe connection to an Industrial or Desktop PC.

Regenerative DC Power Supply A power source for the DC bus connections of each digital drive. Since each Digital Drive board provides minimum DC bus filtering, Litz wire[10] may need to connect the Digital drives to the DC supply. This power supply is designed to *source* or *sink* current depending on the operating conditions of the drives[11]. At the minimum, this supply may consist simply of a large electrolytic capacitor and bridge rectifier.

Motor 1,2,3... A rotary or linear three phase permanent magnet AC servo motor designed to be in close proximity to the respecitive Digital drive that connects to it. The merits of drive and motor being in close proximity to each other is described in greater detail below.

Encoder 1,2,3... A linear or rotary, square wave or sine wave quadrature, optical encoder associated with each Servo motor. Again, the encoder would be designed

[9] In an embedded environment, the Control board would need to be connected to a 3.3 VDC power supply.

[10] A specialized wire designed for high frequency applications.

[11] A detailed description of this power supply is beyond the scope of this document.

to be in close proximity to the associated motor and drive. This is also described in greater detail below.

The system depicted in Figure 4 shows only the interconnections between the control and drives, with simple representations of servo motors and encoder feedback transducers.

Based on the description of the Digital Drive illustrated in Figure 2 and it's associated board components that make up the control and power sections of the drive, one can estimate the volume required to construction such a drive given the power requirements of the application.

Let us assume a drive design with a peak current of 30 amps and a continuous current of 15 amps with a maximum DC bus voltage of 320 VDC. Such a drive would have a continuous rating of roughly 5 HP. The drive system would then be adequate for applications like:

- Torque motor control in small to medium size robotic applications requiring three or more degrees of motion.
- Gantry systems requiring drives for X,Y,Z stages and possible requirements for additional axis's that control or sense rotation and pitch.

Based on the component descriptions for the proposed digital drive of Figure 2 and given the power requirements stated above, this drive could easily be condensed into a volume no larger then 125x75x50 centimeters [12].

Let us apply this new design to a gantry system similar to the that described above.

Typically, a system such of this consists of a base structure (lets call it the X axis) with dual sliding members attached to a rigid base structure. At this power range usually each of the sliding members contains a linear motor and associated drive. Both members are controlled as an *single* axis. This requires a tight coordination of control and trajectory generation to each of the drives [13].

Another sliding member with linear motor is attached on top of the two *base* members, in a perpendicular fashion (we will call this the Y axis) and usually requires only one drive.

Next comes a vertical sliding member which is usually a *stage* (we will call this the Z axis). This member is usually driven by a linear motor as well.

Finally, depending on the application additional axis (we will call these axis *alpha, beta, gamma*) are mounted to the Z member. These axis may use linear and/or rotary servo motors.

In a tradition design, this application would require at least three *cable trays* to bridge the motor power, encoder connections, and status connections (e.g., *limit switch* signals) between each of the sliding members of the gantry and the central control unit.

The central control unit would usually be a NEMA style enclosure containing drive amplifiers and controller.

[12] The required size and mass of the heatsink is not included in this calculation. The reason for this is explained later.

[13] The update rate of the SERDES interface ensures this tight coordination.

Herein lies the problems associated with this type of gantry control implementation:

- The central control unit is usually a NEMA style enclosure containing drive amplifiers and controller situated away from the mechanical structure. Cabling for both power transmission (motor power) and control signaling (encoder feedback and limits) for *all* axis's enter and exit this cabinet.
- At least two cable trays are required to transfer the motor power, encoder feedback and status signals between the X and Y member and also the Y and Z member of the gantry. For say a six axis implementation, the first cable tray (X to Y) would carry six sets of three phase motor power cables, along with six sets of encoder and limit switch connections. Likewise, the second cable tray (Y to Z), would carry five sets of these connections.
- High performance dictates that the PWM operation at the 320 VDC bus be unfiltered for each of the drives and as such, high dv/dt is present on all the motor power cabling that travels through the cable trays to each motor. This exposes all the low voltage signaling (e.g., the encoder feedback signals) to excessive EMI. In addition the structure of the gantry (usually extruded and machined aluminum) acts as a wave guide to the EM radiation.
- The cable trays and wire they contain adds to the inertia of the Y and Z members, reducing the overall performance of the system.
- Expensive motor and encoder cabling is required at the points where the wire enters and exits the cable trays. The higher gauge motor wiring must be flexable enough to withstand repetitive bending without insulation breakdown. The low voltage signal wire must be heavily shielded to deflect the EM radiation of the motor leads. But like the motor wire, this cabling must also be flexable enough to withstand repetitive bending without breakdown.

Now, replacing the traditional implementation of drives and control for the gantry system described above, with a drive and control system depicted in Figure 4 and replacing the SATA cables with a glass optical fiber (GOF) interface would yield the following.

- The central control unit is a NEMA style enclosure that house only the controller and regenerative DC Bus power supply. The digital drive depicted in Figure 2 (and portrayed in a schematic form in Figure 4) are mounted to each of the moving members of the gantry. In other words, each digital drive is mounted to the structure containing the associated linear or rotary AC Servo motor. In most applications, the heat generated by the motor is much greater then the losses of the IPM module. Thus, the structure holding the motor can itself be the heatsink to the digital drive.
- The two cable trays connecting the X and Y member and the Y and Z member can now be greatly reduced in size since only a single DC Bus Power cable and plastic fiber-optic cable traverse the distance between the first motor (axis X) and the last motor (axis *gamma*)
- The length of the conducting wires between the output connections of the digital drive and the connection to associated motor are greatly reduced (depending on the design, as little as a few inch's). This greatly reduces the dv/dt coupling issues mentioned above.

- System inertia is greatly reduced. The added mass of the digital drives mounted to the stages themselves is small compared to the mass of the cabling and large cable trays of the traditional design.

Additional improvements in performance are realized when the drive is mounted next to the motor and hence, next to the position feedback encoder, especially if the encoder produces sine wave output signals. Usually this type of encoder is used with *position multiplication* circuitry.

The digital drive depicted in Figure 2 is ideal for processing the analog (sine/cosine) signals directly on the board through *FPGA #2*. With on board analog processing circuitry (not shown in Figure 2) the high speed multipliers contained in *FPGA #2* can be used for position feedback multiplication. Here, the short length of cable between the encoder and the analog processing components on the board provides a great benefit.

In the tradition gantry design, this cable would run from the encoder through the cable trays to the NEMA enclosure adding *noise* into the torque control algorithm. The usual solution would be to add complex digital *notch* filters into the control algorithm that in most cases degraded overall positioning performance.

Thus, the drive system proposed in this section not only reduces the physical complexity, but also improves performance.

Finally, it should be obvious form the diagram of the digital servo amplifier depicted in 2 that one need only add a heatsink, a DC bus power supply, and enclosure around the PC board to revert to a traditional AC Digital Drive. The actual design of the module would take this into account so that only one PC board design is required for both new and legacy applications.

As illustrated above the physical design of the drive system by itself can contribute to increased performance especially if by the nature of the design, electrical *noise* can be reduced.

This *noise* can also be reduced by selecting a control scheme that can provide modes that are dynamically tailored for the operating points of the motor at any given instant.

3 A Layered Development Simulator

The introduction of VHDL formalized the notion of a layered approach to digital circuit design. The designer had the option of starting with an abstract realization of an idea using the simple constructs provided by the VHDL language (e.g. a *behavioral* representation of the design).

In the case of the design of say a hardware implemented *control* algorithm, both the *plant* and the *control* could be modeled at the behavioral level. The *plant* is usually considered part of the *testbench* in a design simulation. After the control and plant were constructed, a simulation of both could be run as a proof of concept in what is termed as *behavioral* mode. The next step would usually be a reformulation of the control from behavioral level to a *structural* level which usually (but not always) introduced timing constraints into the control portion

of the design. At the structural level, the control and plant would once again be combined and a simulation run.

In the early days of VHDL design, a third and final reformulation of the control to the *register transfer level* would be required, followed by simulation. Upon success, this level was used to construct the actual implementation of the design. With increased power of todays VHDL compilers, manual reformulation to the register transfer level design is no longer required. The final design can usually realized at the *behavioral* level. Hence, the structural representation of the design (together with *cell* libraries), was used as the framework for the design.

Note that through out the steps above, the *plant* portion of the design (which was need for simulation), remained at the *behavioral* level.

Thus, one may think it beneficial to design a control simulator with some of the same characteristics outlined above.

- The ability use a *proof-of-concept* description of the *control* using high level **C++** language constructs.
- To be able to provide a model of the texitplant, also written in **C++**.
- The plant be represented as a list of ordinary differential equations that is executed using a, variable time step sixth order Runge-Kutta method. We shall call these elements *OdeObject's*.
- The capability to solve *coupled* ordinary differential equations.
- The ability to generate accurate command signals, both continuous and discontinuous in nature as a function of the time reference to the Runge-Kutta solver. We shall call these elements *SrcObject's*.
- The capability of executing blocks of sequential statements written in **C** that represent the control (or set of concurrent controls) effecting the plant. In the early stage of development, the statements associated with a given control block can be written in abstract form, similar in idea to the behavioral level of VHDL mentioned above. As the design progresses, these blocks become more specific and refined such that they become the actual code that is executed in the various *soft processing cores* of the *FPGA #1* and *FPGA#2* represented in Figure 2 above[14]. We will call these elements *CtrlObject's*.
- A method of coordinating the timing of the OdeObject, SrcObject, and CtrlObject's such that a sudden changes in the output of a given SrcObject does not disrupt the accuracy of the Runge-Kutta solver. The simulator must cache all the states of the previous iteration so that when required, a recalculation stating at the minimum time step can be made.
- A robust and versatile platform in which to run the simulation. Being that the framework of this simulator is **C++**, it makes sense to use an open architecture such as GNU g++/gdb. Speed of simulation requires that the OdeObject's run as compiled **C** (e.g. not interpreted). However, the complexity of the **C++** object oriented environment dictates that and intermediate representation be adopted to shield the user from the tedious task of writing class constructors

[14]Most importantly throughout this process, the description of the plant remains in it's original (and hopefully accurate) abstract form.

and alike. Thus we also introduce into this development environment, the GNU Flex/Bison parser and compiler generating tools.

To gain confidence as to the accuracy of this simulator, a test circuit was devised that incorporated some of the characteristics of a multi-phase AC motor, such as *coupled* inductive elements. The circuit also needed to be suitable for simulation on the *benchmark* simulator selected for cross comparison. The benchmark simulator selected was PSpice®.

Figure 5 shows a diagram of the test circuit.

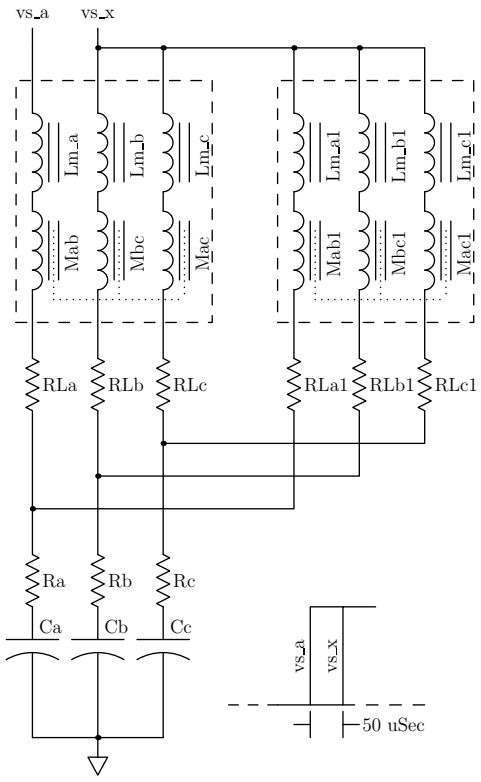

Figure 5: Dual CM choke test circuit

The equations for the circuit shown in Figure 5 are provided in (1) through (9).

$$\frac{dq_a}{dt} =$$
$$i_a + i_{a1} \tag{1}$$
$$\frac{dq_b}{dt} =$$
$$i_b + i_{b1} \tag{2}$$
$$\frac{dq_c}{dt} =$$
$$i_c + i_{c1} \tag{3}$$
$$(Lm_a + L_a)\frac{di_a}{dt} + M_{ab}\frac{di_b}{dt} + M_{ac}\frac{di_c}{dt} =$$
$$vs_a - RL_a i_a - R_a(i_a + i_{a1}) - \frac{q_a}{C_a} \tag{4}$$
$$M_{ab}\frac{di_a}{dt} + (Lm_b + L_b)\frac{di_b}{dt} + M_{bc}\frac{di_c}{dt} =$$
$$vs_b - RL_b i_b - R_b(i_b + i_{b1}) - \frac{q_b}{C_b} \tag{5}$$
$$M_{ac}\frac{di_a}{dt} + M_{bc}\frac{di_b}{dt} + (Lm_c + L_c)\frac{di_c}{dt} =$$
$$vs_c - RL_c i_c - R_c(i_c + i_{c1}) - \frac{q_c}{C_c} \tag{6}$$
$$(Lm_{a1} + L_{a1})\frac{di_{a1}}{dt} + M_{ab1}\frac{di_{b1}}{dt} + M_{ac1}\frac{di_{c1}}{dt} =$$
$$vs_{a1} - RL_{a1} i_{a1} - R_a(i_a + i_{a1}) - \frac{q_a}{C_a} \tag{7}$$
$$M_{ab1}\frac{di_{a1}}{dt} + (Lm_{b1} + L_{b1})\frac{di_{b1}}{dt} + M_{bc1}\frac{di_{c1}}{dt} =$$
$$vs_{b1} - RL_{b1} i_{b1} - R_b(i_b + i_{b1}) - \frac{q_b}{C_b} \tag{8}$$
$$M_{ac1}\frac{di_{a1}}{dt} + M_{bc1}\frac{di_{b1}}{dt} + (Lm_{c1} + L_{c1})\frac{di_{c1}}{dt} =$$
$$vs_{c1} - RL_{c1} i_{c1} - R_c(i_c + i_{c1}) - \frac{q_c}{C_c} \tag{9}$$

Figures 6 and 7 provide plots of the voltage across capacitor C_a for a voltage change of 100 volts applied at the inputs to the circuit. This voltage is applied to the two inputs (labeled *vs_a* and *vs_x*) in a *staggered step* fashion as shown in Figure 5.

As is apparent in Figures 6 and 7 the simulation runs for both this new development simulator and the PSpice simulator show identical results.

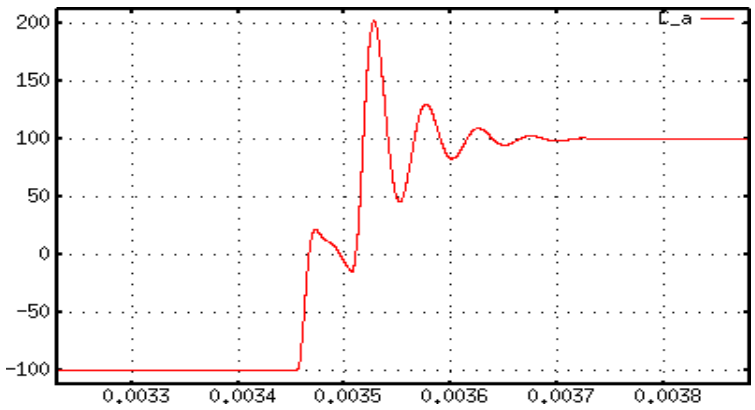

Figure 6: The Development simulation plot for CM choke test circuit.

Figure 7: The PSpice simulation plot for CM choke test circuit.

4 Extensions to Full State Feedback Control

The development of a full state feedback controller for the AC Brushless Servo motor[2] can be made simply by minor modification to the closed loop stepping motor control[1] outlined in the appendix B. The equation for the brushless AC motor are

$$
\begin{aligned}
L_a\frac{di_a}{dt} - M_{ab}\frac{di_b}{dt} - M_{ac}\frac{di_c}{dt} &= v_a - R_a i_a + K_a\omega\sin(N_r\theta) - v_n \qquad (10)\\
-M_{ab}\frac{di_a}{dt} + L_b\frac{di_b}{dt} - M_{bc}\frac{di_c}{dt} &= v_b - R_b i_b + K_b\omega\sin(N_r\theta - 2\pi/3) - v_n\\
-M_{ac}\frac{di_a}{dt} - M_{bc}\frac{di_b}{dt} + L_c\frac{di_c}{dt} &= v_c - R_c i_c + K_c\omega\sin(N_r\theta - 4\pi/3) - v_n\\
J_m\frac{d\omega}{dt} &= -K_a i_a\sin(N_r\theta)\\
&\quad - K_b i_b\sin(N_r\theta - 2\pi/3)\\
&\quad - K_c i_c\sin(N_r\theta - 4\pi/3)\\
&\quad - B_m\omega - C_m sgn(\omega) - D_m sin(2N_r\theta)\\
\frac{d\theta}{dt} &= \omega
\end{aligned}
$$

These three motor phase equations constitute a set of coupled ODE's, unlike the equation for the stepping motor specified in Appendix B. This is because a motor neutral v_n ties all three phases together allowing current to flow from one phase to the other. The simulator also allows imbalances and disturbances to be introduced into the system to be represented (e.g. individual phase inductance L_c and mutual inductance M_{ab} as well as *stick-tion* $C_m sgn(\omega)$, detent torque $D_m sin(2N_r\theta)$ and so forth).

If we were to remove the imbalances and disturbances, we end up with a much simpler set of equations, allowing v_n to be set to zero

$$
\begin{aligned}
L\frac{di_a}{dt} - M\frac{di_b}{dt} - M\frac{di_c}{dt} &= v_a - Ri_a + K\omega\sin(N_r\theta) \qquad (11)\\
-M\frac{di_a}{dt} + L\frac{di_b}{dt} - M\frac{di_c}{dt} &= v_b - Ri_b + K\omega\sin(N_r\theta - 2\pi/3)\\
-M\frac{di_a}{dt} - M\frac{di_b}{dt} + L\frac{di_c}{dt} &= v_c - Ri_c + K\omega\sin(N_r\theta - 4\pi/3)\\
J\frac{d\omega}{dt} &= -Ki_a\sin(N_r\theta)\\
&\quad - Ki_b\sin(N_r\theta - 2\pi/3)\\
&\quad - Ki_c\sin(N_r\theta - 4\pi/3)\\
&\quad - B\omega\\
\frac{d\theta}{dt} &= \omega
\end{aligned}
$$

This allows us to perform a transformation similar to that applied to set of stepping motor equations of Appendix B. By making the assumption that $M = \frac{1}{2}L$ (no leakage inductance) and using the 3 phase - 2 phase transformation

$$\begin{aligned} \psi_{a'} &= \psi_a - \frac{1}{2}\psi_b - \frac{1}{2}\psi_c \\ \psi_{b'} &= \frac{\sqrt{3}}{2}\psi_b - \frac{\sqrt{3}}{2}\psi_c \end{aligned} \tag{12}$$

which applied to (11) yields

$$\begin{aligned} (L+M)\frac{di_{a'}}{dt} &= v_{a'} - Ri_{a'} + \frac{3}{2}K\omega\sin(N_r\theta) \\ (L+M)\frac{di_{b'}}{dt} &= v_{b'} - Ri_{b'} - \frac{3}{2}K\omega\cos(N_r\theta) \\ J\frac{d\omega}{dt} &= -Ki_{a'}\sin(N_r\theta) + Ki_{b'}\cos(N_r\theta) - B\omega \\ \frac{d\theta}{dt} &= \omega \end{aligned} \tag{13}$$

Finally, we simply apply the DQ transformations defined in (16) and (17) of Appendix B to (13) and defining $L_{eq} = L + M$ or $L_{eq} = \frac{3}{2}L$ yields

$$\begin{aligned} L_{eq}\frac{di_d}{dt} &= v_d - Ri_d + \omega N_r L_{eq} i_q \\ L_{eq}\frac{di_q}{dt} &= v_q - Ri_q - \omega N_r L_{eq} i_d - \frac{3}{2}K\omega \\ J\frac{d\omega}{dt} &= Ki_q - B\omega \\ \frac{d\theta}{dt} &= \omega \end{aligned} \tag{14}$$

Since it has been easily demonstrated above that with a few simplifications, the transformed model of the AC brushless motor can be made to look nearly identical to the stepping motor model[1].

Thus, the general principles of optimal control for the stepping motor can be applied to the AC brushless motor. Hence we only need to review Appendix B of this document to arrive at conclusions as to the best approach for controlling the AC brushless motor.

Assuming this is correct, we will conclude this section by making observations as to improvements to the system that apply more to the hardware design aspects of the controller.

Using the custom simulator described in Section 3 above, a system was constructed to model the AC Brushless motor running solely on the feedforward reference commands described by equations (35) and (36) of Section B.2.2, and

a model of the switching amplifier power stage illustrated in Figure 2. For the entire discussion that follows, only the references described by equations (35) and (36) are used to drive the simulated AC brushless motor (no feedback control is implemented for these tests).

Before continuing, we must define the meaning of each waveform in the waveform plots which follow, for the outline of Figure 2, and for selected equations (15) through (42) in Appendix B. This information is provide in Tables 1 and 2.

Plot Label	Equation Variable	Description
omegad	ω_d	Reference Velocity.
Omaga	ω	Actual Velocity.
vdd	v_{dd}	Desired *direct* voltage.
vqd	v_{qd}	Desired *quadrature* voltage.
idd	i_{dd}	Desired *direct* current.
iqd	i_{qd}	Desired *quadrature* current.
VCmd_a	v_a	Commanded Phase A motor voltage.
VCmd_b	v_b	Commanded Phase B motor voltage.
Ia	i_a	Actual current flowing in Phase A of motor.

Table 1: Plot definitions relative to the variables in the equations listed in Section 4 below

Plot Label	Description
VBus_a	Motor phase connection at terminal *Ph-A*.
VBus_b	Motor phase connection at terminal *Ph-B*.
VBus_c	Motor phase connection at terminal *Ph-C*.
Ia	Current flowing through motor phase connection *Ph-A*.
Ia_bus	A reconstruction of the current flowing through motor phase connection *Ph-A*, derived from the actual current flowing through the minus side bus resistor *Rs*.

Table 2: Plot definitions relative to circuit connection in Figures 1 and 2

4.1 Program Description

4.1.1 Reference Generator

As previously stated, this simulation controls a motor driven only by the reference (feedforward) elements of the control equations. Adding the code necessary for simulating the entire control system for handling error cancellation would have complicated the system and is beyond the scope of this document.

As stated above, the purpose of this document is emphasize and analyze in detail, the impact of an actual hardware design on the performance of a motor control system. The elements related to the simulation of the actual control algorithms are left as a future exercise.

For reason of simplicity, a simple *modified sine* ramping profile is implemented. A simulation run of approximately one second is selected. The commanded reference velocity *omegad* (ω_d) as well as the commanded direct and quadrature voltages v_{dd} and v_{qd} are plotted in Figure 8.

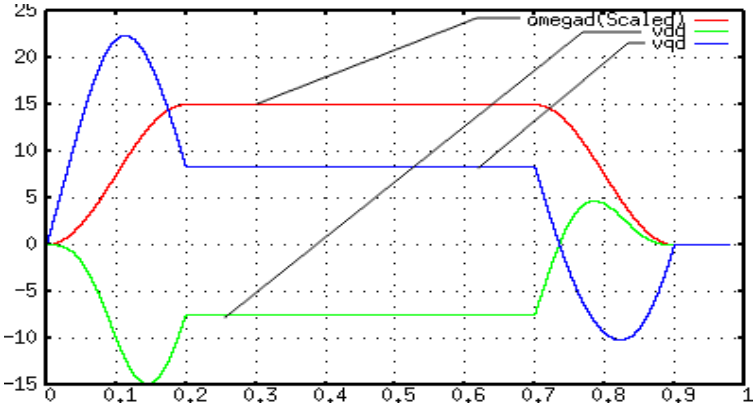

Figure 8: Plot of quadrature and direct voltage command with velocity reference.

Currents i_{dd} and i_{qd} withω_d are plotted in Figure 9.

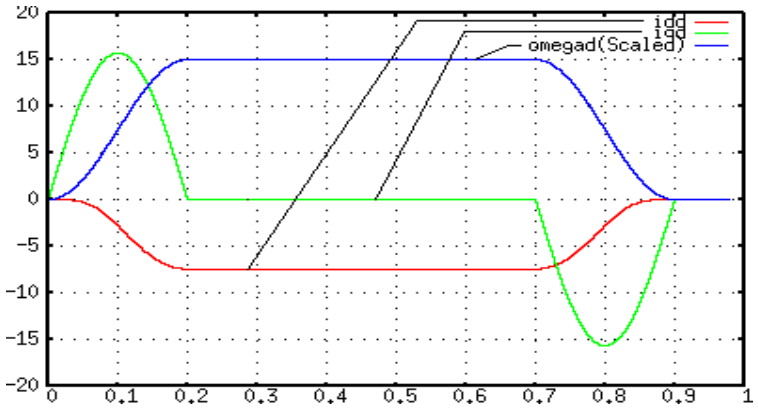

Figure 9: Plot of quadrature and direct current command with velocity reference.

The direct and quadrature voltage and current commands (v_{dd}, v_{qd}, i_{dd}, i_{qd}) of Figure 8 and Figure 9 are derived using equations (31) through (36) in Appendix B.2.1. The reference generator was setup to run at a periodic rate of one iteration every 50 uSec. The value ω_d was scaled to 15 percent of it value to bring the plots into perspective.

Field weakening effects are evident in the voltage and current plots (the *humps* on i_{dd} and v_{dd}) .

The commanded phase voltages for producing v_a and v_b (labeled *VCmd_a* and *VCmd_b* in the plot) are the voltage command signals in the physical plane after the inverse D/Q transformation). These signals are shown in Figure 10. The simulator is setup to operate in Space-Vector PWM mode with *minus side clamping*. In this mode, at any given time there are only two phases out of three that that control voltage. The third phase is clamped to the minus side bus *-VBus* as shown in Figure 1.

There is a very important aspect to the switching characteristic of the Space-Vector PWM *minus side clamping* that is not mentioned it the general literature describing Space-Vector modulation.

As the command amplifier voltage drops to zero, the PWM amplifier switching waveforms go to zero. This has important implications relative to the reduction (and possibly elimination) of switching noise at conditions when the servo control is *in position* and not influenced by external forces (e.g. a horizontal air-bearing application).

Another very important aspect of this type of modulation, is that one of the three phases is always connected to the *-VBus* while the other two transition between the *-VBus* state, the *dead time* state that protects against *shoot-through* and the *-VBus* state.

This makes the common mode of the three phase motor winding as a whole connected to *-VBus* at any given point in time.

In the case of traditional carrier based PWM mode, there are points where all three motor phases are simultaneously in the *dead time* state. This condition can lead to audible noise and more importantly noise that can cause disturbances on the motor torque.

In other words, one can draw a line through the switch transitions on phases A, B and C and see that there are short periods of time where neither of the phases are referenced to either the plus bus or minus bus.

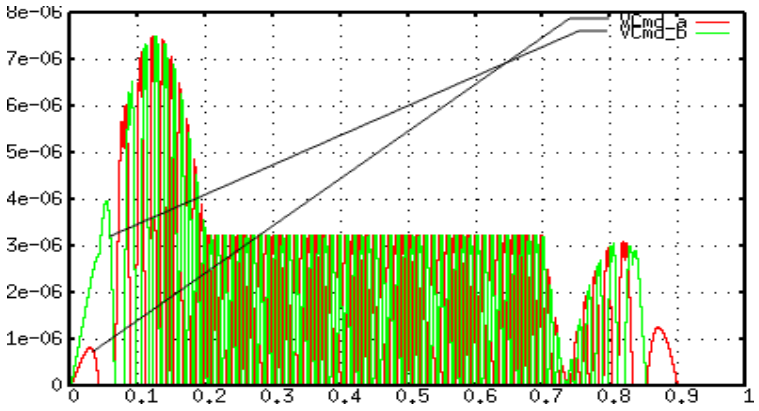

Figure 10: *VCmd_a* and *VCmd_b* voltage commands in the physical plane produced by *vdd* and *vdq* described above and defined in Table 1.

A detailed view of Figure 10 (zoomed in) is shown in Figure 11. The shape of these signals are due to nature in which the Space-Vector PWM algorithm selects the instantaneous reference commands that are to be compared to a reference carrier signal (a triangle wave signal).

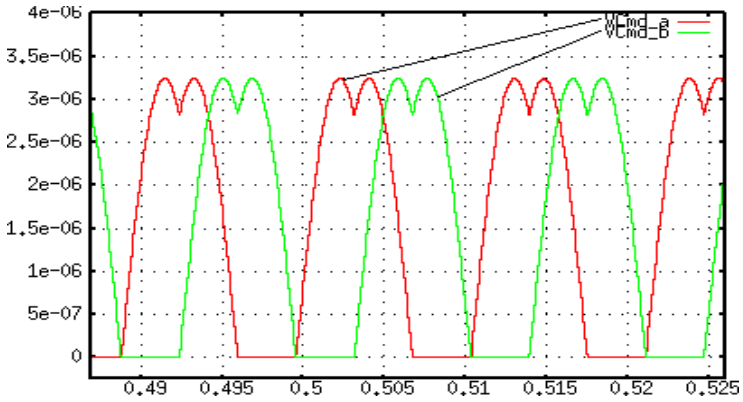

Figure 11: Close up view of the signals *VCmd_a* and *VCmd_b* in Minus side clamped Space-Vector PWM Mode

The simulator can also be setup to operate in traditional carrier based PWM mode. In this case, the command phase voltages or producing v_a and v_b take on the appearance shown in Figure 12.

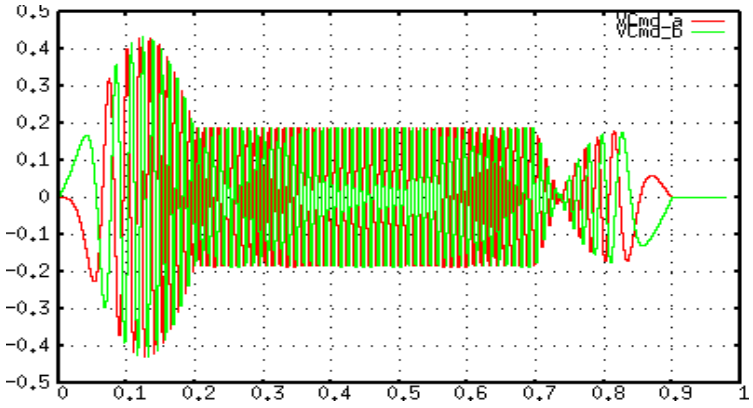

Figure 12: View of the signals *VCmd_a* and *VCmd_b* in traditional carrier based PWM Mode

A zoom-in of the actual switching phase voltages v_a and v_b for Space-Vector PWM *minus side clamping* and traditional carrier based PWM modulation near the beginning of acceleration of the test run is shown in Figures 13 and 14 respectively.

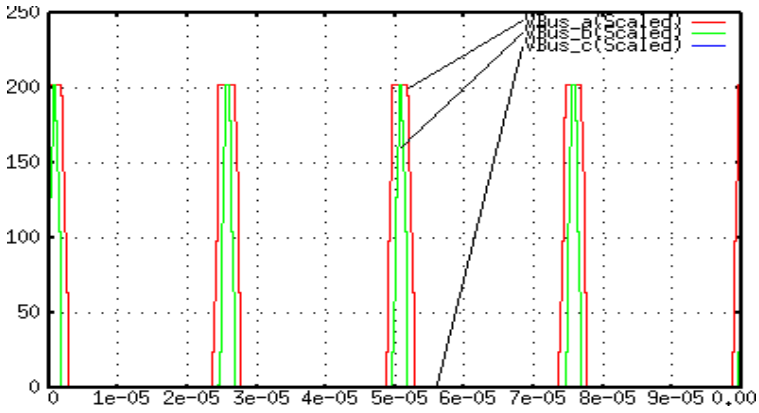

Figure 13: Close up of switching voltages in minus-side Space-Vector mode at the beginning of motor acceleration.

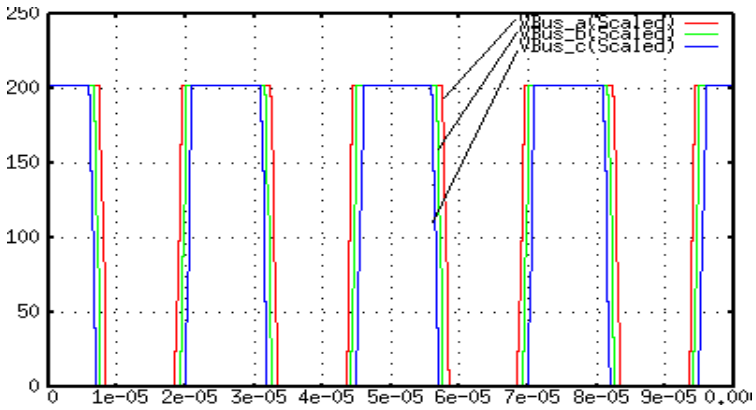

Figure 14: Close up of switching voltages in carrier based mode at the beginning of motor acceleration.

Again, it can be seen that for minus-side Space-Vector mode, the switching voltage to the motor starts at zero. However, in the case of carrier base PWM, switching starts modulation at 50 percent duty cycle for all three motor phases.

Finally, a plot of the actual motor velocity *Omega* (ω) verses the command reference velocity *omegad* (ω_d) is shown in Figure 15. The plot for Figure 15 is done with the simulation set for minus-side clamped Space Vector PWM mode. A virtually identical plot can be shown with the simulator setup to operate in the traditional carrier based PWM mode.

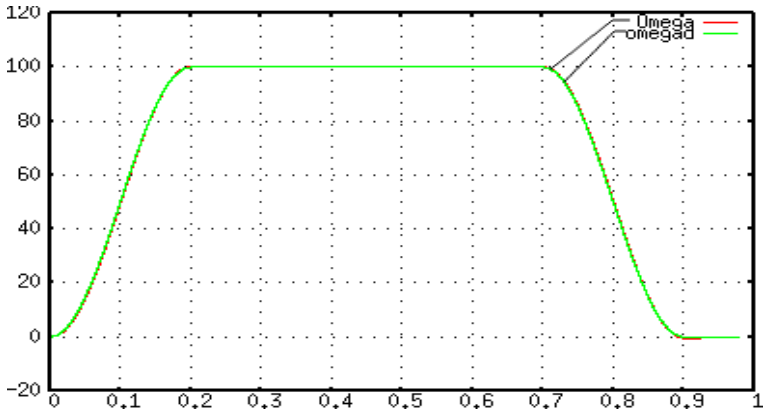

Figure 15: Actual motor velocity *Omega* (ω) and command reference velocity *omegad* (ω_d) for the test run described above.

A strait forward explanation of Space-Vector PWM can be found here [3]. A careful analysis of this document will reveal to the reader that the algorithm relies on modulation around what is termed as a *zero vector*. In a simple PWM three phase bridge (two level) configuration, there are two zero vectors, the state where all three of the phases are connected to the +VBus and the state where all three of the phases are connected to the -VBus.

As stated above, I have proposed a design based on a *zero vector* reference where all three of the phases are connected to the -VBus. This allows the addition of three small LC filter circuits connected from each phase (Ph-A, Ph-B and Ph-C) with reference to the -VBus to be added to the circuit of Figure 1. As the command voltage drops to near zero (e.g. as the servo motor comes into position), the PWM voltages of Figure 13 with the help of the small LC filters vanish.

In our example all switching voltages are referenced to -VBus. It turns out that in Space-Vector PWM (assume two levels), the switching reference can be a point anywhere between -VBus and +VBus. In this case both the -VBus zero state and +VBus zero state combine as *zero* vector for the switch period. Zero voltage is defined when the switching waveforms of Ph-A, Ph-B and Ph-C are aligned. This leads to a reference of another more advanced paper on Space-Vector PWM that deals with the selection of the optimal *zero vector reference* based on a given operating point. In essence this paper proposes a method to determine the optimal *zero vector reference* such that the switching ripple

current is at a minimum. This paper can be found here [4].

4.1.2 Deriving Motor Current from a Single Resistor

The power stage circuit used in this simulation proposes the use of a single current sense resistor *Rs* for deriving current flow in phases *Ph-A*, *Ph-B* and *Ph-C* (see Figure 1 and Figure 2). Currents i_a, i_b and i_c are derived from objects responsible for the simulation of the electrical portion of the motor model (see Equation 10). An example of the ODE simulation equation that is responsible for deriving i_a is documented in Section A.2.1 (OdeObject Ia_Ib_Ic_1).

The current flow through *Rs* is represented by the simulation variable *Ia_bus* (see Table 2) and is created using the simulation code documented in Section A.2.2 (OdeObject ISample). The OdeObject ISample also produces the reconstructed phase currents *Ib_bus* and *Ic_bus*.

The use of a single current sense resistor contained within the bus has both benefits and drawbacks[15]. The benefit is a simplified circuit with one current sense element that requires no level translation or isolation for processing within the FPGA.

Due to the placement of resistor *Rs* in the circuit there are two conditions however that prevent the ability to determine the flow of i_a, i_b and i_c. These condition are when the PWM is in the *zero vector* state in which all of the top switches are turned or all of the bottom switches are turned on.

A plot of i_a for the entire simulation run depicted in Figure 16 with a close-up of i_a and the reconstructed phase current *Ia_bus* shown in Figure 17 Figure 18 below.

[15] In the case where the three phase bridge is implemented using an IPM as shown in Figure 2, a single resistor is our only choice when sensing current at -VBus

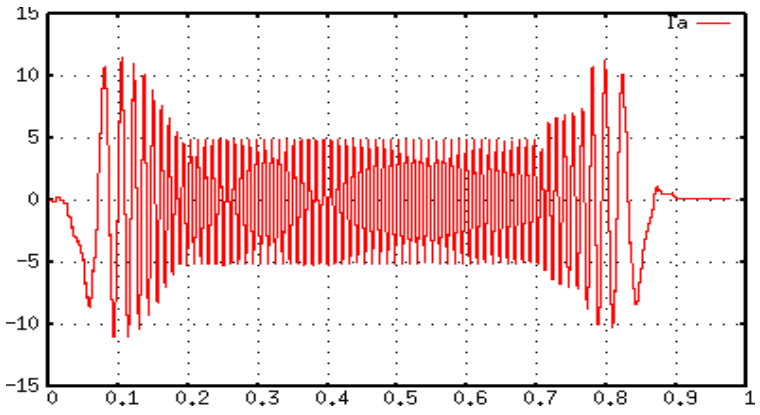

Figure 16: Phase current i_a for entire simulation run.

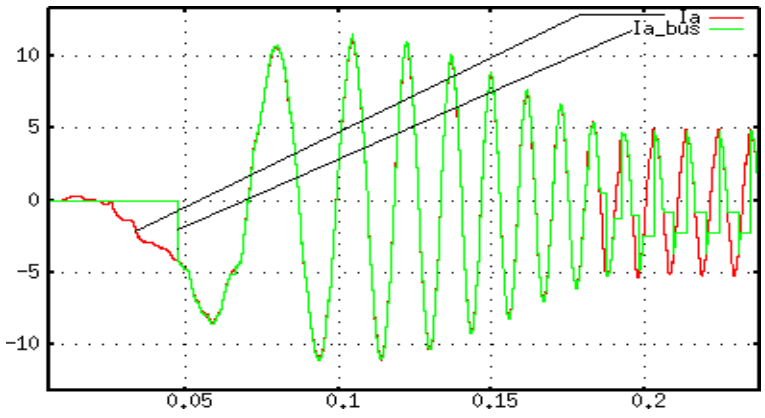

Figure 17: Close-up of Phase current i_a and reconstructed current *Ia_bus* at the start of motor acceleration.

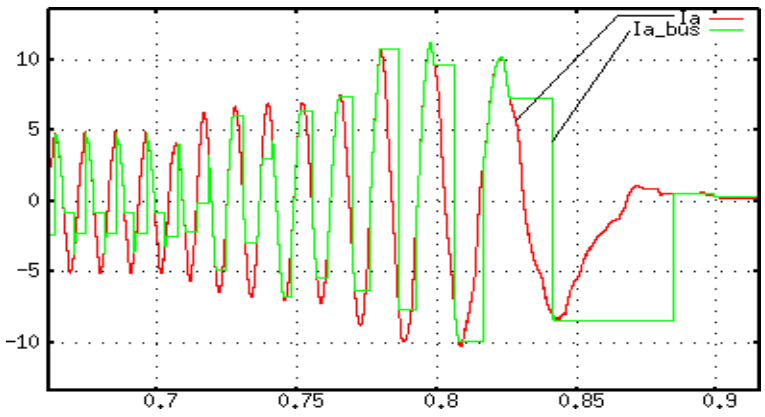

Figure 18: Close-up of Phase current i_a and reconstructed current *Ia_bus* at the start of motor deceleration.

As can be seen in Figure 17 Figure 18 when the switching signal for Ph-A collapses to a point where the bottom IGBT switch of Ph-A can barely turn on, the current cannot be determined by the sampling algorithm. This effect is shown more clearly in Figure 19 and Figure 20.

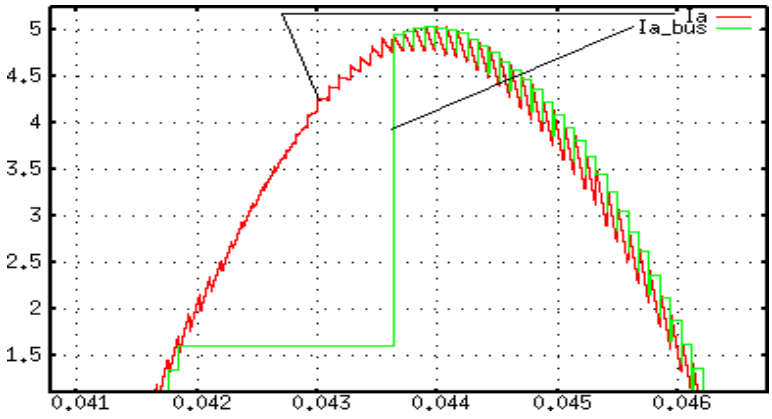

Figure 19: Loss of reconstructed current *Ia_bus* due to inadequate sample time duration.

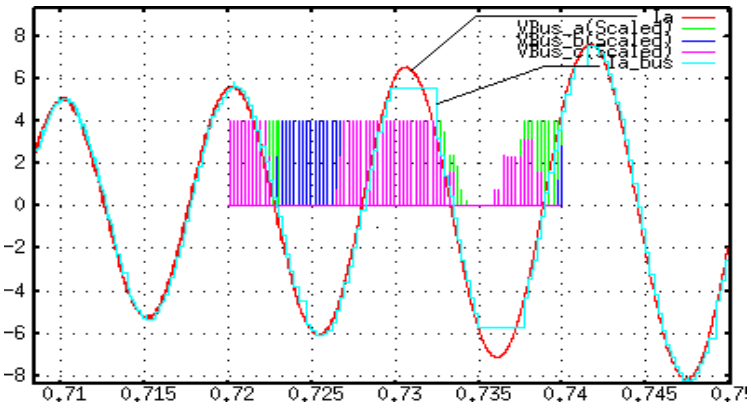

Figure 20: Complete loss of reconstructed current *Ia_bus* at the beginning of motor deceleration due to change in polarity of the commanded phase voltages.

Within the ISample code of Section A.2.2 a special interpolation mode can be enabled such that for a given phase (in this example we use Ph-A), an estimation can be made at the points where direct sampling cannot be performed. This is done using the information derived from the direct sampling of *Ib_bus* and *Ic_bus* which at these points do possess enough *turn on* time to acquire samples [16].

With this mode enabled, continuity can be maintained. The revised plots of *Ia_bus* and i_a for the acceleration and deceleration portion of the run are shown in Figure 21 and Figure 22 (compare these plots to Figure 17 Figure 18 and respectively).

[16] This special interpolation of course applies to the reconstruction of *Ib_bus* and *Ic_bus* in turn.

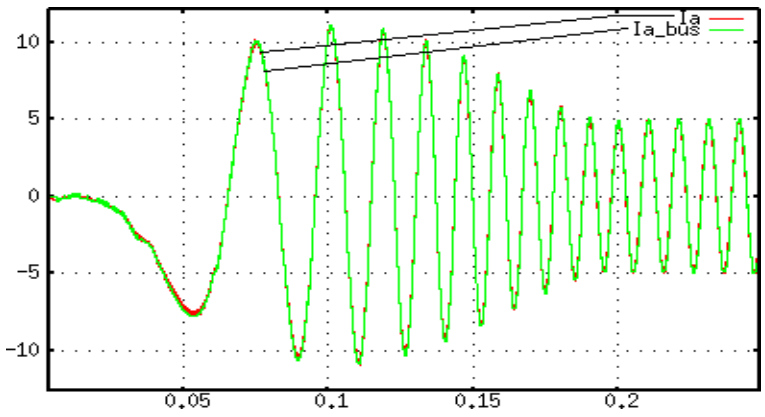

Figure 21: Close-up of Phase current i_a and reconstructed current *Ia_bus* at the start of motor acceleration with the special interpolation mode enabled.

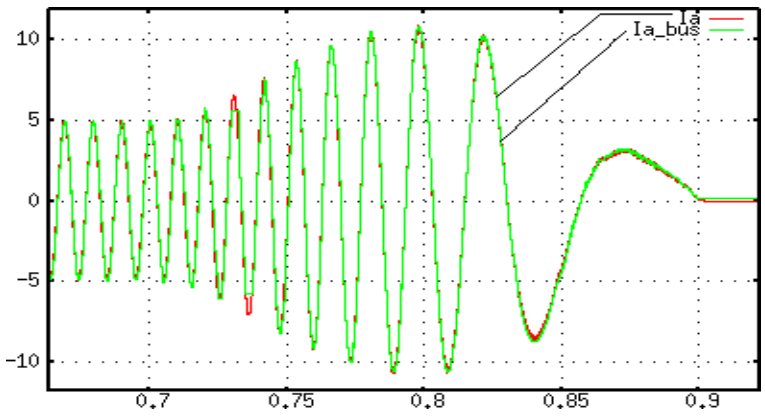

Figure 22: Close-up of Phase current i_a and reconstructed current *Ia_bus* at the start of motor deceleration with the special interpolation mode enabled.

Finally a more detailed view of the effect of this special interpolation mode is shown in Figure 23 (compare this plot to Figure 19).

The derived form of *Ia_bus* is coarse at best but within a reasonable degree, provides a close estimation of the actual phase current i_a.

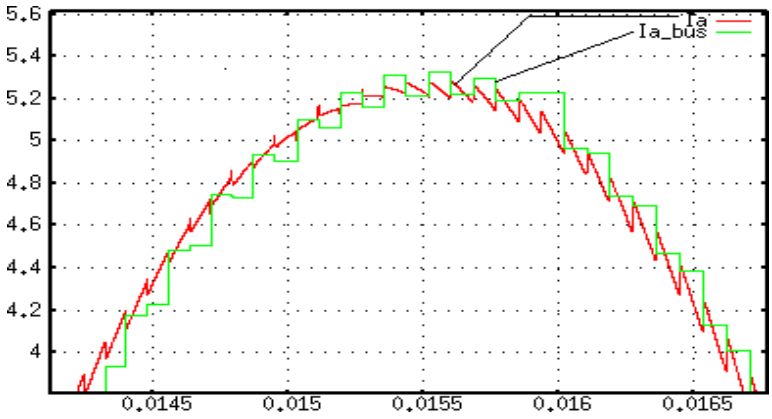

Figure 23: Recovery by interpolation of reconstructed current *Ia_bus* at points where inadequate sample time duration will not allow direct sampling.

5 Alternative Power Circuits

This section presents a couple of alternative arrangements to the three phase bridge shown in Figure 1. Like the preceeding section, these arrangements suggest a design that allows phase currents to be monitored without the use of level shifting or isolation circuitry.

5.1 Power Stage with Three Sense Resistors

The power circuit depicted in Figure 24 is equivalent to the circuit shown in Figure 1. However instead of an IPM, individual IGBT's and gate drive IC's are used. This allows the use of three current sense resistors *Rs-A*, *Rs-B* and *Rs-C*, one for each leg. This simplifies the sampling and reconstruction algorithm used to recreate the current i_a, i_b and i_c flowing through phases *Ph-A*, *Ph-B* and *Ph-C*.

A good example of a gate dive IC is the International Rectifier® IR2110 device. An example of an ultra-fast IGBT is International Rectifier® IRGB4640D (600 Volt, 40 Amp).

Unlike the implementation using a single sense resistor in the minus bus as described in the last section, there is now no need for a special interpolation mode. Current in each of the legs is now observable under all switching conditions[17].

[17] Again, this assumes minus-side Space Vector PWM modulation

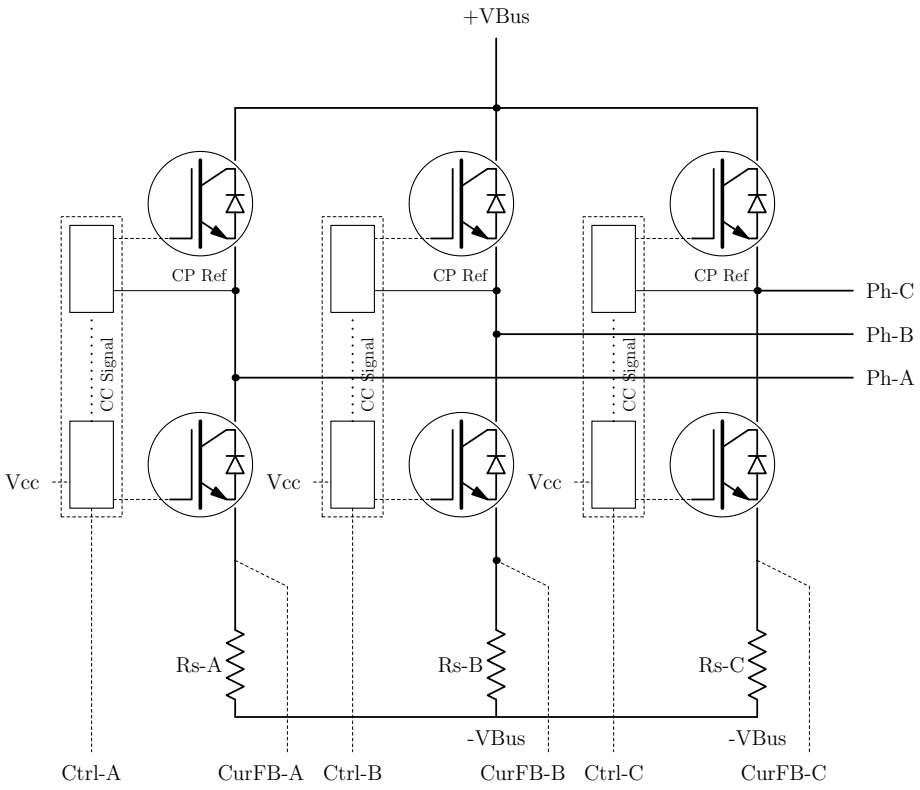

Figure 24: Implementation of a three phase power stage using gate drive IC's and discrete IGBT's

5.2 Three Level Power Stage (Neutral Point Clamp)

There has been an upturn with regard to the use of multi-level converters applied to servo motor control. Historically, the use of these types of converters were reserved for power conversion applications.

One example of a multi-level converter is the *neutral point clamp* typically used in a three level configuration[18].

A diagram of a three level neutral point clamp is shown in Figure 25.

[18]Configurations for neutral-point clamp designs above five levels becomes problematic due to the high number of clamping diodes required.

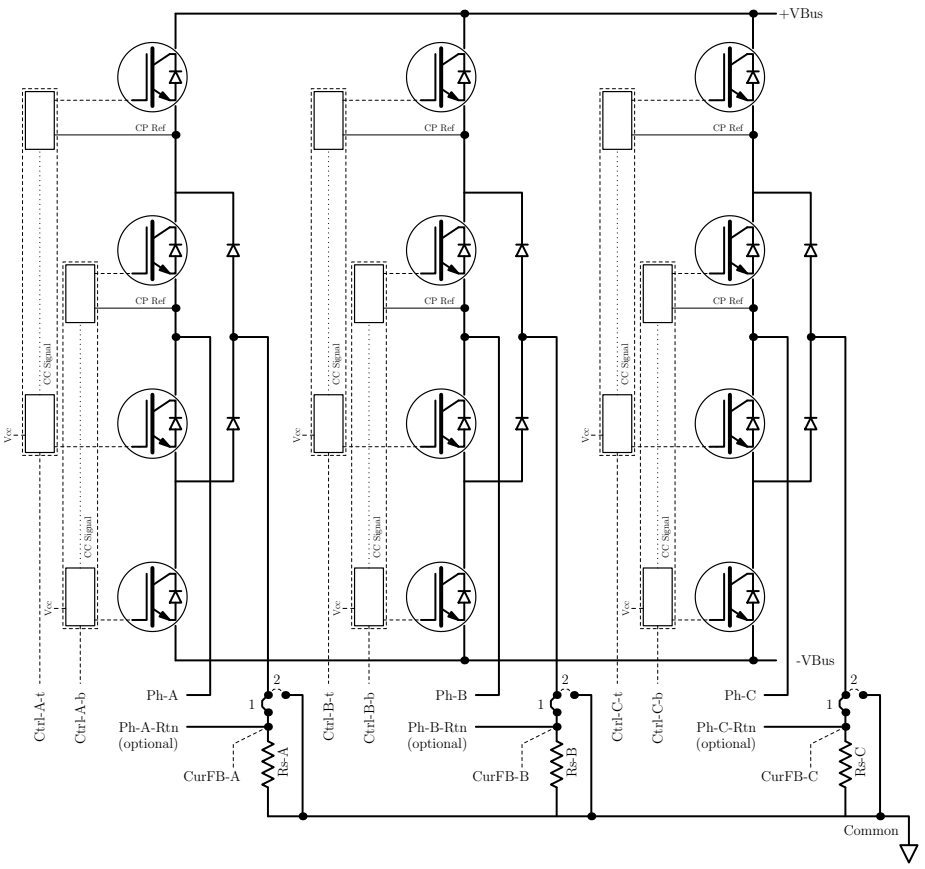

Figure 25: A three phase, three level neutral point clamp implementation of a power stage using gate drive IC's and discrete IGBT's.

Like the two level configurations shown in Figures 1 and 24, Space-Vector PWM can be used to modulate the phase voltages. A very good document describing Three Level Space Vector PWM is provided here [5].

Some additional points can made about the circuit diagram of Figure 25. Like Figure 24 the IR2110 gate driver is employed. However now, some of the unique features of this driver are exposed. Unlike most dual gate drive IC's, this chip contains high voltage level shifting circuits in both the top and bottom gate drive elements. This allows the control signals of the IR2110 to be connected directly to the FPGA (in this case FPGA #1 of Figure 2). Six IR2110 are needed to control the three-phase neutral point clamp circuit of Figure 25.

Another feature provided by the circuit of Figure 25 is the ability to select how phase current sensing is to be performed. Jumpers are provided for this selection. If the jumpers are selected for position "1". A current sampling algorithm must be used similar to that described for the power circuit described

in 5.1 above.

If jumper position "2" is selected, a three phase motor with separated windings (e.g. windings that are electrically isolated[19] not connected in either *wye* or *delta*) can be used. The return connections of each of phases would connect to Ph-A-Rtn, Ph-B-Rtn and Ph-C-Rtn.

This allows current to be sampled directly (no interpolation). In addition, this configuration allows unbalanced windings to be compensated (re-balanced electronically), something that cannot be accomplished in a typical *wye* or *delta* connected three phase motor.

Finally, it should be noted that one can revert to a more traditional way in modulating a three-level converter and still maintain the benefits described above for Space-Vector PWM (*minus side clamping*).

Because, the circuit of Figure 25 has a neutral connection. There are three *zero vectors*, the +VBus, -VBus and the neutral connection (designated by the *Common* symbol in Figure 25.

The neutral connection is *balanced* because it is placed half way between the +VBus and -VBus connections.

This allows traditional *carrier based* PWM to be used instead of Space-Vector PWM. The following plots show a comparison between Space-Vector *minus side clamp* (Figures 26, 28, 30 and 32) and traditional carrier based modulation (Figures 27, 29, 31 and 33).

[19] Electrically isolated but still magnetically coupled.

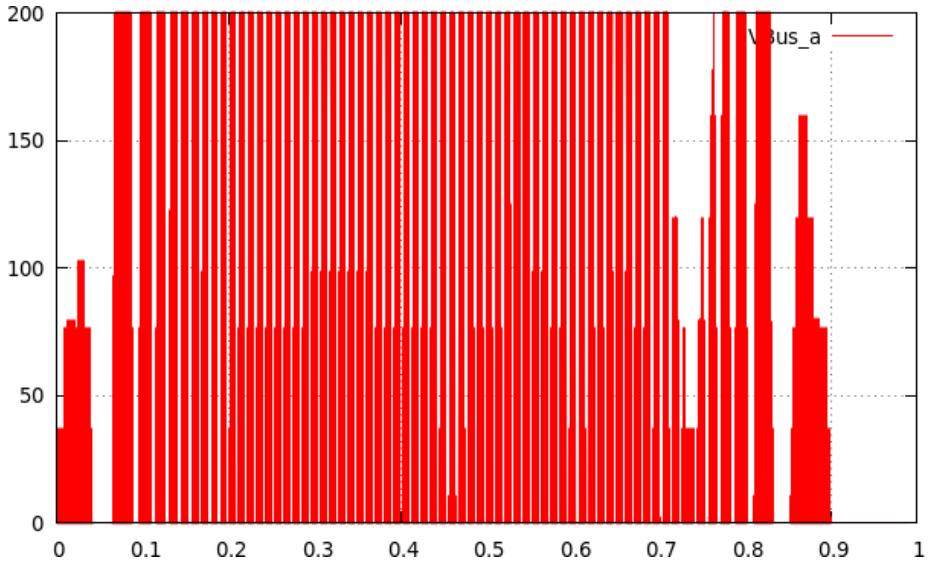

Figure 26: PWM waveform for Phase A using two-level Space-Vector *minus side clamped* modulation.

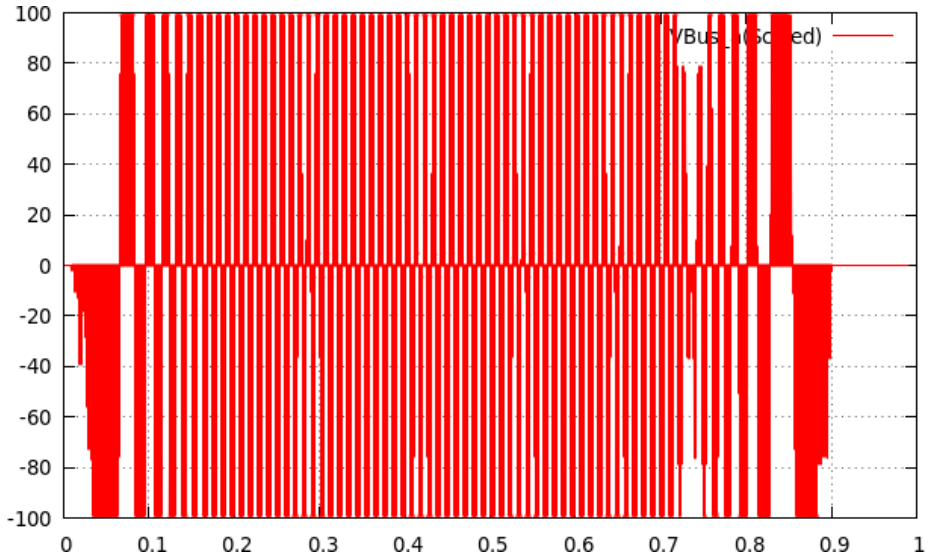

Figure 27: PWM waveform for Phase A using three-level traditional carrier based modulation.

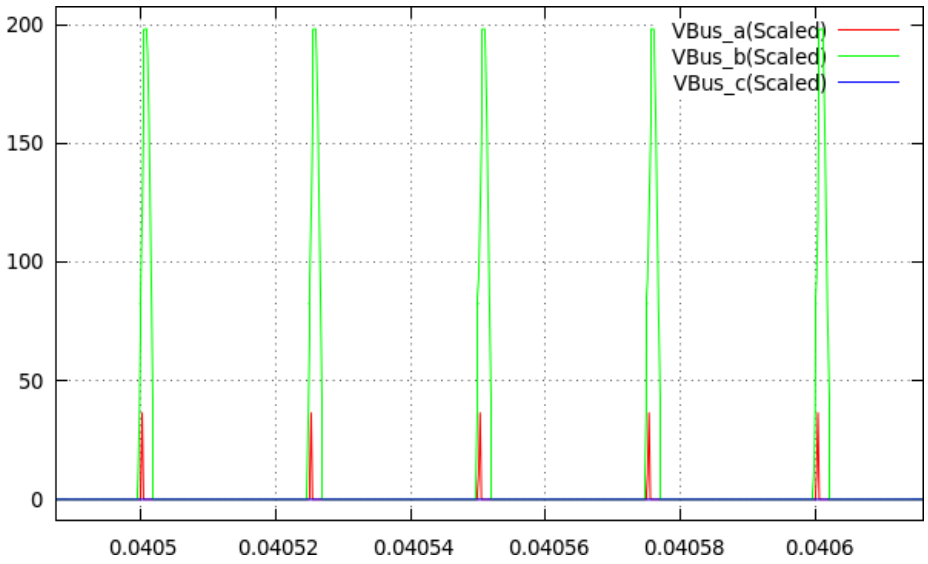

Figure 28: Close-up of PWM waveform for Phases A, B, and C using two-level Space-Vector *minus side clamped* modulation.

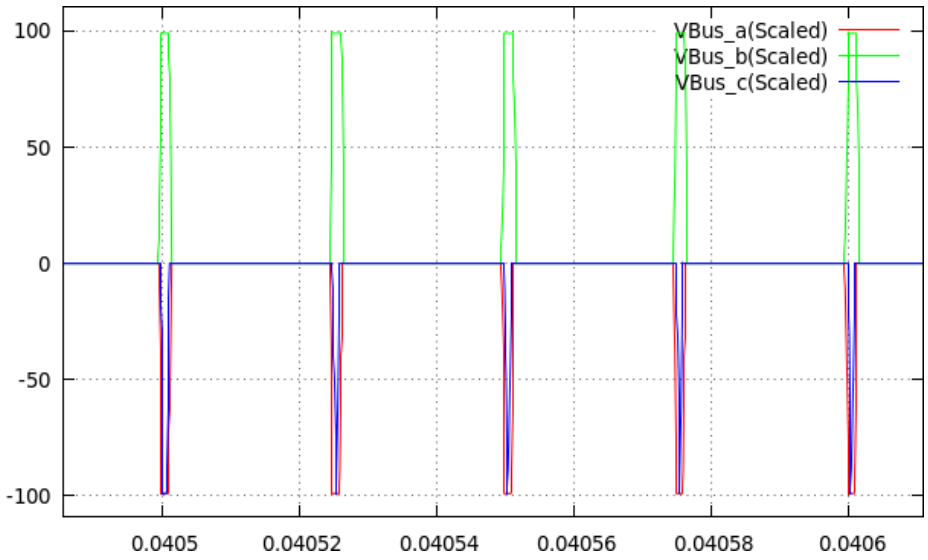

Figure 29: Close-up of PWM waveform for Phases A, B and C using three-level traditional carrier based modulation.

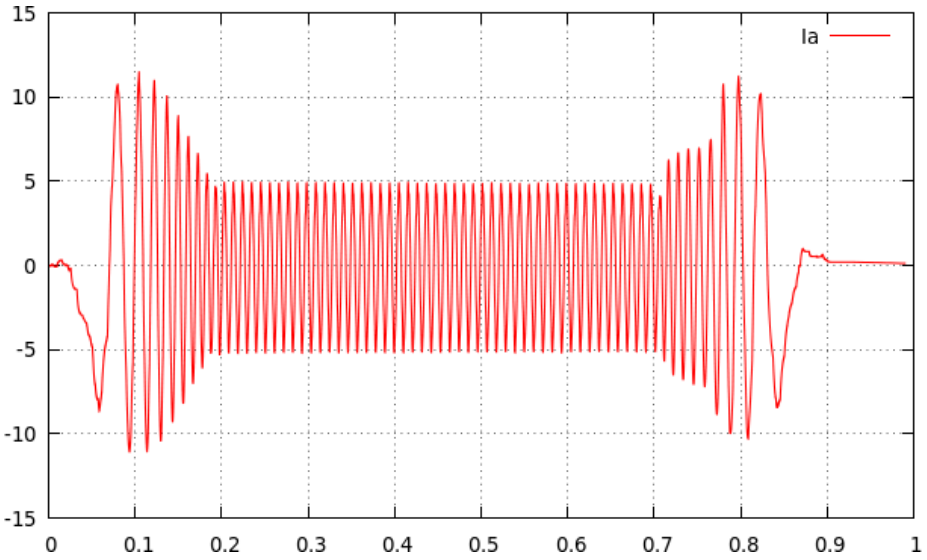

Figure 30: Current flowing in Phase A using two-level Space-Vector *minus side clamped* modulation.

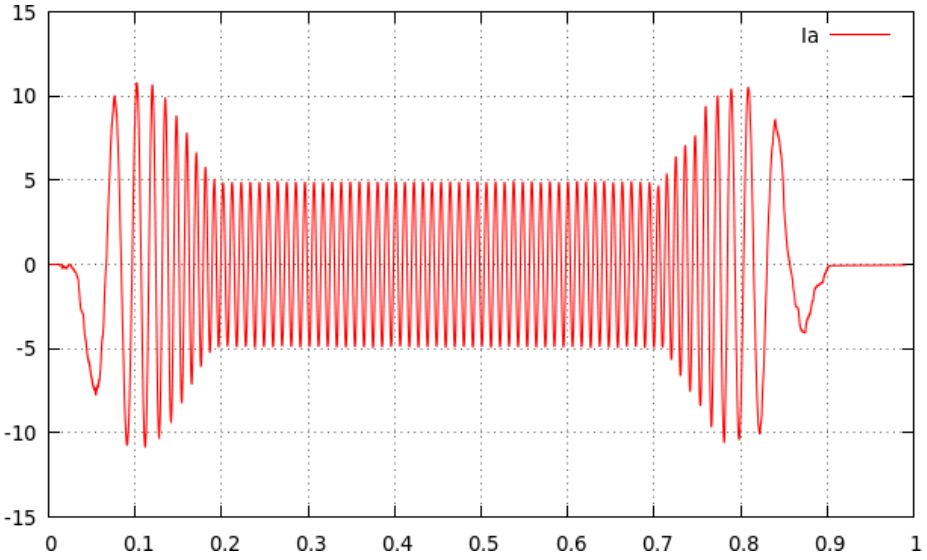

Figure 31: Current flowing in Phase A using three-level traditional carrier based modulation.

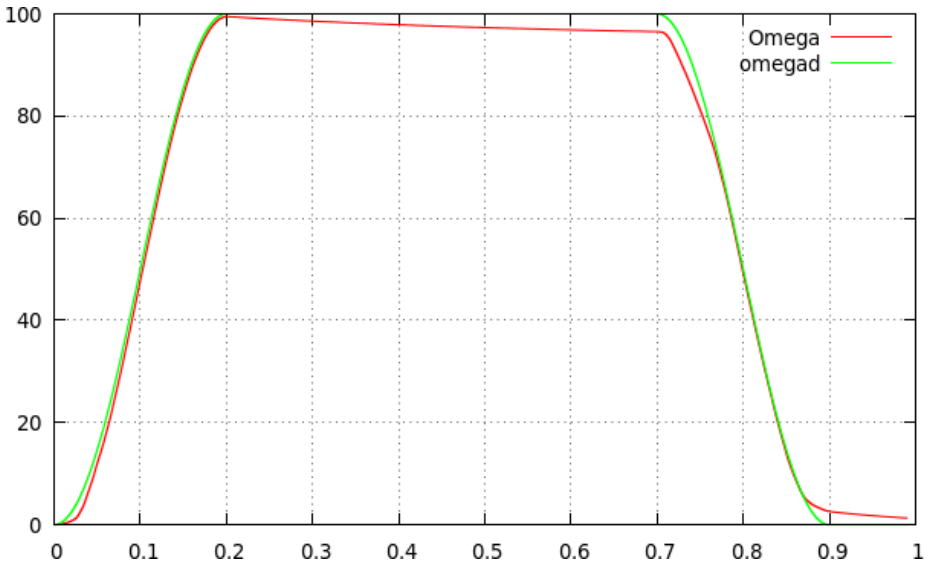

Figure 32: Motor velocity *omegad* using two-level Space-Vector *minus side clamped* modulation.

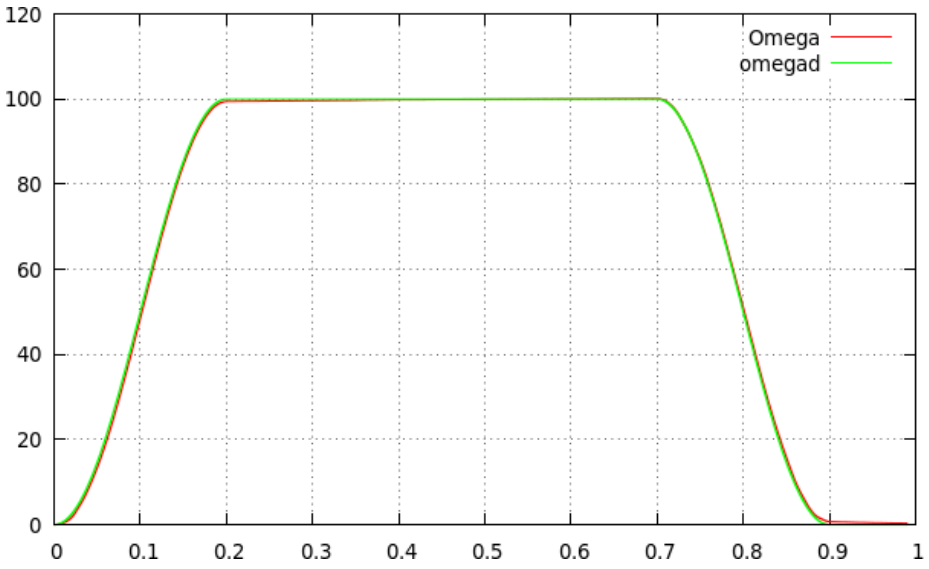

Figure 33: Motor velocity *omegad* using three-level traditional carrier based modulation.

Some things to note in the Figures shown above. In Figure 29 notice that all three phase switch state within a given cycle. If three-level Space-Vector PWM were to be employed similar to that described in document [5], only two of the three phases would switch states every cycle.

A brief discussion as to how Space-Vector PWM can be applied to a three-level switch topology is described in the next section.

5.3 Two Level Space Vector with *Plus Side Clamped* Modulation

Because two-level Space-Vector PWM contains two *zero* vectors (all top switches on or all bottom switches on), clamping can be done with reference to the positive bus as well as the negative bus. A close-up of *plus side clamped* Space-Vector modulation is shown in 34.

It can be shown that *plus side clamped* modulation reproduces the identical simulation run to that shown in the plot of Figure 32.

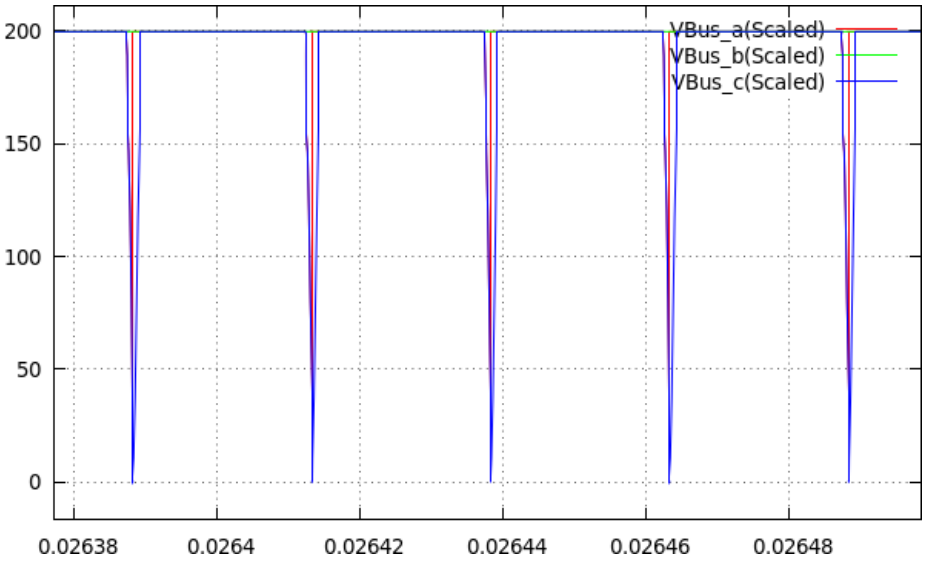

Figure 34: Close-up of PWM waveform for Phases A, B, and C using two-level Space-Vector *plus side clamped* modulation.

If one were to combine the plot of Figure 28 with the plot of 34 phase shifted by 180 degrees, and reduce the bus voltage of both to 100 VDC, we essentially produce the three-level Space-Vector equivalent to the carrier based waveform shown in Figure 29.

The phase shift of 180 degrees insures that we maintain the referenced *zero vector* at the common point (mid-point) of the three level converter. At some

point however, as the pulse widths widen, they must combine so as to essentially make the converter operate as a two-level converter where modulation is between +VBus and -VBus only.

An important point should be made about this sequencing of this modulation. Over one full switching cycle, it can be shown that the current from the +VBus and -VBus is the same. This is important if the bus capacitors for both the +VBus and -VBus are being charged from full wave rectification without an neutral (common) return. Under this condition if the currents were un-equal, the two bus voltages could potentially become unbalanced (see Figure 25).

A Development Simulator Details

A.1 Overview

This section provides some details relative to simulation used to construct the plots presented in this paper.

This simulator uses a custom grammar to describe the program and is for the most part a shorten form of the standard C and C++ language. As mentioned in Section 3, implementation of this simulator involves the use of a special program to parse structured statements for the purpose of generating a C++ file. This auto-generated file is then combined with pre-compiled C and C++ libraries containing the Runge-Kutta simulation algorythm along with the code necessary to connect to the standard GNU *Gnuplot* program used to generate the plots.

As can be seen by studying the statements listed below, one can deduce that the form of this simulator is far from what would be expected of a commercial simulator. The emphasis here is versatility over simplicity.

Again, the intent here was to design a simulator that could be used as a tool in writing the actual C code, that would ultimately run in one or more processing cores. These processing cores could be contained on a device such as an FPGA containing one or more soft processors. An example of one such device is the Xilinx® Artix-7 FPGA configured with one or more Microblaze soft processing cores.

Using the simulation code listed in Section A.2.1 and A.2.2 as reference, the sections of a typical program are as follows.

Begin Definition ... End Definition Statements contained within these delimiters provide an area for C style constructs such as *define*, *type*, and *structure* declarations that exhibit global scope within the program. Statements can be grouped within a single beginning and ending set of quotation marks or individually on a per-line basis.

Begin *object object_name* ... End *object object_name* At present, there exists three types of program objects, *OdeObject*, *CtrlObject* and *SrcObject*. A proposed forth object named *SpiceObject* is to be added in the future. Each of these objects are described in detail below. In general, an *object* defines a class who's instantiation executes concurrently with other instantiated classes, under the control of a variable time stepping algorythm.

Begin Simulation ... End Simulation The parameters of the simulation are specified in this section. The run time for the simulation is specified by the value assigned to *Simutime*. The values assigned to *Reltol* and *AbsTol* dictate the amount of acceptable error for each time step generated by the Runge-Kutta algorythm. Time steps that produce an error outside the range specified by these variables, instructs the algorithm to reduce the step size and recalculate based on the new time increment.

The objects described above are very simular to C++ Classes [20]. Objects

[20] In fact the auto-generated output file constructed from the program listing is a C++ source file.

usually contain a function that allows the execution of sequencial C style statements, delimited by a beginning and ending quotation marks. These statements are scheduled to execute once every time step. The size of the time step varies and is governed by a sixth order Runge-Kutta interpolator operating on the output of each *OdeObject* defined in the program. Each object can contain variables delimited by the keyword *Variables* which have context only in the object that they are declared. There are two reserved variables named y and $\frac{dy}{dt}$. The variable $\frac{dy}{dt}$ applies only to the *OdeObject* and represents it's solution at each time increment. Thus, in the case of the *OdeObject*, the variable y is simply the time integral of $\frac{dy}{dt}$.

The keyword *RValueUpdate* which can be specified in all object types, denotes the update requirements for one or more peer objects declared in the program. The statements contained within this delimiter are considered the "glue" to all other objects contained in the program. For example, the update statement associated with an *OdeObject*

"Ia_Ib_ic_2","Pwm_sig = y"

indicates that the variable "Pwm_sig" is declared in the instance of the object named "Ia_Ib_Ic_2" and should be updated with the state variable "y" at the end of each time step. The example program shows only updates by state variables of the given object. In actuality, ANY variable declared in one object can be used to update one or more variables in any other object using the procedure outlined above.

The keyword *PlotThisOutput* allows the intrinsic variable "y" for a given object to be stored relative to time for outputting to a *GnuPlot* window at the end of simulation. There is also the keyword *Probes* that allows the selection of any variable declared within the given object to be stored and plotted at the end of simulation.

A description of some of the objects used in this simulator are as follows.

OdeObject As outlined above, this object is used to specify a differencial equation with an output that can an be the derivative or one or more state variables (e.g., a *coupled* ODE). This object also controls the time stepping algorithm that by default, is implemented as a sixth-order Runge-Kutta solver. As such, the solver executes the function specified in each of the declared *OdeObject*'s, six times within the current time step quantum to determine a differential error gradient.

This execution is done concurrently on all *OdeObject*'s. At the end of each time step, the size of the time quantum is adjusted by determining the object with the maximum error and comparing this error to the tolerances specified in the *Simulation* section of the program described above.

CtrlObject This object allows for the encapsulation of sequential C that can be scheduled to run at predetermined intervals in the course of a simulation run. There can be multiple *CtrlObject*'s defined to run at the same or at different time intervals. These objects are scheduled to run concurrently with other *CtrlObject*'s, *OdeObject*'s and *SrcObject*'s. Communications with other *CtrlObject*'s, *OdeObject*'s and *SrcObject*'s is accomplished by using *RValueUpdate*

declarations described above. As the name suggests, *control* objects represent the process or processes that would be implemented in an actual control design implemented with a DSP or RISC processor.

Because, the code in these objects is written in C, with usually little to no translation is when finally inserted in the actual processing system. Because of the variable time stepping nature of the simulator, a large degree of granularity error can be generated if the scheduling of these objects happens well into the current time step. To compensate for this, when a scheduled event is detected, the time stepping algorithm reverts back to it's previous cached state, divides the current time step by two and re-executes. This process is repeated until the event is no longer detected. The execution then resumes at the minimum time step interval until the event is again reached. The event is then acknowledged and the simulation resumes in variable time stepping mode.

SrcObject Source objects are used to generate signals or outputs associated with the plant (e.g. a PWM circuit or logic). These objects run in simulated real time concurrent to other *SrcObject*'s and *OdeObject*'s. The input to these objects usually originates from *CtrlObject*'s, but this is not a requirement. Like *CtrlObject*'s, these object's can insert discontinuities into the simulation environment (e.g., objects that are responsible for PWM generation). When these transitions happen within large time steps, excessive simulation error can be generated. Output transitions are handled in an identical fashion as the events described for the *CtrlObject* above.

SpiceObject This is a new object that has been added to the simulator. The current environment allows *OdeObject* to be connected to other OdeObject, *CtrlObject*, and *SrcObject*'s using the *RValueUpdate* sematics described above. However, this communication mechanism is designed only for the updating of internal input variables at end of each time step. These variables have *local scope* only, relative to the objects they are declared in.

A typical spice simulator requires an additional processing element that is executed after the error processing algorithm. It's job is to solve a set of equations who's outputs are dependents of other equations of the same set. This calculation is done between the successive time steps eliminating time as a dependent variable. This requires the use of Gaussian Elimination operating on a matrix whose elements are mostly make up of the state variables of the system (a method known as *nodal analysis*.

Modern spice simulators use methods such as *sparse matrix representation* to control the amount of memory required to represent the system of dependent equations. This new simulator needs only limited nodal analysis capability, for representation of circuits that contain only four intrinsic elements, three of which are passive elements. They are a *resister*, *inductor* (coupled and non-coupled), *capacitor* and *switch*. The *switch* is an active element used to present a diode, MOSFET, IGBT, and similar devices.

Thus, the *SpiceObject* provides a mechanism to read variables contained in selected *OdeObject*, *CtrlObject* and *SrcObject*'s solve a set of simultaneous equations and return a solution vector back to the selected *OdeObject*'s.

A.2 Simulation Program Code

A.2.1 AC Servo Motor

```
Begin OdeObject Ia_Ib_Ic_1

    Variables:
        "
        double vs_a;
        double pwm_sig;
        double ia;
        double omega;
        double theta;
        double v_n;
        int PositionFeedback; //(this will ultimately to "FSFCtrl")
        int VelocityFeedback; //(this will ultimately to "FSFCtrl")
        "

    PlotThisOutput: "TRUE"

    GroupSolve: "3"

    Function:
        "
        #ifdef LINEAR_AMP_MODE
            //generating the voltage command directly through CtrlObject "VDqCmd"
        vs_a = pwm_sig + .5*DC_BUS_VOLTAGE;
        #else
        vs_a = ShapeSquareWaveSource(((pwm_sig ¿ 0) ? DC_BUS_VOLTAGE : 0), SRC_SLEW_RATE ,t);
        #endif
        return(vs_a - Ra*y + Ka*omega*sin(Nr*theta));
        "

    PostOdeFunction:
        "
            //These states create a realistic quantization of Position and Velocity
            //feedback, indicative of actual encoder feedback. These integer variables
            //can later be turned back into double variables for processing by the
            //control algorythm.
        PositionFeedback = (int) (theta/(2*PI) * ENCODER_RESOLUTION);
        VelocityFeedback = (int) (omega/(2*PI) * ENCODER_RESOLUTION);
        "

    GroupSolveFunction:
        "
        #define A1 (Lm_b/Mab + Mbc/Mac)
        #define B1 (- Lm_c/Mac - Mbc/Mab)
        #define A2 (Mab/Lm_a - Lm_b/Mab)
        #define B2 (Mac/Lm_a + Mbc/Mab)

        #define D (B2*A1 - B1*A2)
        #define F (D*B2)
        #define G (Mab/(D*Lm_a) - Mac*A2/(F*Lm_a) + 1/D - A2/F)
        #define H (Mac/(B2*Lm_a) + 1/B2)

        #define I (H - B1*G)
        #define J (B2*G)

        double dia;
        double dib;
        double dic;

        double C1;
        double C2;

        v_n = ((dmdt[2]/Mac - dmdt[1]/Mab)*J + (dmdt[0]/Lm_a + dmdt[1]/Mab)*I - dmdt[0]/Lm_a) /
            (-J/Mab + J/Mac + I/Lm_a + I/Mab - 1/Lm_a);
```

```
        C1 = - v_n/Mab + dmdt[1]/Mab + v_n/Mac - dmdt[2]/Mac;
        C2 = v_n/Lm_a + v_n/Mab - dmdt[0]/Lm_a - dmdt[1]/Mab;

        dib = (B2*C1 - B1*C2)/D;
        dic = (C2 - A2*dib)/B2;
        dia = (Mab*dib + Mac*dic - v_n + dmdt[0])/Lm_a;

        dydt[0] = dia;
        dydt[1] = dib;
        dydt[2] = dic;
        "

    Probes:

        "VBus_a.push_back(vs_a);"

    RValueUpdate:

        "Omega","ia = y"
        "ISample","ia = y"

End OdeObject Ia_Ib_Ic_1
```

Figure 35: Description for Motor phase A current with coupled armature inductance from other phases (Motor Phase B and C are expressed in a similar fashion).

```
Begin OdeObject Omega

    Variables:
        "
        double ia = 0;
        double ib = 0;
        double ic = 0;
        double theta;

    PlotThisOutput: "TRUE"

    Function:
        "
        return((- Ka*ia*sin(Nr*theta) - Kb*ib*sin(Nr*theta - 2*PI/3) -
            Kc*ic*sin(Nr*theta - 4*PI/3) -
            Bm*y - Cm*(y < 0 ? -1 : (y ¿ 0 ? 1 : 0)) -
            Dm*sin(2*Nr*theta))/Jm);
        "

    RValueUpdate:

        "Ia_Ib_Ic_1","omega = y"
        "Ia_Ib_Ic_2","omega = y"
        "Ia_Ib_Ic_3","omega = y"
        "Theta","omega = y"
        "ISample","omega = y"
        "ISample","alpha = dydt"

End OdeObject Omega
```

Figure 36: Description for Motor velocity with modeled disturbance.

```
Begin OdeObject Theta

    Variables:
        "
        double omega;
        "

    PlotThisOutput: "TRUE"

    Function:
        "
        return(omega);
        "

    RValueUpdate:

        "Ia_Ib_Ic_1","theta = y"
        "Ia_Ib_Ic_2","theta = y"
        "Ia_Ib_Ic_3","theta = y"
        "Omega","theta = y"
        "Probe","theta = y"
        "VDqCmd","theta = y"

End OdeObject Theta
```

Figure 37: Description for Motor position.

A.2.2 Current Feedback Synthesis from Bus Resistors

```
Begin OdeObject ISample

    Variables:
        "
            //source for this ODE
        double ia;
        double ib;
        double ic;
        double TriAngWave;
        double V_xo[3];
        double omega;
        double alpha;
        BUS_SAMPLE_STATE PrevBusSampleState;
        BUS_SAMPLE_ACTION PrevBusSampleAction;
        double PrevBusSampleTime;
        double ia_bus;
        double ib_bus;
        double ic_bus;
        double ia_bus_internal;
        double ib_bus_internal;
        double ic_bus_internal;
        "

    Probes:
        "
        Ia_bus.push_back(ia_bus);
        Ib_bus.push_back(ib_bus);
        Ic_bus.push_back(ic_bus);
        "

    PostOdeFunction:
        "
        bool IaSampleValid = FALSE;
        bool IbSampleValid = FALSE;
        bool IcSampleValid = FALSE;
        bool AllowSectorAveraging = FALSE;
        BUS_SAMPLE_STATE BusSampleState;
        BUS_SAMPLE_ACTION BusSampleAction;
        double t_prev;

        #define ENABLE_BUS_SAMPLING //un-comment to enable bus current sampling.
        #ifdef ENABLE_BUS_SAMPLING
            //un-comment to enable determination of "non-observable" phase from present and previous samples.
        #define ESTIMATE_PHASE

            //un-comment to show only the part of the algorythm that estimates the "non-observable"
            //phase from the present and previous samples.
            //("ESTIMATE_PHASE must be defined above to use this option.)
            //#define ONLY_ESTIMATE_PHASE

        if(V_xo[0] > TriAngWave){
            IaSampleValid = TRUE;
        }
        if(V_xo[1] > TriAngWave){
            IbSampleValid = TRUE;
        }
        if(V_xo[2] > TriAngWave){
            IcSampleValid = TRUE;
        }
        if(IaSampleValid && IbSampleValid && IcSampleValid){
            //We have a "0" vector (all low)
            BusSampleState = BSS_111;
        }
        else if(!IaSampleValid && !IbSampleValid && !IcSampleValid){
            //We have a "0" vector (all high)
            BusSampleState = BSS_000;
```

```
}
else if(IaSampleValid && IbSampleValid){
	BusSampleState = BSS_110;
}
else if(IbSampleValid && IcSampleValid){
	BusSampleState = BSS_011;
}
else if(IaSampleValid && IcSampleValid){
	BusSampleState = BSS_101;
}
else if(IaSampleValid){
	BusSampleState = BSS_100;
}
else if(IbSampleValid){
	BusSampleState = BSS_010;
}
else{
	BusSampleState = BSS_001;
}
if(BusSampleState == PrevBusSampleState){
	return;
}
else{
	t_prev = PrevBusSampleTime;
	if((t - t_prev) < MIN_BUS_SAMPLE_TIME_CHANGE){
	return;
	}
	PrevBusSampleTime = t;
}
	//SAMPLE STATES:

switch(BusSampleState){
case BSS_000:
	switch(PrevBusSampleState){
	case BSS_000:
		BusSampleAction = BSA_NONE;
		break;
	case BSS_100:
		BusSampleAction = BSA_000_100;
		break;
	case BSS_010:
		BusSampleAction = BSA_000_010;
		break;
	case BSS_001:
		BusSampleAction = BSA_000_001;
		break;
	case BSS_110:
		BusSampleAction = BSA_000_110;
		break;
	case BSS_011:
		BusSampleAction = BSA_000_011;
		break;
	case BSS_101:
		BusSampleAction = BSA_000_101;
		break;
	case BSS_111:
		BusSampleAction = BSA_NONE;
		break;
	}
break;
case BSS_100:
	switch(PrevBusSampleState){
	case BSS_000:
		BusSampleAction = BSA_100_ZERO;
		break;
	case BSS_100:
		BusSampleAction = BSA_NONE;
		break;
```

```
        case BSS_010:
                BusSampleAction = BSA_ILLEGAL;
                break;
        case BSS_001:
                BusSampleAction = BSA_ILLEGAL;
                break;
        case BSS_110:
                BusSampleAction = BSA_100_110;
                break;
        case BSS_011:
                BusSampleAction = BSA_ILLEGAL;
                break;
        case BSS_101:
                BusSampleAction = BSA_100_101;
                break;
        case BSS_111:
                BusSampleAction = BSA_100_ZERO;
                break;
        }
break;
case BSS_010:
        switch(PrevBusSampleState){
        case BSS_000:
                BusSampleAction = BSA_010_ZERO;
                break;
        case BSS_100:
                BusSampleAction = BSA_ILLEGAL;
                break;
        case BSS_010:
                BusSampleAction = BSA_NONE;
                break;
        case BSS_001:
                BusSampleAction = BSA_ILLEGAL;
                break;
        case BSS_110:
                BusSampleAction = BSA_010_110;
                break;
        case BSS_011:
                BusSampleAction = BSA_010_011;
                break;
        case BSS_101:
                BusSampleAction = BSA_ILLEGAL;
                break;
        case BSS_111:
                BusSampleAction = BSA_010_ZERO;
                break;
        }
break;
case BSS_001:
        switch(PrevBusSampleState){
        case BSS_000:
                BusSampleAction = BSA_001_ZERO;
                break;
        case BSS_100:
                BusSampleAction = BSA_ILLEGAL;
                break;
        case BSS_010:
                BusSampleAction = BSA_ILLEGAL;
                break;
        case BSS_001:
                BusSampleAction = BSA_NONE;
                break;
        case BSS_110:
                BusSampleAction = BSA_ILLEGAL;
                break;
        case BSS_011:
                BusSampleAction = BSA_001_011;
                break;
```

```
        case BSS_101:
                BusSampleAction = BSA_001_101;
                break;
        case BSS_111:
                BusSampleAction = BSA_001_ZERO;
                break;
        }
break;
case BSS_110:
        switch(PrevBusSampleState){
        case BSS_000:
                BusSampleAction = BSA_110_ZERO;
                break;
        case BSS_100:
                BusSampleAction = BSA_110_100;
                break;
        case BSS_010:
                BusSampleAction = BSA_110_010;
                break;
        case BSS_001:
                BusSampleAction = BSA_ILLEGAL;
                break;
        case BSS_110:
                BusSampleAction = BSA_NONE;
                break;
        case BSS_011:
                BusSampleAction = BSA_ILLEGAL;
                break;
        case BSS_101:
                BusSampleAction = BSA_ILLEGAL;
                break;
        case BSS_111:
                BusSampleAction = BSA_110_ZERO;
                break;
        }
break;
case BSS_011:
        switch(PrevBusSampleState){
        case BSS_000:
                BusSampleAction = BSA_011_ZERO;
                break;
        case BSS_100:
                BusSampleAction = BSA_ILLEGAL;
                break;
        case BSS_010:
                BusSampleAction = BSA_011_010;
                break;
        case BSS_001:
                BusSampleAction = BSA_011_001;
                break;
        case BSS_110:
                BusSampleAction = BSA_ILLEGAL;
                break;
        case BSS_011:
                BusSampleAction = BSA_NONE;
                break;
        case BSS_101:
                BusSampleAction = BSA_ILLEGAL;
                break;
        case BSS_111:
                BusSampleAction = BSA_011_ZERO;
                break;
        }
break;
case BSS_101:
        switch(PrevBusSampleState){
        case BSS_000:
                BusSampleAction = BSA_011_ZERO;
```

```
                break;
            case BSS_100:
                BusSampleAction = BSA_101_100;
                break;
            case BSS_010:
                BusSampleAction = BSA_ILLEGAL;
                break;
            case BSS_001:
                BusSampleAction = BSA_101_001;
                break;
            case BSS_110:
                BusSampleAction = BSA_ILLEGAL;
                break;
            case BSS_011:
                BusSampleAction = BSA_ILLEGAL;
                break;
            case BSS_101:
                BusSampleAction = BSA_NONE;
                break;
            case BSS_111:
                BusSampleAction = BSA_101_ZERO;
                break;
            }
        break;
        case BSS_111:
            switch(PrevBusSampleState){
            case BSS_000:
                BusSampleAction = BSA_NONE;
                break;
            case BSS_100:
                BusSampleAction = BSA_111_100;
                break;
            case BSS_010:
                BusSampleAction = BSA_111_010;
                break;
            case BSS_001:
                BusSampleAction = BSA_111_001;
                break;
            case BSS_110:
                BusSampleAction = BSA_111_110;
                break;
            case BSS_011:
                BusSampleAction = BSA_111_011;
                break;
            case BSS_101:
                BusSampleAction = BSA_111_101;
                break;
            case BSS_111:
                BusSampleAction = BSA_NONE;
                break;
            }
        break;
}
//record present state for next sample.
PrevBusSampleState = BusSampleState;

#if ((defined ESTIMATE_PHASE) && (!defined ONLY_ESTIMATE_PHASE ))
if((omega > - OMEGA_SAMPLE_LIMIT) && (omega < OMEGA_SAMPLE_LIMIT) &&
        (alpha > - ALPHA_SAMPLE_LIMIT) && (alpha < ALPHA_SAMPLE_LIMIT)){
        AllowSectorAveraging = TRUE;
}
#endif
switch(BusSampleActionBS){
        case BSA_000_100:
        case BSA_111_011:
        case BSA_010_011:
        case BSA_001_011:
        case BSA_110_100:
        case BSA_101_100:
```

```
#ifdef ONLY_ESTIMATE_PHASE
	ia_bus_internal = ia;
#else
	ia_bus = ia;
#endif
#ifdef ESTIMATE_PHASE
	switch(PrevBusSampleAction){
	case BSA_000_010:
	case BSA_111_010:
	case BSA_000_101:
	case BSA_111_101:
	case BSA_100_101:
	case BSA_001_101:
	case BSA_110_010:
	case BSA_011_010:
#ifdef ONLY_ESTIMATE_PHASE
	ic_bus = - (ib_bus_internal + ia_bus_internal);
#else
	ic_bus = - (ib_bus + ia_bus);
#endif
break;
	case BSA_000_001:
	case BSA_111_001:
	case BSA_000_110:
	case BSA_111_110:
	case BSA_100_110:
	case BSA_010_110:
	case BSA_011_001:
	case BSA_101_001:
#ifdef ONLY_ESTIMATE_PHASE
	ib_bus = - (ic_bus_internal + ia_bus_internal);
#else
	ib_bus = - (ic_bus + ia_bus);
#endif
break;
	case BSA_000_010:
	case BSA_111_101:
	case BSA_100_101:
	case BSA_001_101:
	case BSA_110_010:
	case BSA_011_010:
#ifdef ONLY_ESTIMATE_PHASE
	ib_bus_internal = ib;
#else
	ib_bus = ib;
#endif
#ifdef ESTIMATE_PHASE
	switch(PrevBusSampleAction){
	case BSA_000_100:
	case BSA_111_100:
	case BSA_000_011:
	case BSA_111_011:
	case BSA_010_011:
	case BSA_001_011:
	case BSA_110_100:
	case BSA_101_100:
#ifdef ONLY_ESTIMATE_PHASE
	ic_bus = - (ia_bus_internal + ib_bus_internal);
#else
	ic_bus = - (ia_bus + ia_bus);
#endif
break;
	case BSA_000_001:
	case BSA_111_001:
	case BSA_111_110:
	case BSA_000_110:
	case BSA_100_110:
	case BSA_010_110:
```

```
        case BSA_011_001:
        case BSA_101_001:
#ifdef ONLY_ESTIMATE_PHASE
        ia_bus = - (ic_bus_internal + ib_bus_internal);
#else
        ia_bus = - (ic_bus + ib_bus);
#endif
break;
        case BSA_000_001:
        case BSA_111_110:
        case BSA_100_110:
        case BSA_010_110:
        case BSA_011_001:
        case BSA_101_001:
#ifdef ONLY_ESTIMATE_PHASE
        ic_bus_internal = ic;
#else
        ic_bus = ic;
#endif
#ifdef ESTIMATE_PHASE
        switch(PrevBusSampleAction){
        case BSA_000_100:
        case BSA_111_100:
        case BSA_000_011:
        case BSA_111_011:
        case BSA_010_011:
        case BSA_001_011:
        case BSA_110_100:
        case BSA_101_100:
#ifdef ONLY_ESTIMATE_PHASE
        ib_bus = - (ia_bus_internal + ic_bus_internal);
#else
        ib_bus = - (ia_bus + ic_bus);
#endif
break;
        case BSA_000_010:
        case BSA_111_010:
        case BSA_000_101:
        case BSA_111_101:
        case BSA_100_101:
        case BSA_001_101:
        case BSA_110_010:
        case BSA_011_010:
#ifdef ONLY_ESTIMATE_PHASE
        ia_bus = - (ib_bus_internal + ic_bus_internal);
#else
        ia_bus = - (ib_bus + ic_bus);
        break;
}
#endif
break;
        case BSA_000_110:
        case BSA_111_001:
#ifdef ONLY_ESTIMATE_PHASE
        ic_bus_internal = ic;
#else
        ic_bus = ic;
#endif
if(AllowSectorAveraging){
        //half the sample, negate and assign to the other two.
        ia_bus = ib_bus = -.5*ic;
}
break;
        case BSA_000_011:
        case BSA_111_100:
#ifdef ONLY_ESTIMATE_PHASE
        ia_bus_internal = ia;
#else
```

```
        ia_bus = ia;
    #endif
    if(AllowSectorAveraging){
        //half the sample, negate and assign to the other two.
        ib_bus = ic_bus = -.5*ia;
    }
    break;
        case BSA_000_101:
        case BSA_111_010:
    #ifdef ONLY_ESTIMATE_PHASE
        ib_bus_internal = ib;
    #else
        ib_bus = ib;
    #endif
    if(AllowSectorAveraging){
        //half the sample, negate and assign to the other two.
        ia_bus = ic_bus = -.5*ib;
    }
    break;
    case BSA_ILLEGAL:
        Exception("Illegal Space Vector sequence detected.");
    break;
    //record present action for next sample.
    PrevBusSampleAction = BusSampleAction;
    #endif
    "

End OdeObject ISample
```

Figure 38: Motor phase current reconstruction from a state machine that processes current flowing in the bus resistors.

B State Feedback Controller using Feedback Linearization

One can find numerous information on web related to the state space control of permanent magnet AC motors. Here is one paper I believe that provides a very well rounded approach of a modern method for controlling a synchronous motor (see [1]). This paper focuses on the closed loop control of a permanent magnet. A permanent magnet stepping motor can be considered a two phase motor with no magnetic coupling between the two phases[21]. However, when one transforms the AC physical plane of a multi-phase motor with balanced windings into the D/Q plane, it does not matter if the windings are or are not coupled since the result is a *Direct* and *Quadrature* plane that is decoupled.

Section B.1.1 below titled **Mathematical Model and Feedback Linearization** is a partial reprint of [1]. Again, this paper describes a method for the closed loop control of a permanent magnet stepping motor. However, once the motor description is transformed into the D/Q plane the analysis and approach to control can be applied to the the permanent magnet AC motor. Section 4 describes the permanent magnet AC motor in the physical plane up to and including the transformation into the D/Q plane. However, there is no description for the control this motor. We can use the following section to derive insight into how the permanent magnet AC motor can be controlled.

Before we continue into the next section, note the comments I have added in boldface text:

Added comments in boldface text ...

B.1 Mathematical Model and Feedback Linearization

B.1.1 Mathematical Model

The PM stepping motor consists of a slotted stator with two phases and a permanent magnet rotor. One side of the rotor is a *north* pole and the other side is the corresponding *south* pole. The teeth on each side of the rotor are out of alignment by a tooth-width. The mathematical model for the PM stepping motor is given below.

$$\begin{aligned}
\frac{di_a}{dt} &= (v_a - Ri_a + K_m i_a \omega \sin(N_r\theta))/L \\
\frac{di_b}{dt} &= (v_b - Ri_b - K_m i_b \omega \cos(N_r\theta))/L \\
\frac{d\omega}{dt} &= (-K_m i_a \sin(N_r\theta) + K_m i_b \cos(N_r\theta) - B\omega)/J \\
\frac{d\theta}{dt} &= \omega
\end{aligned} \tag{15}$$

[21] This of course considered as an ideal condition. There is always some form of residual coupling.

where v_a, v_b and v_a, v_b are the voltages and currents in phases A and B respectively. Further, ω is the rotor (angular) speed, θ is the rotor (angular) position, B is the viscous-friction coefficient, J is the inertia of the motor plus load, K_m is the motor torque constant, R is the resistance of the phase winding, L is the inductance of the phase winding and N_r is the number of rotor teeth.

Table 1 specifies the parameters for a typical PM Stepper motor.

Parameter	Value
L	1.5 mH
R	.55 Ω
J	4.5x10^{-5} $k_g \cdot m^2$
K_m	.19 N·m/A
N_r	50
B	8.0x10^{-5} N·m·s/rad

Table 3: Stepper Motor Parameters

Note the model (15) is sufficient for control design. However, the model (15) is nonlinear, due to the sinusoidal functions and their multiplications with state variables. the Sinusoidal terms vary at the electrical frequency of the motor which, for a 50 pole stepper motor, is 50 times the mechanical frequency, ie., $\omega_e = N_r \omega_m$. As discussed later, this large difference between the electrical and mechanical bandwidth has a significant impact on the control design.

The AC Three phase brushless servo motor has a typical pole count of 8 making the difference between electrical and mechanical bandwidth less of an issue. The lower pole count also reduces the detent torque.

B.1.2 Feedback Linearization

The point of feedback linearization control is to find a (nonlinear) state space transformation such that, in the new coordinates, the nonlinearities may be cancelled out by state feedback. For the PM stepping motor, the appropriate nonlinear coordinate transformation is known as the Direct-Quadrature (DQ) transformation. The DQ transformation for the phase voltages and currents are defined as follows:

$$\begin{bmatrix} v_d \\ v_q \end{bmatrix} = \begin{bmatrix} \cos(N_r\theta) & \sin(N_r\theta) \\ -\sin(N_r\theta) & \cos(N_r\theta) \end{bmatrix} \begin{bmatrix} v_a \\ v_b \end{bmatrix} \tag{16}$$

$$\begin{bmatrix} i_d \\ i_q \end{bmatrix} = \begin{bmatrix} \cos(N_r\theta) & \sin(N_r\theta) \\ -\sin(N_r\theta) & \cos(N_r\theta) \end{bmatrix} \begin{bmatrix} i_a \\ i_b \end{bmatrix} \tag{17}$$

The direct current i_d represents the component of the stator magnetic field along the axis of the rotor magnetic field, while the quadrature current i_q gives the component of the stator magnetic field perpendicular to the rotor.

The application of the DQ transformation to the original system (15) yields the following system of equations.

$$\begin{aligned}
\frac{di_d}{dt} &= (v_d - Ri_d + N_r L\omega i_q)/L \\
\frac{di_q}{dt} &= (v_q - Ri_q - N_r L\omega i_d - K_m\omega)/L \\
\frac{d\omega}{dt} &= (K_m i_q - B\omega)/J \\
\frac{d\theta}{dt} &= \omega
\end{aligned} \tag{18}$$

Variable v_d is the direct voltage, v_q is the quadrature voltage, i_d is the direct current, i_q is the quadrature current, ω is the angular velocity and θ is the angular position.

Although the resulting system (18) is till nonlinear, the nonlinear terms can now be canceled by state feedback. Another significant advantage of this coordinate transformation is that the transformed currents i_d and i_q now vary approximately at the mechanical frequency of the motor. This enables the controller to be operated at a much lower frequency, since the mechanical variables typically have bandwidths in the range of 0-100 Hz compared with 0-5 kHz bandwidth for v_a, v_b, i_a and i_b.

The form of system (18) suggests choosing a v_d and v_q to be of the following form:

$$\begin{aligned}
v_d &= Ri_d - N_r L\omega i_q + Lu_d \\
v_q &= Ri_q + N_r L\omega i_d + K_m\omega + Lu_q
\end{aligned} \tag{19}$$

The system then becomes

$$\frac{di_d}{dt} = u_d \tag{20}$$

$$\frac{di_q}{dt} = u_q \tag{21}$$

$$\frac{d\omega}{dt} = (K_m i_q - B\omega)/J \tag{22}$$

$$\frac{d\theta}{dt} = \omega \tag{23}$$

Note that the original fourth-order system has been transformed into a first-order linear system (20), decoupled from a third-order linear system (equations (21) through (23)). Therefore, linear control techniques can be used for the system in the new variables (20) and (21). This is discussed in the next section. Note that, while the controller has been presented in a simple and intuitive manner, it can be shown to be a special example of feedback linearization theory.

B.1.3 Design of the Current References

While the linearized system is decoupled, constraints on the magnitude of the voltages introduce couplings. To achieve a desired speed ω_d and acceleration α_d, the required quadrature current is found from equation (22) to be

$$i_{qd} = \frac{J}{K_m}\alpha_d + \frac{B}{K_m}\omega_d \tag{24}$$

The quadrature component i_q of the current produces torque while the direct component i_d does not produce any torque. However, in order to attain higher speeds, it is necessary to apply a *negative* direct current to cancel the effect of the back-emf of the motor. Specifically, note form the second equation of (18 that the "back-emf" term in the DQ coordinates is $K_m\omega$. With K_m taken to be 0.2 (see Table 1), the back-emf voltage at a speed of 2000 rpm is 41 volts. A decoupling control law as in (19 requires that v_q *cancel* this back-emf term. For our setup, the source voltage is 40 volts, so that cancellation of the back-emf would lead to saturation at the input. However, if i_d were forced to be negative (field-weakening) by the correct choice of v_d, the term $N_r L\omega i_d$ would help to cancel the back-emf term $K_m\omega$ (see (18) or (19). the design of an appropriate reference for for i_d is thus essential to avoid saturation of the phase voltages at high rotor speeds.

The desired direct current i_{dd} is found by maximizing the torque (*i.e.*, i_q) at constant speed subject to the constraint $v_d^2 + v_q^2 = v_a^2 + v_b^2 \leq V^2$. However, the latter constraints leads to a simple analytical expression for i_{dd} as is now shown. Using the standard Lagrange multiplier technique, the maximum is shown to correspond to

$$\frac{v_d}{v_q}(\omega) = -\frac{N_r\omega L}{R} \tag{25}$$

$$i_{dd}(\omega) = -\frac{N_r L K_m \omega^2}{R^2 + (N_r\omega L)^2} \tag{26}$$

Field weakening is a important part of modern motor control. This applies to AC Brushless servo motors, Induction motors and Stepping Motors as described above. Equation (26) is based on Lyapunov. The high pole count of the Stepping motor mandates this type of control, if any reasonable high speed operation is to be expected (In industry, this is usually termed as phase advance). As stated above, the choice of the constraint $v_d^2 + v_q^2 \leq V^2$ ***may simplify the derivation leading up to (25) and (26) but may not be the optimum when factoring in the operation of the servo amplifier. Selecting the constraints*** $|v_a| \leq v_{max}, |v_a| \leq v_{max}$ ***in general presents a more complicated derivation, essentially providing the same result.***

When factoring in the behavior of the amplifier, both constraints ignore the fact that the servo amplifier may generate more absolute current than may be necessary to produce torque at a given operating point.

The servo amplifier sees current as loss no matter if this current is real or reactive, relative to the motor. These loses come in three forms: simple resistive loses, DC losses across the forward conduction points of the semiconductor elements, and switching losses of the semiconductors. More importantly, the amplifier may not be capable of producing the current required to satisfy (26). To protect the amplifier, we need to apply a clamp based on the result of the sum-of-squares-square-root of i_{dd} and i_{dq} (absolute magnitude). Without altering the effect of 26 within the safe operating range, a fix to this problem that when we exceed the absolute magnitude we reduce i_{dd}.

The relationship (25) is often referred to as the optimal lead-angle. The inverse tangent of (25) yields the angle advance that the phase voltages v_a,v_b need to be commanded relative to the angle $N_r\theta$ (corresponding to the position of the rotor). The equivalent relationship (26) is not usually derived in the literature, but is better suited to our purposes. Although (25) and (26) were derived assuming constant speed, it has been shown by Brown et.al. that the time optimal control (point-to-point, final speed unconstrained) of a stepper motor is only 1% better than the optimal lead-angle control (25) and (26).

B.2 Feedback Controller Development

B.2.1 Reference Trajectory

A specified position θ_d, speed $d\omega_d = \theta_d/dt$ and acceleration $\alpha_d = d\omega_d/dt$ are chosen. Figure 5 shows a typical speed profile. Position and acceleration of course are represented by the integral and derivative of this profile, respectively. After choosing the desired reference trajectory $[\theta_d, \omega_d, \alpha_d]^T$, the corresponding state trajectory $[i_{dd}, i_{qd}, \theta_d, \omega_d]^T$ and reference input voltages v_{dd}, v_{qd} were chosen. They were found so as to satisfy.

$$\frac{di_{dd}}{dt} = (v_{dd} - Ri_{dd} + N_r L\omega i_q)/L \tag{27}$$

$$\frac{di_{qd}}{dt} = (v_{qd} - Ri_{qd} - N_r L\omega i_{dd} - K_m\omega_d)/L \tag{28}$$

$$\frac{d\omega_d}{dt} = (K_m i_{qd} - B\omega_d)/J \tag{29}$$

$$\frac{d\theta_d}{dt} = \omega_d \tag{30}$$

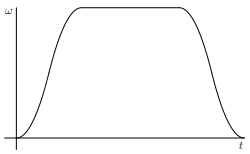

Figure 39: Reference trajectory.

The desired quadrature current was found by solving (29) for i_{qd} (see(24)) and its derivative, di_{qd}/dt, is then found by differentiating this expression. The desired direct current i_{dd} was defined by (26) and its derivative is found by simply differentiating this expression. Doing these calculations gives:

$$i_{dd}(\omega_d) = -\frac{N_r L K_m \omega_d^2}{R^2 + (N_r \omega L)^2} \tag{31}$$

$$i_{qd} = \frac{J}{K_m}\alpha_d + \frac{B}{K_m}\omega_d \tag{32}$$

$$\frac{di_{dd}}{dt}(\omega_d) = -\frac{2N_r L K_m R^2 \omega_d \alpha_d}{R^2 + (N_r \omega L)^2} \tag{33}$$

$$\frac{di_{qd}}{dt} = \frac{J}{K_m}\dot{\alpha_d} + \frac{B}{K_m}\alpha_d \tag{34}$$

The reference trajectories for i_{dd} and i_{qd} corresponding to θ_d and ω_d ($omega_d$ as represented by Figure 6 with it's integral θ_d). Solving (27)-(30) for v_{dd} and v_{qd} gives

$$v_{dd} = L\frac{di_{dd}}{dt} + Ri_{dd} - N_r\omega_d Li_{qd} \tag{35}$$

$$v_{qd} = L\frac{di_{qd}}{dt} + Ri_{qd} + N_r\omega_d Li_{dd} + K_m\omega_d \tag{36}$$

Simulation has shown that these reference input voltages v_{dd} and v_{qd} do not saturate the amplifier.

The above state reference trajectory and reference input voltage were used for the design of a state feedback controller (SFC).

B.2.2 Full State Feedback Controller Development (SFC)

We assume that current measurements are available and use them in the linear controller by letting

$$v_d = -N_r\omega Li_q + v_{dd} + N_r\omega_d Li_{qd} + Lu_d \tag{37}$$

$$v_q = N_r\omega Li_d + v_{qd} - N_r\omega_d Li_{dd} + Lu_q \tag{38}$$

Substituting equations (37) and 38) for v_d and v_q result in the following:

$$\frac{di_d}{dt} = (Lu_d - Ri_d + v_d + N_r L\omega_d i_{qd})/L \tag{39}$$

$$\frac{di_q}{dt} = (Lu_q - Ri_q - K_m\omega + v_{qd} - N_r L\omega_d i_{dd})/L \tag{40}$$

$$\frac{d\omega}{dt} = (K_m i_q - B\omega)/J \tag{41}$$

$$\frac{d\theta}{dt} = \omega \tag{42}$$

Equations (37) and (38) contain the nonlinear cancellation terms, the desired reference voltages and the new inputs u_d and u_q which are used to stabilize the eerror system as is now shown.

$$\epsilon = \begin{bmatrix} i_d - i_{dd} \\ i_q - i_{qd} \\ \omega - \omega_d \\ \theta - \theta_d \\ \xi - \xi_d \end{bmatrix} \tag{43}$$

Where $\xi = \int_0^t \theta(t)\, dt$. The integrator is added to the controller to eliminate the steady-state error due to disturbances.

Subtracting (27)-(30) from equations (39)-(42) results in the following error dynamics:

$$\dot{\epsilon} = \begin{bmatrix} -R/L & 0 & 0 & 0 & 0 \\ 0 & -R/L & -K_m/L & 0 & 0 \\ 0 & K_m/J & -B/J & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \epsilon + \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} u_d \\ u_q \end{bmatrix} \tag{44}$$

or, more compactly,

$$\dot{\epsilon} = A\epsilon + Bu \tag{45}$$

with the obvious definitions for A and B.

Through the use of a nonlinear state transformation, input transformation, and nonlinear feedback, a *linear time-invariant* error system has been obtained. The feedback is defined to be

$$u = -K\epsilon \tag{46}$$

with K defined to ave the form

$$K = \begin{bmatrix} k_{11} & 0 & 0 & 0 & 0 \\ 0 & k_{22} & k_{23} & k_{24} & k_{25} \end{bmatrix} \tag{47}$$

As the system is controllable, Ackerman's formula was used to find the gain matrix K that places the poles of the closed-loop error system.

References

[1] Marc Bodson, John N. Chiasson, Robert T. Novotnak and Ronold B. Rekoswski
High Performance Nonlinear Feedback Control of a Permanent Magnet Stepping Motor.
First IEEE Conference on Control Applications, 1992. ISBN 0-7803-0047-5.

[2] Marc Bodson and John N. Chiasson
Brushless DC Motor Model (class notes)

[3] Jin-Woo Jung
Project #2 Space Vector PWM Inverter, February 20, 2005.
Mechatronic Systems Laboratory, Department of Electrical and Computer Engineering, Ohio State University
`http://meaconsultingdotorg.files.wordpress.com/2015/12/spacevector_pwm_inverter.pdf`

[4] Raja Ayyanar, D. Zhao, H. Krishnamurthy, G. Narayanan
Space Vector Methods for AC Drives to Achieve High Efficiency and Superior Waveform Quality, March 31, 2004
Office for Research and Sponsored Projects Administration, Arizona State University
`http://meaconsultingdotorg.files.wordpress.com/2015/12/gettrdoc1.pdf`

[5] Vieri Xue
Center-Aligned SVPWM Realization for 3-Phase 3-Level Inverter
Texas Instruments Application Report, SPRABS6, October 2012
`http://meaconsultingdotorg.files.wordpress.com/2015/12/sprabs6-three-level-space-vector.pdf`