# Quantized State Simulation of Electrical Power Systems

by

Joseph Micah Hood

Bachelor of Science
University of South Carolina 2007

Master of Science
University of South Carolina 2010

---

Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy in

Electrical Engineering

College of Engineering and Computing

University of South Carolina

2023

Accepted by:

Roger Dougal, Major Professor

Enrico Santi, Committee Member

Xiaofeng Wang, Committee Member

Peter Binev, Committee Member

Cheryl L. Addy, Interim Vice Provost and Dean of the Graduate School

# Abstract

An alternative is proposed to the current state-of-the-art simulation methods for the transient simulation of electrical power systems. The proposed method combines the Latency Insertion Method (LIM), the Quantized Discrete Event Specification (QDEVS), and the Quantized State System (QSS) method of integration. Using LIM, the power system state equations are decoupled in a way that allows the formulation of the system into a QDEVS-compliant model, which can then be directly solved with various QSS integration techniques. This combination of methods is called the Quantized DEVS-LIM method, or simply, QDL. A key feature of QDL is the asynchronous updates of all system states, which has many implications. Some benefits of QDL for power system simulation are improved computational efficiency and the ability to model, within the same simulation, the wide range of fast and slow dynamic behavior inherent to power systems. Also, by careful selection of the quantization step size for each state, the desired balance of accuracy and speed can be achieved. Two object-oriented QDL simulator prototypes are developed, a MATLAB implementation for the efficient simulation of linear QDL systems, and a Python implementation for non-linear QDL systems using advanced QSS integration techniques. The suitability of QDL for very stiff systems is demonstrated by successfully simulating a system with a stiffness ratio around $10^9$. Also, the feasibility of applying the QDL method to real-world power systems is tested with a reasonably large (32 state), multi-machine, highly nonlinear power system. Some challenges encountered are the presence of steady-state oscillations, the difficulty in selecting acceptable quantization step sizes for larger systems, and inefficiencies in larger simulations due to excessive numerical

noise. Several solutions to these problems are proposed and tested to various levels of success. Overall, the QDL method is shown to be a promising new approach to power system simulation that can provide important benefits over existing methods, especially if the remaining limitations can be adequately addressed.

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

The purpose of this work is to investigate the use of a novel quantized state method for the time-domain simulation of electrical power systems, and to determine if this method is a feasible and advantageous approach to simulating power system dynamics compared to the current state-of-the-art methods.

## 1.1 The Proposed Simulation Method: QDL

The proposed modeling and simulation method combines three concepts: the Latency Insertion Method (LIM), Quantized Discrete Event Specification (QDEVS), and Quantized State Systems (QSS). The method is therefore called QDEVS-LIM, (abbreviated throughout this document as QDL). The LIM, described in [1], can be thought of as a system partitioning method, such as using Bergeron transmission lines to decouple large electrical network models by exploiting the latency inherent to long transmission lines. The LIM, however, takes this concept all the way down to the individual electrical node level of the network, and removes instantaneous coupling between all system states. This produces a model that can be formulated using Quantized Discrete Event Specification (QDEVS). QDEVS, proposed in [2], is a specification for simulating a discrete event system with quantized signals. The specification defines the concept of "atoms", which are atomic agents in the system that have state. These atoms obey specific rules for updating themselves and communicating information to other atoms ([2]). Finally, once the system model is QDEVS compliant, Quantized State Systems (QSS) integration methods, first described in

[3], can solve the time evolution of a system.

In summary, QDL uses LIM to decompose an electrical system model into QDEVS atoms so QSS solutions can be used to produce a solution to the time evolution of the system. The following sections will attempt to explain why this is desirable.

## 1.2 Contrasting Quantized State Solutions to Time-slicing Solutions

There are significant differences between time-slicing and quantized methods. In short, time-slicing methods answer the question "At each time step, what is the system state?", while quantized state methods answer the question "At what times did each state move up or down by one quantization step?"

Time-slicing integration methods typically solve all system states synchronously at a fixed or variable time step. Given the system's previous state, the set of system parameters, and system's boundary conditions, a snapshot of the system state is created at each time step. The solution is therefore "sliced" along the dimension of time. In contrast, quantized state integration methods, effectively slice the solution along the dimension of state quantities. A graphical representation of this is shown in figure 1.1, depicting the output of a conceptual $2^{nd}$ order dynamical system simulation. The curves on the left plot represents a typical, fixed time step, numerical integration solution. This solution may be, for example, from an implicit method with high accuracy, where each time step's solution is very close to the exact (analytical) solution. Typically, one would apply a linear interpolation between the points (as shown) for post-simulation visualization and analysis.

In the quantized solution on the right in figure 1.1, the quantized state results are not solved synchronously for all system states, and are only defined for each state at multiples of that state's quantization step size ($\Delta Q$). Note that the quantized state solution points are connected with a zero-order hold curve. This is because the

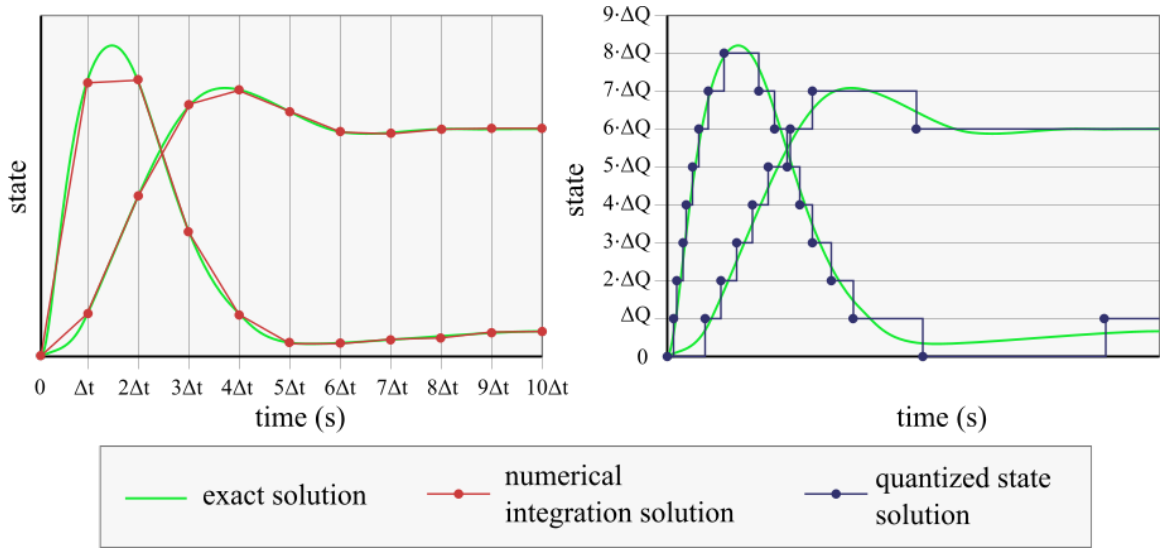



Figure 1.1 Example simulation results for a conceptual $2^{nd}$ order dynamical system, comparing a time-slicing numerical integration method to a quantized state method, where $\Delta t$ is the time step, and $\Delta Q$ is the quantization step size.

results of a $1^{st}$ order quantized state solution are defined as piece-wise constant [4] and they are typically rendered accordingly in plots.

## 1.3 Rationale for Applying Quantized State Solutions to Power Systems

In order to simulate the full range of dynamic behavior in a typical electrical power system with a single dynamical system model, a stiffness ratio ($\lambda_{max} \backslash \lambda_{min}$) of $10^6$ or higher must be supported. Time constants of a power system typically span from the turbo-governor (with time constants on the order of 1 second), to the very fast dynamics of a switching converter (with micro-second time constants). The uniform time-slicing of nodal analysis can potentially simulate such a high-bandwidth system model only by using time steps on the order of micro-seconds. Each of these micro-second time steps requires a full update of the system Jacobian matrix (in the case of non-linear models, which is required for practical, real-world system models). This

constant update of the Jacobian matrix forces a constant matrix factorization at each micro-second time step. The computational load (flops per simulation second) for this matrix factorization grows exponentially with increases in system size. For large transmission grids, the electrical node count (and thus the order of the Jacobian matrix) can be as high as hundreds of thousands. Variable time step methods are of little use here, as the simulation of a detailed switching converter requires dense time-slicing between each switch transition, even when the system is in a quasi-steady-state condition.

Quantized state solutions can potentially solve this extreme stiffness problem, and because most of the current literature on quantized state integration revolves around small, linear systems, this is a novel area for research. The simulation of larger, stiffer, and highly non-linear systems is a new area for the application of quantized methods.

## 1.4 Document Outline

This dissertation will begin in chapter 2 by describing the Latency Insertion Method, specifically the formulation most relevant to our needs. We cannot apply quantized state solution methods directly to a traditional formulation of a power system. The LIM will allow us to cast the power system model into a form that can be used in a discrete event context by enforcing latency at every system state. The QSS integration concepts are then introduced in chapter 3, describing the basic properties of Quantized State Systems. The formal QDEVS specification for a LIM-based system model is then developed in chapter 4, defining the QDEVS functions that operate on a LIM-based system model. The MATLAB and Python implementations of QDL simulations is discussed in chapter 5 (with the full source code of these implementations listed in Appendix A).

The next three chapters contain simulations of systems of increasing complexity. Chapter 6 presents several interesting test cases for the QDL method that demon-

strate various properties and advantages of the method. Among these are a 40 state, linear grid of LIM branches and nodes, with an extreme stiffness ratio designed to test the limits of the method. Following those examples, chapters 7 and 8 finally apply the QDL method to realistic power system models and systems. Chapter 7 presents a $7^{th}$ order, three-phase synchronous machine attached to an infinite bus. The feasibility of the QDL method for such a model is demonstrated, and the simulation result match the reference simulation very well. The synchronous machine simulation is then used to investigate the relationship between quantization step size and error. Chapter 8 applies QDL to a relatively large problem: a multi-machine power system network with 12 device models including cables, buses, a synchronous generator, an induction machine, loads, an (average model) converter, and control devices (an exciter and a governor). This system is complex and realistic enough to test the thesis of this work: to determine if a quantized state method is a feasible and advantageous approach to simulating power system dynamics

Chapter 9 explores the important question of how to select the quantization step size ($\Delta Q$) to optimize simulation performance and control error. A method is proposed to intelligently select $\Delta Q$ based on the propagation of error within the QDL system model. The problems with steady-state behavior of the QDL method is discussed in chapter 10, along with a proposal for a method of detecting steady-state conditions and mitigating steady-state noise. Finally, chapter 11 describes possible next steps for the research and development of this new method, including further investigation into better methods for selecting $\Delta Q$, a search for better error and noise mitigation, and integrating the most recent, advanced QSS integration methods.

# Chapter 2

# Latency Insertion Method

## 2.1 Rationale for using LIM for Quantized State Simulation

In order to apply Quantized State System (QSS) integration methods, the study system must be described fully by a set of differential equations without any instantaneous coupling between any of the state variables in the system [3]. To achieve this, we will use a modeling approach called the Latency Insertion Method (LIM). The LIM, described in [1] exploits existing latency in the network and inserts small, fictitious latency at nodes and branches where no significant physical latency exists. It provides generic node and branch models, that can be combined into arbitrarily complex typologies to model various electrical devices (or any dynamical devices that can be modeled with an equivalent circuit model). The equivalent circuit model representation is powerful because it imposes conservation law inherently, and it is flexible enough to model dynamical subsystems of many engineering disciplines (mechanical, thermal, etc.).

With latency at every system node and on every system branch, there is no instantaneous coupling between any states. One way to understand this is to imagine a state space system with no off-diagonal components in the state matrix. All of state coupling occurs within a dynamically-updated input vector. And, although there are proposed versions of the LIM for dealing with algebraic sub-networks [5], these methods do not create a model that can be QDEVS compliant. We will therefore focus on creating fully latent networks using the original LIM formulation.

## 2.2 Atomic LIM Components

The LIM method uses two primary generic components from which the network model can be developed: a LIM *node* and a LIM *branch*. The LIM node defines a KCL-based ODE for the state of a node voltage, and the LIM branch defines a KVL-based ODE for the current flow between two arbitrary nodes. For the QDL method development, we will use the extended LIM node and branch formulations that include dependent sources as described in [6]. This extended formulation provides a straight-forward means of modeling energy-conversion coupling circuits, which are common in power system models.

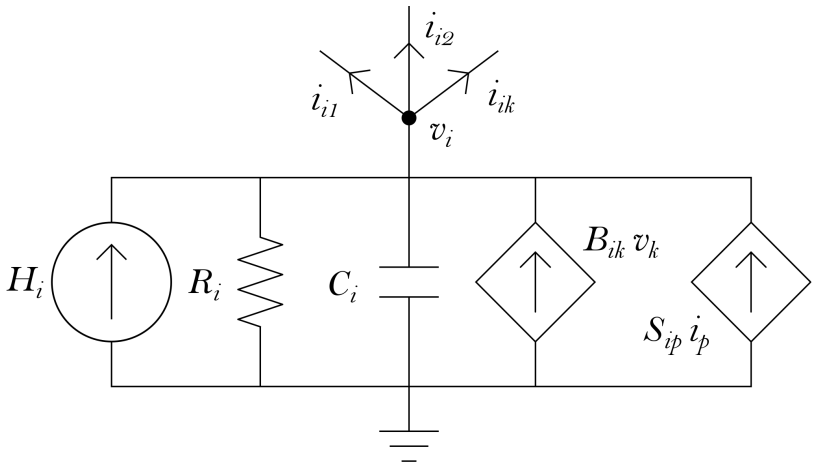



Figure 2.1 Generic LIM Node with Dependent Sources

The generic LIM branch model with dependent sources is shown in figure 2.1. The model includes a voltage-controlled current source (VCCS) and a current-controlled current source (CCCS). The KCL equation for the $i^{th}$ node is

$$C_i \frac{d}{dt} v_i(t) + G_i v_i(t) - H_i(t) - B_{ik} v_k(t) - S_{ip} i_p(t) = \sum_{M_i}^{k=1} i_{ik}(t) \tag{2.1}$$

Note that these parameters, injections and coefficients can be time-varying and non-linear in general.

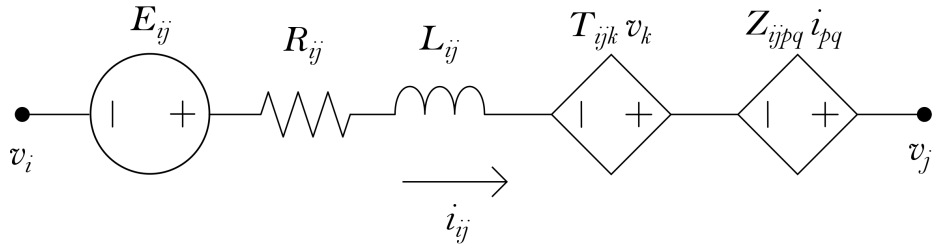



Figure 2.2 Generic LIM Branch with Dependent Sources

The generic LIM branch model with dependent sources is shown in figure 2.2. The model includes a voltage-controlled voltage source (VCVS) and a current-controlled voltage source (CCVS). The KVL equation for a branch from $i^{th}$ to the $j^{th}$ node is

$$v_i(t) - v_j(t) = L_{ij}\frac{d}{dt}i_{ij}(t) + R_{ij}i_{ij}(t) - e_{ij}(t) - T_{ijk}v_k(t) - Z_{ijpq}i_{pq}(t) \qquad (2.2)$$

Note that these parameters, injections and coefficients can be time-varying and non-linear in general.

The LIM formulation also allows for externally-controlled, ideal current source branches and voltage source nodes (see figs. 2.2 and 2.2). Although these components are somewhat trivial, they are useful in the derivation of many device models and are worth noting. Because the voltages at the branch ports are effectively dc quantities (as far as the branch model is concerned) during the span of a time step, an ideal source does not require any special consideration by the branch model. Note that ideal voltage nodes may not be added in parallel attached to the same electrical node, and ideal current source branches may not be added in series with other branch models.

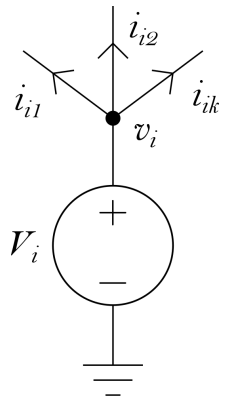



Figure 2.3 Ideal Voltage Source Node

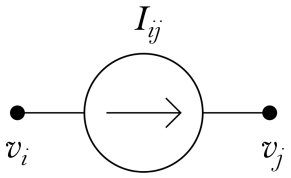



Figure 2.4 Ideal Current Source Branch

## 2.3 LIM System Model

It is convenient to represent a LIM system model as a state space model, with partitions as shown in (eq. 2.3). By definition, the LIM state space model does not contain an algebraic output equation but is described fully with the state equation $\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t)$. The motivation for developing the LIM system state space model is two-fold. We wish to provide a trusted benchmark solution with which to compare QDL simulation accuracy and computational performance, and the state space model is a common format that can be solved with various third-party tools. Also, various components of the LIM state space model will be used directly in the formulation of the QDL atomic model equations described in chapter 4.

$$\frac{d}{dt}\begin{bmatrix} v \\ i \end{bmatrix} = \begin{bmatrix} C^{-1}(B-G) & C^{-1}(S-A) \\ L^{-1}(T-A^T) & L^{-1}(Z-R) \end{bmatrix} \begin{bmatrix} v \\ i \end{bmatrix} + \begin{bmatrix} C^{-1} & 0 \\ 0 & L^{-1} \end{bmatrix} \begin{bmatrix} H \\ E \end{bmatrix} \tag{2.3}$$

where:

$v$ : the set of node voltages of length $n$

$i$ : the set of branch currents of length $k$

$C$ : the set of node shunt capacitance values of length $n$

$G$ : the set of node shunt conductance values of length $n$

$H$ : the set of node shunt current injections of length $n$

$B$ : the set of node VCCS gains of size $(n \times n)$

$S$ : the set of node CCCS gains of size $(n \times k)$

$L$ : the set of branch series inductance values of length $k$

$R$ : the set of branch series resistance values of length $k$

$E$ : the set of branch series voltage sources of length $k$

$T$ : the set of branch VCVS gains of size $(k \times n)$

$Z$ : the set of branch CCVS gains of size $(k \times k)$

$A$ : the port connection incidence matrix size $(n \times k)$

and the elements of the incidence matrix $A$ are determined by

$$A_{i,k} = \begin{cases} 1, & \text{if the } i^{th} \text{ node is connected to the } i^{th} \text{ terminal of the } k^{th} \text{ branch} \\ -1, & \text{if the } i^{th} \text{ node is connected to the } j^{th} \text{ terminal of the } k^{th} \text{ branch} \\ 0, & \text{otherwise} \end{cases} \tag{2.4}$$

## 2.4 Example LIM System Derivation

In order to provide a concrete example of a simple power system formulated using LIM, a dc motor model is presented (fig. 2.4). This system includes an electro-mechanical energy conversion in the form of back-to-back VCVS and CCCS sources.

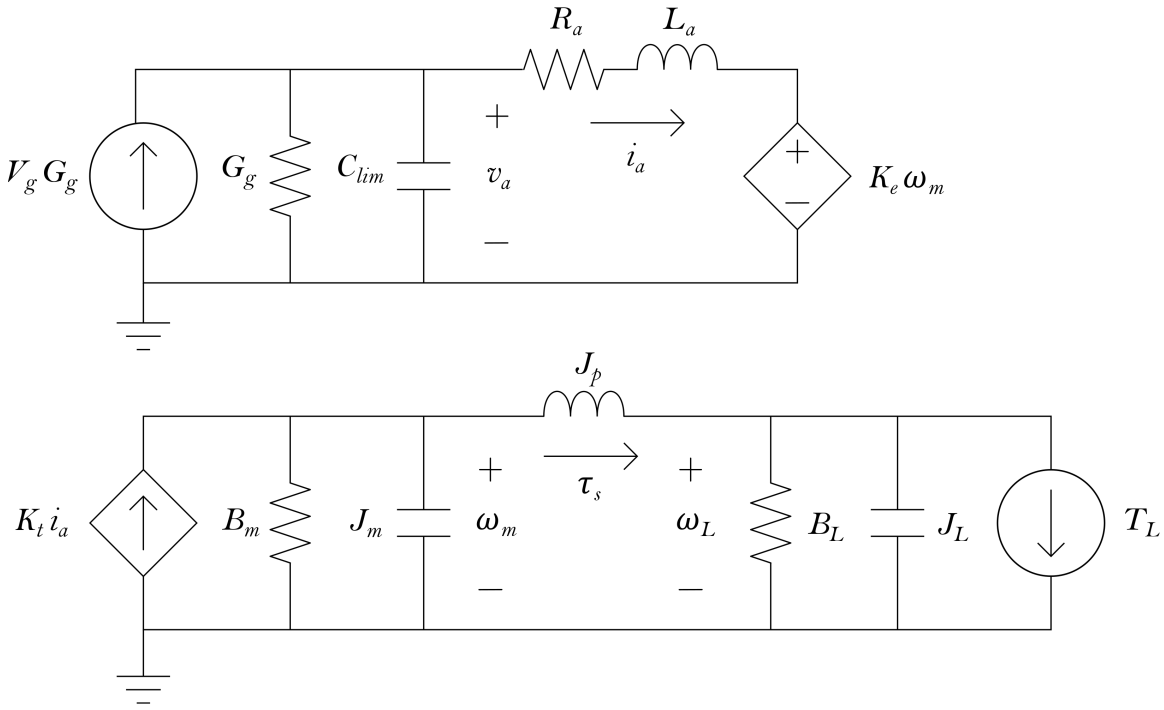



Figure 2.5 Example DC Motor System LIM Model with Dependent Sources

The LIM system is then defined with the following components:

$$v = [v_a \;\; \omega_m \;\; \omega_L]^\intercal, \quad i = [i_a \;\; \tau_s]^\intercal \tag{2.5}$$

$$H = [V_g G_g \;\; 0 \;\; T_L]^\intercal, \quad E = [0 \;\; 0]^\intercal \tag{2.6}$$

$$R = \begin{bmatrix} R_a & 0 \\ 0 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} L_a & 0 \\ 0 & J_p \end{bmatrix}, \quad G = \begin{bmatrix} G_g & 0 & 0 \\ 0 & B_m & 0 \\ 0 & 0 & B_L \end{bmatrix},$$
$$C = \begin{bmatrix} C_{lim} & 0 & 0 \\ 0 & J_m & 0 \\ 0 & 0 & J_L \end{bmatrix}, \quad S = \begin{bmatrix} 0 & 0 \\ K_t & 0 \\ 0 & 0 \end{bmatrix}, \quad T = \begin{bmatrix} 0 & K_e & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{2.7}$$

Note that the armature terminal node does not include any real latency, and so a small, fictitious capacitance of $C_{lim}$ is included in order to ensure that the system has full latency and can be represented by a valid LIM system model.

# Chapter 3

# Quantized State Systems

The Quantized Discrete Event Specification (QDEVS) [7] provides a formal DEVS specification for a discrete event description of quantized systems. The Quantized State System (QSS) methods are a series of integration methods based on the QDEVS specification and described in Zeigler et al. [7]. These QSS methods provide a QDEVS-compliant way to simulate continuous systems.

## 3.1 Summary of QSS

This approach begins with the assumption that a generic continuous state equation system (SES)

$$\dot{\mathbf{x}} = f(\mathbf{x}(t), \mathbf{u}(t)) \tag{3.1}$$

can be approximated by a *Quantized* State Systems (QSS) in the form

$$\dot{\mathbf{x}} = f(\mathbf{q}(t), \mathbf{u}(t)) \tag{3.2}$$

where $\mathbf{q}(t)$ is the quantized state vector that follows a piecewise constant trajectory, and is related to the state vector $\mathbf{x}(t)$ by the quantum size $\Delta Q$. The states are quantized based on their trajectory using various hysteresis quantization functions depending on the specific QSS algorithm used. In [8] and [9] the structure and implementation of atomic DEVS models and general-purpose simulators for QSS systems are defined. In the linear case, the QSS approach guarantees a bounded error,

[4] so analytically stable systems cannot become numerically unstable when being simulated by a fully coupled QSS algorithm. [9] Several variations of QSS offer different features. The simplest formulation, QSS1, developed in Zeigler et al. [7], and Kofman and Junco [4], relies on explicit integration and uses first-order estimates of state derivatives to predict the time at which the continuous state $x_i(t)$ will increase or decrease by amount $\Delta Q$ (quantization step size) from the current quantized value $q_i(t)$ to the next higher or lower quantized value.

## 3.2 QSS Method Selection for QDL

Although QSS1 has some advantages, such as being easy to implement, its disadvantage is that it uses a first-order approximation of the state trajectory to calculate the time to the next event. To get accurate results, $\Delta Q$ has to be quite small, which produces a large number of steps. The QSS2 [10] and QSS3 [11] methods use more accurate second- and third-order approximations, respectively, for the state trajectory, however, the computational cost grows with the square root and cubic root of the desired accuracy. Because we are interested in simulating realistic, stiff power systems that include both fast electrical dynamics and slow mechanical dynamics, we require a QSS method that can handle stiff systems. This requirement eliminates QSS1, QSS2, and QSS3 because these methods create fictitious high-frequency oscillations for stiff systems which in turn generate large numbers of steps. These steps are costly in computational cost and memory size, even when a system is nominally in steady state. To address this, a Linear-Implicit QSS (LIQSS1) algorithm is used as proposed in [12]. The LIQSS1 method uses a semi-implicit solution in the quantization function to force steady-state derivatives to zero and prevent troublesome oscillations for stiff systems. We will use the LIQSS1 algorithm for the development of the QDL method in the following chapter.

# Chapter 4

# The Quantized DEVS-LIM (QDL) Method

The formal QDEVS function definitions for the proposed QDL method are presented in this chapter.

## 4.1 QDL QDEVS Specification

The original DEVS specification is outlined in [13], and further developed for quantized systems (QDEVS) in [2]. The specification consists of several functions that need to be defined for a specific QDEVS model. These are

$f$: Derivative Update function

$\delta_{int}$: Internal State Transition Function

$\delta_{ext}$: External State Transition Function

$ta$: Time Advance Function, and

$Q$: Quantization Function.

These are defined below for the LIM atomic models presented in chapter 2, and for the LIQSS1 integration method in [14].

### 4.1.1 QDL Derivative Update Function

The derivative estimate is then determined from the LIM model for the nodes with

$$d_i = \frac{H_{ii}}{C_{ii}} - \frac{G_{ii}}{C_{ii}} \cdot q_i + \frac{1}{C_{ii}} B_i \cdot q^{node} + \frac{1}{C_{ii}} \left(S_i - A_i\right) \cdot q^{branch}, \tag{4.1}$$

and for the branches with

$$d_k = \frac{E_{kk}}{L_{kk}} - \frac{R_{kk}}{L_{kk}} \cdot q_k + \frac{1}{L_{kk}} Z_k \cdot q^{node} + \frac{1}{L_{kk}} \left(T_k - A_k^T\right) \cdot q^{branch}, \tag{4.2}$$

where $C_{ii}$, $G_{ii}$, $H_{ii}$, $R_{kk}$, $L_{kk}$, and $E_{kk}$, are quantities from the LIM model at node $i$ and branch $k$. $B_i$, $S_i$, and $B_i$ are quantities from the LIM model at row $i$. $Z_k$, $T_k$, and $Z_k$ quantities from the LIM model at row $k$. $q_i$ is the quantized state at node $i$, $q_k$ is the quantized state at branch $k$, $q^{node}$ is the vector of node quantized states, and $q^{branch}$ is the vector of branch quantized states.

### 4.1.2 QDL Internal State Transition Function

The internal state for each QDL atom is calculated by the internal transition function $\delta_{int}$. An internal transition occurs when the simulation time has advanced to the atom's $t^{next}$ value determined by its time advance function $ta$.

In this function, the node voltage states are updated as

$$v_i = v_i^{last} + d_i^{last} \cdot \left(t - t_i^{last}\right), \tag{4.3}$$

and the branch currents are updated as

$$i_k = i_k^{last} + d_k^{last} \cdot \left(t - t_k^{last}\right), \tag{4.4}$$

where $v_i$ and $i_k$ are the voltage at node $i$ and the current at branch $k$ respectively.

After the states are updated, the $t^{last}$ values are saved:

$$t_i^{last} = t, \quad t_k^{last} = t. \tag{4.5}$$

### 4.1.3 QDL External State Transition Function

In the case of QDL, the *behavior* of the internal $\delta_{int}$ and the external $\delta_{ext}$ transition functions are identical. The difference is in how each is invoked. The internal transition is triggered when the simulation time $t$ has advanced to the atom's $t^{next}$ value determined in the $ta$ function, and the external transition is triggered when one or more connected atoms' quantized states change. Because the behavior of the internal and external transition functions is identical, the confluent transition function $\delta_{con}$ is not required.

### 4.1.4 QDL Time Advance Function

The time until the next internal transition is determined from the time advance function $ta$.

The time advance calculation for node $i$ is

$$t_i^{next} = \begin{cases} t_i + (\bar{q}_i - v_i)/d_i, & \text{if } d_i^{last} > 0 \\ t_i + (\underline{q}_i - v_i)/d_i, & \text{if } d_i^{last} < 0 \\ \infty, & \text{otherwise} \end{cases}, \tag{4.6}$$

and for branch $k$ is

$$t_k^{next} = \begin{cases} t_k + (\bar{q}_k - i_k)/d_k, & \text{if } d_k(t_k^{last}) > 0 \\ t_k + (\underline{q}_k - i_k)/d_k, & \text{if } d_k(t_k^{last}) < 0 \\ \infty, & \text{otherwise} \end{cases}, \tag{4.7}$$

where $\bar{q}$ and $\underline{q}$ are the upper and lower quantization limits respectively, and are dynamically updated by the quantization function described in the following section.

#### 4.1.5 QDL Quantization Function

The quantization function quantizes the internal state after a transition has occurred. The LIQSS1 method uses an advanced hysteresis function that tracks the sign of the derivative, determines when the state is oscillating between two quantized levels ($\pm\Delta Q$), and sets the quantized value such that the derivative is zero, using an approximation of the Jacobian diagonal. Below is the description of the LIQSS1 atomic DEVS model quantization method which is described in [12].

Note that the quantization function is the same for node voltages and branch currents, but only the equations for $\mathbf{v}$ and $v_i$ are included below for brevity. The same equations apply to the branch currents by substituting $\mathbf{i}$ and $i_k$ in place of $\mathbf{v}$ and $v_i$.

The following definitions will be used in the description of the quantization function:

$\mathbf{v}$ is the set of node voltages,

$\mathbf{u}$ is the set of external inputs,

$\mathbf{q}$ is the set of quantized states of all nodes and branches,

$\mathbf{J}$ is the Jacobian matrix,

$v_i$ is the value for the $i^{th}$ voltage at time $t$,

$\dot{v}_i$ is the time derivative of $v_i$,

$f_i$ is the time derivative function of $v_i$,

$\Delta Q_i$ is the quantization step size for the $i^{th}$ node,

$q_i$ is the quantized value of $v_i$,

$\underline{q}_i$ is the lower quantum limit of $v_i$,

$\bar{q}_i$ is the upper quantum limit of $v_i$,

$\hat{q}_i$ is the quantized value for which $\dot{v}_i = 0$, and

$\tilde{q}_i$ is the approximate quantized value for which $\dot{v}_i = 0$.

Note that the dependence of $v$, $u$, and $q$ on the current simulation time $t$ is implicit, except where the superscript *last* is used to denote quantities at the simulation time $t^-$, or the time of the previous quantization update.

$$q_i = \begin{cases} \underline{q}_i, & \text{if } f_i(\mathbf{q},\mathbf{u})\left(\underline{q}_i - v_i\right) \geq 0 \\ \bar{q}_i, & \text{if } f_i(\mathbf{q},\mathbf{u})\left(\bar{q}_i - v_i\right) \geq 0 \wedge f_i(\mathbf{q},\mathbf{u})\left(\underline{q}_i - v_i\right) < 0 \\ \tilde{q}_i, & \text{otherwise} \end{cases}, \tag{4.8}$$

with the upper and lower quantized limits defined as

$$\underline{q}_i = \begin{cases} \underline{q}_i^{last} - \Delta Q_i, & \text{if } v_i - \underline{q}_i^{last} \leq 0 \\ \underline{q}_i^{last} + \Delta Q_i, & \text{if } v_i - \underline{q}_i^{last} \geq 2\Delta Q_i \\ \underline{q}_i^{last}, & \text{otherwise} \end{cases}, \tag{4.9}$$

$$\bar{q}_i = \underline{q}_i^{last} + 2\Delta Q_i, \text{ and} \tag{4.10}$$

$$\tilde{q}_i = \begin{cases} \bar{q}_i - (1/A_{ii}) \cdot f_i(\bar{\mathbf{q}}^i), & \text{if } J_{ii} \neq 0 \\ q_i^{last}, & \text{otherwise} \end{cases}, \tag{4.11}$$

where $\bar{\mathbf{q}}^i$ is equal to $\mathbf{q}^{last}$ except for the $i^{th}$ component, where it is equal to $\bar{q}_i$ and $J_{ii}$ is the $ii^{th}$ component of the Jacobian matrix evaluated at $\bar{\mathbf{q}}^i$, .i.e.,

$$J_{ii} = \left.\frac{\partial f_i}{\partial v_i}\right|_{\bar{\mathbf{q}}^i, u^{last}}, \tag{4.12}$$

and therefore, when $J_{ii} \neq 0$, setting $q_i = \tilde{q}_i$ will force $\dot{v}_i = 0$ in the linear case. The calculation of $\tilde{q}_i$ is achieved by taking $\hat{\mathbf{q}}^i$ equal to $\bar{\mathbf{q}}^i$ except for the $i^{th}$ component is $\hat{q}_i$.

We solve for $\hat{q}$ (the point where $f_i = 0$) as

$$\hat{q}_i = \bar{q}_i - \frac{f_i\left(\bar{\mathbf{q}}^i, \mathbf{u}\right)}{J_{ii}} + \frac{g\left(\bar{\mathbf{q}}^i, \mathbf{u}\right) - q\left(\hat{\mathbf{q}}^i, \mathbf{u}\right)}{J_{ii}}, \tag{4.13}$$

where $g(\mathbf{x}, \mathbf{u}) = f(\mathbf{x}, \mathbf{u}) - J_i \cdot \mathbf{x}$, $J_{ii}$ is approximated by

$$J_{ii} \approx \frac{f_i\left(\bar{\mathbf{q}}^i, \mathbf{u}\right) - f_i\left(\underline{\mathbf{q}}^i, \mathbf{u}\right)}{\bar{q}_i - \underline{q}_i}. \tag{4.14}$$

### 4.1.6 Simulation Procedure

The atomic models are coupled via the topological connections as encoded in the LIM system port connection incidence matrix $A$, as well as the controlled source matrices $B$, $S$, $T$ and $Z$ (see eq. 2.3). These coupled models become the Coupled DEVS System as described in [2] and [15]. For the simple but illustrative case of two coupled LIM atoms, a node and a branch, the data flow can be visualized as in figure 4.1. Each atom maintains its internal state, and broadcasts changes in its external quantized state to the other atom via an event queue that manages the internal and external events of both atoms.

The full DEVS simulation procedure that implements the Coupled DEVS simulation is described in the flow chart in 4.2.

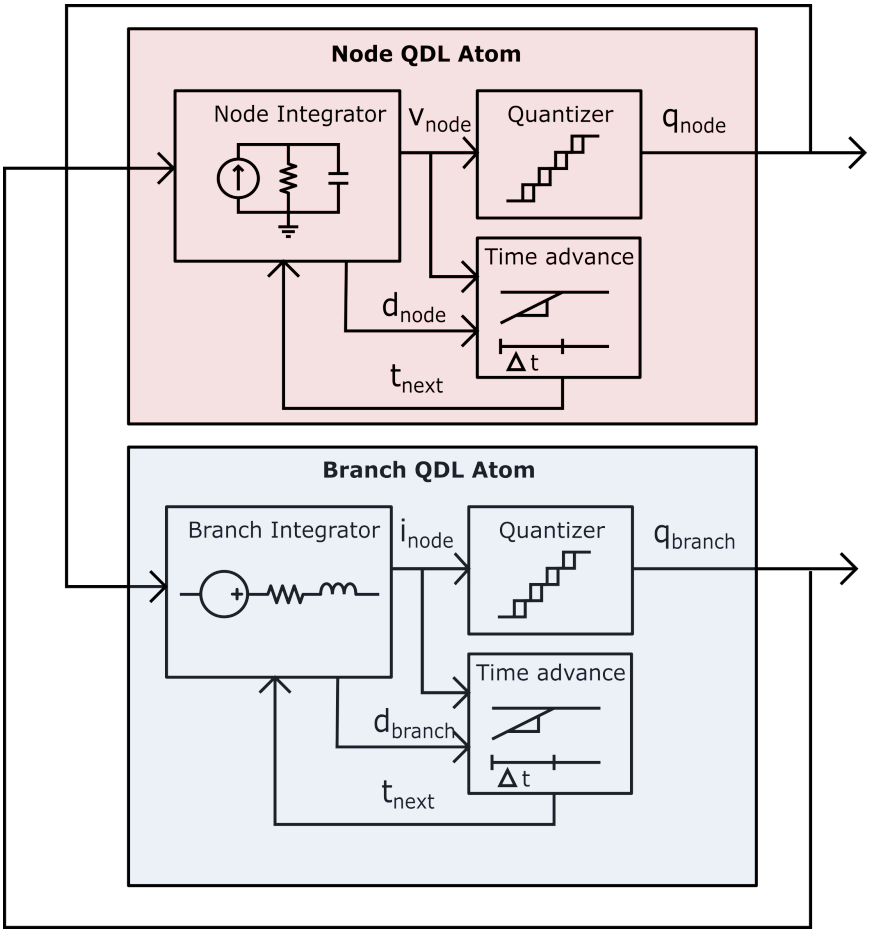



Figure 4.1 $2^{nd}$ order QDL system data flow.

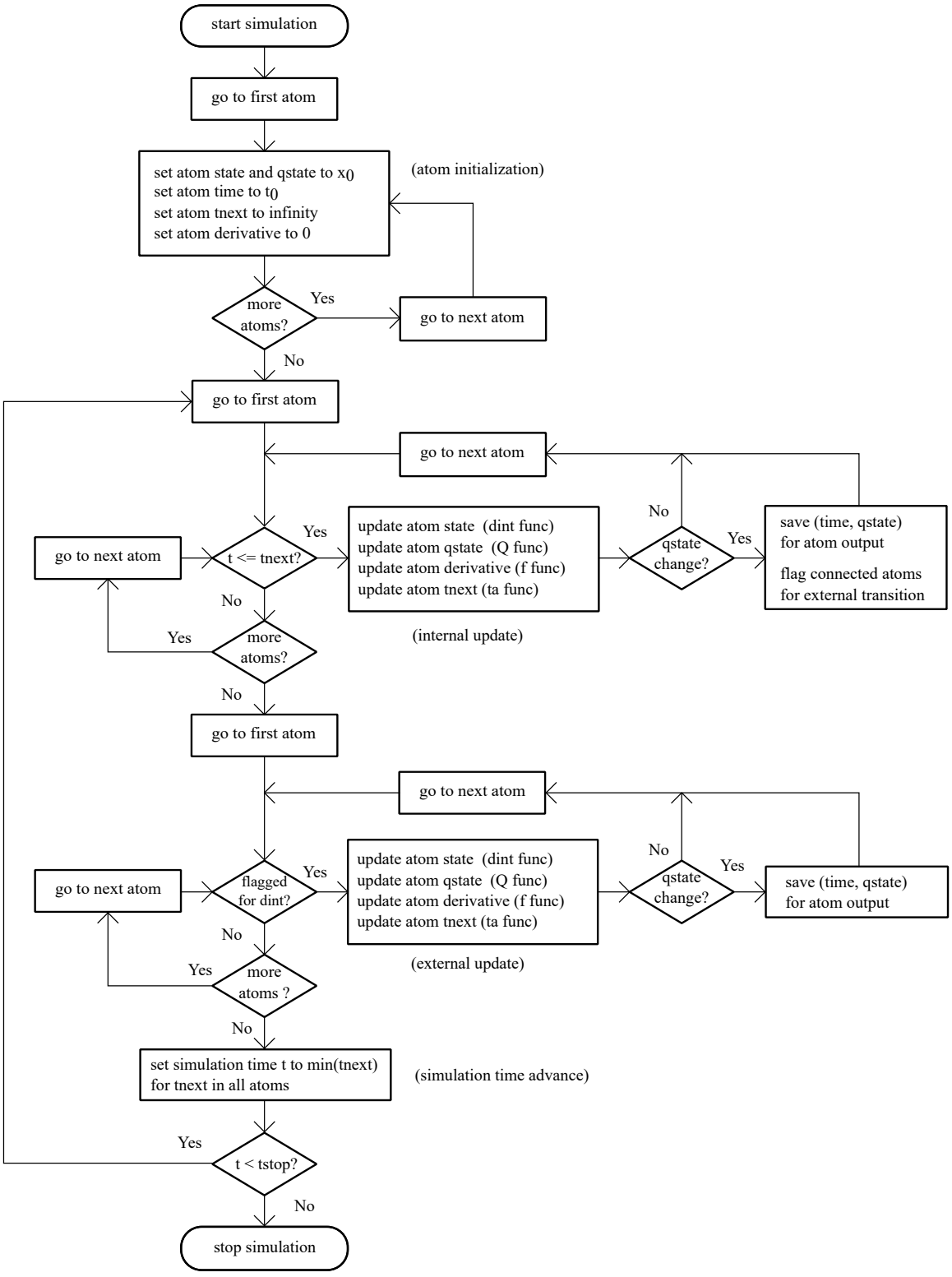



Figure 4.2 Simplified QDL simulation procedure flowchart.

# Chapter 5

# QDL Simulator Implementations

Two implementations have been developed for simulating systems with the QDL method; a MATLAB implementation for efficient simulation of linear systems, and a Python implementation for fast prototyping of a QDL simulator that can support non-linear models and advanced QSS integration methods.

## 5.1 MATLAB QDL Implementation

An object-oriented implementation of a QDL simulator was implemented in MATLAB script that is suitable for linear QDL system models. The full source code of this implementation is included in Appendix A.

The key objects of the MATLAB script QDL implementation are

- **QdlSystem**: Contains the collection of QDL atom objects, the system matrices, the main simulator algorithm, stimulus implementations and plotting utilities.
- **QdlNode**: The generic QDL node atom model
- **QdlBranch**: The generic QDL branch atom model
- **QdlSimulum**: The generic QDL stimulus interface.

Example source code for a QDL simulation system definition is listed below for a 30 second simulation of a dc motor. The motor is excited by a PWM voltage source and loaded with a torque load that steps from 0 to $0.5Nm$ at 10 seconds.

```
dQ = 0.1

Va = 10;
Ra = 0.1;
La = 0.01;
Ke = 0.1;
Kt = 0.1;
Jm = 0.1;
Bm = 0.001;
Tm = 0.0;

sys = QdlSystem(dQ);

va = QdlNode('va (V) (100Hz d=0.5)', 0, 0, 0);
va.stim_type = QdlSystem.StimSource;
va.source_type = QdlSystem.SourcePWM;
va.freq = 100;
va.duty = 0.5;
va.v1 = Va;
va.v2 = 0;

gnd = QdlNode('ground', 0, 0, 0);
gnd.stim_type = QdlSystem.StimSource;
gnd.source_type = QdlSystem.SourceDC;
gnd.vdc = 0.0;

nmech = QdlNode('shaft speed (rad/s)', Jm, Bm, Tm);

barm = QdlBranch('armature current (A)', La, Ra, 0);
barm.connect(va, gnd);

barm.add_tnode(nmech, -Ke);
nmech.add_sbranch(barm, Kt);

sys.add_node(va);
sys.add_node(gnd);
sys.add_node(nmech);
sys.add_branch(barm);

sys.init();
sys.runto(10);
sys.H(3) = -0.5;  % step load torque
sys.runto(20);
```

The node and branch device constructors have arguments for the primary component coefficients (namely $L$, $R$, $E$, $C$, $B$, and $H$), and the controlled source connections are realized via add_tnode(), add_zbranch(), add_sbranch() and add_bnode() functions for the branch and node atoms. This allows an arbitrary number of controlled sources to be added to the node and branch atoms. The simulator is run via the runto() function, that can be called multiple times sequentially, allowing discrete event changes to be introduced to any system variable during the simulation. Various quantized source types are available. Used here is a PWM source. These sources create discrete events that are pushed into the event queue along with the QSS integrator scheduled events.

## 5.2 Python QDL Implementation

For the fast prototyping of more advanced integrators, quantizers, sources, devices and system models, Python was chosen as the language of choice. The full source code of this implementation is included in Appendix B.

The key objects of the Python QDL framework are

- **qdl.System**: Contains the collection of QDL device objects, the main simulator algorithms, reference ODE solutions, stimulus implementations and plotting utilities.
- **qdl.Atom**: A generic QDL atom model
- **qdl.StateAtom**: A QDL state atom model where the QDEVS functions are implemented.
- **qdl.SourceAtom**: A source atom model for quantized input signal components, and
- **qdl.Device**: The generic QDL device interface, a collection of QDL atoms, connection ports, time derivative and Jacobian cell functions.

In addition to the framework objects, the Python QDL implementation also has an extensive hierarchical device library, including complex devices such as the three-phase electric machines used in chapters 7 and 8.

As a first example of QDL Python model definitions, the LIM Branch and Node components are listed below. The LIM atom coefficients and Jacobian matrix components are determined at run time using delegate functions (such as $a_{ii}$, and $b_{ij}$, etc.), allowing non-linear and time-varying behavior to be modeled.

```
1 class LimNode(Device):
2
3     def __init__(self, name, c, g=0.0, h0=0.0, v0=0.0,
4                  source_type=SourceType.CONSTANT, h1=0.0,
```

```
                 h2=0.0, ha=0.0, freq=0.0, phi=0.0, duty=0.0,
                 t1=0.0, t2=0.0, dq=None):

        Device.__init__(self, name)

        self.c = c
        self.g = g

        self.h = SourceAtom("h", source_type=source_type,
                            x0=h0, x1=h1, x2=h2, xa=ha,
                            freq=freq, phi=phi, duty=duty,
                            t1=t1, t2=t2, dq=dq, units="A")

        self.voltage = StateAtom("voltage", x0=0.0,
                                 coeffunc=self.aii,
                                 dq=dq, units="V")

        self.add_atoms(self.h, self.voltage)

        self.voltage.add_connection(self.h, coeffunc=self.bii)

        self.voltage.add_jacfunc(self.voltage, self.aii)

    def connect(self, branch, terminal="i"):

        if terminal == "i":
            self.voltage.add_connection(branch.current,
                                        coeffunc=self.aij)

            self.voltage.add_jacfunc(branch.current, self.aij)

        elif terminal == "j":
            self.voltage.add_connection(branch.current,
                                        coeffunc=self.aji)

            self.voltage.add_jacfunc(branch.current, self.aji)

    def aii(self, v):
        return -self.g / self.c

    def bii(self):
        return 1.0 / self.c

    def aij(self, v, i):
        return -1.0 / self.c

    def aji(self, i, v):
        return 1.0 / self.c


class LimBranch(Device):

    def __init__(self, name, l, r=0.0, e0=0.0, i0=0.0,
                 source_type=SourceType.CONSTANT, e1=0.0,
                 e2=0.0, ea=0.0, freq=0.0, phi=0.0,
                 duty=0.0, t1=0.0, t2=0.0, dq=None):

        Device.__init__(self, name)

        self.l = l
        self.r = r

        self.e = SourceAtom("e", source_type=source_type,
                            x0=e0, x1=e1, x2=e2, xa=ea,
                            freq=freq, phi=phi, duty=duty,
                            t1=t1, t2=t2, dq=dq, units="V")

        self.current = StateAtom("current", x0=0.0,
                                 coeffunc=self.aii,
                                 dq=dq, units="A")

```

```
76          self.add_atoms(self.e, self.current)
77
78          self.current.add_connection(self.e,
79                                      coeffunc=self.bii)
80
81          self.current.add_jacfunc(self.current, self.aii)
82
83      def connect(self, inode, jnode):
84
85          self.current.add_connection(inode.voltage,
86                                      coeffunc=self.aij)
87
88          self.current.add_connection(jnode.voltage,
89                                      coeffunc=self.aji)
90
91          self.current.add_jacfunc(inode.voltage, self.aij)
92          self.current.add_jacfunc(jnode.voltage, self.aji)
93
94      def aii(self, i):
95          return -self.r / self.l
96
97      def bii(self):
98          return 1.0 / self.l
99
100     def aij(self, i, v):
101         return 1.0 / self.l
102
103     def aji(self, v, i):
104         return -1.0 / self.l
```

As an example of a more complex device, the source for the Synchronous Machine in the DQ frame is listed below. This example contains many complex derivatives and Jacobian matrix components encoded in delegate functions in the Python class definition.

```
1  class SyncMachineDQ(Device):
2
3      def __init__(self, name, Psm=25.0e6, VLL=4160.0, ws=60.0*PI
       ,
4          P=4, pf=0.80, rs=3.00e-3, Lls=0.20e-3, Lmq=2.00e-3,
5          Lmd=2.00e-3, rkq=5.00e-3, Llkq=0.04e-3, rkd=5.00e-3,
6          Llkd=0.04e-3, rfd=20.0e-3, Llfd=0.15e-3, vfdb=90.1,
7          Kp=10.0e4, Ki=10.0e4, J=4221.7, fkq0=0.0, fkd0=0.0,
8          ffd0=0.0, wr0=60.0*PI, th0=0.0, iqs0=0.0, ids0=0.0,
9          dq_i=1e-2, dq_f=1e-2, dq_wr=1e-1, dq_th=1e-3, dq_v=1e0)
       :
10
11          self.name = name
12          Device.__init__(self, name)
13
14          self.Psm  = Psm
15          self.VLL  = VLL
16          self.ws   = ws
17          self.P    = P
18          self.pf   = pf
19          self.rs   = rs
20          self.Lls  = Lls
21          self.Lmq  = Lmq
22          self.Lmd  = Lmd
23          self.rkq  = rkq
24          self.Llkq = Llkq
25          self.rkd  = rkd
26          self.Llkd = Llkd
27          self.rfd  = rfd
28          self.Llfd = Llfd
```

```
        self.vfdb = vfdb
        self.Kp   = Kp
        self.Ki   = Ki
        self.J    = J

        # set intial conditions:

        self.iqs0 = iqs0
        self.ids0 = ids0
        self.fkq0 = fkq0
        self.fkd0 = fkd0
        self.ffd0 = ffd0
        self.wr0  = wr0
        self.th0  = th0

        # cached derived impedances:

        self.Lq = Lls + (Lmq * Llkq) / (Llkq + Lmq)
        self.Ld = (Lls + (Lmd * Llfd * Llkd)
                  / (Lmd * Llfd + Lmd * Llkd + Llfd * Llkd))

        # atoms:

        self.ids = StateAtom("ids", x0=ids0, derfunc=self.dids,
                             units="A",     dq=dq_i)

        self.iqs = StateAtom("iqs", x0=iqs0, derfunc=self.diqs,
                             units="A",     dq=dq_i)

        self.fkq = StateAtom("fkq", x0=fkq0, derfunc=self.dfkq,
                             units="Wb",    dq=dq_f)

        self.fkd = StateAtom("fkd", x0=fkd0, derfunc=self.dfkd,
                             units="Wb",    dq=dq_f)

        self.ffd = StateAtom("ffd", x0=ffd0, derfunc=self.dffd,
                             units="Wb",    dq=dq_f)

        self.wr  = StateAtom("wr",  x0=wr0,  derfunc=self.dwr,
                             units="rad/s", dq=dq_wr)

        self.th  = StateAtom("th",  x0=th0,  derfunc=self.dth,
                             units="rad",   dq=dq_th)

        self.add_atoms(self.ids, self.iqs, self.fkq, self.fkd,
                       self.ffd, self.wr,  self.th)

        # internal connections:

        self.ids.add_connection(self.iqs)
        self.ids.add_connection(self.wr )
        self.ids.add_connection(self.fkq)
        self.iqs.add_connection(self.ids)
        self.iqs.add_connection(self.wr )
        self.iqs.add_connection(self.fkd)
        self.iqs.add_connection(self.ffd)
        self.fkd.add_connection(self.ffd)
        self.fkd.add_connection(self.ids)
        self.fkq.add_connection(self.iqs)
        self.ffd.add_connection(self.ids)
        self.ffd.add_connection(self.fkd)
        self.wr.add_connection( self.ids)
        self.wr.add_connection( self.iqs)
        self.wr.add_connection( self.fkq)
        self.wr.add_connection( self.fkd)
        self.wr.add_connection( self.ffd)
        self.wr.add_connection( self.th )
        self.th.add_connection( self.wr )

```

```
98          # jacobian:
99
100         self.ids.add_jacfunc(self.ids, self.jids_ids)
101         self.ids.add_jacfunc(self.iqs, self.jids_iqs)
102         self.ids.add_jacfunc(self.fkq, self.jids_fkq)
103         self.ids.add_jacfunc(self.wr , self.jids_wr )
104         self.iqs.add_jacfunc(self.ids, self.jiqs_ids)
105         self.iqs.add_jacfunc(self.iqs, self.jiqs_iqs)
106         self.iqs.add_jacfunc(self.fkd, self.jiqs_fkd)
107         self.iqs.add_jacfunc(self.ffd, self.jiqs_ffd)
108         self.iqs.add_jacfunc(self.wr , self.jiqs_wr )
109         self.fkq.add_jacfunc(self.iqs, self.jfkq_iqs)
110         self.fkq.add_jacfunc(self.fkq, self.jfkq_fkq)
111         self.fkd.add_jacfunc(self.ids, self.jfkd_ids)
112         self.fkd.add_jacfunc(self.fkd, self.jfkd_fkd)
113         self.fkd.add_jacfunc(self.ffd, self.jfkd_ffd)
114         self.ffd.add_jacfunc(self.ids, self.jffd_ids)
115         self.ffd.add_jacfunc(self.fkd, self.jffd_fkd)
116         self.ffd.add_jacfunc(self.ffd, self.jffd_ffd)
117         self.wr.add_jacfunc( self.wr , self.jwr_wr  )
118         self.wr.add_jacfunc( self.ids, self.jwr_ids )
119         self.wr.add_jacfunc( self.iqs, self.jwr_iqs )
120         self.wr.add_jacfunc( self.fkq, self.jwr_fkq )
121         self.wr.add_jacfunc( self.fkd, self.jwr_fkd )
122         self.wr.add_jacfunc( self.ffd, self.jwr_ffd )
123         self.wr.add_jacfunc( self.th , self.jwr_th  )
124         self.th.add_jacfunc( self.wr , self.jth_wr  )
125
126         # DQ ports:
127
128         self.id = self.iqs
129         self.iq = self.ids
130         self.vd = None
131         self.vq = None
132
133         # input port:
134
135         self.input = None  # avr output vref connection
136
137     def connect(self, bus, avr):
138
139         self.vd = self.iqs.add_connection(bus.vq)
140         self.vq = self.ids.add_connection(bus.vd)
141         self.input = self.ffd.add_connection(avr.vfd, self.vfdb
    )
142
143         self.iqs.add_jacfunc(bus.vq, self.jiqs_vq)
144         self.ids.add_jacfunc(bus.vd, self.jids_vd)
145         self.ffd.add_jacfunc(avr.vfd, self.jffd_vfd)
146
147     def vtermd(self):
148         return self.vd.value()
149
150     def vtermq(self):
151         return self.vq.value()
152
153     def vfd(self):
154         return self.input.value()
155
156     # Jacobian functions:
157
158     def jffd_vfd(self, ffd, vfd):
159         return self.vfdb
160
161     def jids_vd(self, ids, vd):
162         return -1 / self.Lls
163
164     def jiqs_vq(self, iqs, vq):
```

```
        return -1 / self.Lls

    def jids_ids(self, ids):
        return -self.rs/self.Lls

    def jids_iqs(self, ids, iqs):
        return -(self.Lq * self.wr.q) / self.Lls

    def jids_fkq(self, ids, fkq):
        return (self.Lmq * self.wr.q) / (self.Lls
        * (self.Lmq + self.Llkq))

    def jids_wr(self, ids, wr):
        return (((self.Lmq * self.fkq.q)
        / (self.Lmq + self.Llkq)
        - self.Lq * self.iqs.q) / self.Lls)

    def jiqs_ids(self, iqs, ids):
        return (self.Ld * self.wr.q) / self.Lls

    def jiqs_iqs(self, iqs):
        return -self.rs / self.Lls

    def jiqs_fkd(self, iqs, fkd):
        return ((self.Lmd * self.wr.q) / (self.Llkd
        * self.Lls * (self.Lmd / self.Llkd + self.Lmd
        / self.Llfd + 1)))

    def jiqs_ffd(self, iqs, ffd):
        return ((self.Lmd * self.wr.q) / (self.Llfd
        * self.Lls * (self.Lmd / self.Llkd + self.Lmd
        / self.Llfd + 1)))

    def jiqs_wr(self, iqs, wr):
        return ((self.Ld * self.ids.q + (self.Lmd
        * (self.fkd.q / self.Llkd + self.ffd.q
        / self.Llfd)) / (self.Lmd / self.Llkd
        + self.Lmd / self.Llfd + 1)) / self.Lls)

    def jfkq_iqs(self, fkq, iqs):
        return -((self.Lls - self.Lq) * self.rkq)
        / self.Llkq

    def jfkq_fkq(self, fkq):
        return -((1 - self.Lmq / (self.Lmq
        + self.Llkq)) * self.rkq) / self.Llkq

    def jfkd_ids(self, fkd, ids):
        return -((-self.Lls - self.Ld)
        * self.rkd) / self.Llkd

    def jfkd_fkd(self, fkd):
        return (-((self.Lmd / (self.Llkd * (self.Lmd
        / self.Llkd + self.Lmd / self.Llfd + 1)) + 1)
        * self.rkd) / self.Llkd)

    def jfkd_ffd(self, fkd, ffd):
        return -(self.Lmd * self.rkd) / (self.Llfd
        * self.Llkd * (self.Lmd / self.Llkd
        + self.Lmd / self.Llfd + 1))

    def jffd_ids(self, ffd, ids):
        return -((-self.Lls - self.Ld) * self.rfd)
        / self.Llfd

    def jffd_fkd(self, ffd, fkd):
        return (-(self.Lmd * self.rfd) / (self.Llfd
        * self.Llkd * (self.Lmd / self.Llkd + self.Lmd
```

```
        / self.Llfd + 1)))

    def jffd_ffd(self, ffd):
        return (-((self.Lmd / (self.Llfd * (self.Lmd
        / self.Llkd + self.Lmd / self.Llfd + 1)) + 1)
        * self.rfd) / self.Llfd)

    def jwr_ids(self, wr, ids):
        return ((0.75*self.P*(-self.Lq*self.iqs.q+self.Ld
        * self.iqs.q-(self.Lmq*self.fkq.q)/(self.Lmq
        + self.Llkq)))/self.J)

    def jwr_iqs(self, wr, iqs):
        return ((0.75*self.P*(-self.Lq*self.ids.q+self.Ld
        * self.ids.q+(self.Lmd*(self.fkd.q/self.Llkd
        + self.ffd.q/self.Llfd))/(self.Lmd/self.Llkd
        + self.Lmd/self.Llfd+1.0)))/self.J)

    def jwr_fkq(self, wr, fkq):
        return (-(0.75 * self.Lmq * self.P * self.ids.q)
        / (self.J * (self.Lmq + self.Llkq)))

    def jwr_fkd(self, wr, fkd):
        return ((0.75 * self.Lmd * self.P * self.iqs.q)
        / (self.J * self.Llkd * (self.Lmd / self.Llkd
        + self.Lmd / self.Llfd + 1)))

    def jwr_ffd(self, wr, ffd):
        return ((0.75 * self.Lmd * self.P * self.iqs.q)
        / (self.J * self.Llfd * (self.Lmd / self.Llkd
        + self.Lmd / self.Llfd + 1)))

    def jwr_wr(self, wr):
        return -self.Kp / self.J

    def jwr_th(self, wr, fthkq):
        return self.Ki / self.J

    def jth_wr(self, th, wr):
        return -1.0

    # derivative functions:

    def dids(self, ids):
        return ((-self.wr.q * self.Lq * self.iqs.q + self.wr.q
        * (self.Lmq / (self.Lmq + self.Llkq) * self.fkq.q)
        - self.vtermd() - self.rs * ids) / self.Lls)

    def diqs(self, iqs):
        return ((self.wr.q * self.Ld * self.ids.q + self.wr.q
        * (self.Lmd * (self.fkd.q / self.Llkd + self.ffd.q
        / self.Llfd) / (1.0 + self.Lmd / self.Llfd + self.Lmd
        / self.Llkd))
        - self.vtermq() - self.rs * iqs) / self.Lls)

    def dfkq(self, fkq):
        return (-self.rkq / self.Llkq * (fkq - self.Lq
        * self.iqs.q - (self.Lmq / (self.Lmq + self.Llkq)
        * fkq) + self.Lls * self.iqs.q))

    def dfkd(self, fkd):
        return (-self.rkd / self.Llkd * (fkd - self.Ld
        * self.ids.q + (self.Lmd * (fkd / self.Llkd
        + self.ffd.q / self.Llfd) / (1.0 + self.Lmd
        / self.Llfd + self.Lmd / self.Llkd)) - self.Lls
        * self.ids.q))

```

```
300     def dffd(self, ffd):
301         return (self.vfd() - self.rfd / self.Llfd
302         * (ffd - self.Ld * self.ids.q + (self.Lmd
303         * (self.fkd.q / self.Llkd + ffd / self.Llfd)
304         / (1.0 + self.Lmd / self.Llfd + self.Lmd
305         / self.Llkd)) - self.Lls * self.ids.q))
306
307     def dwr(self, wr):
308         return ((3.0 * self.P / 4.0 * ((self.Ld
309         * self.ids.q + ((self.Lmd * (self.fkd.q
310         / self.Llkd + self.ffd.q / self.Llfd)
311         / (1.0 + self.Lmd / self.Llfd + self.Lmd
312         / self.Llkd)))) * self.iqs.q - (self.Lq
313         * self.iqs.q + (self.Lmq / (self.Lmq
314         + self.Llkq) * self.fkq.q)) * self.ids.q))
315         + (self.Kp * (self.ws - wr) + self.Ki
316         * self.th.q)) / self.J
317
318     def dth(self, th):
319         return (self.ws - self.wr.q)
```

A system simulation is defined in a Python script. An example simulation script is listed below for a similar dc motor system as the previous MATLAB example.

```
1  dq = 1e-2
2  f = 100.0
3  d = 0.5
4
5  sys = System()
6
7  ground = GroundNode("ground")
8  source = PwmSourceNode("source", vlo=0.0, vhi=1.0, freq=f, duty
       =d)
9
10 motor = DCMotor("motor", ra=0.1, La=0.01, Jm=0.1, Bm=0.001, Kt
       =0.1,
11                     Ke=0.1, ia0=0.0, wr0=0.0, dq_ia=dq, dq_wr=dq)
12
13 motor.connect(source, ground)
14
15 sys.add_devices(ground, source, motor)
16
17 tstop = 10.0
18 dt = 1e-4
19 dc = True
20 optimize_dq = 0
21 ode = True
22 qss = True
23
24 chk_ss_delay = 2.0
25
26 def event(sys):
27     source.voltage.duty = 0.8
28
29 sys.schedule(event, tstop*0.5)
30
31 sys.initialize(dt=dt, dc=dc)
32
33 sys.run(tstop, ode=ode, qss=qss, verbose=True, ode_method="
       LSODA",
34         optimize_dq=optimize_dq, chk_ss_delay=chk_ss_delay)
```

To define a system in the Python implementation, the device models are instantiated, parameterized, and then connected via the appropriate connect functions that

map LIM node and branch ports. There are also input/output signal port connections for signal devices such as the AVR. Instead of using multiple sequential runto() function calls, discrete events like parameter changes are implemented as local functions, and scheduled with the simulator to be triggered at the specified time. Advanced features, such as higher-order ODE reference solution options, steady-state detection, automatic $\Delta Q$ optimization, and dc operating point solutions are also available in the Python implementation.

# Chapter 6

# QDL Simulation Examples

## 6.1 Second Order Linear Circuit

We first illustrate the QDL method with a very simple example. A linear, $2^{nd}$ order system consisting of one node atom and one branch atom is presented in figure 6.1. For this case with two atoms, the simulation process can be visualized as in figure 4.1 The simulation results are shown in figure 6.2 compared against the implicit state space solution (dashed lines). The QDL results are piece-wise constant values. Note that the quantum size $\Delta Q$ is different for the node voltage and the branch current. In general, $\Delta Q$ can be specific to each atom.

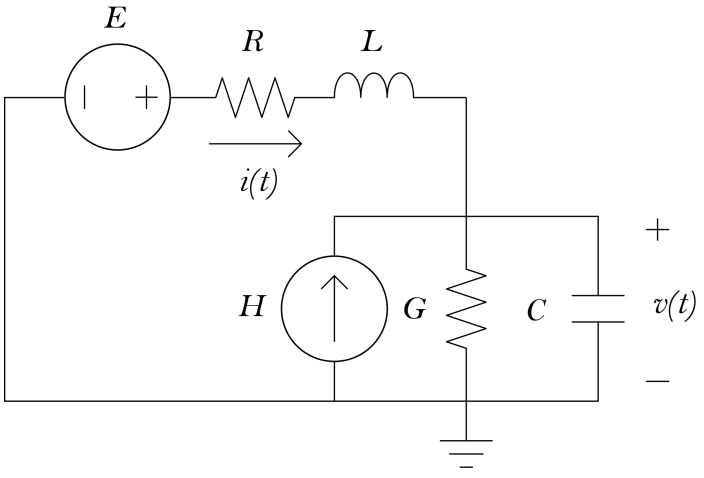



Figure 6.1 $2^{nd}$ order QDL system.

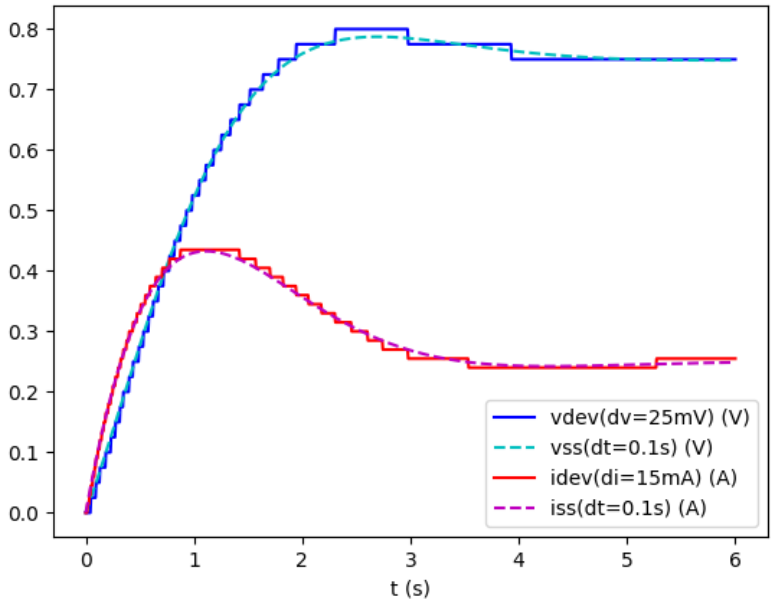



Figure 6.2 $2^{nd}$ order QDL system simulation results (where $v_{dev}$ and $i_{dev}$ are the QDL results, and $v_{ss}$ and $i_{ss}$ are the state-space reference solution results).

## 6.2 Distributed Transmission Line

The next test is a RLCG distributed transmission line model with 5 nodes and 4 branches. The first node includes the external source injection. The results for all of the node voltages ($n1$ through $n5$) and branch currents ($b1$ through $b4$) are shown in figures 6.3 and 6.4. A step change in the source occurs midway through the simulation to provide an additional transient disturbance.

Because the benchmark state space solution and the frequency of updates is an important metric when analyzing the results, the plots contain several types of overlaid information to help visualize these data. First, the reference implicit state space solution results are shown (dashed cyan line). The QDL results are shown as black markers, where each marker is plotted at the time of the atom's external quantized state update. Lastly, the frequency of QDL updates (how often the state is recom-

puted and an update event is broadcast from the atom) is shown as a histogram of updates per the span of time shown in the plot legend. Note the change in update frequency between the transient and steady-state portions of the simulation.

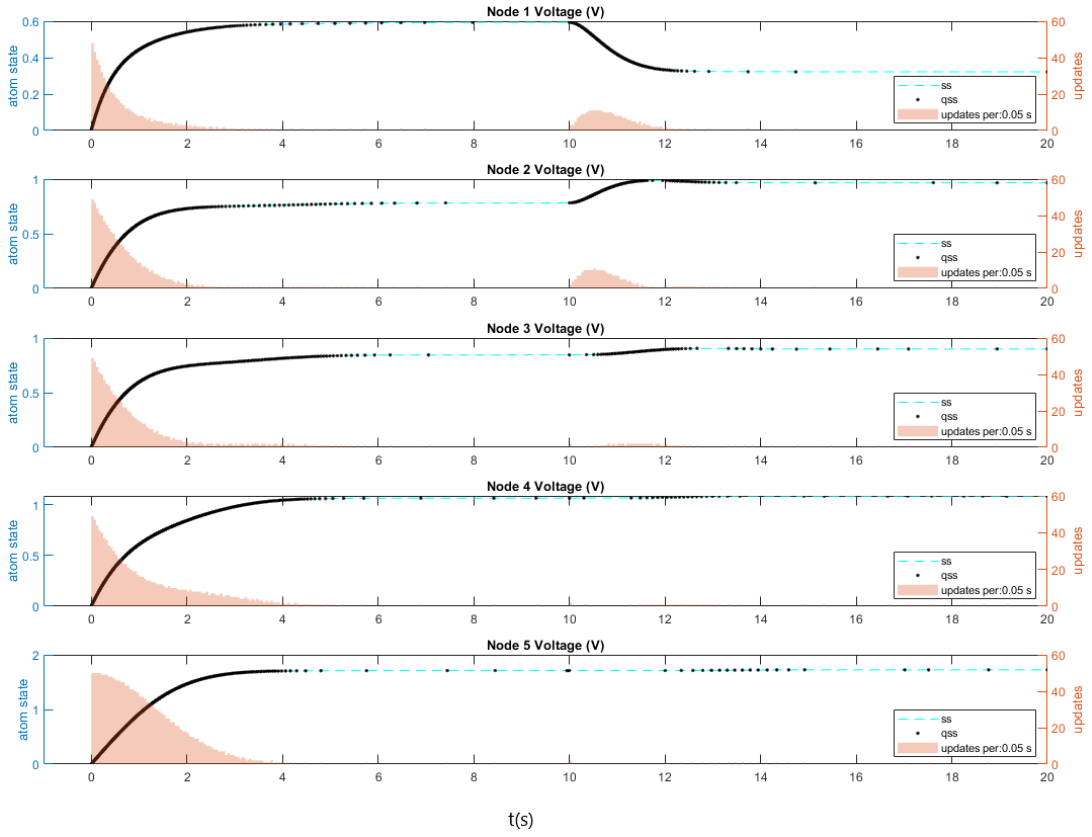



Figure 6.3 $9^{th}$ Order Distributed Transmission Line Voltages

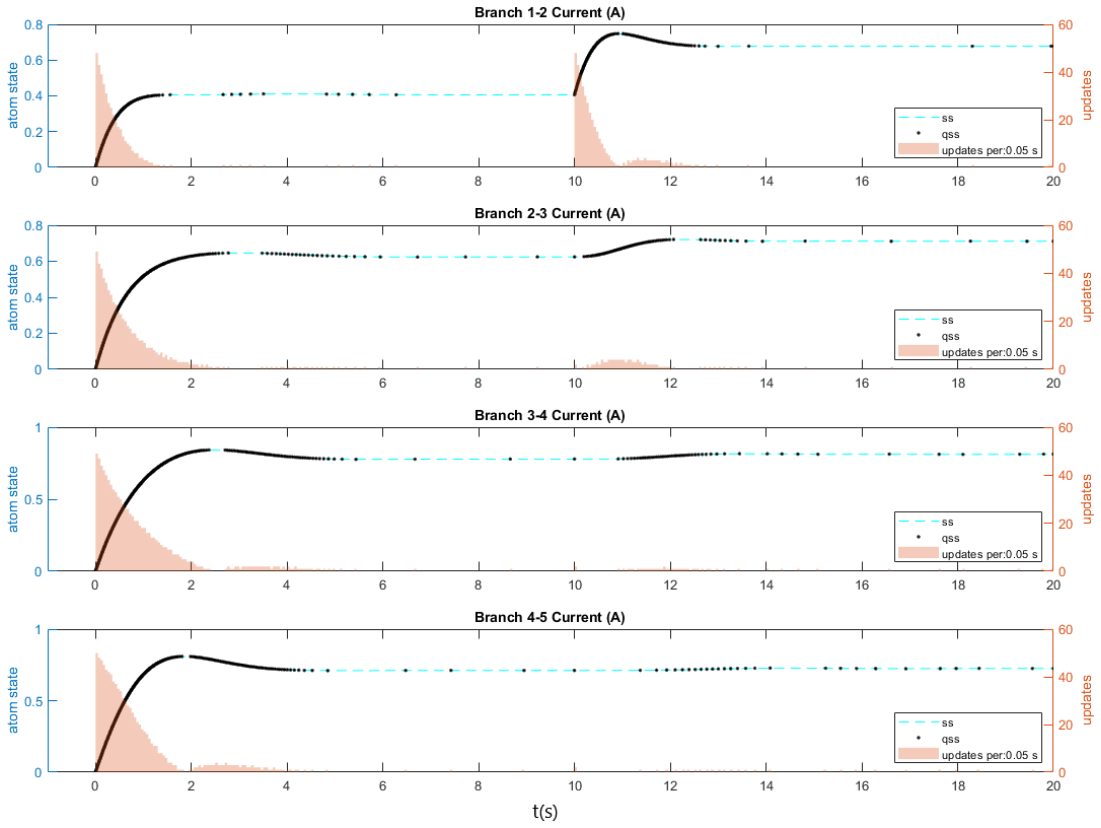



Figure 6.4 $9^{th}$ Order Distributed Transmission Line Currents

## 6.3 Very Stiff Grid Simulation

In order to test the capabilities of the QDL method with stiff systems, a dense grid was created with a very wide range of time constants. This grid has 16 LIM nodes and 24 LIM branches (see figure 6.5). Each quadrant of the grid contains different levels of latency, controlled by setting the capacitance values for the nodes within in the quadrants. The eigenvalues of the system range approximately from $10^{-3}$ seconds to $10^{6}$ seconds, creating a stiffness ratio of approximately $10^{9}$. The voltages at the corner nodes are used as representative states for analysis of the results of the simulation, and are conveniently named Node 1, Node 2, Node 3, and Node 4, corresponding to their latency quadrant.

The simulation was run for $10^{4}$ simulation seconds. The current injection at Node

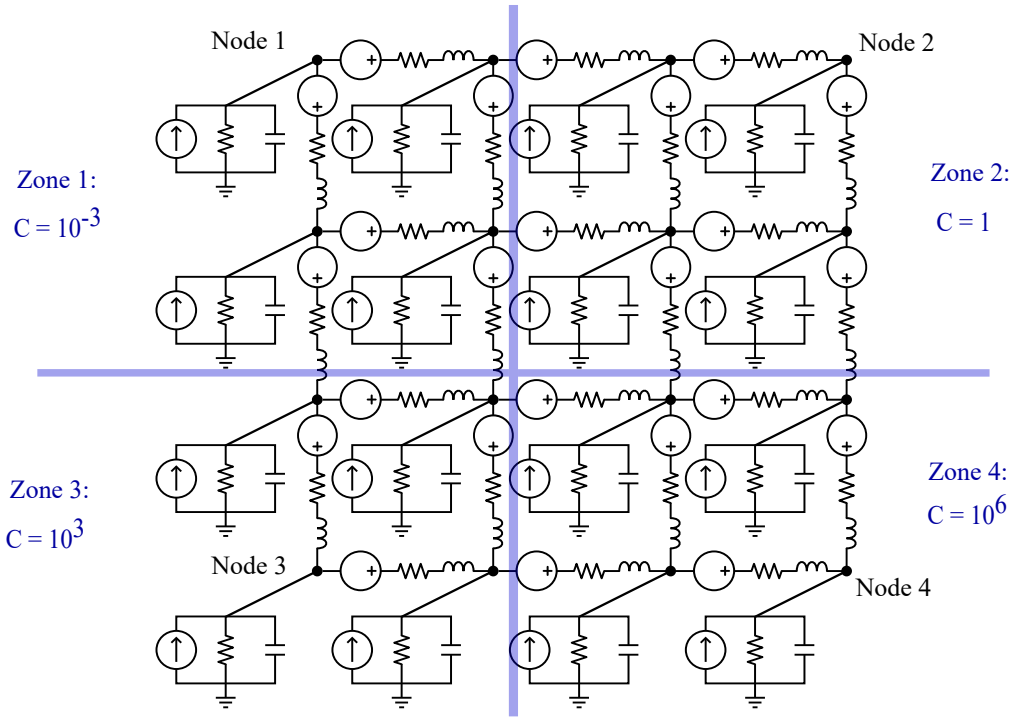



Figure 6.5 Stiff LIM grid with four latency zones.

1 is stepped from 0 to $1A$ at $t = 0$, and from $1A$ to $10A$ at $t = tsim/2$ to provide a perturbation to create a dynamic response. The results are shown in figure 6.6. Because the results from a simulation with such a large stiffness ratio are difficult to visualize on one time scale, zoomed plots of the faster transients are included for corner Node 1, Node 2 and Node 3 (see figure 6.7). Note that the dynamic response of Node 4 is too slow to warrant a zoomed plot.

Included in each plot are two axes, a left axis for the voltage quantity, and a right axis for the update frequency. The update frequency is the rate at which the asynchronous state updates occur in the simulation for each atom. As expected and desired, the update frequencies are relatively high during the transients, and very low or non-existent during the steady-state portions. Each plot's legend shows the update frequency histogram bin size in seconds. Note that these bin sizes vary with each plot as they were chosen for readability.

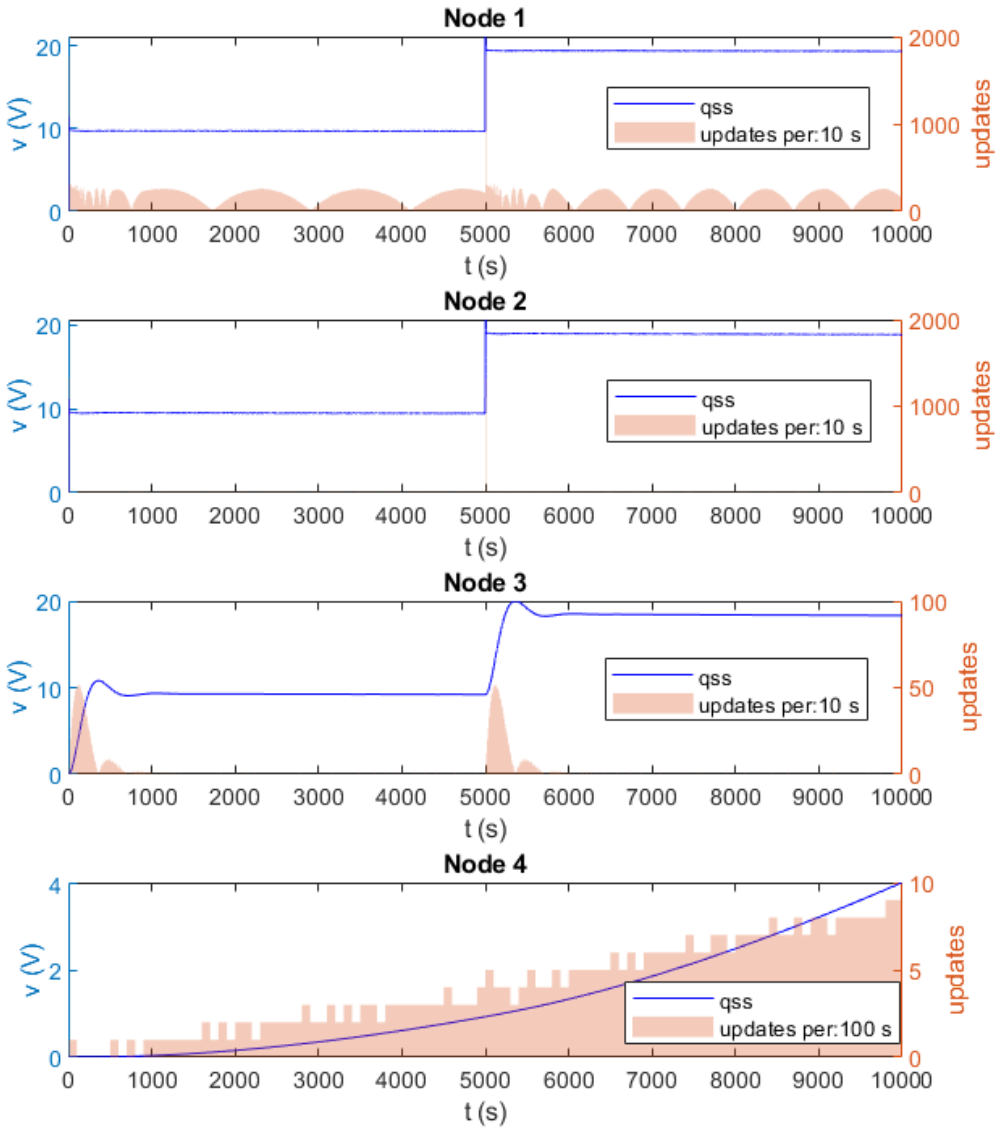



Figure 6.6 Stiff grid corner node voltage dynamic response.

An implicit state space solution as a benchmark was impractical to create for this example because of the extreme stiffness and long simulation time of this simulation. The run-time of such a simulation would take many hours or days to simulate on typical desktop hardware using an implicit state space solution with a time step small enough to capture the fast dynamics. For example, a time step of $\lambda_{min}/10$ would require the solution of a 40-state system for $10^8$ time steps. The QDL system,

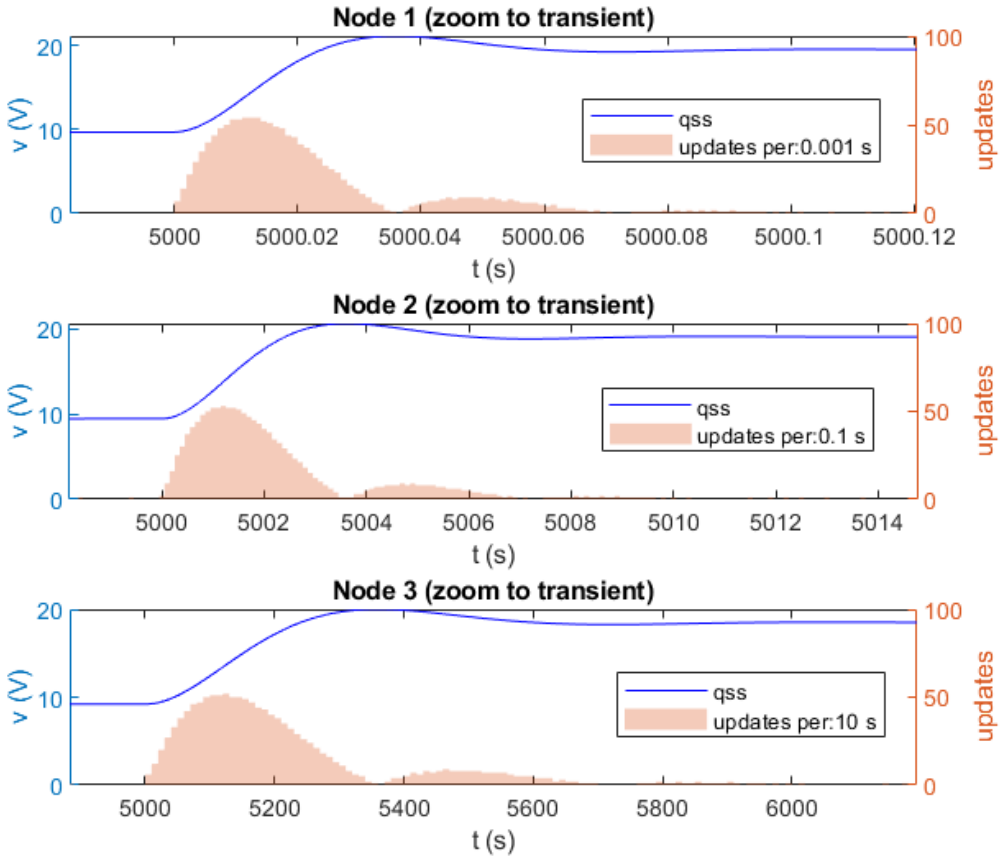



Figure 6.7 Stiff grid corner node voltage dynamic response (zoom to transient).

however, requires total updates on the order of $10^5$ for all states combined, and runs in less than 5 mins on a laptop computer as a single-threaded application. Note that this is not an exhaustive performance or accuracy analysis. The quantification of computational efficiency and error will need to be part of future work. Also, because an implicit state space solution is impractical to perform on these types of the extremely stiff systems, a reasonable method of bench-marking the performance and results will have to be determined that does not require running the full simulation.

## 6.4 DC Motor with a PWM Source

A simulation was performed using QDL for a dc motor excited with a pulse-width modulated voltage source. This system highlights additional features of the QDL method that are not apparent in the previous transmission line or grid models. The DC motor model includes an energy-conversion coupling between the electrical and mechanical sub-networks of the system. This coupling method is very important for the simulation of power systems for the energy transfer in machines, transformers, converters, and loads. The first motor simulation, with results in figure 6.4 involved a simple dc voltage source armature excitation at startup and shows the resulting armature current and rotor speed. There is an increase in mechanical torque load at the halfway mark. The system is somewhat stiff, with stiffness ratio around 10 between the electric branch and mechanical node (shaft). As expected, very few updates occur in the slower rotor speed state versus the electrical current state, and both atoms update very seldom during the steady-state portions of the simulation.

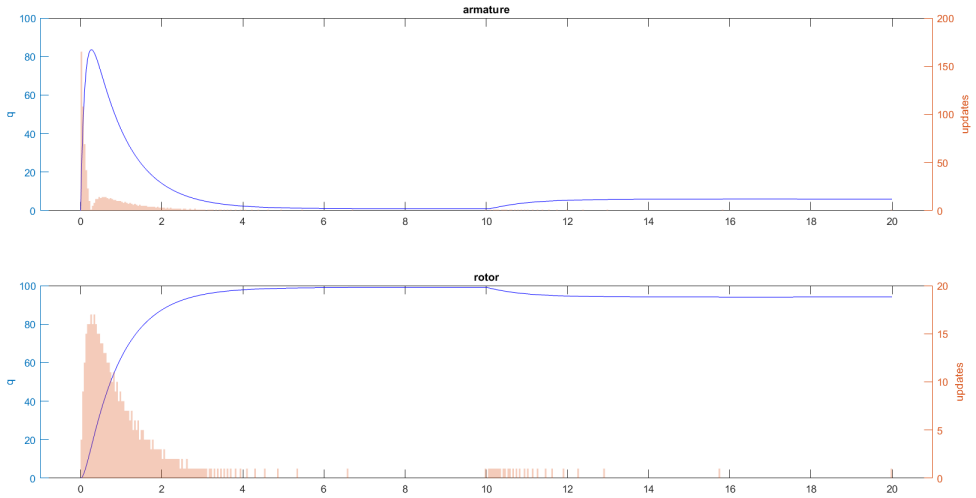



Figure 6.8 DC Motor Simulation

To further test the QDL implementation, the DC motor model was simulated

with an ideal PWM source at the armature terminals. Results for the PWM source simulation are shown for small $\Delta Q$ (figure 6.4) as well as large $\Delta Q$ (figure 6.4). In the first case with small $\Delta Q$, there are many updates of the external PWM state (as expected), and also many updates of the external branch current state. The current ripple is recorded in full detail, with updates on the order of 2000 per simulation second. The updates on the rotor speed node are very few and practically non-existent during steady-state. The large $\Delta Q$ simulation demonstrates the ability of QDL to ignore details like PWM-induced ripple while still effectively simulating the system. The $\Delta Q$ is chosen to be larger than the ripple magnitude, which results in the lack of ripple in the output, but more importantly very few updates in the external atom states (total updates are on the order of 10 for each atom).

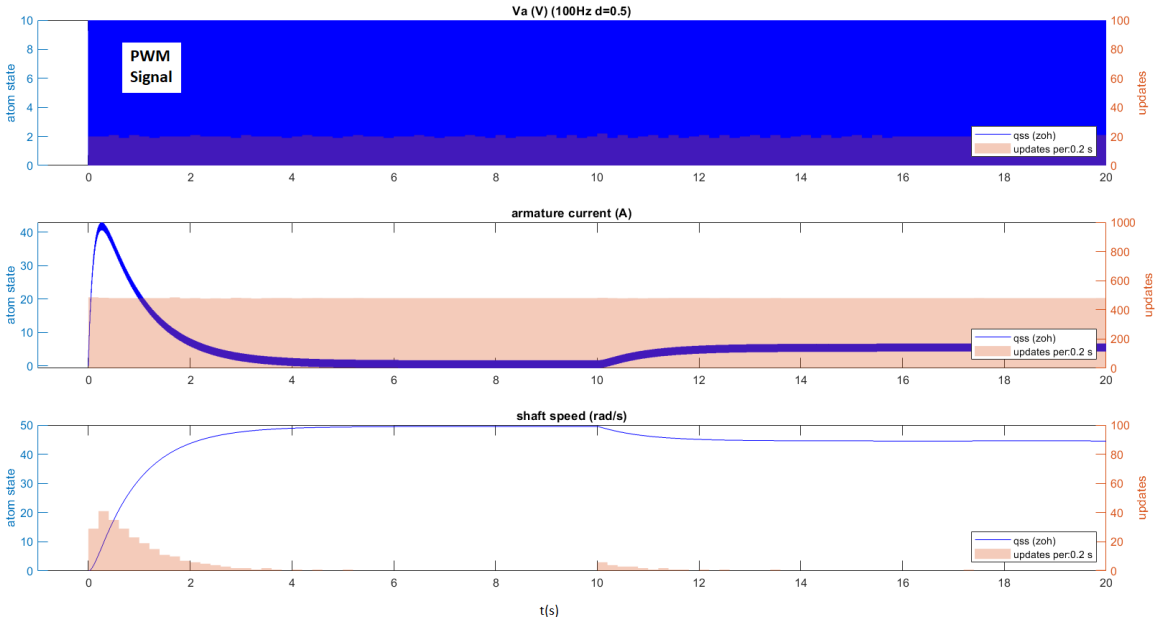



Figure 6.9 DC Motor with PWM Source (Small quantum, with ripple)

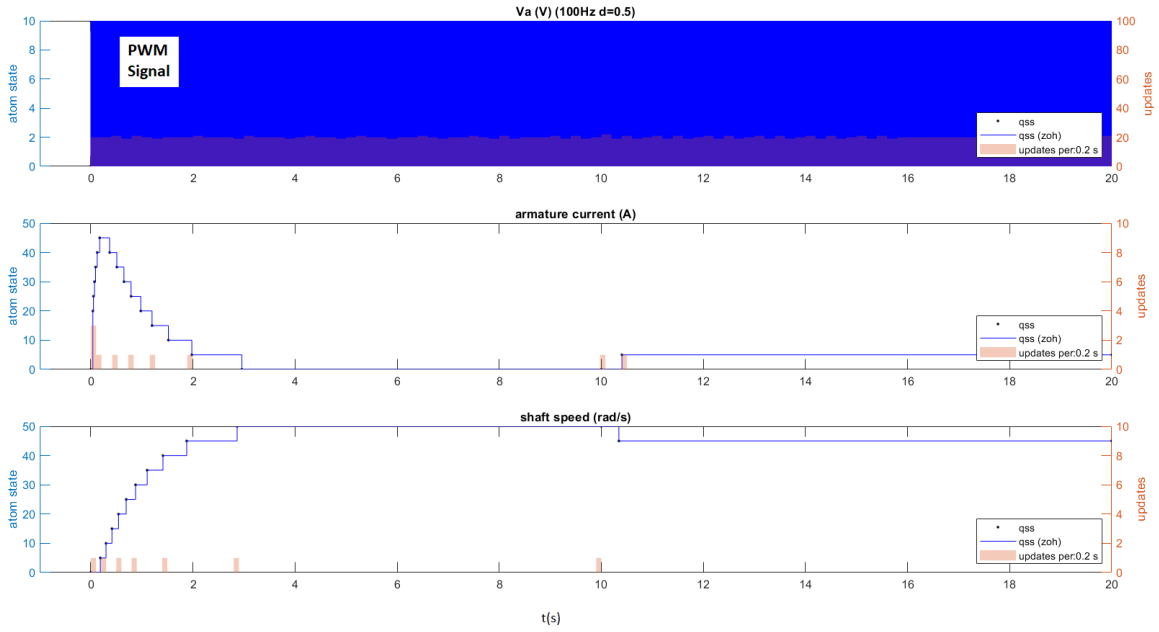



Figure 6.10 DC Motor with PWM Source (Large quantum, no ripple)

# Chapter 7

# Synchronous Machine Simulation

The purpose of this chapter is to assess the performance of the QDL method to the time-domain simulation of a highly non-linear, complex component of power systems: a three-phase synchronous generator.

In chapter 6, a dense, linear network having a very high stiffness ratio ($10^9$) was simulated using a combination of LIQSS1 and latency methods [1]. The QDL method successfully solved the system response when conventional methods could not (or would have required impractically long simulation times). That work did not address the implications of non-linear system elements. In the previous works surveyed that use QSS integration, the feasibility and performance of QSS-based methods were investigated only for linear systems or for very small non-linear systems (In Di Pietro et al., [16] for example, a four-stage interleaved converter was simulated using QSS). This chapter attempts to take this a step further by simulating a higher order non-linear power system, specifically with detailed synchronous machine model.

Systems that include synchronous machines often operate for a long duration in mostly steady-state conditions. These steady conditions are occasionally punctuated by abrupt changes of conditions that must be accurately simulated. The QDL method has the potential for high computational efficiency for simulation of such systems. Non-linearity, high stiffness, and a necessity for algebraic constraints to enforce circuit conservation laws are the important aspects of these systems that we feel QDL is well-suited for.

## 7.1 Study System

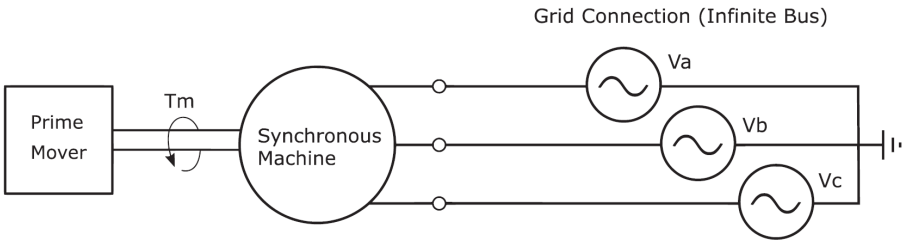



Figure 7.1 Synchronous generator connected to an infinite bus.

The study system, shown in figure 7.1, has three major components: a prime mover, a synchronous machine (which can act as either a generator or a motor depending on the direction of power flow), and an simple ac power grid model. Models of the prime mover and the grid are simplified. The torque source is an ideal time-dependent source with zero inertia, and the power system is represented as an ideal three-phase sinusoidal ac voltage source with zero impedance and constant frequency (i.e. an infinite bus). This study system was chosen because it is of widespread interest in power systems analysis (the so-called single machine infinite bus, or SMIB system), and because it demonstrates suitability of the QDL method to efficiently solve non-linear systems that are characterized by coupled fast and slow dynamics.

## 7.2 Synchronous Machine Model

The model of the electric machine is formulated in the synchronous reference frame to eliminate the periodic sinusoidal variations of all voltages and currents via the standard Park transformation [17]. This maximizes the benefit of using the QDL method for the analysis of ac systems, as the potential for slower updates during steady-state conditions relies on flat (zero-derivative) states in steady-state. Although QDL can be used to simulate systems having arbitrary wave forms, the sinusoidal

voltage and current oscillations inherent in an ac power system would require rapid state updates that would negate any benefits offered by the QDL method.

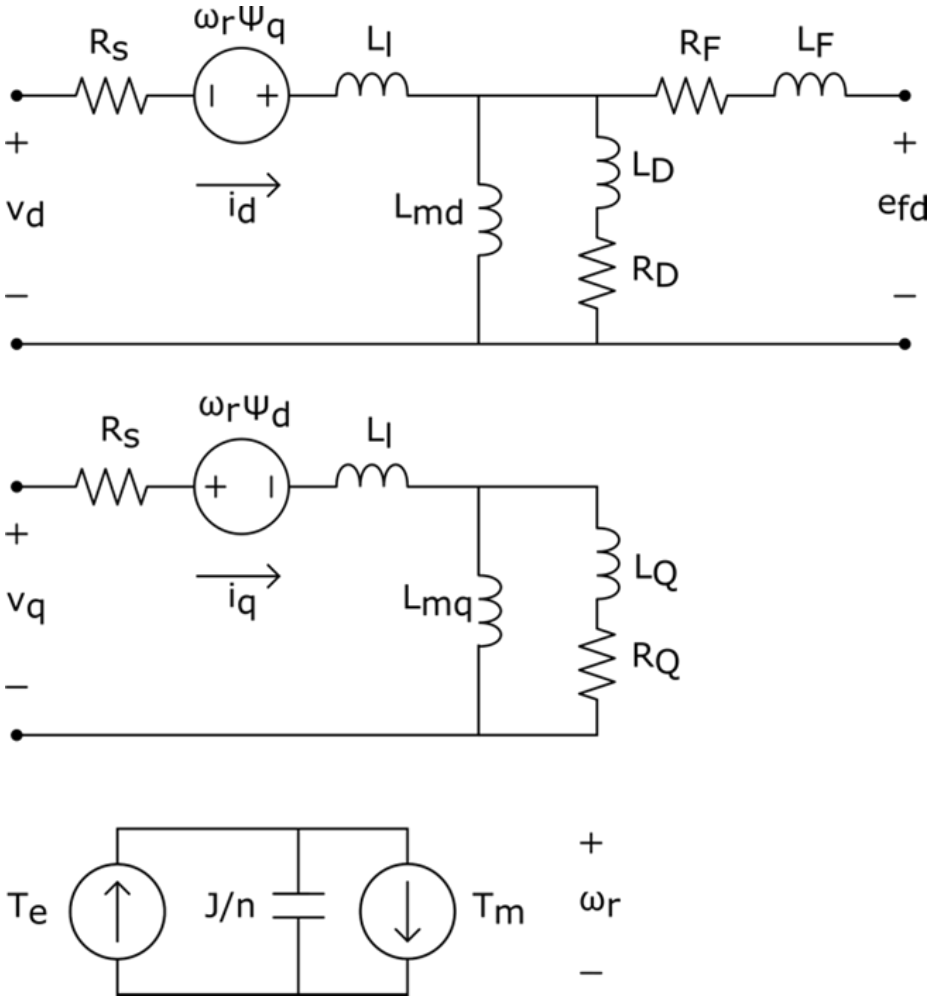



Figure 7.2 Direct, quadrature, and mechanical equivalent circuits of the synchronous machine.

A common model of the synchronous generator formulated in the synchronous reference frame is described by the set of equivalent circuits shown in figure 7.2. The fundamental frequency of the study system is 50 Hz. The seventh-order set of nonlinear equations that describe the dynamics of the transformed circuit is described in Equations 7.1 through 7.7, and the algebraic constraints that apply to the network solution are defined by Equations 7.8 and 7.9. Figure 7.2 and the equations follow

the model described in [18].

$$\frac{d}{dt}\psi_d = v_d - R_s i_d + \omega_r \psi_q \tag{7.1}$$

$$\frac{d}{dt}\psi_q = v_q - R_s i_q - \omega_r \psi_d \tag{7.2}$$

$$\frac{d}{dt}\psi_F = e_{fd} - i_F R_F \tag{7.3}$$

$$\frac{d}{dt}\psi_D = -i_D R_D \tag{7.4}$$

$$\frac{d}{dt}\psi_Q = -i_Q R_Q \tag{7.5}$$

$$\frac{d}{dt}\omega_r = \frac{n}{J}(i_q \psi_d - i_d \psi_q - T_m) \tag{7.6}$$

$$\frac{d}{dt}\theta = \omega_r - \omega_b \tag{7.7}$$

$$\begin{bmatrix} i_{dr} \\ i_F \\ i_D \end{bmatrix} = \begin{bmatrix} L_{md} + L_L & L_{md} & L_{md} \\ L_{md} & L_F + L_{md} & L_{md} \\ L_{md} & L_F + L_{md} & L_F + L_{md} \end{bmatrix}^{-1} \cdot \begin{bmatrix} \psi_{dr} \\ \psi_F \\ \psi_D \end{bmatrix} \tag{7.8}$$

$$\begin{bmatrix} i_{qr} \\ i_Q \end{bmatrix} = \begin{bmatrix} L_{mq} + L_L & L_{mq} \\ L_{mq} & L_{mq} \end{bmatrix}^{-1} \cdot \begin{bmatrix} \psi_q \\ \psi_Q \end{bmatrix} \tag{7.9}$$

Where the terms in these equations are defined as:

$\psi_d$ : direct-axis and stator flux (Wb),

$\psi_q$ : quadrature-axis stator flux (Wb),

$\psi_D$ : direct-axis damper flux (Wb),

$\psi_Q$ : quadrature-axis damper flux (Wb),

$v_d$ : direct-axis terminal voltage (V),

$v_q$ : quadrature-axis terminal voltage (V),

$R_s$ : stator series resistance ($\Omega$),

$\omega_r$ : rotor speed (rad/s),

$\omega_b$ : base frequency ($2\pi 50$ rad/s),

$i_d$ : direct-axis stator current (A),

$i_q$ : quadrature-axis stator current (A),

$\psi_F$ : stator field flux (Wb),

$i_F$ : stator field current (A),

$e_{fd}$ : field voltage (V),

$R_F$ : field resistance ($\Omega$),

$i_D$ : direct-axis internal equivalent current (A),

$i_Q$ : quadrature-axis internal equivalent current (A),

$R_D$ : direct-axis equivalent circuit resistance ($\Omega$),

$R_Q$ : quadrature-axis equivalent circuit resistance ($\Omega$),

$i_{dr}$ : direct-axis rotor current (A),

$i_{qr}$ : quadrature-axis rotor current (A), and

$\theta$ : rotor angle relative to synchronous reference frame (rad).

## 7.3 Simulation Scenario and Reference Solution

The scenario is a typical simulation that initializes the system to steady-state at a specific operating point, and then adds a perturbation to produce a transient response. The simulation starts with the synchronous generator rotating at steady-state in synchronism with the connected grid. This steady-state condition continues for the first 15 seconds. Beginning at time t = 15s, the torque applied by the prime mover to the shaft of the machine begins to ramp up. The torque ramp continues until time t = 20s, at which time the torque has reached 25% of the machine's rated torque. At 25% of rated torque, the phase of the stator voltage leads that of the grid voltage, and the machine drives about $83MW$ active power and $13MVAr$ reactive power into the grid.

The system was first simulated using a reference (benchmark) simulation to provide a basis for performance and accuracy comparison of the QDL simulation. The reference solution was obtained by using an implicit Euler integration in MATLAB. A fixed time step of $(10^{-4}s)$ was chosen for the Euler solution to ensure stability considering the entire range of eigenvalues. The QDL simulation was obtained by using the LIQSS1 integration method described below.

## 7.4 Implementation of the QDL Model

The specific QSS integration method used for this example is LIQSS1, described in [16] and [12]. The system model comprises seven QDL atom models, one for each of the seven state variables. Figure 7.3 shows the seven QDL atoms comprising the synchronous machine with directional arrows indicating the external transition graph connections. The system is tightly coupled and components have bidirectional connections. Bi-directional arrows indicate that a state update of either atom requires an update of the other. As an example, following the string of just one arrow, any

update to the output of $\psi_{dr}$ requires $\omega_r$ to update the time at which it expects to reach its next quantum level. If the update of $\psi_{dr}$ results in a change of the quantized state of $\omega_r$, then the $\theta$ atom will update the estimate of its transition time to the next quantum transition, and that will feed back to require a next update of $\psi_{dr}$.

To start the solution, the next update time of each atom is initialized to infinity. Each atom then independently computes its own next update time (the time at which it should arrive at its next higher or lower quantum state). The atom with the nearest update time is then updates its external state ($q$) to the appropriate adjacent quantum level. This external state update then flags the connected atoms to update their internal states and next external transition time, and the loop continues. A flowchart of the process is shown in chapter 4 figure 4.2.

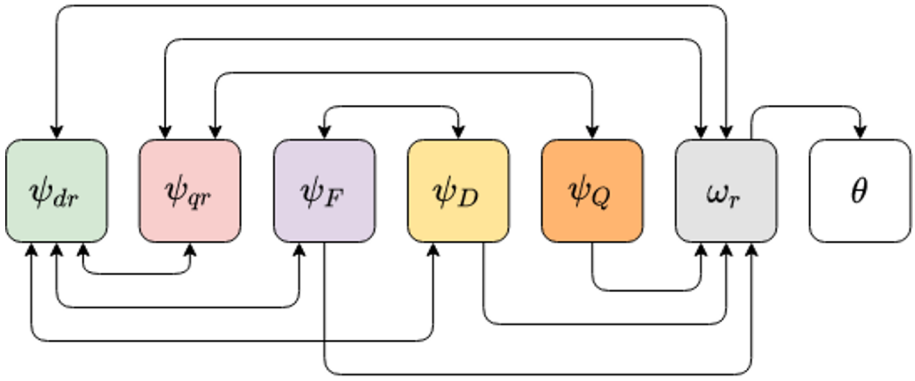



Figure 7.3 Synchronous machine model QDL atoms showing the external state transition connections.

## 7.5 QDL Performance

The performance of the QDL method is shown in comparison to the performance of the reference solution in a series of plots. Each plot shows the trajectory of a state variable and the cumulative number of updates (or re-evaluations) of the QDL atom for that state variable. All states in the system use the same quantization step size

($\Delta Q$) of $10^{-4}Wb$, except the machine rotor speed ($\omega_r$) for which we choose a $\Delta Q$ $1/10^{th}$ as large as the others ($10^{-5}rad/s$) since the dynamic range of the rotor speed is much less than that of the system fluxes. For simplicity, this simulation does not use a complex methodology for choosing the best $\Delta Q$. A $\Delta Q$ of roughly 0.1% of the total expected deviation (the absolute value of the dynamic range of the quantity in the reference simulation) of the quantized state variable. Another objective of the this simulation is to investigate the relationship between $\Delta Q$ and solution error. Understanding this relationship could lead to a more rigorous methodology for choosing $\Delta Q$ based on a desired error bound. Figures 7.4, 7.5 and 7.6 show the accuracy with which the QDL method tracks the reference solution. The cumulative number of updates for a particular atom is shown by the red lines in the following charts, with each state's cumulative updates being unique. Further discussion on the relationship between $\Delta Q$ and error, and alternative methods for $\Delta Q$ selection can be found in chapter 9.

An performance advantage of the QDL method over the reference solution can be seen as the system reaches the new steady state condition and the rate of atom updates markedly decreases. For example, in figure 7.4, during the interval from 0 to 15 seconds, while the system is in steady-state, there are very few state updates. During the period from 15 sec to 35 seconds, however, while the machine is accelerating (and the other states of the system are rapidly changing), the slope of the curve representing the cumulative number of updates is large. Finally, after around 35 seconds, the system approaches a new steady-state and the update rate decreases. This aspect of the QDL method allows the simulation to advance rapidly during steady-state conditions, and speed up again when fast transient behavior requires more time resolution. Also, the cumulative number of updates depends on the chosen $\Delta Q$. In general, a smaller $\Delta Q$ will tend to require more frequent updates, and the simulation will tend to advance more slowly through time. However, it is important

to note that the effects of changing the $\Delta Q$ on performance and accuracy is complex, and warrants further discussion and investigation (see chapter 9).

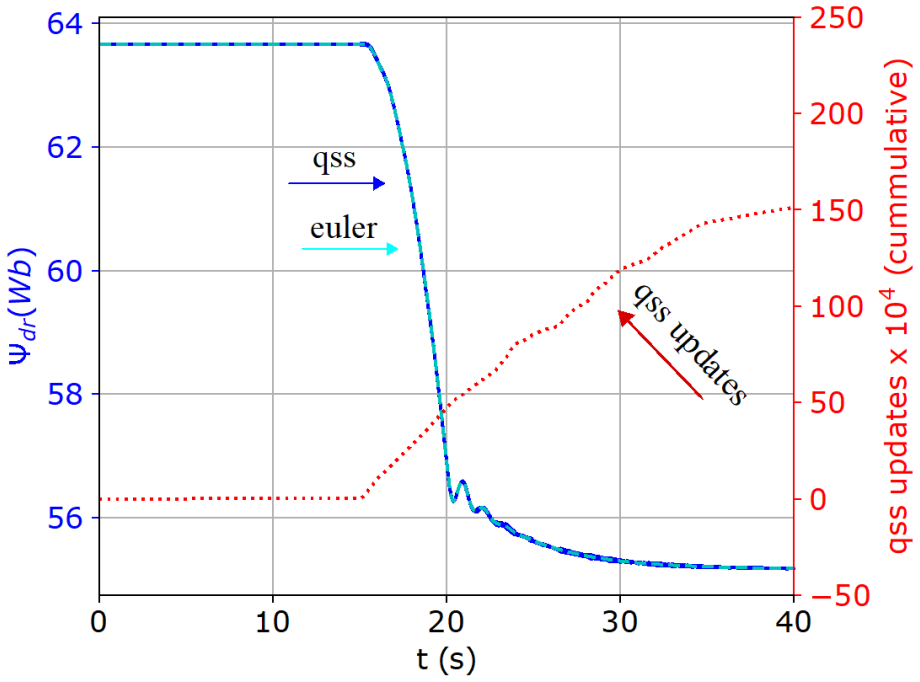



Figure 7.4 Rotor d-axis flux. The flux computed by the QDL method is nearly identical to that computed by the reference method so the two lines are nearly indistinguishable. Cumulative count of $\psi_{dr}$ atom updates shows little activity prior to torque ramp, higher activity during torque ramp, and a return to little activity as new steady state is attained.

The trajectory of rotor angle ($\theta$) , as shown in figure 7.7, is particularly interesting in that it shows how the count of this atom's updates increases immediately after the start of the torque ramp-up, then tapers off while the torque slew rate is constant during the interval between 15 to 20 seconds, then increases again at 20 seconds when the torque stops moving, and finally becomes small again as the rotor angle reaches its final steady state value. Figure 7.8 shows the trajectory of the rotor speed which always remains very close to 100 rad/s, but shows structure near the beginning and ending of the torque ramp, and a small speed increase during the torque ramp-up.

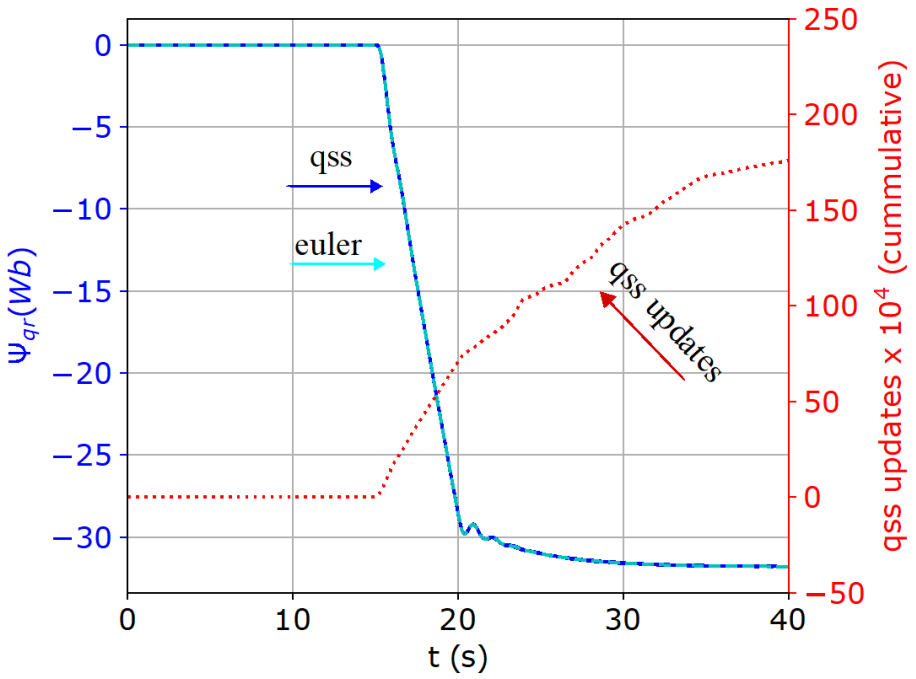



Figure 7.5 Rotor q-axis flux. The values computed by the QDL method and the reference method are nearly indistinguishable.

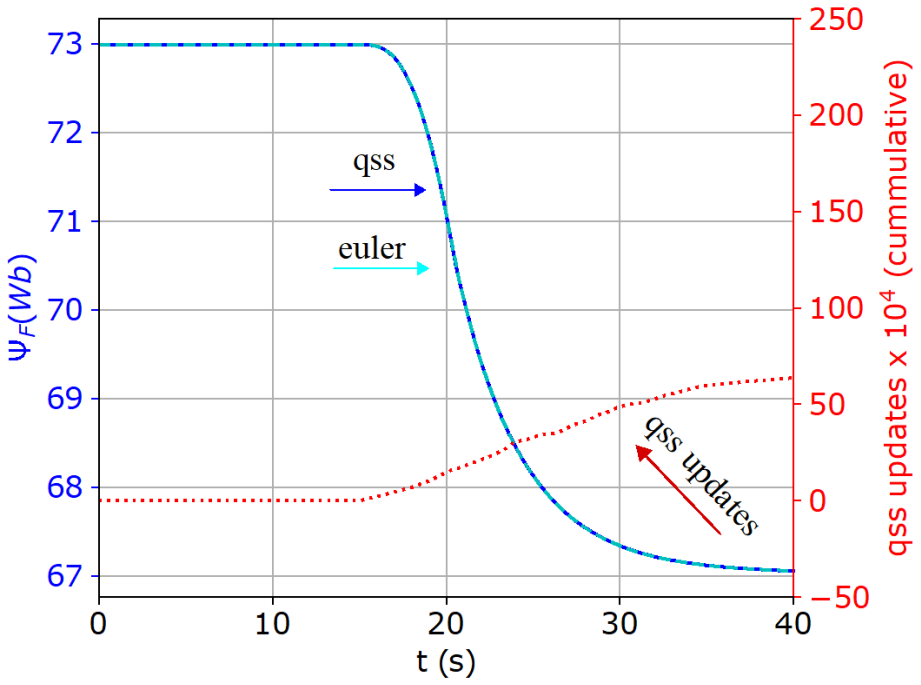



Figure 7.6 Field flux, showing good agreement between both computing methods and a total number of QDL atom updates that is smaller than the counts for d- and q-axis fluxes.

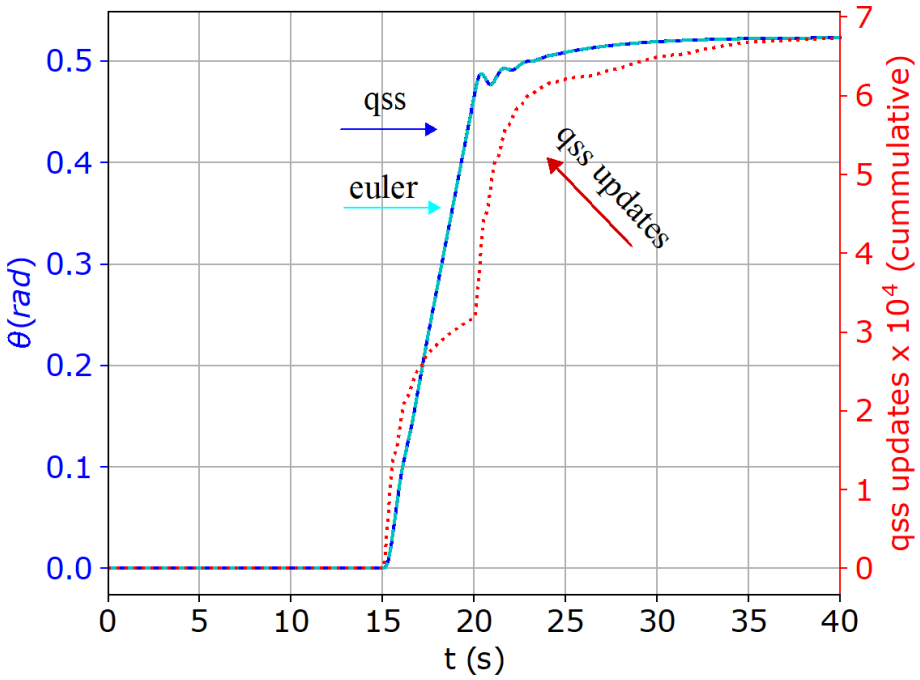



Figure 7.7 Rotor angle. QDL update rate shows interesting behavior with faster rates associated with beginning and ending of the torque ramp.

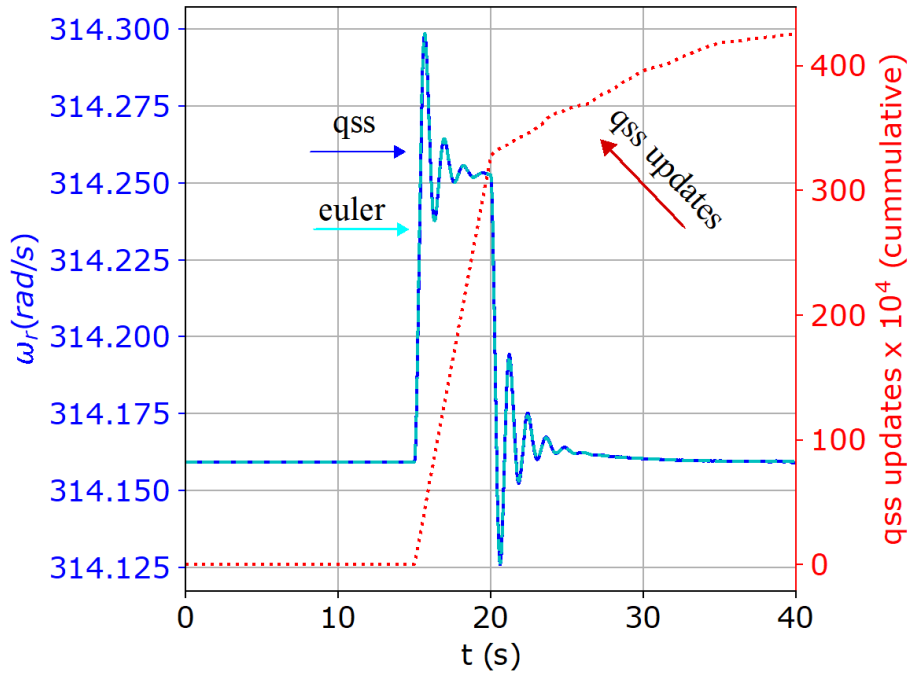



Figure 7.8 Rotor speed trajectory and update rates.

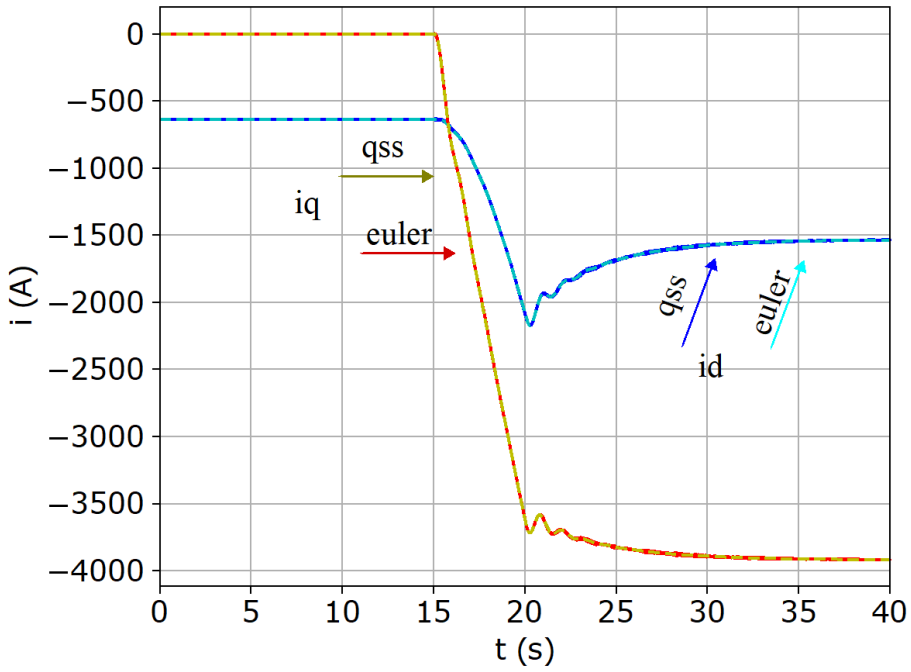



Figure 7.9 Comparisons of d- and q-axis rotor currents computed by the QDL and reference solutions.

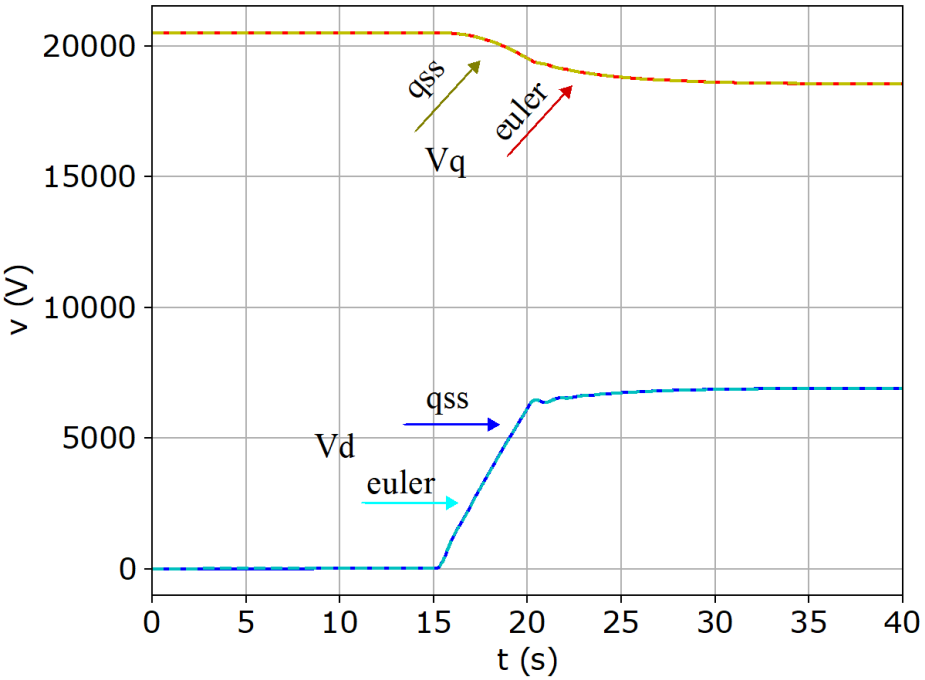



Figure 7.10 Comparisons of d- and q-axis voltages computed by the QDL and reference solutions.

## 7.6 Accuracy and Error Analysis

In [4], the authors proved that, for linear time-invariant systems, the global error in QSS integration solutions can be bounded by a constant proportional to the quantization step size ($\Delta Q$). However, our reference system is highly non-linear, so it is interesting to explore the behavior of error with varying $\Delta Q$. The effect $\Delta Q$ on simulation accuracy is shown in figures 7.11, 7.12 and 7.13, where several system variables are plotted for a range of $\Delta Q$ values, with enhanced detail during particular time periods. A larger quantization step size results in both a lower update rate and a higher error amplitude compared to a situation with a smaller $\Delta Q$. In each case, the quoted $\Delta Q$ applies to all of the state variables except rotor speed, for which the quantization step is $1/10^{th}$ that of the other variables. In each plot, the trajectory with green color corresponds to the largest $\Delta Q$ of $10^{-3}$, while blue and red correspond to smaller sizes ($\Delta Q = 10^{-4}$ and $\Delta Q = 10^{-5}$ respectively). The rotor speed calculated with the largest $\Delta Q$ has the largest error in comparison to the reference solution. This is evident in both figure 7.12, in which resolution is increased near the apex of the speed trajectory, and in figure 7.13 where higher resolution shows a slightly oscillating speed trajectory after the torque ramp. These high frequency oscillations are inherent to the QDL method and the amplitude of these oscillations increases as the $\Delta Q$ increases. The sensitivity data provided in this chapter have been empirically determined from simulation of this specific system with specific component parameters. Although this empirical data does provide useful insight into the relationship between $\Delta Q$ and error, it does not solve the problem for the general case.

Figures 7.14 and 7.15 describe the same behavior as was shown in figures 7.11, 7.12, and 7.13 but from the error perspective. Figure 7.14 shows how the pointwise absolute error (the difference between the QDL simulation and the reference simulation) varies over a particular half-second interval. The time variation of error in the state variable $\psi_{dr}$ is shown for several different $\Delta Q$ ranging from $\Delta Q = 10^{-6}$

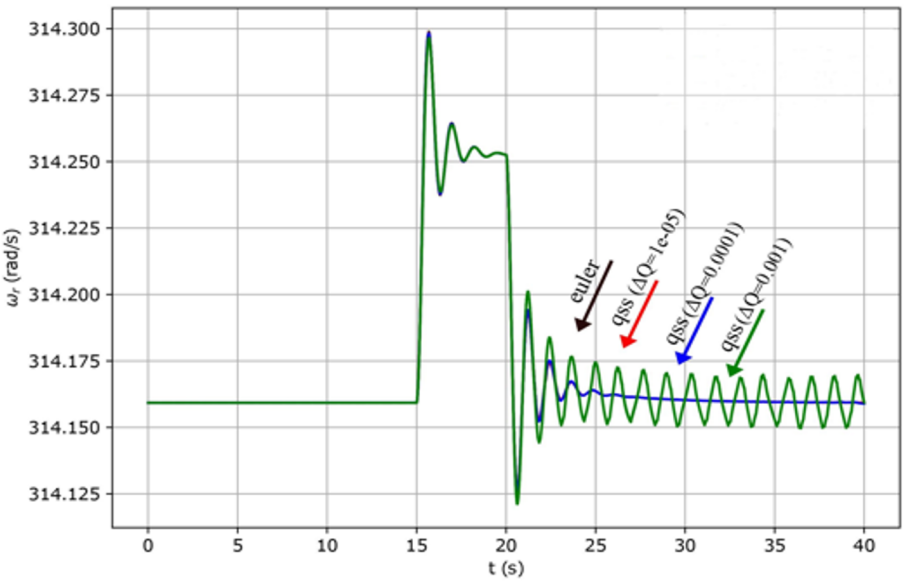



Figure 7.11 Rotor speed using different quantization sizes ($\Delta Q = 10^{-5}$, $\Delta Q = 10^{-4}$, $\Delta Q = 10^{-5}$ vs Euler method reference solution.

to $\Delta Q = 10^{-2}$. Clearly, a larger $\Delta Q$ produces a larger error in this case, but the relationship was not linear. Before the torque ramps up, the different $\Delta Q$ values all produce negligible errors. After the torque starts to ramp up, the number of updates starts to grow and the models using larger quantization sizes produce larger errors. Although a small quantization size improves the simulation accuracy, it also causes a larger model update rate. Therefore, if computing speed is important, and if a larger percent error is tolerable, one might choose a large $\Delta Q$ in order to achieve the requisite computing speed.

To generate the data shown in figure 7.15 we quantized the rotor speed at $\Delta Q = 10^{-7}$ and the other state variables at $\Delta Q = 10^{-4}$. This was an experiment to see if choosing a relatively small quantization size for some particular interesting state, but leaving other quanta larger, would produce fewer updates for the whole system, while still having a small error for the particular state of interest. Figure 7.15 shows that the state of rotor speed experienced a high number of updates, but the error did not

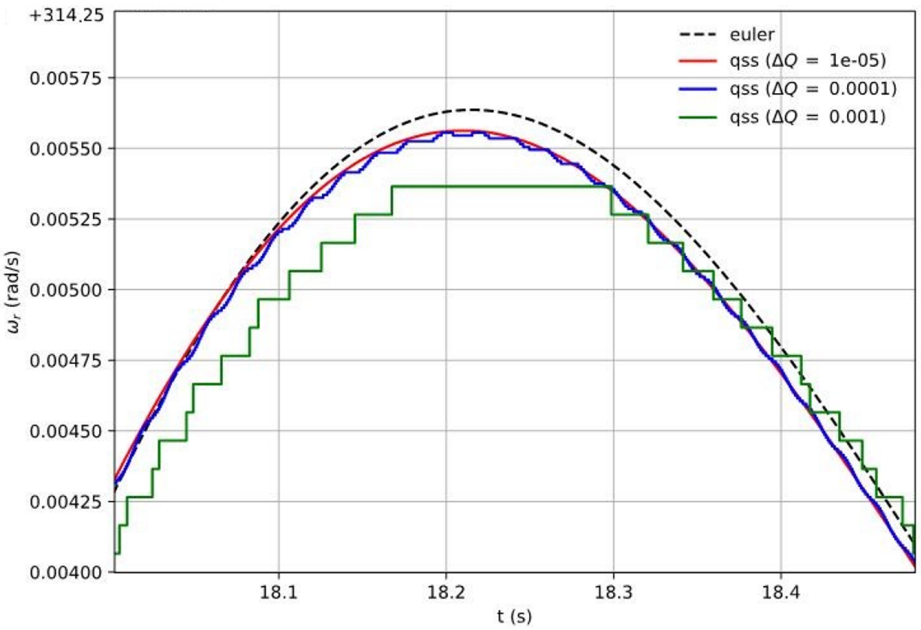



Figure 7.12 Zoom plots from 18 sec to 18.5 sec of rotor speed using different quantization sizes $\Delta Q = 10^{-5}$, $\Delta Q = 10^{-4}$, $\Delta Q = 10^{-3}$ vs Euler method reference solution.

reduce compared to the simulation with system-wide $\Delta Q = 10^{-4}$, which produced the same output with a smaller number of updates.

Figure 7.16 shows the maximum error among any system variable at any time during the simulation interval. The graph is plotted on a logarithmic scale as a function of the $\Delta Q$, which was varied from $10^{-6}$ to $10^{-2}$ Wb. Also plotted is the corresponding sum of all updates of all atoms over the entire simulation. For small $\Delta Q$ values of $10^{-6}$ to $10^{-5}$ Wb, the error is very small and largely independent of $\Delta Q$. A logarithmic scale is used to emphasize that for very small $\Delta Q$ values ($10^{-5}$ Wb), a decrease in $\Delta Q$ does not improve the accuracy, but it does impose a penalty on computational intensity (simulation update rate), and the simulation takes longer to advance through time with no- significant error reduction. A sweet spot is evident around $\Delta Q$ between $10^{-5}$ to $10^{-4}$, where computational intensity has become relatively low while error also remains low. Above $\Delta Q$ values of around $10^{-4}$,

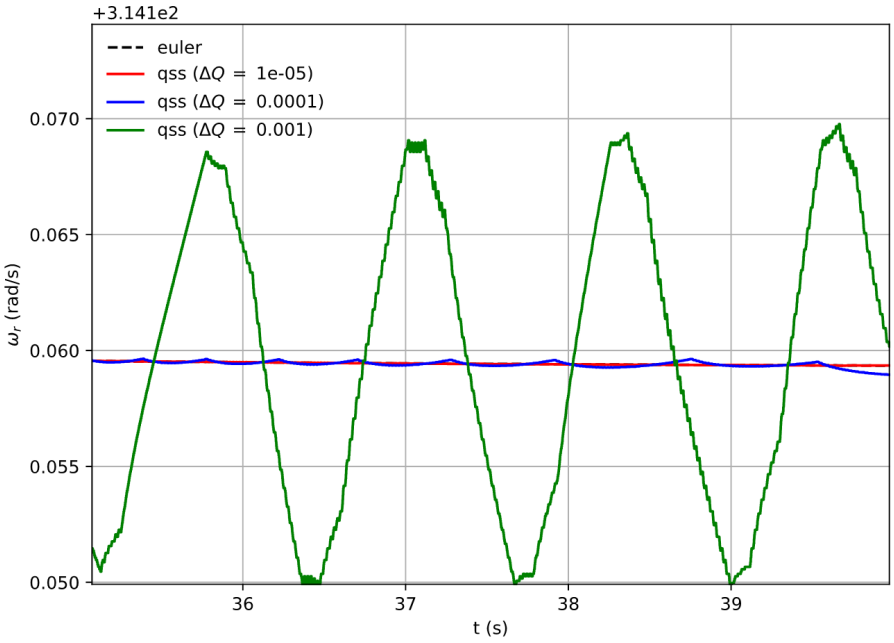



Figure 7.13 Zoom-in plots that show details for steady state situation of rotor speed using different quantization sizes $\Delta Q = 10^{-5}$, $\Delta Q = 10^{-4}$, $\Delta Q = 10^{-5}$ vs Euler reference solution. The oscillations have very small amplitudes.

the error increases rapidly, but without a consistent reductions in computational intensity.

Although the QDL method does accurately track the reference solution, the method does inherently exhibit steady-state oscillations. These oscillations are evident in figure 7.17, which shows the direct axis rotor flux at high resolution just near the onset of the torque ramp at 15 seconds. The amplitudes of these high frequency oscillations reduce as the $\Delta Q$ is reduced, but the oscillations are always present. Each oscillation represents an update event, so reducing the oscillation size comes at the expense of computation time. Figures 7.17 and 7.18 show these high frequency oscillations at $\Delta Q$ values of $10^{-4}$ and $10^{-5}$. The mitigation of this high-frequency oscillation behavior is discussed further in chapter 10.

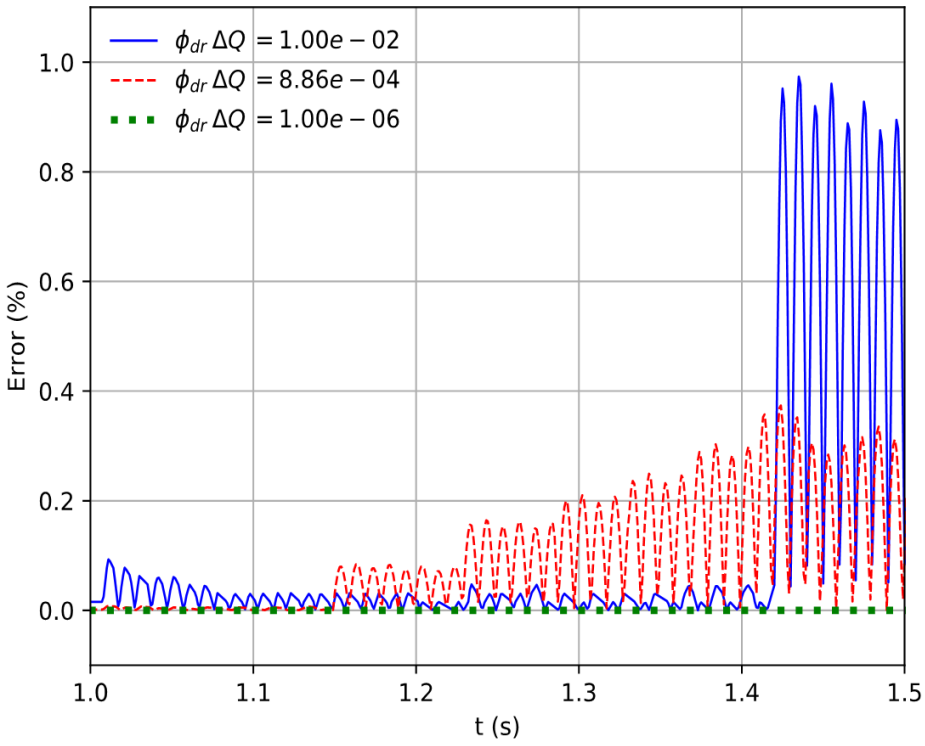



Figure 7.14 Error between reference solution and QDL solution of rotor d-axis flux for several different quantization sizes $\Delta Q = 10^{-2}$ , $\Delta Q = 8.86 \cdot 10^{-4}$, $\Delta Q = 10^{-6}$.

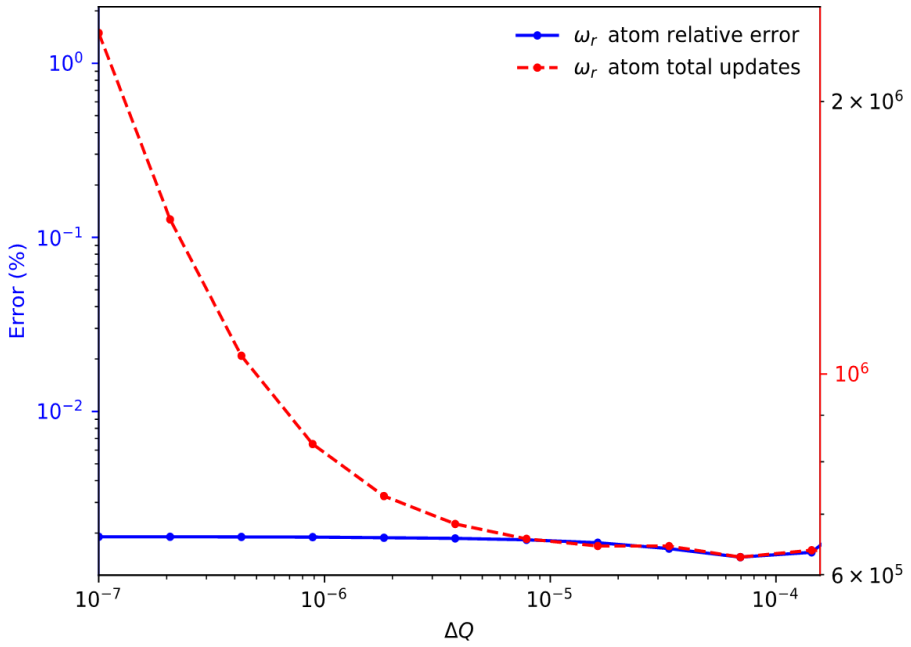



Figure 7.15 Rotor speed updates vs relative error when rotor speed $\Delta Q$ is set to $10^{-7}$ and the quantization size of the rest of the system variables is $\Delta Q = 10^{-4}$. Possibly unnecessary high precision with low benefit of error reduction

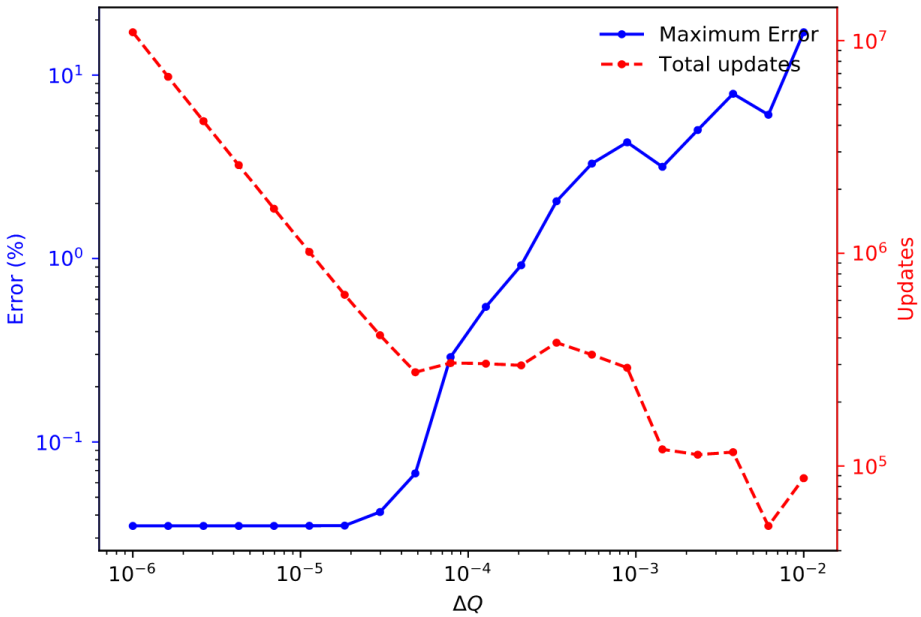



Figure 7.16 Maximum Error of all atoms for different $\Delta Q$ values. Total number of the updates decreases as the system is simulated with bigger $\Delta Q$ values at the expense of increasing the error.

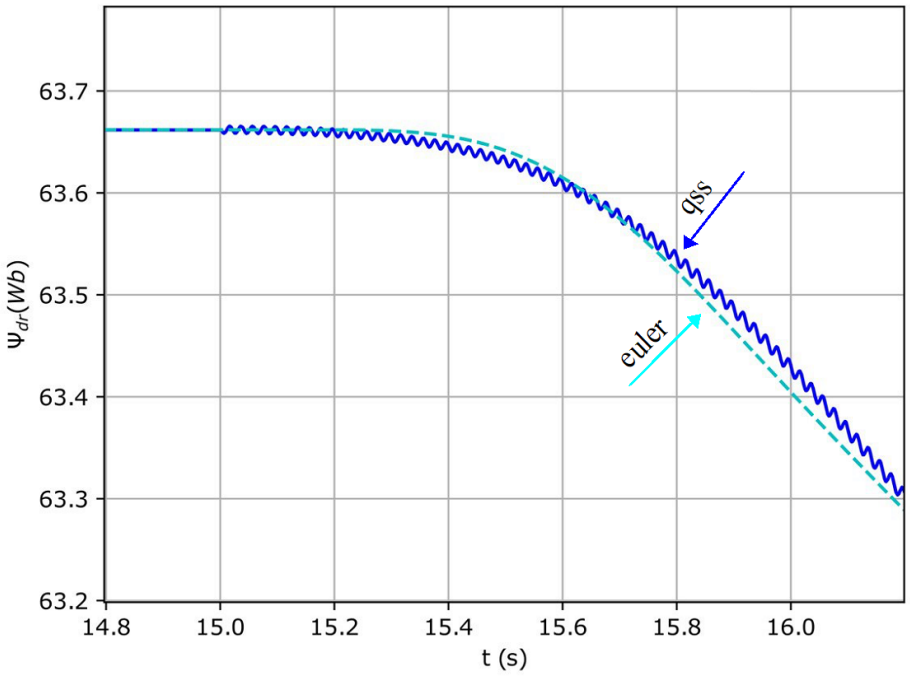



Figure 7.17 High resolution plot showing the ripples in QDL solution of the rotor d-axis flux $\psi_{dr}$ with $\Delta Q$ of $10^{-4}$.

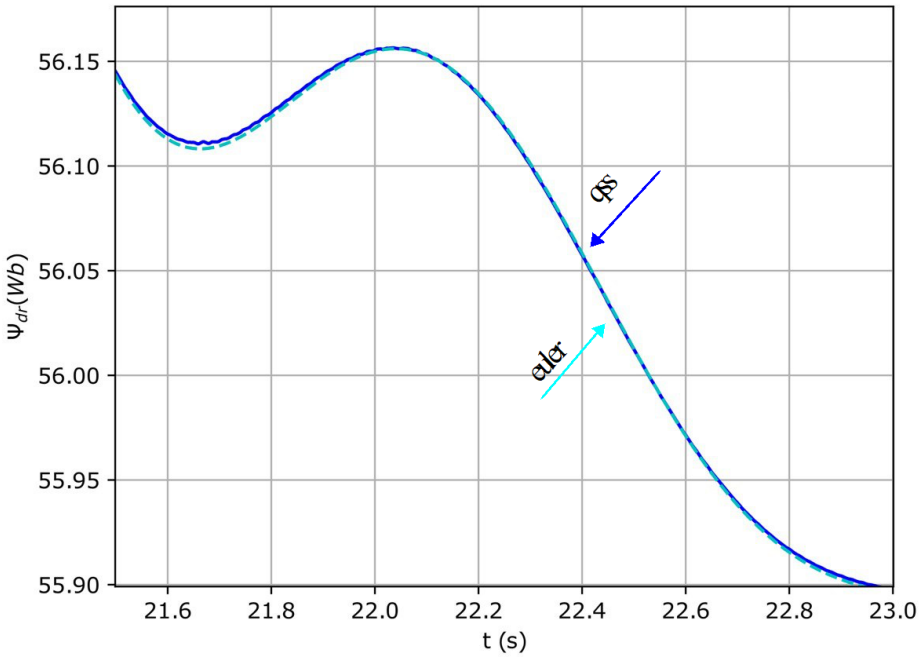



Figure 7.18 High resolution plot showing the ripples in the QDL solution of the rotor d-axis flux $\psi_{dr}$ with $\Delta Q$ of $10^{-5}$. The amplitude of the ripples is smaller compared to $\Delta Q$ of $10^{-4}$.

## 7.7 Study Summary

The performance of the QDL method for analyzing the dynamics of non-linear power system that includes a detailed synchronous machine model has been investigated. Uniform quantization of system state variables at 0.01 percent was found to yield accuracy within 0.4 percent of that achieved with a conventional implicit state-space solution, but with a significant advantage in computational intensity, especially for systems that operate for long times in a quasi-steady-state. Since the QDL method enables the user to individually set the quantization step size ($\Delta Q$) values of each state, we evaluated performance as a function of $\Delta Q$. When the system was simulated using one uniform quantization size for all states, the total number of state updates generally decreased as $\Delta Q$ increased, but above a quantization size of about $10^{-4}$, further increase in $\Delta Q$ did not significantly reduce the computational cost (but it did decrease the simulation accuracy). When $\Delta Q$ for a single state of interest was set smaller than the uniform $\Delta Q$ of other states, refining the $\Delta Q$ of that particular state variable did not necessarily improve the error of that particular state, but it did logarithmically increase the state update intensity of the whole system. The observations of the effects of $\Delta Q$ selection are limited by the particular system that was studied. Other systems, especially those having state variables that are widely different in magnitude, may behave differently.

# Chapter 8

# Power System Simulation

The purpose of this chapter is to test the feasibility and performance of the QDL method for the modeling and simulation of the dynamic performance of a relatively large and realistic power system model. The system model created for this evaluation is a three-phase network consisting of a synchronous machine, a turbine, a governor, an exciter, an induction motor, an ac/dc converter, a dc load, and a constant impedance load. All of these devices are connected through a three-phase network consisting of several buses and transmission lines. This study system contains a high range of transient behavior, from the slow mechanical transients of the rotating machines, to the very fast transients of the electrical quantities on the network. This is therefore mathematically a very stiff system model, which poses a challenge of effectively simulating the system over significant periods of simulation time while capturing the full range of transient behavior using a single representation of the power system. Because the QDL approach primarily quantizes the simulation states (and not time), it should therefore be well-suited to the task of simulating the slow and fast dynamics of system without some of the numerical stability and efficiency issues inherent in solving stiff systems using traditional time-slicing methods.

The benefits of QDL methods are best leveraged by systems whose states have derivatives that approach zero at steady-state. A ac power system that has sinusoidal steady-state behavior in the stationary reference frame is therefore best modeled in the rotating reference frame using a Park transformation. The QDL method than then take advantage of the benefits of the QDL steady-state performance behavior.

The variable time spans between state updates should be longer during steady-state conditions without sacrificing accuracy during the fast transients.

## 8.1 System Model

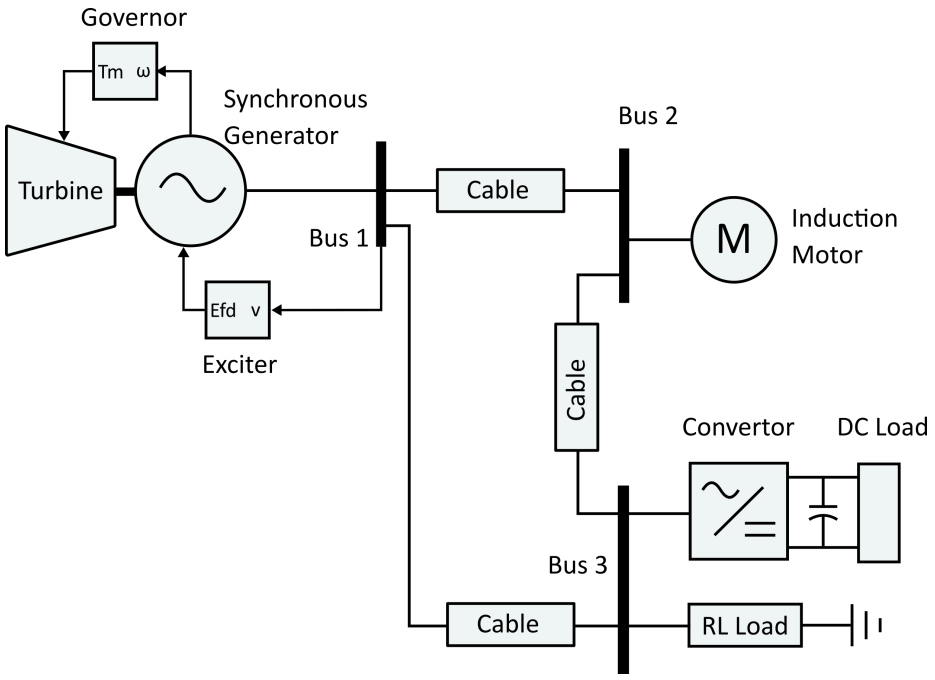



Figure 8.1 Power system schematic

The reference system is a three-phase, 60 Hz, 4160 V (line-line RMS) power system consisting of a synchronous machine being operated as a generator, connected to three loads via a network of buses and cables. An induction motor is connected to bus two and the other two loads, a dc load behind a converter and a constant impedance (RL) load are connected to bus three. The converter is represented as a non-linear dq averaged model. The cables are represented using a standard pi model. With 32 states, this power system is designed to be simple enough for the testing of a novel simulation method, while complex enough to represent a realistic, non-linear power system. For each device, a model was QDL compliant device model was formulated. These models are described in the next section.

The descriptions for each of the device models for the study system are given below, along with the derivation of the QDL model.

## 8.2 Cable Model

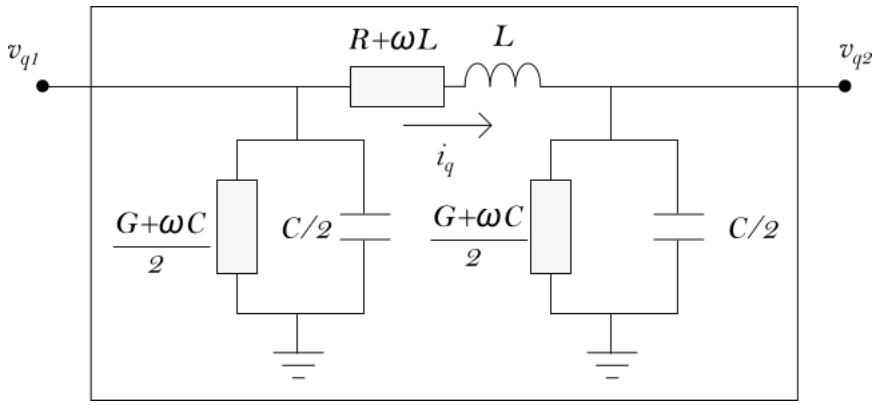



Figure 8.2 Cable model

Each cable is represented as a lumped pi model of a transmission line as described in [19]. The equivalent circuit is shown in figure 8.2. Note that only the q-axis model is shown. The d-axis portion of model and is identical to q-axis portion, except the voltage and current quantities are their d-axis counterparts. This model differs from the cable model in [19], in that the shunt conductance and capacitance elements have been added to the ends of the branch in order to provide full latency required by the QDL and simulation method. This model provides full branch and node latency by exploiting the inherit latency from the physical branch inductance and the node capacitance. The dynamic equations that describe the cable model are

$$\frac{d}{dt}i_q = \frac{-(R+\omega L)i_q + (v_{q1} - v_{q2})}{L} \tag{8.1}$$

$$\frac{d}{dt}v_{q1} = \frac{-(\frac{G+\omega C}{2})v_{q1} + \sum i_{branch}}{C/2} \tag{8.2}$$

Where all quantities are those represented in figure 8.2, and $\sum i_{branch}$ is the sum of all currents injected from all branches connected to node $v_{q1}$. Similar equations are used for the other states $v_{q2}$, $v_{d1}$, $v_{d2}$ and $i_d$.

To construct the system model, the LIM node conductance and capacitance quantities are summed together automatically with other devices connected to the same bus. In other words, each bus's LIM node model is created by combining the contributions from the terminals of all connected devices with shunt conductance and capacitance elements.

## 8.3 RL Load Model

The RL Load is modeled as dq series impedance branches connected to ground. The equivalent circuit is shown in figure 8.3

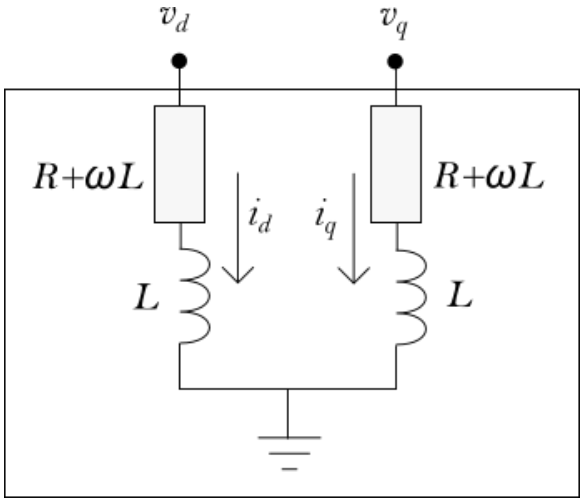



Figure 8.3 RL Load

Because this model is fully described by LIM atoms, the equations are not given here (see the LIM node atom description in chapter 2, figure 2.1).

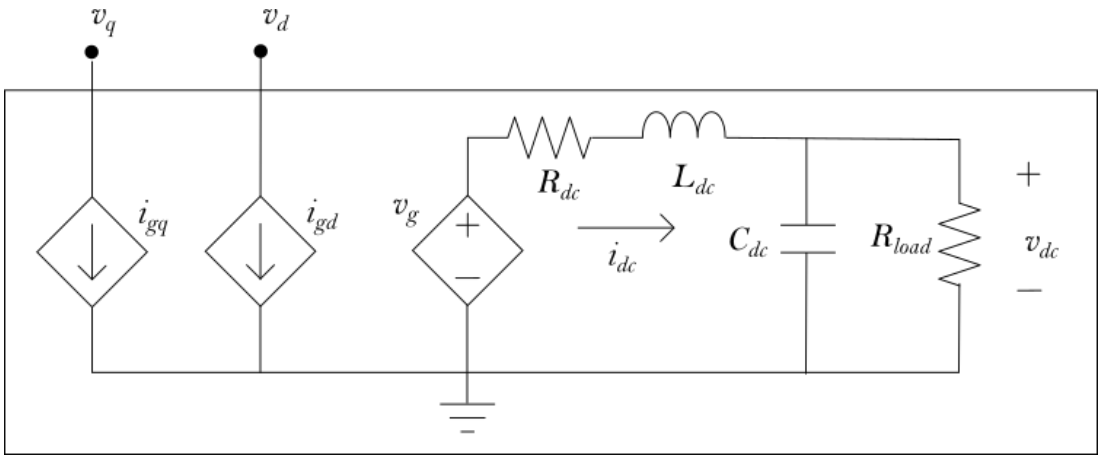



Figure 8.4 Transformer-Rectifier Load

## 8.4 Transformer-Rectifier Model

The Transformer-rectifier load model is based on the model found in [20], and includes a non-linear coupling between d and q-axis constant current branches and a dc side with a LIM branch and node. The equivalent circuit model is shown in figure 8.4. The ac-dc coupling is defined by

$$i_{gd} = S_d i_{dc} \tag{8.3}$$

$$i_{gq} = S_q i_{dc} \tag{8.4}$$

$$v_g = \frac{v_d}{S_d} + \frac{v_q}{S_q} \tag{8.5}$$

where the coupling coefficients are

$$S_d = 2\sqrt{\frac{2}{3}}\frac{\sqrt{3}}{\pi}\cos\phi \tag{8.6}$$

$$S_q = -2\sqrt{\frac{2}{3}}\frac{\sqrt{3}}{\pi}\sin\phi. \tag{8.7}$$

where $\phi$ is the firing angle. The dc subsystem of this model is a LIM branch, the equations for which are given in chapter 2, figure 2.2.

## 8.5 Synchronous Machine

The synchronous machine [20] is modeled at two LIM branches for the electrical model (one for each dq axis), and a LIM node for the mechanical dynamics. The equivalent circuit is shown in figure 8.5

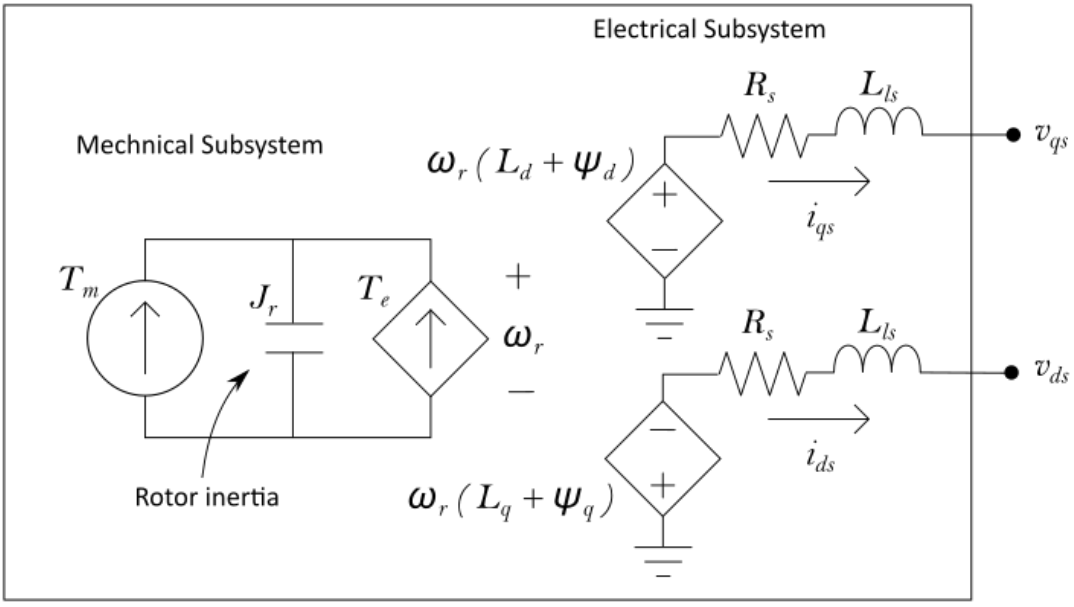



Figure 8.5 Synchronous Machine

The electro-mechanical dynamics are modeled with the flux differential equations

$$\frac{d}{dt}\psi_{kq} = \frac{r_{kq}}{L_{lkq}} \left(\psi_{kq} - L_q i_{qs} - \psi_q i_{qs} + L_{ls} i_{qs}\right) \tag{8.8}$$

$$\frac{d}{dt}\psi_{kd} = \frac{r_{kd}}{L_{lkd}} \left(\psi_{kd} - L_d i_{ds} - \psi_d i_{ds} + L_{ls} i_{ds}\right) \tag{8.9}$$

$$\frac{d}{dt}\psi_{fd} = \frac{r_{fd}}{L_{lfdd}} \left(\psi_{fd} - L_d i_{ds} - \psi_d i_{ds} + L_{ls} i_{ds} - v_{fd}\right) \tag{8.10}$$

where the electrical torque $T_e$ is coupled to the terminal currents $i_{qs}$, $i_{ds}$ as

$$T_e = \frac{3P}{4}\left(\psi_{ds} i_{qs} - \psi_{qs} i_{ds}\right) \tag{8.11}$$

and the q- and d-axis equivalent fluxes and inductances are given by

$$L_q = L_{ls} + \frac{L_{mq} L_{lkq}}{L_{lkq} + L_{mq}} \tag{8.12}$$

$$L_d = L_{ls} + \frac{L_{md} L_{lfd} L_{lkd}}{L_{md} L_{lfd} + L_{md} L_{lkd} + L_{lfd} L_{lkd}} \tag{8.13}$$

$$\psi_q = \frac{L_{mq}}{L_{mq} + L_{kd}} \psi_k d \tag{8.14}$$

$$\psi_d = \frac{L_{md}\left(\frac{\psi_{kd}}{L_{lkd}} + \frac{\psi_{fd}}{L_{lfd}}\right)}{1 + \frac{L_{md}}{L_{lfd}} + \frac{L_{md}}{L_{lkd}}} \tag{8.15}$$

.

## 8.6 Turbo-Governor Model

The dynamics of the prime mover and the speed governor are combined into a single model described by

$$T_m = K_p \delta_r + K_i \theta_r \tag{8.16}$$

where $T_m$ in the mechanical torque, $K_p$ and $K_i$ are the PI controller constants, and the speed deviation $\delta_r$ and rotor angle $\theta_r$ are related to the synchronous speed ($\omega_s$) and the machine rotor speed ($\omega_r$) as

$$\delta_r = \omega_s - \omega_r \tag{8.17}$$

$$\frac{d}{dt}\theta = \delta_r. \tag{8.18}$$

.

## 8.7 Exciter Model

The synchronous machine terminal voltage is regulated using an exciter. The exciter model used is a simplified IEEE AC8B Exciter model. This is modeled as four differential equations for three internal signal states as well as the output voltage.

The input is the per unit terminal voltage magnitude, and the output is the per unit field voltage $v_{fd}$. The differential equations are

$$\frac{d}{dt}x_1 = v_{term,pu} - \frac{1}{T_{dr}}x_1 \tag{8.19}$$

$$\frac{d}{dt}x_2 = x_1 \tag{8.20}$$

$$\frac{d}{dt}x_3 = \left(K_{ir} - \frac{K_{dr}}{T_{dr}^2}\right)x_1 + \frac{K_{ir}}{T_{dr}}x_2 - \frac{1}{T_a}x_3 + \left(\frac{K_{dr}}{T_{dr}} + K_{pr}\right)v_{term,pu} \tag{8.21}$$

$$\frac{d}{dt}v_{fd,pu} = \frac{K_a}{T_a T_e}x_3 - \frac{K_e}{T_e}v_{fd,pu} \tag{8.22}$$

where the per unit terminal voltage $v_{t,pu}$ is related to the machine dq quantities and the rated line-line terminal voltage magnitude $V_{base,LL}$ as

$$v_{term,pu} = \frac{\sqrt{v_{qs}^2 + v_{ds}^2}}{V_{base,LL}} \tag{8.23}$$

and the per unit output of the exciter $v_{fd,pu}$ is then scaled by the base voltage $V_{base,LL}$ before being input into to synchronous machine model as

$$v_{fd} = v_{fd,pu} \cdot V_{base,LL} \tag{8.24}$$

.

## 8.8 Induction Machine Model

The induction machine is modeled as two LIM branches for the electrical model (one for each dq axis) and a LIM node for the mechanical dynamics as shown in figure 8.8

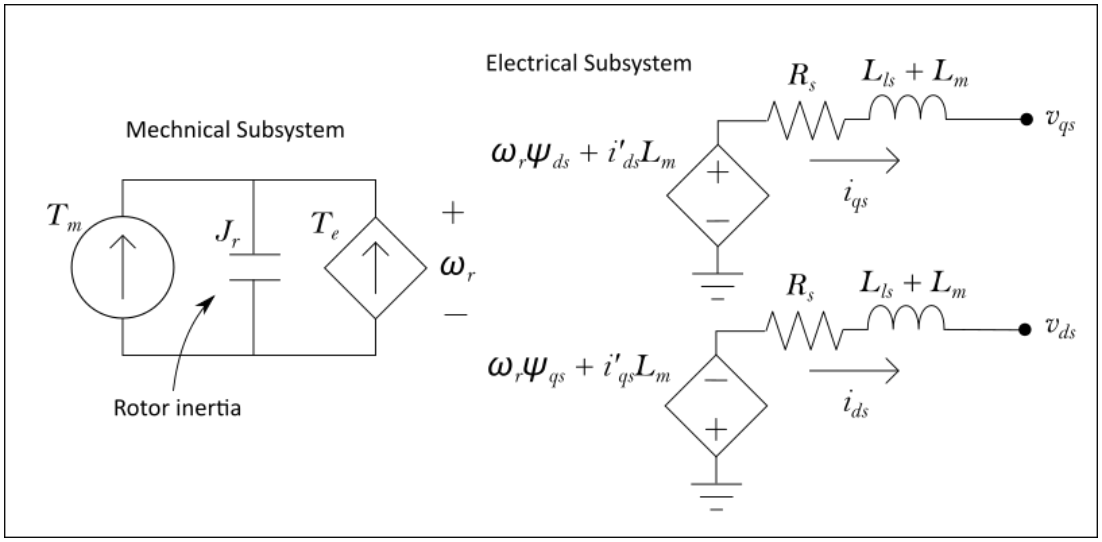



Figure 8.6 Induction Machine

The LIM sources are updated using the flux quantities defined in terms of the instantaneous stator and rotor d- and q- access currents using

$$\psi_{qs} = L_{ls}i_{qs} + l_m(i_{qs} + i_{qr}) \tag{8.25}$$

$$\psi_{ds} = L_{ls}i_{ds} + l_m(i_{ds} + i_{dr}) \tag{8.26}$$

$$\psi_{qr} = L_{lr}i_{qr} + l_m(i_{qr} + i_{qs}) \tag{8.27}$$

$$\psi_{dr} = L_{lr}i_{dr} + l_m(i_{dr} + i_{ds}) \tag{8.28}$$

,

And the mechanical dynamics are resolved using the LIM node model in figure 8.8, where the electrical torque $T_e$ is determined from the currents using

$$T_e = \frac{P}{2J}\left(\frac{3P}{4}\left(L_{ls}i_{ds} + L_m\left(i_{ds} + i_{dr}\right)\right)i_{qs} - \left(L_{ls}i_{qs} + L_m\left(i_{qs} + i_{qr}\right)\right)i_{ds} - T_b\left(\frac{\omega_r}{\omega_s}\right)^3\right) \tag{8.29}$$

where $P$ is the number of pole pairs, $T_b$ is the base torque, and $\omega_s$ and $\omega_r$ are the synchronous and rotor speeds respectively.

## 8.9 Simulation Scenario and Benchmark Solution

In the simulation scenario, the system starts in steady-state (all state derivatives are zero), using a standard iterative ODE operating point solution. At time $t = 0$ seconds, the active power of RL load is increased by 20%. A reference solution (denoted "ODE" in the output plots) is run with identical parameters and events to compare the results and quantify error.

In order to provide results for performing accuracy and performance bench-marking, a traditional integration method was used that is well-suited to non-linear, stiff systems. The modeling method used to benchmark accuracy and performance uses a representation of the system as a set of non-linear, first order differential equations. The QDL formulation of the power system provides a full state-space description of the system, including the system Jacobian matrix that can be used directly with many numerical integration techniques. The integration method used to compute the benchmark is an implicit Runge-Kutta method of the Radau IIA family of order 5, a method well-suited for these types of stiff systems. The fixed time step was chosen to be $10\mu s$ to easily accommodate the fastest eigenvalues of the system.

## 8.10 Simulation Results

Each plot presented for the simulation results includes the QDL results as the solid color curves, the benchmark results as a dashed gray curve, and the cumulative atom updates as a dashed color curve with a separated $y$ axis on the right. Each plotted atoms' results are shown for the entire 60 seconds of simulation, along with zoom plots to the transient around the disturbance at 1 second simulation time. The time range of each zoom plot is chosen to highlight the performance of QDL for the specific dynamic behavior for that specific state, as some states move much faster than others. Some plots included filtered results to reduce spurious noise as discussed in section 10.2.

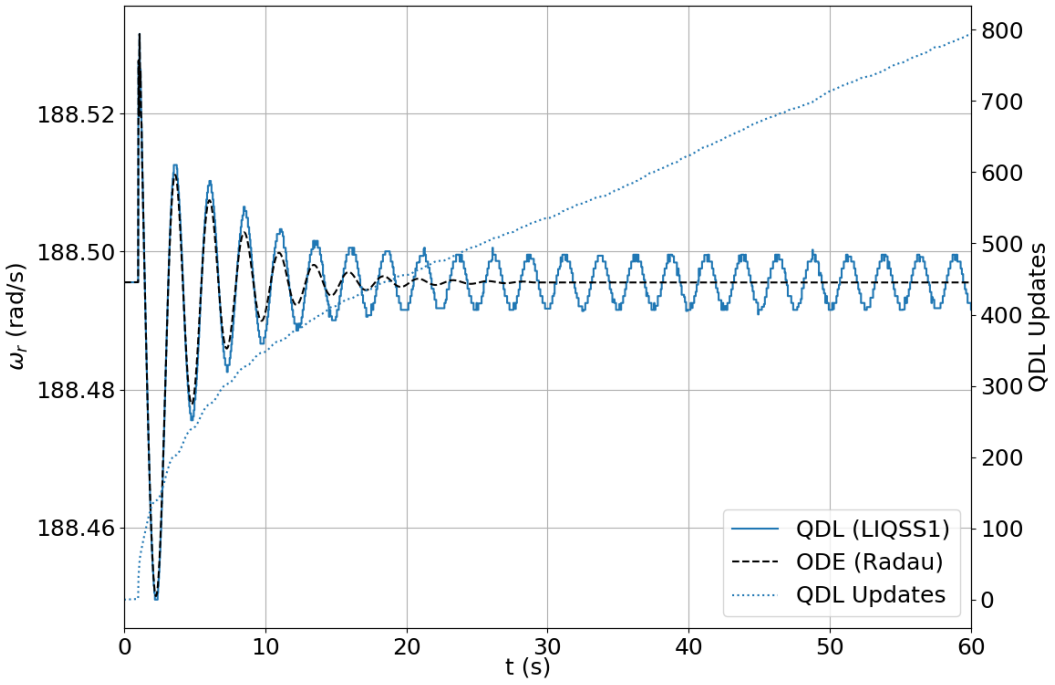



Figure 8.7 Synchronous machine speed.

From inspection of the plots in figures 8.7 to 8.20, the transient behavior of all states tracts the benchmark solution very well. The steady-state behavior, although bounded, shows significant oscillations. This highlights the importance of the topics

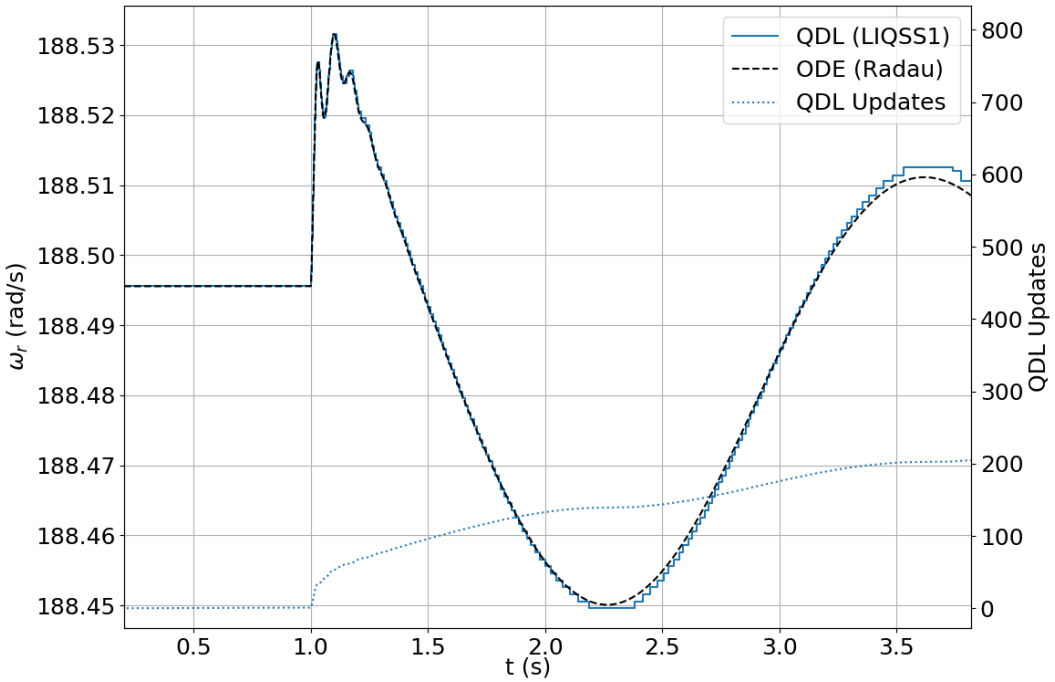



Figure 8.8 Synchronous machine speed, zoom to transient.

discussed in chapter 10 for detecting steady-state, and filtering spurious oscillations from the results using post-simulation filtering.

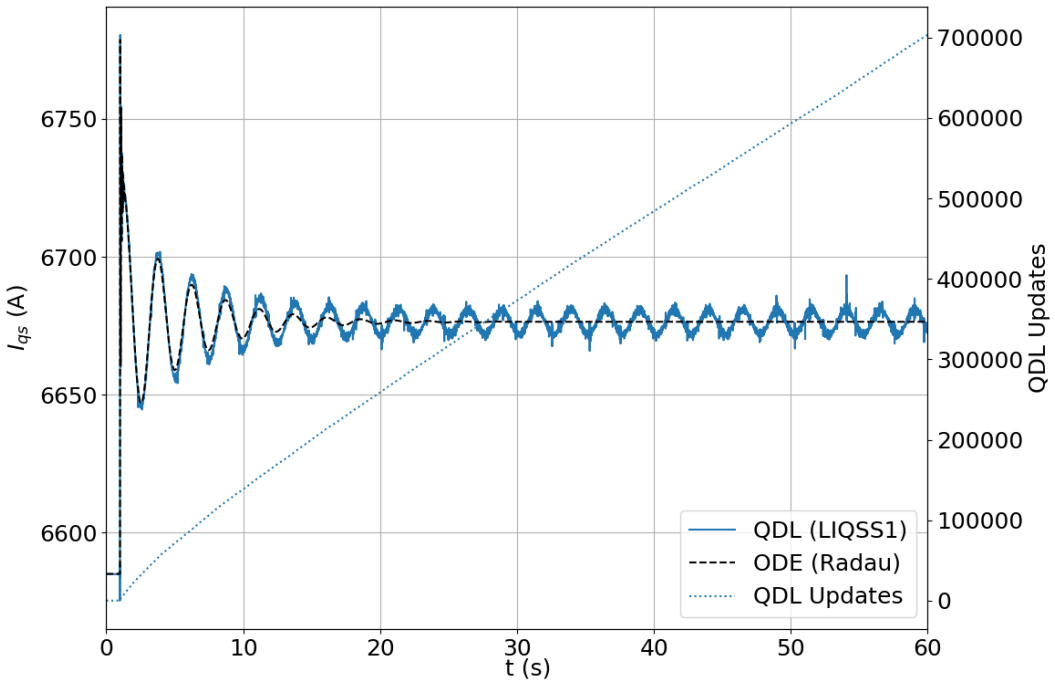



Figure 8.9 Synchronous machine q-axis stator current (filtered, $fc = 100Hz$).

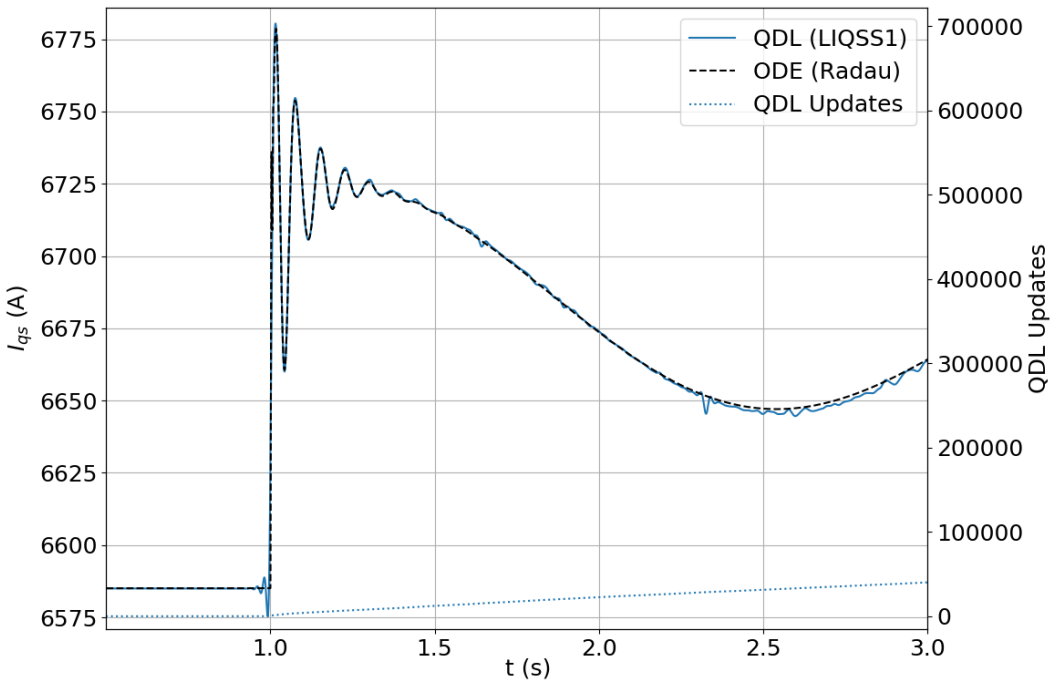



Figure 8.10 Synchronous machine q-axis stator current, zoom to transient (filtered, $fc = 100Hz$)

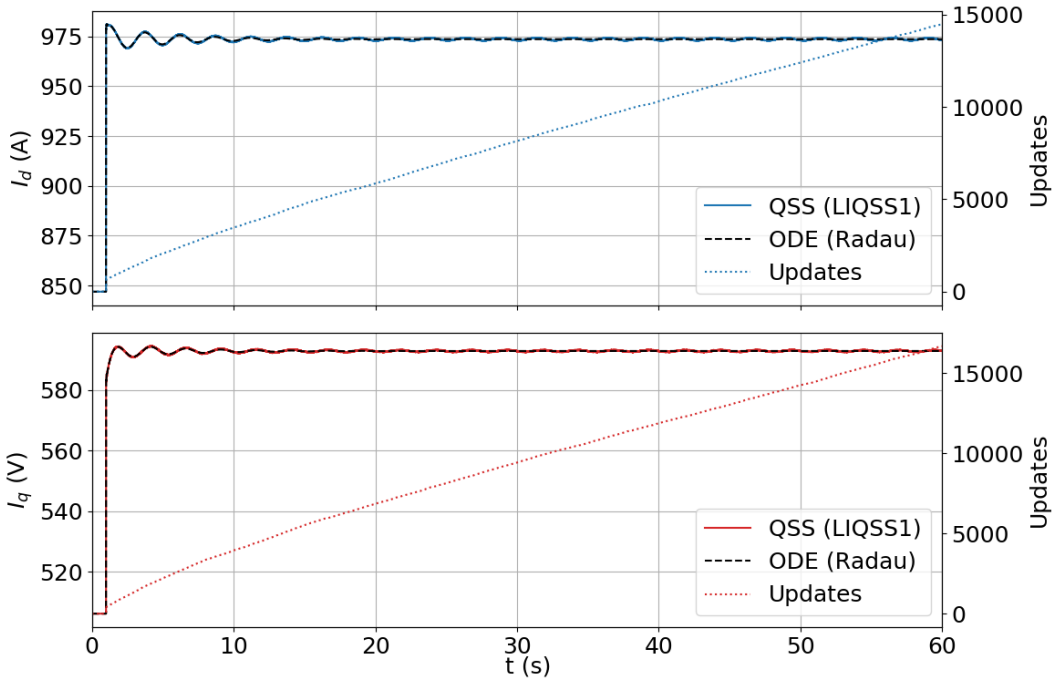



Figure 8.11 RL Load currents.

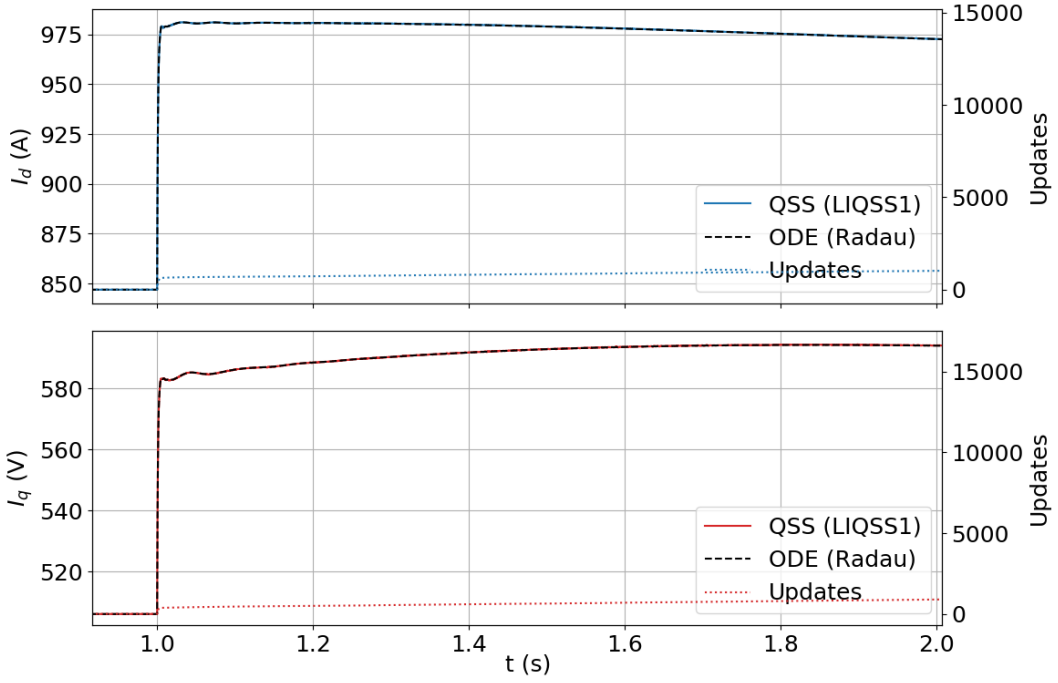



Figure 8.12 RL Load currents, zoom to transient.

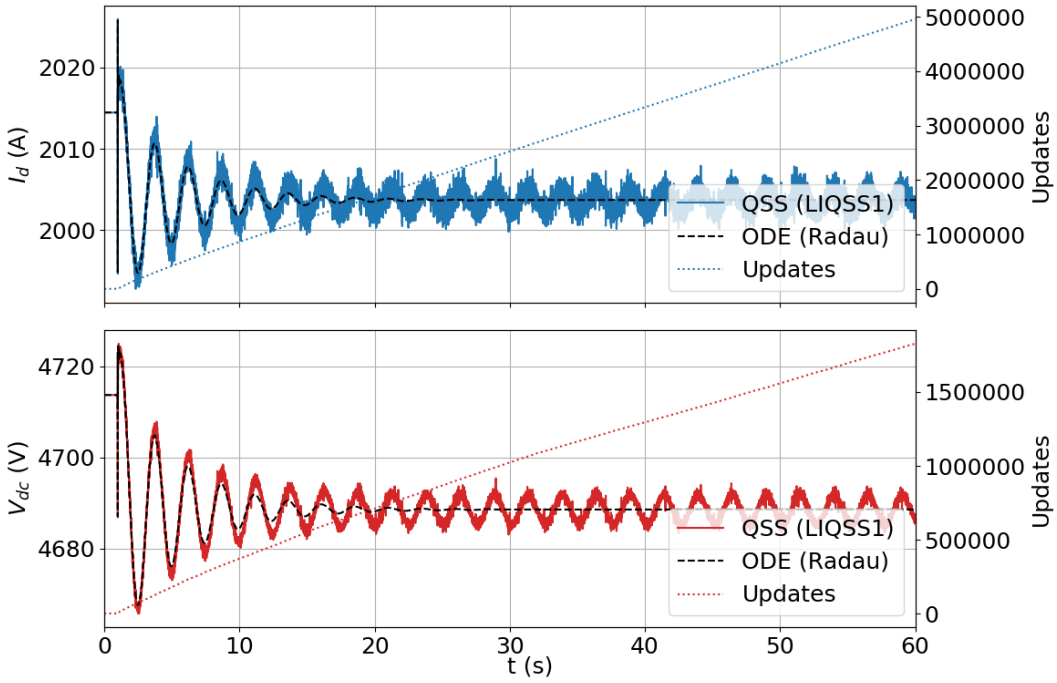



Figure 8.13 Transformer-rectifier load d-axis current and dc voltage.

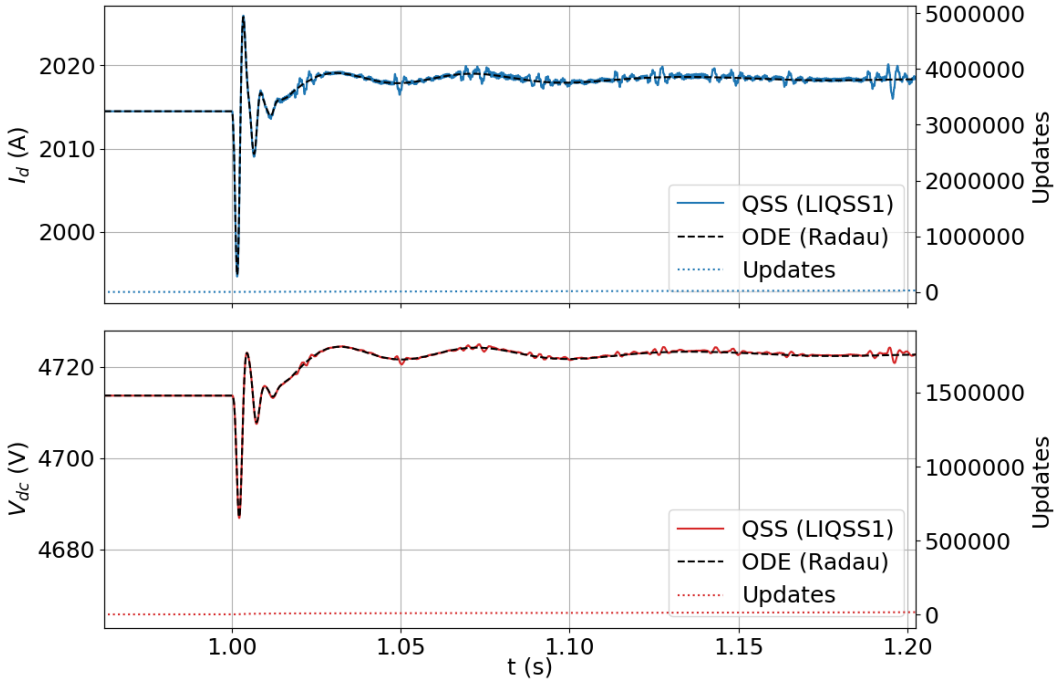



Figure 8.14 Transformer-rectifier load d-axis current and dc voltage, zoom to transient.

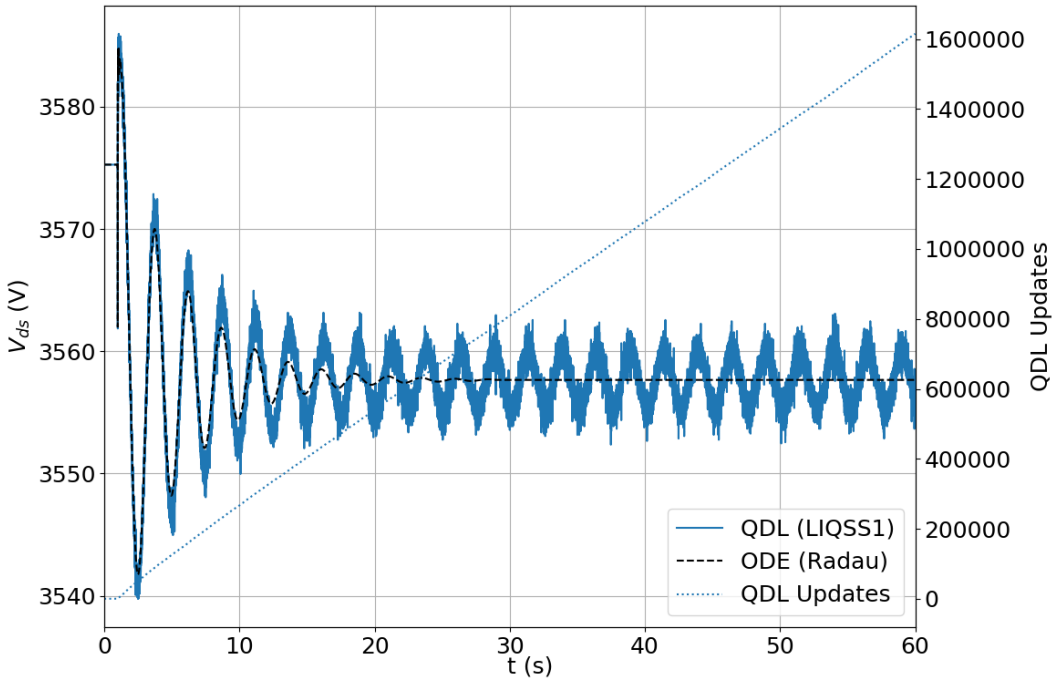



Figure 8.15 Bus 1 d-axis voltage.

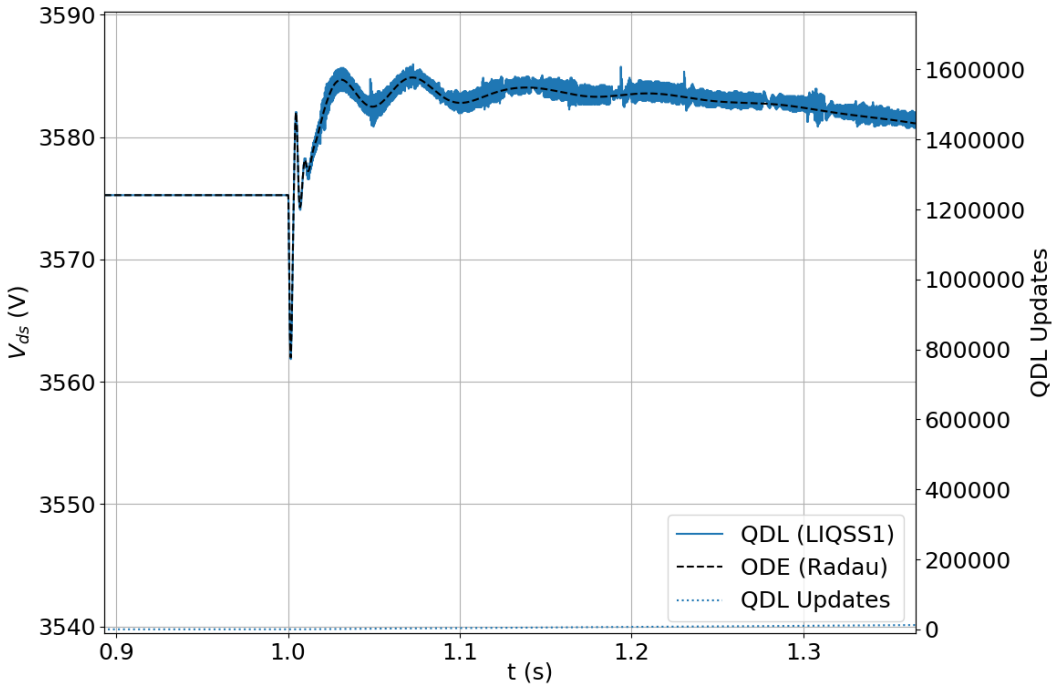



Figure 8.16 Bus 1 d-axis voltage, zoom to transient.

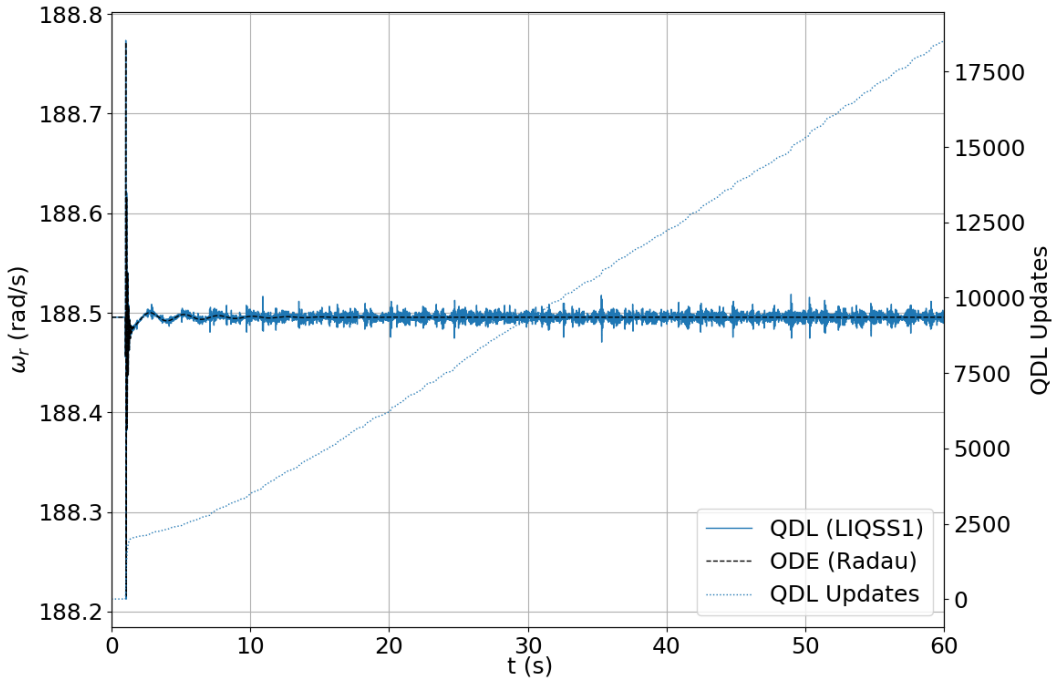



Figure 8.17 Induction machine speed.

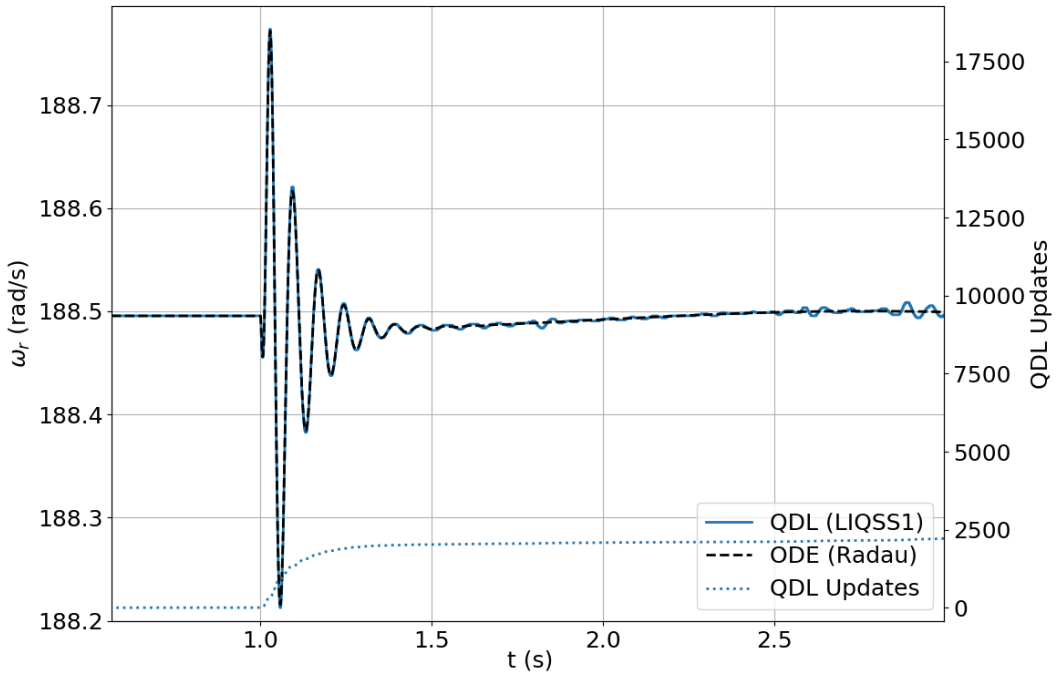



Figure 8.18 Induction machine speed, zoom to transient.

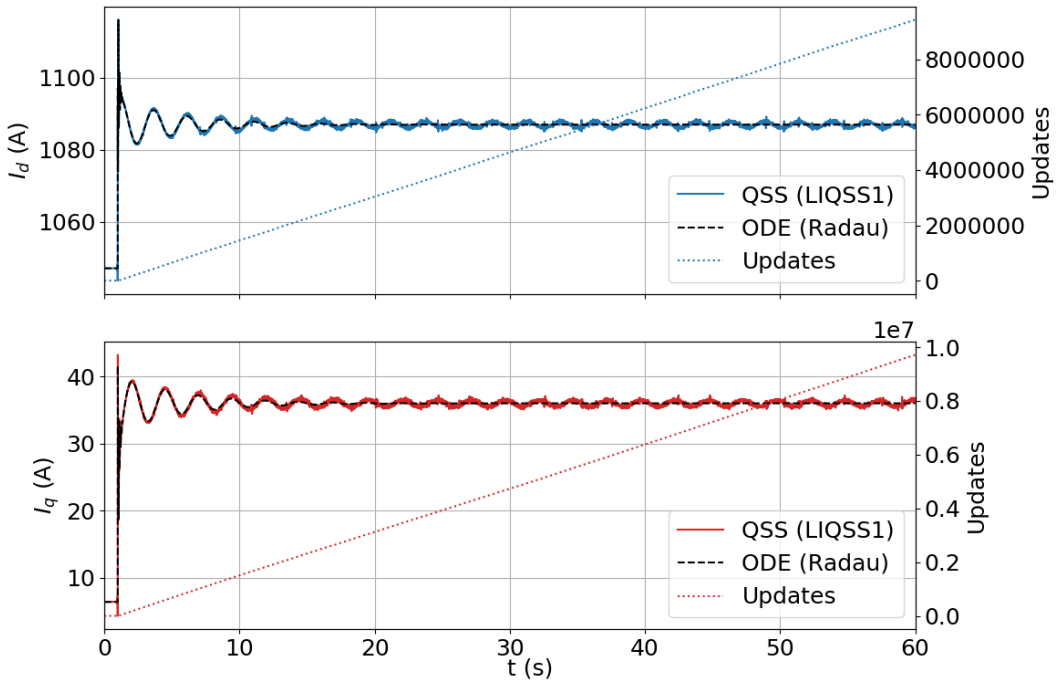



Figure 8.19 Cable currents from bus 2 to bus 3 q-axis (filtered, $f_c = 100Hz$).

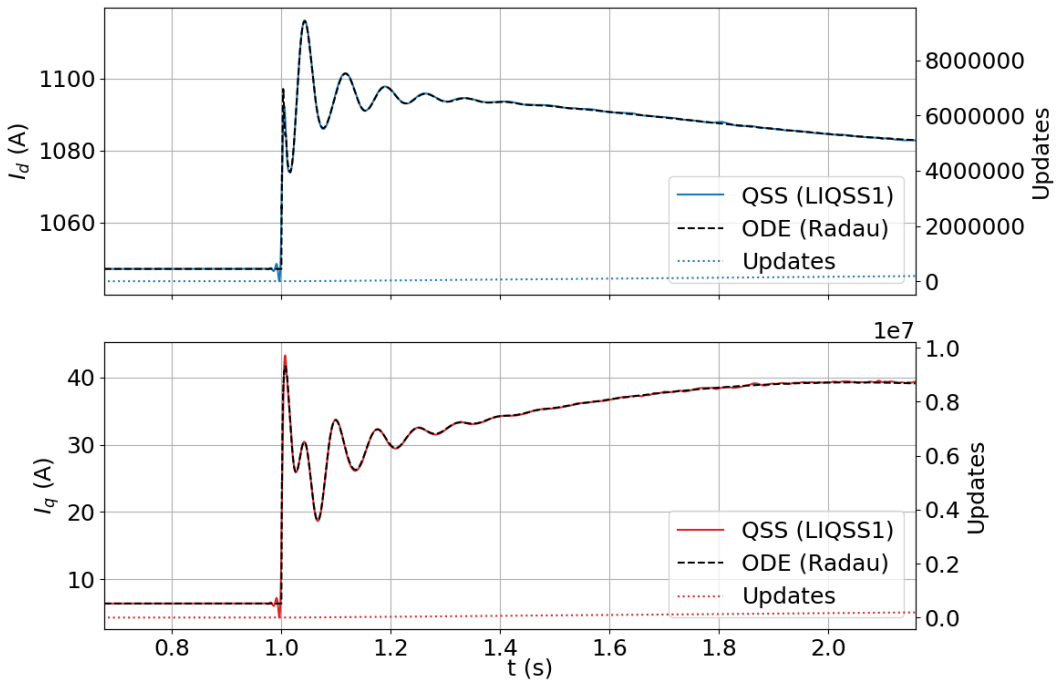



Figure 8.20 Cable currents from bus 2 to bus 3 q-axis, zoom to transient (filtered, $f_c = 100Hz$).

## 8.11 Error Analysis

Error was quantified using the normalized RMS deviation calculation defined as

$$\text{NRMSD} = \frac{\sqrt{\frac{\sum_{i=1}^{N}(y_i - q_i)^2}{N}}}{\max y - \min y}. \tag{8.30}$$

where $y_i$ is the benchmark (ODE) solution result for the state variable at point i, $q_i$ is the quantized QDL solution result for the state variable at point i, $N$ is the number of solution points, and the quantity $\max y - \min y$ is the dynamic range of the variable in the benchmark simulation.

This method of producing aggregated error values was chosen in order to provide error results that could be meaningfully compared across the states of the simulation, given the vastly different offsets and dynamic ranges between the states. The errors observed for the load increase simulation are shown in the sorted bar chart in figure 8.21.

The q-axis current on the cable from bus 2 to 3 is the state with the highest aggregated error in the simulation. This is not surprising given the level of high-amplitude noise seen in the simulation output for this variable. See chapter 10 for further discussion on post-process filtering of these data to improve readability of results, and to reduce error.

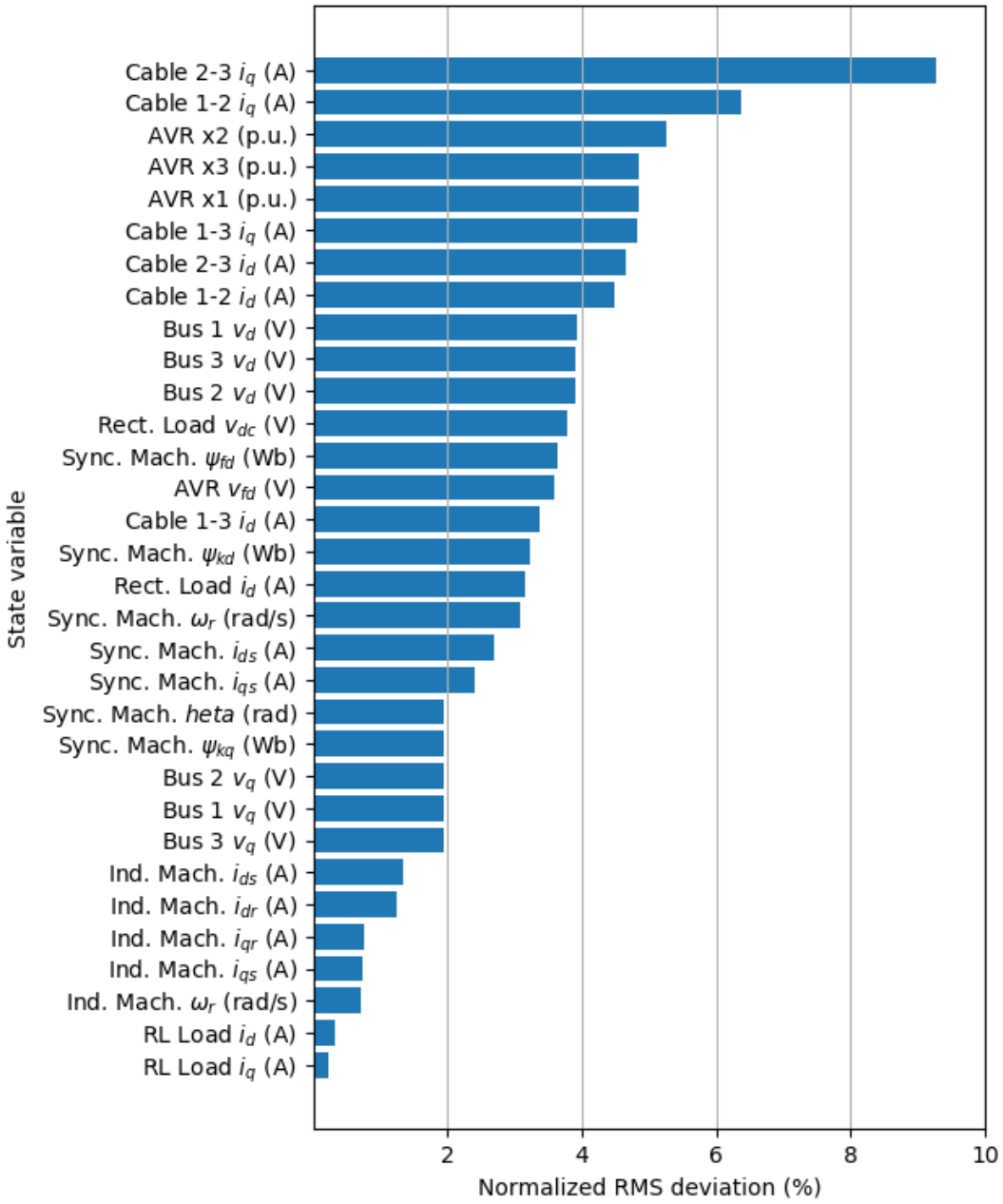



Figure 8.21 Simulation NRMSD error by state variable

## 8.12 Performance Analysis

To analyze the performance of the QDL method for this system, a proxy for computation intensity was used to estimate the relative performance of the QDL method against the benchmark ODE solution. This proxy is the cumulative number of updates for each atom. Although not directly comparable to time-slicing method time step counts, it gives us a good approximation of the relative computational efficiency of the QDL method vs. the reference ODE solution. Note that, in order to produce a stable simulation, a time step of $10\mu s$ was used, which results in $6 \times 10^5$ time steps for a 60 second (simulation time) simulation.

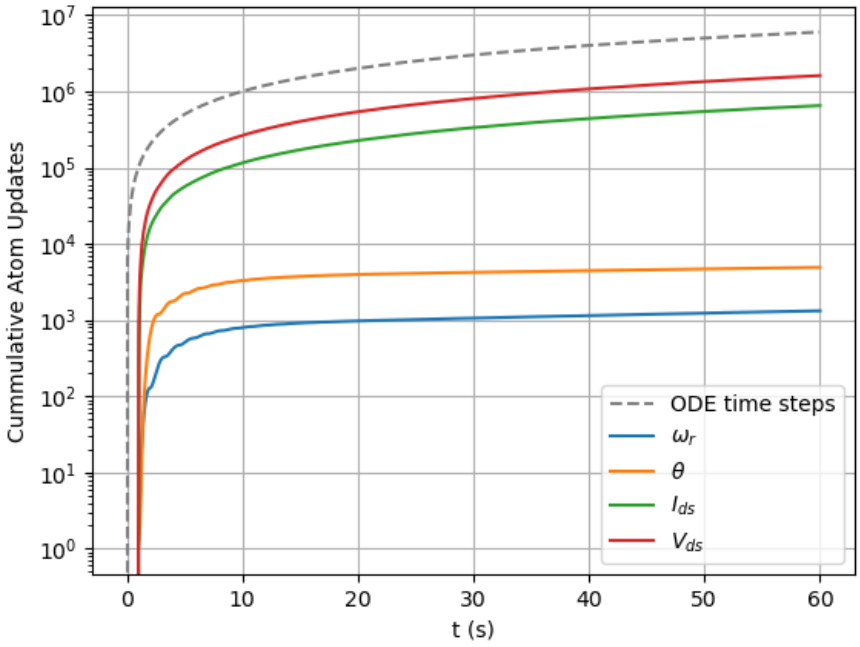



Figure 8.22 Cumulative QDL atom updates vs. simulation time, synchronous machine states

The key result shown in the plot in figures 8.22 and 8.23 is that the cumulative updates of the QDL atom states are less frequent than the required ODE benchmark solution time steps, in some cases by several orders of magnitude. Also, in the case of some states, such as the synchronous machine mechanical quantities (the rotor

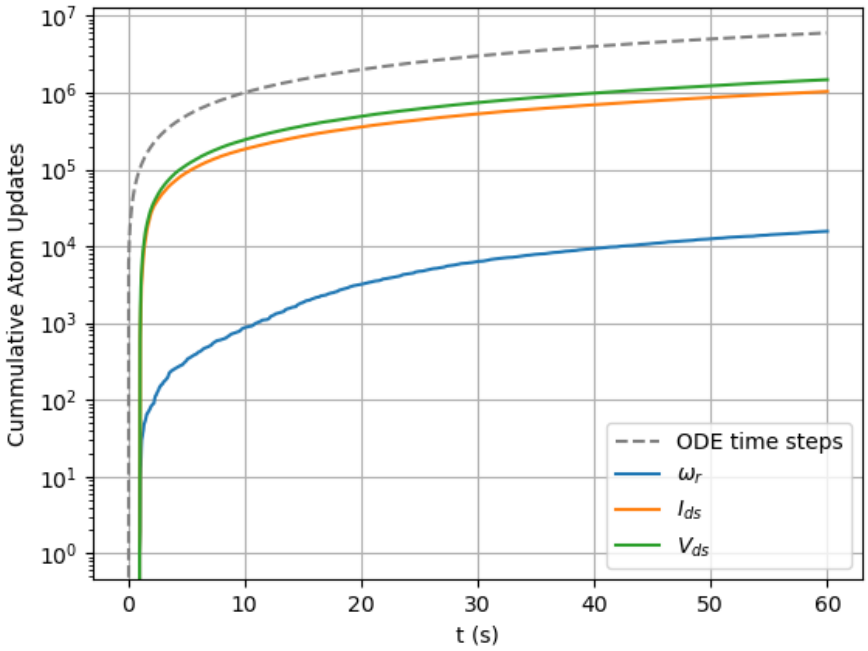



Figure 8.23 Cumulative QDL atom updates vs. simulation time, induction machine states

speed $\omega_r$ and rotor angle $\theta$), the trajectories of the cumulative curves become very flat, meaning there are very few updates needed for these states at steady-state.

Because the accuracy and computational intensity are related to the chosen quantization step size in QSS-based integration methods (a trade-off that does not exist with traditional methods), these metrics are difficult to compare directly. Also, different quantization step sizes were selected for each QDL atom state based on the dynamic range of each variable in the simulation scenarios. With the selected quantization step sizes, the simulation results are accurate to within a relative error of $\pm 5\%$ relative to the reference simulation (a 5th order Implicit Runge-Kutta algorithm with a fixed time step of $10\mu s$). Also, the proxy metrics for computational intensity (the number of state updates for the QSS solution, and the number of time steps for the reference solution) show that the theoretical performance of fully optimized code would result in faster performance for the QDL versus the reference simulation.

# Chapter 9

# Quantization Step Size Selection

The selection of a time step ($\Delta t$) for a time-slicing simulation method usually involves a straight-forward trade-off between accuracy and performance. For time-slicing, the smaller the $\Delta t$, the more accurate the results (with a lower bound on $\Delta t$ for numerical stability). One might assume that the same straight-forward trade-off exists between accuracy and performance when selecting the quantization step size ($\Delta Q$) for a QSS-based simulation. After all, there is a direct relationship between $\Delta Q$ and the amount of elapsed time before the next update will need to be updated. For a first-order QSS integrator of a single state variable, this is a simple relationship of $\Delta t = \Delta Q \cdot d$, where $d$ is the current derivative of the state. of However, although it is generally true that lower $\Delta Q$ tends to result in more computational load and higher simulation accuracy, the relationship is complicated.

To illustrate this, the performance of the synchronous machine simulation from chapter 7 is repeated here in figure 9.1, and shows the error and computational load of the synchronous machine simulation as a function of $\Delta Q$. In this simulation, all flux and speed states of the machine have the same $\Delta Q$ for each simulation run, with the exception that the machine rotor angle is scaled to be $1/10^{th}$ of the $\Delta Q$ of the the other states to account for the finer resolution needed due to a significantly lower dynamic range of this state. The error quantity is the largest normalized time-averaged error relative to reference ODE simulation (the maximum normalized root mean squared deviation, or NRMSD, among all system states). Although the error generally increases with an increasing quantization step size (as expected), the

relationship is non-linear, sporadic at higher $\Delta Q$ values, and shows significant diminishing returns at lower $\Delta Q$ values. These results, and those from other simulations performed in this document, show that the selection of optimal (or even *suitable*) $\Delta Q$ values for a simulation is not a straight-forward process.

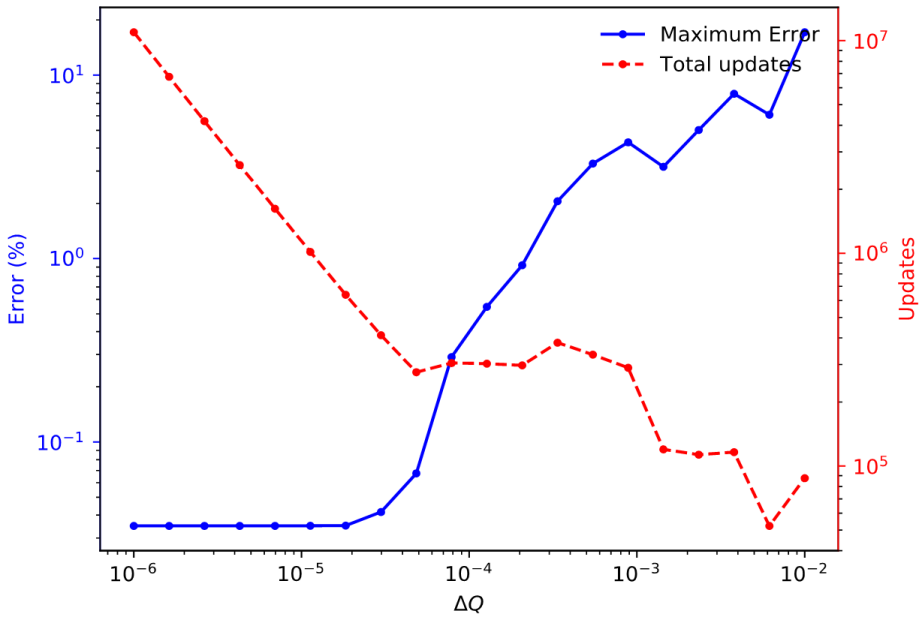



Figure 9.1 Maximum Error (among all system states) and QDL updates for various base values of $\Delta Q$ from the synchronous machine system simulation from chapter 7.

The selection of $\Delta Q$ is also complicated by the fact that each QDL atom in general needs its own $\Delta Q$ value for optimal simulation performance and suitability of the results. This can be clearly understood by considering the case of a system that contains a per unit PID controller model, along with a Megawatt level electrical load. The controller states may require quantization steps on the order of $10^{-6}$, while the load current states may be quantized with a step on the order of $10^{-1}$.

Ideally, the engineer should provide the desired acceptable error levels for the outputs of interest in a simulation, and the optimal $\Delta Q$ values for each atom are then automatically determined in order to maximize the quantization step size given the acceptable error constraints, while minimizing the computational load for the

simulation. Again, for time-slicing simulation methods, this is typically as simple as setting the time step at large as possible given the stability constraints determined by an eigenvalue analysis, perhaps with a margin added to account for the effects of non-linearity. Several methods for approaching an optimal selection of $\Delta Q$ for a QSS-based methods were attempted.

The initial approach to providing suitable $\Delta Q$ values was to assume that the error of each state is bounded by exactly $+/-\ \Delta Q$, and choose the $\Delta Q$ values accordingly. This has the advantage of being very simple, but the core assumption proved to be very incorrect. Because of the propagation of error that occurs between coupled states, the error levels at any given state can be larger than $\pm\Delta Q$ in general. The next attempt involved setting the $\Delta Q$ based on the dynamic range of each state. For example, each state's $\Delta Q$ can be set equal to 1% of the known dynamic range of the state variable as determined by a reference traditional ODE simulation result. This provided a good way to test the QDL method, but it has the obvious problem of requiring a solved reference simulation before the QDL simulation can be performed. This is therefore not suitable as a general method for selecting $\Delta Q$. Finally, a method that exploits the signal propagation behavior of the QDL method was developed. This method is described below.

## 9.1 The Error Propagation Method for $\Delta Q$ Selection

If the Jacobian matrix of he dynamic system can be determined at a given operating point, the linear sensitivities between external state transitions and the internal state of each atom is known. The amount of change possible during a given atom's external state transition can therefore be limited to arbitrary bounds by setting the quantization step size of all connected atoms based on their respective sensitivities. Stated another way: each atom's error is controlled by setting its connected neighbor atoms' quantization steps appropriately. This method uses common uncertainty calculations

to attempt to maximize each state's $\Delta Q$ by interpreting uncertainty as a proxy for desired error bounds.

Given the input variances for a state vector $\sigma^{\mathbf{x}}$, the output variances $\sigma^{\mathbf{y}}$ are found using the additive propagation rule

$$\sigma_i^y = \sqrt{\sum_{j=1}^{N} \left(\frac{\partial f_i}{\partial x_j}\right)^2 (\sigma_j^x)^2} \tag{9.1}$$

where $(\partial f_i/\partial x_j)$ is the element of the Jacobian between the $i^{th}$ and $j^{th}$ atoms. We then replace the variance vectors $\sigma^{\mathbf{x}}$ and $\sigma^{\mathbf{y}}$ with the quantization step vectors $\Delta Q^y$ and $\Delta Q^x$ as

$$\Delta Q_i^y = \sqrt{\sum_{j=1}^{N} \left(\frac{\partial f_i}{\partial x_j}\right)^2 (\Delta Q_j^x)^2} \tag{9.2}$$

where $\Delta Q_i^y$ is the output quantization step for the $i^{th}$ atom, and $Q_j^x$ is the input quantization step for the $j^{th}$ atom. For our purposes, the output quantization step vector can be interpreted as the desired maximum allowed error in the quantized output (this is the error bound vector), and the input quantization step vector will be calculated in order to honor the desired maximum error (this is the QDL $\Delta Q$ vector). Therefore, the QDL $\Delta Q$ vector can be found by solving for $Q_j^x$. We define the squared quantization step vectors as

$$\mathbf{s}^{\mathbf{y}} = \begin{bmatrix} (\Delta Q_1^y)^2 & (\Delta Q_2^y)^2 & \dots & (\Delta Q_N^y)^2 \end{bmatrix}^\top \tag{9.3}$$

$$\mathbf{s}^{\mathbf{x}} = \begin{bmatrix} (\Delta Q_1^x)^2 & (\Delta Q_2^x)^2 & \dots & (\Delta Q_N^x)^2 \end{bmatrix}^\top \tag{9.4}$$

and rewrite (9.2) in matrix form

$$\mathbf{s}^{\mathbf{y}} = \left(\mathbf{J}^\top \mathbf{J}\right) \mathbf{s}^{\mathbf{x}} \tag{9.5}$$

and rearrange to solve for $\mathbf{s}^{\mathbf{x}}$

$$\mathbf{s^x} = \left(\mathbf{J}^\top \mathbf{J}\right)^{-1} \mathbf{s^y} \tag{9.6}$$

where $\mathbf{J}$ is the full Jacobian matrix of the system. The quantization steps used in the simulator are found as the square root of the elements of $\mathbf{s^x}$

$$\mathbf{\Delta Q} = \left[\sqrt{|s_1^x|} \quad \sqrt{|s_2^x|} \quad \dots \quad \sqrt{|s_N^x|}\right]^\top \tag{9.7}$$

For the linear, time-invariant case, the Jacobian is equal to the state matrix, and does not change throughout the simulation. For the time-varying, non-linear case, however, the Jacobian is a function of time and the current state vector, and using the Jacobian to determine the error propagation is only an approximation. The approximation of $\Delta Q$ can be improved by calculating the Jacobian at the initial and final states when a system perturbation event occurs. The minimum values of $\Delta Q$ can then be used from solving (9.6) for both $\mathbf{J}_0$ and $\mathbf{J}_\infty$.

## 9.2 Error Control Example

An interesting test case is a second-order, non-linear pendulum. It is small enough to keep the analysis simple, but has some features of the non-linear dynamics of complex power systems. For this example, the choice of $\Delta Q$ is intentionally chosen to be large relative to the amplitude of the transients of the simulation so the effects of the error method can be seen clearly.

The dynamics of a damped pendulum are described by the second order system

$$\frac{d^2}{dt^2}\theta = -r\frac{d}{dt}\theta - \frac{g}{l}\sin\theta. \tag{9.8}$$

Which can be rewritten as two first-order equations as

$$\frac{d}{dt}\omega = -r\omega - \frac{g}{l}\sin\theta, \tag{9.9}$$

$$\frac{d}{dt}\theta = \omega. \tag{9.10}$$

Linearizing about the operating point $(\omega_0, \theta_0)$, the system can be rewritten in the linear form

$$\frac{d}{dt}\mathbf{x} = \mathbf{J} \cdot \mathbf{x} \tag{9.11}$$

where $\mathbf{x}$ is the state vector $\begin{bmatrix}\omega & \theta\end{bmatrix}^\top$ and $\mathbf{J}$ is the Jacobian matrix

$$\begin{bmatrix} -r & -\frac{g}{l}\cos\theta \\ 1 & 0 \end{bmatrix}. \tag{9.12}$$

The desired error for the QDL simulation is

$$\mathbf{\Delta Q^y} = \begin{bmatrix}0.2\,\text{rad/s} & 0.4\,\text{rad}\end{bmatrix}^\top \tag{9.13}$$

Using $\Delta Q^y$ directly as the $\Delta Q$ vector for the QDL simulation results in a $\theta$ trajectory that deviates significantly outside of the desired error bounds, while the trajectory of $\omega$ remains within bounds at steady state, while deviating only slightly outside of the bounds during the transient figure 9.2.

To apply the selection process from (9.6), we need to find two Jacobian matrices, one for the initial state of $\begin{bmatrix}1\,\text{rad/s} & 1\,\text{rad}\end{bmatrix}$, which is

$$\mathbf{J}_0 = \begin{bmatrix} -0.4 & -0.6626 \\ 1 & 0 \end{bmatrix} \tag{9.14}$$

,

and another for the final equilibrium state of the $\begin{bmatrix}0\,\text{rad/s} & 0\,\text{rad}\end{bmatrix}$

$$\mathbf{J}_\infty = \begin{bmatrix} -0.4 & -1.2263 \\ 1 & 0 \end{bmatrix} \tag{9.15}$$

.

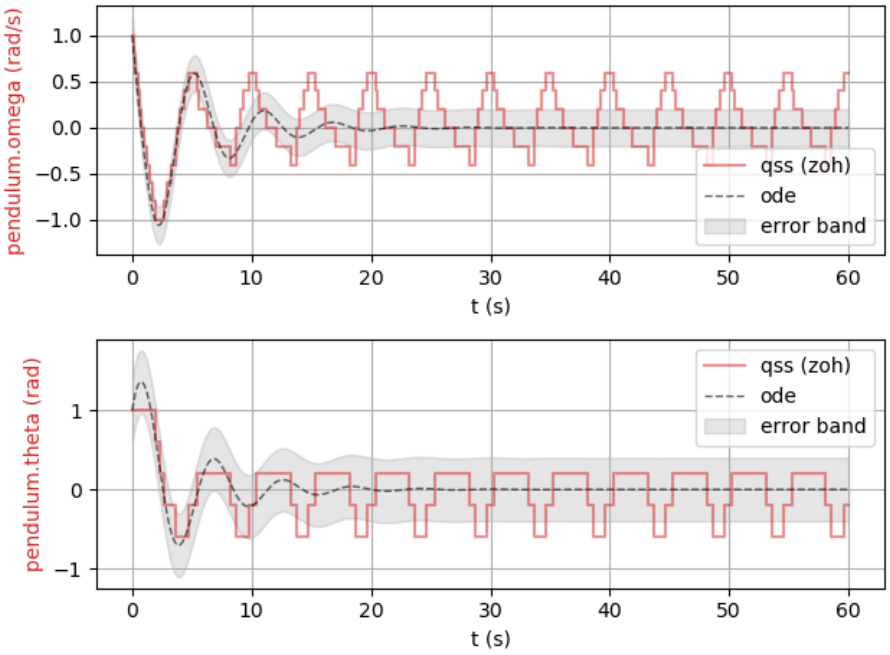



Figure 9.2 QDL pendulum simulation results with $\Delta Q_\omega = 0.2$ and $\Delta Q_\theta = 0.4$

Note that the equilibrium state of the damped, non-forced pendulum system is defined as $\begin{bmatrix} 0\,\text{rad/s} & 0\,\text{rad} \end{bmatrix}$. In general, finding $\mathbf{J}_\infty$ at the final equilibrium point post-disturbance requires solving the non-linear system using a Newton-Raphson solution or some other iterative approach.

From $\mathbf{J}_0$ and $\mathbf{J}_\infty$ we use (9.6) to obtain the two $\Delta Q^x$ vectors

$$\mathbf{\Delta Q_0^x} = \begin{bmatrix} 0.4\,\text{rad/s} & 0.1811\,\text{rad} \end{bmatrix}^\top \tag{9.16}$$

$$\mathbf{\Delta Q_\infty^x} = \begin{bmatrix} 0.4\,\text{rad/s} & 0.0979\,\text{rad} \end{bmatrix}^\top \tag{9.17}$$

.

Taking the minimum values from the elements of these two candidate quantization step vectors, the final $\mathbf{\Delta Q}$ vector used for the QDL simulation is

$$\mathbf{\Delta Q} = \begin{bmatrix} 0.4\,\text{rad/s} & 0.0979\,\text{rad} \end{bmatrix}^{\top} \tag{9.18}$$

An important feature of this method is that the resultant $\mathbf{\Delta Q}$ vector can have elements larger than the error vector $\mathbf{\Delta Q^y}$.

The simulation results with the new $\mathbf{\Delta Q}$ is shown in figure 9.2, where the amplitude of the steady-state oscillations remain bounded within the desired error band (denoted with the gray filled area around the ODE curve).

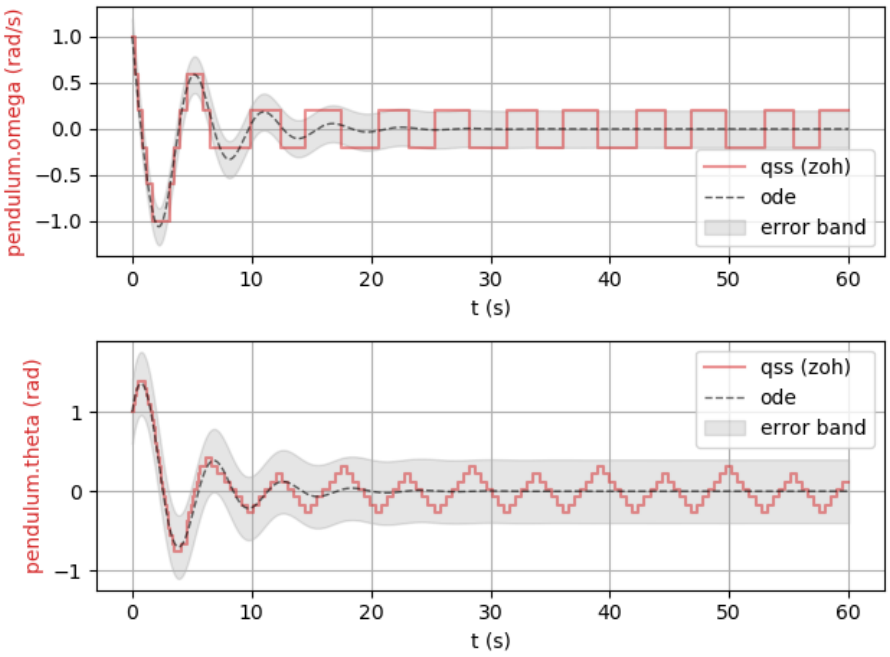



Figure 9.3 QDL pendulum simulation results with $\Delta Q_\omega = 0.05$ and $\Delta Q_\theta = 0.1$

Note that, although the external state trajectories of both system states plotted in figure 9.2 remain bounded within the desired error band, there is still significant movement about an average values in the nominally steady-state portion of the simulation. Because the error bounds were selected to be large for this simulation to amplify the performance of the $\Delta Q$ selection method, the oscillations typical of the QDL steady-state results are easy seen. This behavior often takes the form of an apparent limit

cycle, sometimes with a clear period. The hypotheses is that this perpetuated limit cycle behavior is an artifact of the QDL method caused by fictitious energy injection arising from the difference between the internal state values and their propagated quantized states. This phenomenon, along some potential mitigation methods, are explored in the following chapter.

# Chapter 10

# Steady-state Behavior Improvements

Two areas of improvement are investigated in this chapter related to the problematic steady-state behavior of the QDL method: detecting steady-state detection (with run-time correction), and post-simulation filtering for steady-state noise and error reduction.

## 10.1 Steady-state Detection and Correction

As seen in the example in previous chapters, many system simulations using QDL do not drive the states to a true steady-state condition with all state derivatives at zero. In fact, there is often a consistent quantized limit cycle phenomenon present. Although this phenomenon is not fully understood, if the steady-state condition can be detected, the derivatives can be forced to zero. Once the derivatives are zero, the QSS events of the integrators are all scheduled for $t = \infty$, and the simulation can run indefinitely with *zero* computational load until the next scheduled discrete event.

This steady-state detection and correction scheme consists of monitoring the proximity of the state to the next equilibrium, defining a region about this equilibrium as an *effective* steady-state condition, applying a time delay, and finally forcing the derivatives to zero. The next equilibrium is found immediately after applying a disturbance and is calculated using the dc operating point solution. The state vector for the next equilibrium state is stored and used to check the proximity at regular intervals during the transient simulation. The steady state distance is defined using the l-2 norm as

$$\|\mathbf{q_e}\|_2 < K_m \|\mathbf{Q}\|_2 \tag{10.1}$$

where $\mathbf{\Delta Q}$ is the vector of quantization step sizes for all simulation state atoms, $K_m$ is a multiplier to define the size of the effective steady-state region, and $\|\mathbf{q_e}\|_2$ is the distance of the quantized state vector $q$ from the known equilibrium state vector $\tilde{\mathbf{x}}$ defined as

$$\| (\mathbf{q} - \tilde{\mathbf{x}}) \|_2 = \sum_{j=1}^{N} (q_i - \tilde{x}_i)^2 \tag{10.2}$$

where $\mathbf{q}$ is the vector of quantized states at the current simulation time, and $\tilde{\mathbf{x}}$ is the next equilibrium point vector determined from the operating point solution.

The results of this steady-state detection and correction scheme are applied to the pendulum system in figure 10.2 (no detection) and figure 10.3 (with detection). The parameters of the detector are a time delay of 5 seconds, and a region magnitude multiplier of $K_m$ value of 2. Figure 10.3 visualizes this scheme with the phase plot of the angle versus velocity. The trajectory of the curve can be seen entering the steady-state region, continuing for some time, and then being forced to the equilibrium point in the center. This visualization method is not possible for higher-order systems.

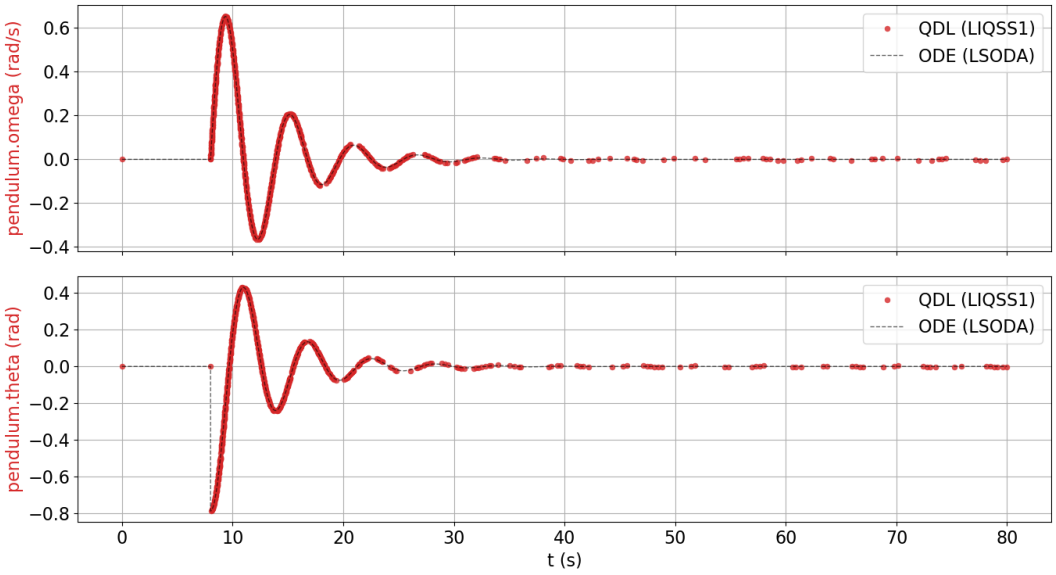



Figure 10.1 Pendulum simulation without steady-state detection

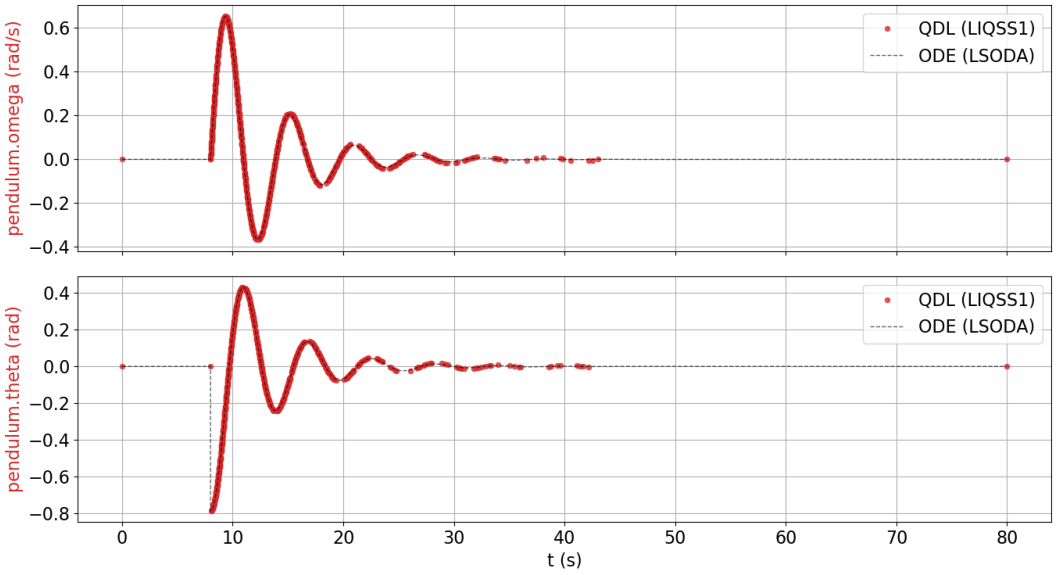



Figure 10.2 Pendulum simulation with steady-state detection

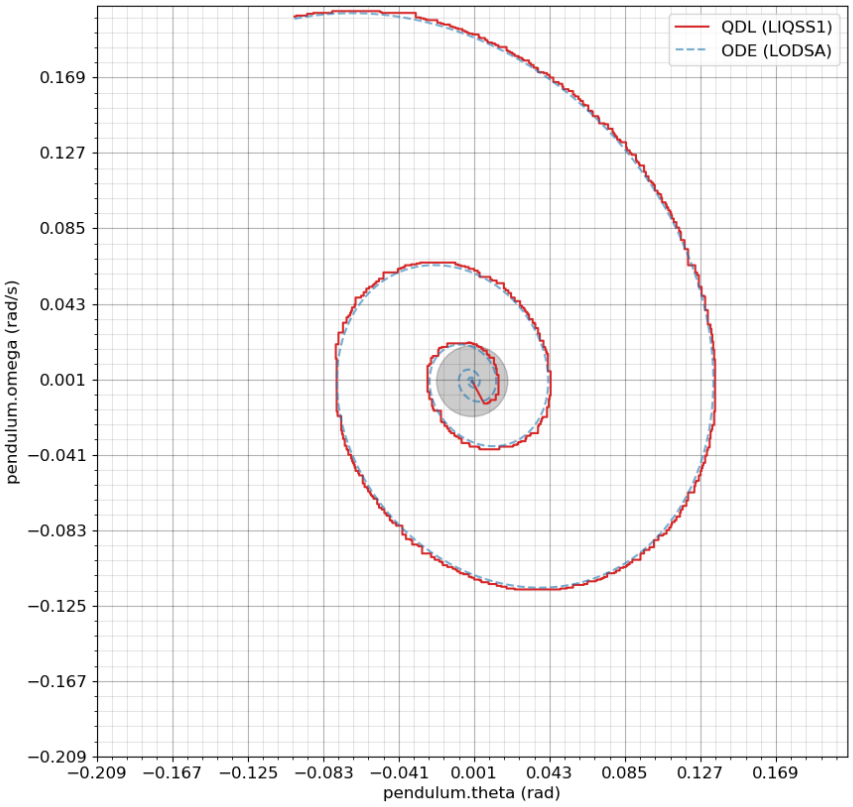



Figure 10.3 Pendulum phase diagram showing steady-state detection region

## 10.2 Post-simulation Filtering

When using the QDL method for larger, non-linear, very stiff systems, the amplitude of numerical noise in some output signals can obfuscate the results.

The worst case for this was found in the study system in chapter 8, specifically in the quadrature-axis current of the cable from bus 2 to 3. The frequency band of excessive noise is much higher than the dominant frequency content of the transient behavior. A filter was applied to the simulation output post-simulation. The filter used is a $6^{th}$ order discrete butterworth low-pass filter with a cutoff frequency of 100 Hz. The filter was applied in both the forward and backwards directions to cancel the phase shift in order to keep the transient response of interest intact as much as possible. The time series plots showing the results of this filtering are shown in figures 10.4, 10.5, and 10.6. Filtering the results reduces the NRMSD error for the filtered state from 9.30% to 1.02% (or a *97.9%* reduction in the error magnitude), bringing the error into a reasonable range of simulation accuracy. The initial transient response tracks the benchmark simulation very well, although there is a filtering artifact at the step change in load impedance. It is expected that filtering can be applied to the other states with similar results.

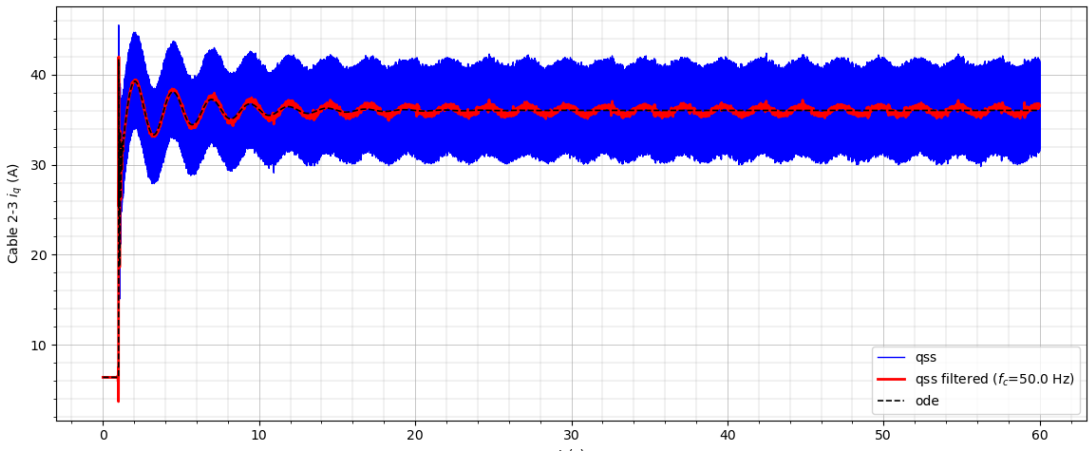



Figure 10.4 Cable 2-3 q-axis current, load increase scenario, full simulation

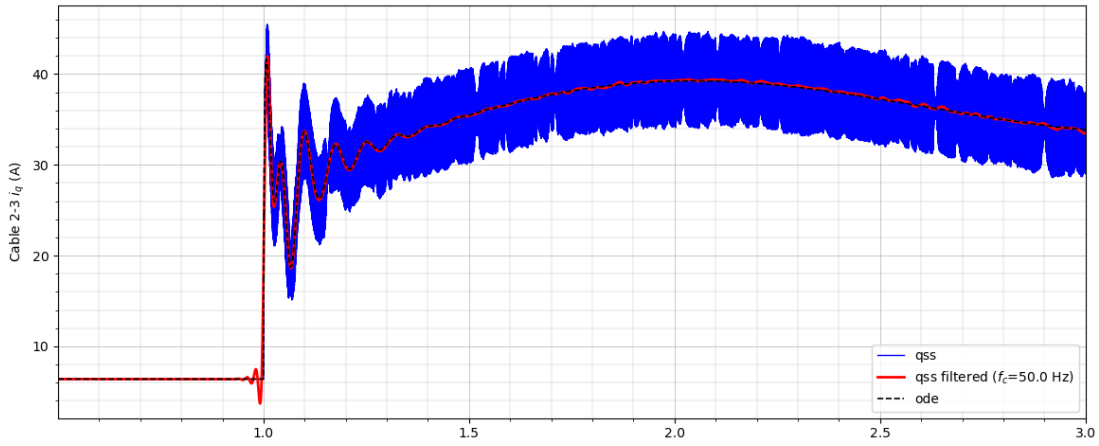



Figure 10.5 Cable 2-3 q-axis current, load increase scenario, zoom to initial transient

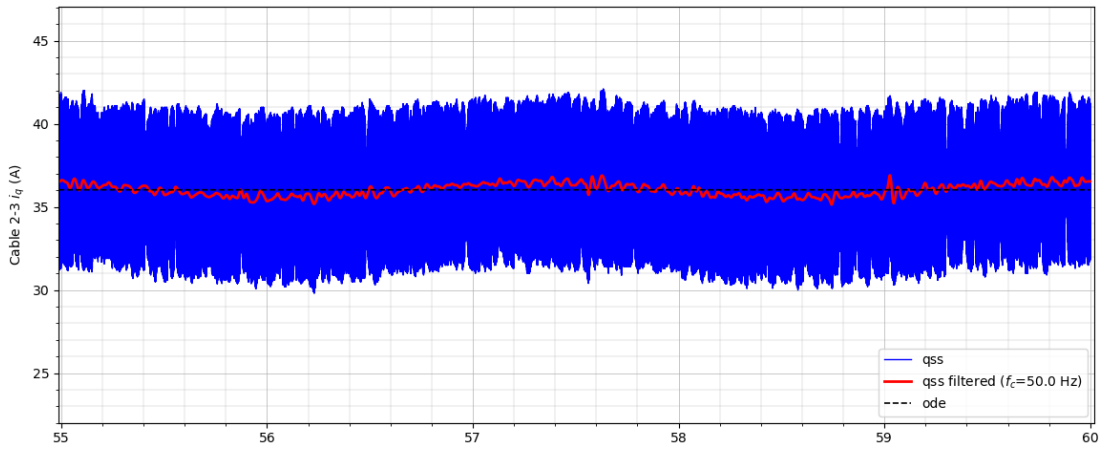



Figure 10.6 Cable 2-3 q-axis current, load increase scenario, zoom to steady-state

An issue with this filtering method is the requirement to know the frequency content of interest before tuning the filters. Because we have a benchmark solution, the useful frequency band can be determined by inspection, and the accuracy of the filtered results can be easily verified. In general, however, this benchmark solution is not available. One approach to finding an appropriate pass band for the filter is via an eigenvalue analysis of the system to determine the dominant modes of interest. This is not investigated further here and is left as potential future work.

# Chapter 11

# Conclusions and Future Work

## 11.1 Feasibility and Benefits of the QDL Method

The stated goal of this work is to demonstrate that the proposed QDL method for transient simulation of electrical power systems is both feasible and advantageous over the current state-of-the-art time-slicing numerical integration methods.

As far as feasibility, the results produced by QDL match the benchmark solution reasonably well, and the performance (computational efficiency) is comparable or faster than time-slicing methods. Several simulation systems and many device models have been created and tested using the method, and all were solvable using the method. The test systems include many smaller examples that highlight the particular features of QDL. Ultimately, in order to demonstrate suitability for real-world power system models, the final test in this work is a relatively large (32 state), non-linear power system model with machines, converters, and other common power system devices. From these tests, it is concluded that the QDL method is at least a feasible solution to the transient simulation of large, real-world power systems.

## 11.2 Opportunities for Future Research

The steady-state behavior of the QDL simulator when simulating large, non-linear systems is problematic. There is significant high frequency noise in the quantized output quantities, particularly affecting the fast electrical quantities (bus voltages and cable currents). There are also lower frequency oscillations in the signal that corre-

spond roughly to the natural mechanical oscillation modes of the system dominated by the machine inertia. It is expected that the noise and oscillations are artifacts from the time delays introduced to the dynamical system from the QDL method. The oscillation and noise do not appear to be damped, and therefore appear as limit cycles in the output. Possible methods of mitigating this behavior are better tuning of the quantization step size, and adding additional inserted latency when the system is detected to be close to steady-state. These mitigation possibilities were investigated in chapter 10, and proved modestly successful, at least for smaller test systems. Future work should include scaling these proposed solutions, and investigating other possible solutions to the noise and error propagation problem.

In addition to the steady-state issues, problems were also found running transient fault analysis. Typical power system studies include the analysis of faults. Transient fault analysis is important for testing the robustness of the power system and its protection system. For the QDL approach to be a viable alternative to other methods, the ability to properly simulate faults is important. However, performance and technology issues prevented fault scenarios from being included in the paper. QSS integration methods in their current form have a specific disadvantage in simulating faults. Because fault scenarios involve large, abrupt changes to system quantities (a voltage quantity quickly moving from its rated value to near zero, for example), the system states move very quickly through many quantization steps, and these quantization changes cascade widely to the rest of the system causing extremely high computational intensity during the transients of the fault scenarios. Strategies for overcoming or compensating for these issues are required that are beyond the scope of this initial feasibility evaluation. These strategies could include code optimization, memory management, or even modifications to the core QSS algorithms to mitigate the problems. It is not expected that the latest *modified* LIQSS methods (such as mLIQSS1 and mLIQSS2) will have better performance with fault scenarios on power

systems models. These updated methods address different problems than those posed by transient fault analysis, and the additional computation steps they require per update are likely to exacerbate this problem. Future work should include upgrading the QDL simulator implementations with the latest LIQSS methods from the most recent literature.

# Bibliography


[1] J. E. Schutt-Aine. Latency insertion method (lim) for the fast transient simulation of large networks. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 48(1):81–89, Jan 2001.

[2] Bernard P. Zeigler. Devs theory of quantized systems, 1999.

[3] François E. Cellier, Ernesto Kofman, Gustavo Migoni, and Mario Bortolotto. Quantized state system simulation, 2008.

[4] Ernesto Kofman and Sergio Junco. Quantized-state systems: A devs approach for continuous system simulation. *Trans. Soc. Comput. Simul. Int.*, 18:123–132, 2001.

[5] Toshimori Sekine and H. Asai. Block-latency insertion method (block-lim) for fast transient simulation of tightly coupled transmission lines. *2009 IEEE International Symposium on Electromagnetic Compatibility*, pages 253–257, 2009.

[6] Pollyanna Goh, J. E. Schutt-Aine, D. Klokotov, Jilin Tan, Ping Liu, Wenliang Dai, and Feras Al-Hawari. Partitioned latency insertion method with a generalized stability criteria. *IEEE Transactions on Components, Packaging and Manufacturing Technology*, 1:1447–1455, 2011.

[7] Bernard P Zeigler, Alexandre Muzy, and Ernesto Kofman. *Theory of modeling and simulation*. Academic Press, San Diego, CA, 3 edition, August 2018.

[8] Ernesto Kofman. Quantized-state control: A method for discrete event control of continuous systems. Technical report, Laboratorio de Sistemas Din´amicos, FCEIA - UNR. CONICET, 2001.

[9] F E Cellier and Håkan Elmqvist. Automated formula manipulation supports object-oriented continuous-system modeling. *IEEE Control Systems*, 13:28–38, 1993.

[10] Ernesto Kofman. A second-order approximation for DEVS simulation of continuous systems. *Simulation*, 78(2):76–89, February 2002.

[11] Ernesto Kofman. A third order discrete event simulation method for continuous system simulation. *Latin American Applied Research*, 36, 2006.

[12] Gustavo Migoni and Ernesto Kofman. Linearly implicit discrete event methods for stiff odes. part i: Theory. *Latin American applied research*, 39, 07 2009.

[13] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation*. Academic Press, 2000.

[14] Gustavo Migoni, Mario Bortolotto, Ernesto Kofman, and François E. Cellier. Linearly implicit quantization-based integration methods for stiff ordinary differential equations. *Simulation Modelling Practice and Theory*, 35:118–136, 2013.

[15] Ernesto Kofman. Discrete event simulation of hybrid systems. *SIAM J. Scientific Computing*, 25:1771–1797, 2004.

[16] Franco Di Pietro, Gustavo Migoni, and Ernesto Kofman. Improving linearly implicit quantized state system methods. *Simulation*, page 003754971876668, May 2018.

[17] Kumarraja Andanapalli and B. R. K. Varma. Park's transformation based symmetrical fault detection during power swing. In *2014 Eighteenth National Power Systems Conference (NPSC)*, pages 1–5, 2014.

[18] IEEE. IEEE guide for synchronous generator modeling practices and applications in power system stability analyses. Technical report, IEEE, Piscataway, NJ, USA, 2008.

[19] Navid Gholizadeh, Joseph M Hood, and Roger A Dougal. Evaluation of linear implicit quantized state system method for analyzing mission performance of power systems. *The Journal of Defense Modeling and Simulation*, 0(0):15485129211061702, 0.

[20] S. Abdelwahed, A. Asrari, J. Crider, R.A. Dougal, M.O. Faruque, Y. Fu, J. Langston, Y. Lee, H.A. Mohammadpour, A. Ouroua, E. Santi, K. Shoder, S.D. Sudhoff, Y. Zhang, H. Zheng, and E. Zivi. Reduced order modeling of a shipboard power system. In *2013 IEEE Electric Ship Technologies Symposium (ESTS)*, pages 256–263, 2013.

# Appendix A

# MATLAB Source Code

## A.1 MATLAB QDL Simulator Source

```
1
2  classdef QdlSystem < handle
3
4
5      properties (Constant)
6
7          StimNone    = 0;
8          StimSource  = 1;
9          StimSignal  = 2;
10
11         SourceNone  = 0;
12         SourceDC    = 1;
13         SourceSINE  = 2;
14         SourcePWM   = 3;
15
16         SignalNone  = 0;
17         SignalBGain = 1;
18         SignalSGain = 2;
19         SignalTGain = 3;
20         SignalZGain = 4;
21
22         DefaultNpoints = 1e6;
23         DefaultDtmin = 1e-12;
24
25     end
26
27     properties
28
29         % scalar variables:
```

```
30
31          nnode       % number of nodes
32          nbranch     % number of branchest
33          nstim       % number of stimulus devices
34          n           % nnode + nbranch (total number of lim atoms)
35          ndevice     % nnode + nbranch + nstim (total number of qdl devices)
36          npt         % size of output vectors
37          dtmin       % minimum time step
38          time        % current simualtion time
39          tstop       % simulation stop time
40          def_dqmin   % default minimum dq
41          def_dqmax   % default maximum dq
42          def_dqerr   % default maximum error target
43
44          % devs atom objects:
45
46          nodes     % array of LIM nodes and ideal voltage sources (nnode)
47          branches % array of LIM branches and ideal current sources (nbranch)
48          stims     % array of stimuli (nstim)
49
50          % liqss variables
51
52          dq      % quantum level (n)
53          dqmin   % min quantum level (n)
54          dqmax   % max quantum level (n)
55          dqerr   % nominal allowed error (n)
56          qlo     % lower boundary of quantized state (n)
57          qhi     % upper boundary of quantized state (n)
58          tlast   % last internal transition time of state (n)
59          tnext   % next predicted internal transition (n)
60          x       % internal states (n)
61          x0      % initial internal states (n)
62          trig    % trigger flags for external transition (n)
63          iout    % output index for sparse output arrays (n)
64          d       % state derivatives (n, 2)
65          q       % quantized state (n, 2)
66          M       % maps each atom to all receiver atoms for update broadcasting (n,n)
67
68          % liqss output data arrays output :
69
70          qout % quantized states (n, npt)
71          tout % output time points (n, npt)
72          nupd % update count (n, npt)
73          tupd % update count times (n, npt)
```

```
        iupd % update count array size (n, npt)

        % enhanced LIM model:

        A  % incidence matrix between node i and branch k (nnode, nbranch)

        % data for latency nodes:
        Cinv % 1/capacitance at node i (nnode)
        G    % conductance at node i (nnode)
        H    % current injection at node i (nnode)
        B    % vccs gain matrix between node i and node p (nnode, nnode)
        S    % cccs gain matrix between node i and branch k (nnode, nbranch)

        % data for latency branches:
        Linv % 1/inductance in branch k (nbranch)
        R    % series resistance in branch k (nbranch)
        E    % voltage at branch k from node i to node j (nbranch)
        T    % vcvs gain matrix between branch k and node i (nbranch, nnode)
        Z    % ccvs gain matrix between branch k and branch q (nbranch, nbranch)

        % stimulus params
        stim_type   % value from above stim type enum values (n)
        source_type % value from above source type enum values (n)
        signal_type % value from above signal type enum values (n)
        xdc         % offset value (n)
        xa          % sine amplitude (n)
        x1          % pwm first value (n)
        x2          % pwm second value (n)
        freq        % sine wave freqnecy or pwm switching freqency (Hz) (n)
        duty        % pwm duty cycle (pu) (n)
        phi         % sine wave phase (rad) (n)
        period      % cached 1/f (s) (n)

        % state space
        dtss % state space timestep
        tss  % state space current time
        xss  % state space state vector (for stepping method)
        Ass  % state space A matrix
        Bss  % state space B matix
        Uss  % state space U vector
        Apr  % state space descretixed A' matrix (backwards Euler)
        Bpr  % state space descretixed B' matrix (backwards Euler)

        timer
```

```
118
119         iupdates
120
121     end
122
123     methods
124
125         function self = QdlSystem(dqmin, dqmax, dqerr)
126
127             self.def_dqmin = dqmin;
128             self.def_dqmax = dqmax;
129             self.def_dqerr = dqerr;
130             self.npt      = self.DefaultNpoints;
131             self.nnode    = 0;
132             self.nbranch  = 0;
133             self.nodes    = QdlNode.empty(0);
134             self.branches = QdlBranch.empty(0);
135             self.dtmin    = self.DefaultDtmin;
136
137         end
138
139         function copyto(self, other)
140
141             % scalar variables:
142
143             other.nnode      = self.nnode;
144             other.nbranch    = self.nbranch;
145             other.n          = self.n;
146             other.npt        = self.npt;
147             other.dtmin      = self.dtmin;
148             other.time       = self.time;
149             other.tstop      = self.tstop;
150             other.def_dqmin  = self.def_dqmin;
151             other.def_dqmax  = self.def_dqmax;
152             other.def_dqerr  = self.def_dqerr;
153
154             % devs atom objects:
155
156             other.nodes    = self.nodes;
157             other.branches = self.branches;
158
159             % liqss variables
160
161             other.dq     = self.dq;
162             other.dqmin  = self.dqmin;
163             other.dqmax  = self.dqmax;
164             other.dqerr  = self.dqerr;
165             other.qlo    = self.qlo;
```

```
            other.qhi    = self.qhi;
            other.tlast  = self.tlast;
            other.tnext  = self.tnext;
            other.x      = self.x;
            other.x0     = self.x0;
            other.trig   = self.trig;
            other.iout   = self.iout;
            other.d      = self.d;
            other.q      = self.q;
            other.M      = self.M;

            % enhanced LIM model:

            other.A    = self.A;

            % data for latency nodes:
            other.Cinv = self.Cinv;
            other.G    = self.G;
            other.H    = self.H;
            other.B    = self.B;
            other.S    = self.S;

            % data for latency branches:
            other.Linv = self.Linv;
            other.R    = self.R;
            other.E    = self.E;
            other.T    = self.T;
            other.Z    = self.Z;

            % source stimulus params
            other.stim_type   = self.stim_type;
            other.source_type = self.source_type;
            other.signal_type = self.signal_type;
            other.xdc         = self.xdc;
            other.xa          = self.xa;
            other.x1          = self.x1;
            other.x2          = self.x2;
            other.freq        = self.freq;
            other.duty        = self.duty;
            other.phi         = self.phi;
            other.period      = self.period;

            % state space
            other.dtss = self.dtss;
            other.tss  = self.tss;
            other.xss  = self.xss;
            other.Ass  = self.Ass;
            other.Bss  = self.Bss;
```

```
214             other.Uss  = self.Uss;
215             other.Apr  = self.Apr;
216             other.Bpr  = self.Bpr;
217
218         end
219
220         function build_lim(self)
221
222             self.n = self.nnode + self.nbranch; % total number of atoms
223
224             self.A = zeros(self.nnode, self.nbranch); % connectivity matrix
225
226             self.M = zeros(self.n, self.n);  % trigger map
227
228             self.Cinv = zeros(self.nnode, 1);
229             self.G    = zeros(self.nnode, 1);
230             self.H    = zeros(self.nnode, 1);
231             self.B    = zeros(self.nnode, self.nnode);
232             self.S    = zeros(self.nnode, self.nbranch);
233
234             self.Linv = zeros(self.nbranch, 1);
235             self.R    = zeros(self.nbranch, 1);
236             self.E    = zeros(self.nbranch, 1);
237             self.T    = zeros(self.nbranch, self.nnode);
238             self.Z    = zeros(self.nbranch, self.nbranch);
239
240             self.x0    = zeros(self.n, 1);
241             self.dq    = zeros(self.n, 1);
242             self.dqmin = zeros(self.n, 1);
243             self.dqmax = zeros(self.n, 1);
244             self.dqerr = zeros(self.n, 1);
245
246             self.xdc = zeros(self.n, 1);
247             self.xa  = zeros(self.n, 1);
248             self.x1  = zeros(self.n, 1);
249             self.x2  = zeros(self.n, 1);
250
251             self.stim_type   = zeros(self.n, 1);
252             self.source_type = zeros(self.n, 1);
253             self.signal_type = zeros(self.n, 1);
254             self.freq        = zeros(self.n, 1);
255             self.period      = zeros(self.n, 1);
256             self.duty        = zeros(self.n, 1);
257             self.phi         = zeros(self.n, 1);
258
```

```
259             for inode = 1:self.nnode
260
261                 self.Cinv(inode) = 1 / self.nodes(inode).C;
262                 self.G(inode) = self.nodes(inode).G;
263                 self.H(inode) = self.nodes(inode).H;
264
265                 self.x0(inode) = self.nodes(inode).v0;
266                 self.dqmin(inode) = self.nodes(inode).dqmin;
267                 self.dqmax(inode) = self.nodes(inode).dqmax;
268                 self.dqerr(inode) = self.nodes(inode).dqerr;
269
270                 self.stim_type(inode)   = self.nodes(inode).stim_type;
271                 self.source_type(inode) = self.nodes(inode).source_type;
272                 self.signal_type(inode) = self.nodes(inode).signal_type;
273                 self.xdc(inode)         = self.nodes(inode).vdc;
274                 self.xa(inode)          = self.nodes(inode).va;
275                 self.x1(inode)          = self.nodes(inode).v1;
276                 self.x2(inode)          = self.nodes(inode).v2;
277                 self.freq(inode)        = self.nodes(inode).freq;
278                 self.period(inode)      = 1/self.nodes(inode).freq;
279                 self.duty(inode)        = self.nodes(inode).duty;
280                 self.phi(inode)         = self.nodes(inode).phi;
281
282                 % add entries to B matrix:
283                 for ibnodes = 1:self.nodes(inode).nbnode
284                     self.B(inode, self.nodes(inode).bnodes(ibnodes).index) ...
285                         = self.nodes(inode).B(ibnodes);
286                 end
287
288                 % add entries to S matrix:
289                 for isbranch = 1:self.nodes(inode).nsbranch
290                     self.S(inode, self.nodes(inode).sbranches(isbranch).bindex) ...
291                         = self.nodes(inode).S(isbranch);
292                 end
293
294             end
295
296             for ibranch = 1:self.nbranch
297
298                 iatom = self.nnode + ibranch;
299
300                 self.A(self.branches(ibranch).inode.index, ibranch) = 1;
301                 self.A(self.branches(ibranch).jnode.index, ibranch) = -1;
302
303                 self.Linv(ibranch) = 1 / self.branches(ibranch).L;
304                 self.R(ibranch) = self.branches(ibranch).R;
```

```
305                 self.E(ibranch) = self.branches(ibranch).E;
306
307                 self.x0(iatom) = self.branches(ibranch).i0;
308                 self.dqmin(iatom) = self.branches(ibranch).dqmin;
309                 self.dqmax(iatom) = self.branches(ibranch).dqmax;
310                 self.dqerr(iatom) = self.branches(ibranch).dqerr;
311
312                 self.stim_type(iatom)   = self.branches(ibranch).stim_type;
313                 self.source_type(iatom) = self.branches(ibranch).source_type;
314                 self.signal_type(iatom) = self.branches(ibranch).signal_type;
315                 self.xdc(iatom)         = self.branches(ibranch).idc;
316                 self.xa(iatom)          = self.branches(ibranch).ia;
317                 self.x1(iatom)          = self.branches(ibranch).i1;
318                 self.x2(iatom)          = self.branches(ibranch).i2;
319                 self.freq(iatom)        = self.branches(ibranch).freq;
320                 self.period(iatom)      = 1/self.branches(ibranch).freq;
321                 self.duty(iatom)        = self.branches(ibranch).duty;
322                 self.phi(iatom)         = self.branches(ibranch).phi;
323
324                 % add entries to T matrix:
325                 for itnodes = 1:self.branches(ibranch).ntnode
326                     self.T(ibranch, self.branches(ibranch).tnodes(itnodes).index) ...
327                         = self.branches(ibranch).T(itnodes);
328                 end
329
330                 % add entries to Z matrix:
331                 for izbranch = 1:self.branches(ibranch).nzbranch
332                     self.Z(ibranch, self.branches(ibranch).zbranches(izbranch).bindex) ...
333                         = self.branches(ibranch).Z(izbranch);
334                 end
335
336             end
337
338         end
339
340         function build_map(self)
341
342             for inode = 1:self.nnode
343
344                 % add connections from to B nodes:
345                 for ibnodes = 1:self.nodes(inode).nbnode
346                     if self.nodes(inode).bnodes(ibnodes).source_type == self.StimNone
347                         self.M(self.nodes(inode).bnodes(ibnodes).index, self.nodes(inode).index) =
    1;
348                     end
349                 end
```

```
350
351                 % add connections from to S branches:
352                 for isbranch = 1:self.nodes(inode).nsbranch
353                     if self.nodes(inode).sbranches(isbranch).source_type == self.SourceNone
354                         self.M(self.nodes(inode).sbranches(isbranch).index, self.nodes(inode).
    index) = 1;
355                     end
356                 end
357
358             end
359
360             for ibranch = 1:self.nbranch
361
362                 iatom = self.nnode + ibranch;
363
364                 if self.branches(ibranch).inode.source_type == self.SourceNone
365                     self.M(iatom, self.branches(ibranch).inode.index) = 1;
366                 end
367
368                 if self.branches(ibranch).jnode.source_type == self.SourceNone
369                     self.M(iatom, self.branches(ibranch).jnode.index) = 1;
370                 end
371
372                 if self.branches(ibranch).source_type == self.SourceNone
373                     self.M(self.branches(ibranch).inode.index, iatom) = 1;
374                     self.M(self.branches(ibranch).jnode.index, iatom) = 1;
375                 end
376
377                 % add connections from T nodes:
378                 for itnodes = 1:self.branches(ibranch).ntnode
379                     if self.branches(ibranch).tnodes(itnodes).source_type == self.SourceNone
380                         self.M(self.branches(ibranch).tnodes(itnodes).index, self.branches(ibranch
    ).index) = 1;
381                     end
382                 end
383
384                 % add connections from Z branches:
385                 for izbranch = 1:self.branches(ibranch).nzbranch
386                     if self.branches(ibranch).zbranches(izbranch).source_type == self.SourceNone
387                         self.M(self.branches(ibranch).zbranches(izbranch).index, self.branches(
    ibranch).index) = 1;
388                     end
389                 end
390
391             end
392
393         end
```

```
394
395         function add_node(self, node)
396
397             if node.dqmin <= 0
398                 node.dqmin = self.def_dqmin;
399             end
400
401             if node.dqmax <= 0
402                 node.dqmax = self.def_dqmax;
403             end
404
405             if node.dqerr <= 0
406                 node.dqerr = self.def_dqerr;
407             end
408
409             self.nnode = self.nnode + 1;
410             node.index = self.nnode;
411             self.nodes(self.nnode) = node;
412
413             % increment branch indeces as these are offest by the nodes:
414
415             for kbranch = 1:self.nbranch
416                 self.branches(kbranch).index = self.branches(kbranch).index + 1;
417             end
418
419         end
420
421         function add_branch(self, branch)
422
423             if branch.dqmin <= 0
424                 branch.dqmin = self.def_dqmin;
425             end
426
427             if branch.dqmax <= 0
428                 branch.dqmax = self.def_dqmax;
429             end
430
431             if branch.dqerr <= 0
432                 branch.dqerr = self.def_dqerr;
433             end
434
435             self.nbranch = self.nbranch + 1;
436             branch.bindex = self.nbranch; % not offset by nnode
437             branch.index = self.nbranch + self.nnode;  % offest by nnode
438             self.branches(self.nbranch) = branch;
439
440         end
441
```

```
442         function init(self)
443
444             self.build_lim();
445             self.build_map();
446             self.build_qdl();
447
448             self.time = 0.0;
449
450         end
451
452         function build_qdl(self)
453
454             % dimension liqss arrays:
455
456             self.qlo    = zeros(self.n, 1);
457             self.qhi    = zeros(self.n, 1);
458             self.tlast  = zeros(self.n, 1);
459             self.tnext  = zeros(self.n, 1);
460             self.trig   = zeros(self.n, 1);
461             self.x      = zeros(self.n, 1);
462             self.d      = zeros(self.n, 2);
463             self.q      = zeros(self.n, 2);
464
465             % initialize liqss arrays:
466
467             self.tnext(:) =  inf;
468             self.dq(:)    =  self.dqmin(:);
469             self.qhi(:)   =  self.dq(:);
470             self.qlo(:)   = -self.dq(:);
471             self.x(:)     =  self.x0(:);
472             self.q(:, 1)  =  self.x0(:);
473             self.q(:, 2)  =  self.x0(:);
474
475             % output data arrays:
476
477             self.qout = zeros(self.n, self.npt);
478             self.tout = zeros(self.n, self.npt);
479             self.iout = zeros(self.n, 1);
480             self.iout(:) = 1;
481
482             % set up counters:
483
484             self.nupd = zeros(self.n, self.npt);
485             self.tupd = zeros(self.n, self.npt);
486             self.iupd = zeros(self.n, 1);
```

```
            self.iupd(:) = 1;

        end

        function runto(self, tstop)

            self.tstop = tstop;

            self.iupdates = 0

            % force initial update and save for this run period:

            for iatom = 1:self.n
                self.internal_update(iatom);
                self.save(iatom);
            end

            % now update intially externally triggered atoms:

            while any(self.trig)
                for iatom = 1:self.n
                   if self.trig(iatom)
                       self.internal_update(iatom);
                   end
                end
            end

            % now advance the simulation until we reach tstop:

            %self.timer = timer();
            %self.timer.ExecutionMode = 'fixedRate';
            %self.timer.Period = 1;

            %self.timer.TimerFcn = {@(~,~,obj, x) disp(['sim time: ' ...
            %    num2str(round(obj.time)) ' of ' x]), self, num2str(tstop)};

            %start(self.timer);

            tdisp = 0;
            tint = 10;

            while self.time < self.tstop
                self.advance();
                if self.time - tdisp > tint
                    disp(['sim time = ' num2str(round(self.time))]);
                    tdisp = self.time;
                end
```

```
534             end
535
536             %stop(self.timer);
537
538             % force final update and save for this run period:
539
540             self.time = self.tstop;
541
542             for iatom = 1:self.n
543                 self.internal_update(iatom);
544                 self.save(iatom);
545             end
546         end
547
548         function advance(self)
549
550             % determine next time and advance time:
551
552             self.time = min(min(self.tnext), self.tstop);
553             self.time;
554
555             % set force flag if we are at the simulation stop time:
556
557             force = self.time >= self.tstop;
558
559             % perform next scheduled internal updates:
560
561             for iatom = 1:self.n
562                 if self.tnext(iatom) <= self.time || force
563                    self.internal_update(iatom);
564                    self.iupdates = self.iupdates + 1;
565                 end
566             end
567
568             % now update externally triggered atoms:
569
570             while any(self.trig)
571                 for iatom = 1:self.n
572                     if self.trig(iatom)
573                        self.internal_update(iatom);
574                        self.iupdates = self.iupdates + 1;
575                     end
576                 end
577             end
578
579         end
580
```

```
581         function internal_update(self, iatom)
582 
583             self.trig(iatom) = 0;  % reset trigger flag
584 
585             self.dint(iatom);      % update internal state (delta_int function)
586             self.quantize(iatom);  % quantize internal state
587             self.d(iatom, 2) = self.f(iatom, self.q(iatom, 2));  % update derivative
588             self.ta(iatom);        % calculate new tnext
589 
590             % trigger external update if quatized output changed:
591 
592             if self.q(iatom, 2) ~= self.q(iatom, 1)
593                 self.save(iatom); % save output
594                 self.q(iatom, 1) = self.q(iatom, 2);  % save last q
595                 self.trigger(iatom); % set trigger flags
596                 self.update_dq(iatom);
597             end
598 
599         end
600 
601         function external_update(self, iatom)
602 
603             self.trig(iatom) = 0;  % reset trigger flag
604 
605             self.dext(iatom);      % update internal state
606             self.quantize(iatom);  % quantize internal state
607             self.d(iatom, 2) = self.f(iatom, self.q(iatom, 2));  % update derivative
608             self.ta(iatom);        % calculate new tnext
609 
610             % trigger external update if quatized output changed:
611 
612             if self.q(iatom, 2) ~= self.q(iatom, 1)
613                 self.save(iatom); % save output
614                 self.q(iatom, 1) = self.q(iatom, 2);  % save last q
615                 self.trigger(iatom); % set trigger flags replace with tnext = time + dtmin for all
    trigger items?
616                 self.update_dq(iatom);
617             end
618 
619         end
620 
621         function dint(self, iatom)
622 
623             if self.source_type(iatom) == self.SourceNone
624 
625                 self.x(iatom) = self.x(iatom) + self.d(iatom, 2) * (self.time - self.tlast(iatom))
```

```
    ;
626
627             elseif self.source_type(iatom) == self.SourceDC
628
629                 self.x(iatom) = self.xdc(iatom);
630
631             elseif self.source_type(iatom) == self.SourcePWM
632
633                 if self.duty(iatom) == 0
634                     self.x(iatom) = self.x1(iatom);
635
636                 elseif self.duty(iatom) == 1
637                     self.x(iatom) = self.x2(iatom);
638
639                 else
640
641                     w = mod(self.time, self.period(iatom));
642
643                     if self.isnear(w, self.period(iatom))
644                         self.x(iatom) = self.x1(iatom);
645
646                     elseif self.isnear(w, self.period(iatom) * self.duty(iatom))
647                         self.x(iatom) = self.x2(iatom);
648
649                     elseif w < self.period(iatom) * self.duty(iatom)
650                         self.x(iatom) = self.x1(iatom);
651
652                     else
653                         self.x(iatom) = self.x2(iatom);
654
655                     end
656
657                 end
658
659             elseif self.source_type(iatom) == self.SourceSINE
660
661                 self.x(iatom) = self.xdc(iatom) + self.xa(iatom) * ...
662                     sin(2*pi*self.freq(iatom)*self.time + self.phi(iatom));
663
664             end
665
666             self.tlast(iatom) = self.time;
667
668         end
669
670         function interp = quantize(self, iatom)
671
```

```
672             if self.source_type(iatom) == self.SourceNone
673
674                 % save old derivative:
675                 self.d(iatom, 1) = self.d(iatom, 2);
676
677                 interp = 0;
678                 change = 0;
679
680                 if self.x(iatom) >= self.qhi(iatom)
681                     self.q(iatom, 2) = self.qhi(iatom);
682                     self.qlo(iatom) = self.qlo(iatom) + self.dq(iatom);
683                     change = 1;
684                 elseif self.x(iatom) <= self.qlo(iatom)
685                     self.q(iatom, 2) = self.qlo(iatom);
686                     self.qlo(iatom) = self.qlo(iatom) - self.dq(iatom);
687                     change = 1;
688                 end
689
690                 self.qhi(iatom) = self.qlo(iatom) + 2 * self.dq(iatom);
691
692                 if change  % we've ventured out of qlo/self.qhi(iatom) bounds
693
694                     % calculate new derivative:
695                     self.d(iatom, 2) = self.f(iatom, self.q(iatom, 2));
696
697                     % if the derivative has changed signs, then we know
698                     % we are in a potential oscillating situation, so
699                     % we will set the q such that the derivative=0:
700
701                     if self.d(iatom, 2) * self.d(iatom, 1) < 0  % if derivative has changed sign
702                         flo = self.f(iatom, self.qlo(iatom));
703                         fhi = self.f(iatom, self.qhi(iatom));
704                         a = (2 * self.dq(iatom)) / (fhi - flo);
705                         self.q(iatom, 2) = self.qhi(iatom) - a * fhi;
706                         interp = 1;
707                     end
708
709                 end
710
711             else
712
713                 self.q(iatom, 2) = self.x(iatom);  % all sources fix q=x
714
715             end
716
717         end
```

```
718
719         function ta(self, iatom)
720
721             if self.source_type(iatom) == self.SourceNone
722
723                 if (self.d(iatom, 2) > 0)
724                     self.tnext(iatom) = self.time + (self.qhi(iatom) - self.x(iatom)) / self.d(
    iatom, 2);
725                 elseif (self.d(iatom, 2) < 0)
726                     self.tnext(iatom) = self.time + (self.qlo(iatom) - self.x(iatom)) / self.d(
    iatom, 2);
727                 else
728                     self.tnext(iatom) = inf;  % derivative == 0, so tnext is inf
729                 end
730
731             elseif self.source_type(iatom) == self.SourceDC
732
733                 self.tnext(iatom) = inf;
734
735             elseif self.source_type(iatom) == self.SourcePWM
736
737                 if self.duty(iatom) == 0 || self.duty(iatom) == 1
738                     self.tnext(iatom) = inf;
739
740                 else
741
742                     w = mod(self.time, self.period(iatom));
743
744                     if self.isnear(w, self.period(iatom))
745                         self.tnext(iatom) = self.time + self.period(iatom) * self.duty(iatom);
746                     elseif self.isnear(w, self.period(iatom) * self.duty(iatom))
747                         self.tnext(iatom) = self.time + self.period(iatom) - w;
748                     elseif w < self.period(iatom) * self.duty(iatom)
749                         self.tnext(iatom) = self.time + self.period(iatom) * self.duty(iatom) - w;
750                     else
751                         self.tnext(iatom) = self.time + self.period(iatom) - w;
752                     end
753
754                 end
755
756             elseif self.source_type(iatom) == self.SourceSINE
757
758                 [q, self.tnext(iatom)] = self.update_sine(self.xdc(iatom), ...
759                     self.xa(iatom), self.freq(iatom), self.phi(iatom), ...
760                     self.time, self.dq(iatom));
761
```

```
762             end
763
764             % force delta t to be greater than 0:
765
766             self.tnext(iatom) = max(self.tnext(iatom), self.tlast(iatom) + self.dtmin);
767
768         end
769
770         function save(self, iatom)
771
772             if 0  % (enable zoh recording by changing to if 1)
773                 self.iout(iatom) = self.iout(iatom) + 1;
774                 self.tout(iatom, self.iout(iatom)) = self.time - self.dtmin/2;
775                 self.qout(iatom, self.iout(iatom)) = self.q(iatom, 1);
776             end
777
778             %if ~self.isnear(self.time, self.tout(iatom, self.iout(iatom)))
779             %if abs(self.time - self.tout(iatom, self.iout(iatom))) >= self.dtmin
780             if self.time ~= self.tout(iatom, self.iout(iatom))
781                 self.iout(iatom) = self.iout(iatom) + 1;
782                 self.tout(iatom, self.iout(iatom)) = self.time;
783             end
784
785             self.qout(iatom, self.iout(iatom)) = self.q(iatom, 2);
786
787
788             if self.time ~= self.tupd(self.iupd)
789                 self.iupd(iatom) = self.iupd(iatom) + 1;
790                 self.tupd(iatom, self.iupd(iatom)) = self.time;
791             end
792
793             self.nupd(iatom, self.iupd(iatom)) = self.nupd(iatom, self.iupd(iatom)) + 1;
794
795         end
796
797         function d = f(self, iatom, qval)
798
799             d = 0;
800
801             if self.source_type(iatom) == self.SourceNone  % only for latent atoms
802
803                 if iatom <= self.nnode  % node
804
805                     %  v(i)' = 1/c(i) * [h(i) + b(i,p) * v(p) + s(i,q) * i(q) + sum(ik) - g(i) * v
    (i)]
806
807                     isum = self.A(iatom, :) * self.q(self.nnode+1:end, 2);
```

```
808
809                     % d = 1/C * [ H + B * qnode + S * qbranch - q * G - isum ]
810
811                     d = self.Cinv(iatom) * (self.H(iatom) + self.B(iatom,:) * self.q(1:self.nnode,
     2) ...
812                         + self.S(iatom,:) * self.q(self.nnode+1:end, 2) - qval * self.G(iatom) -
    isum);
813
814                 else  % branch
815
816                     % i(k)' = 1/L(k) * [e(k) + t(k,p) * v(p) + z(k,q) * i(q) + (v(i) - v(j)) - r(k
    ) * i(k)]
817
818                     ibranch = iatom-self.nnode;
819
820                     vij = self.A(:, ibranch)' * self.q(1:self.nnode, 2);
821
822                     % d = 1/L * [ E + T * qnode + Z * qbranch - q * R - vij ]
823
824                     d = self.Linv(ibranch) * (self.E(ibranch) + self.T(ibranch,:) * self.q(1:self.
    nnode, 2) ...
825                         + self.Z(ibranch,:) * self.q(self.nnode+1:end, 2) - qval * self.R(ibranch)
     + vij);
826
827                 end
828
829             elseif self.source_type(iatom) == self.SourceSINE
830
831                 d1 = 2 * pi * self.freq(iatom) * self.xa(iatom) * ...
832                     cos(2 * pi * self.freq(iatom) * self.time + self.phi(iatom));
833
834                 d2 = -4 * pi * pi * self.freq(iatom) * self.freq(iatom) * self.xa(iatom) * ...
835                     sin(2 * pi * self.freq(iatom) * self.time + self.phi(iatom));
836
837                 if abs(d1) > abs(d2)
838                     d = d1;
839                 else
840                     d = d2;
841                 end
842
843             end
844
845         end
846
847         function dext(self, iatom)
848
849             if self.source_type(iatom) == self.SourceNone
```

```
850
851                 self.x(iatom) = self.x(iatom) + f(self, iatom, self.x(iatom)) * (self.time - self.
    tlast(iatom));
852
853             end
854
855             self.tlast(iatom) = self.time;
856
857         end
858
859         function trigger(self, iatom)
860
861             for jatom = 1:self.n
862                 if self.M(iatom, jatom)
863                     self.trig(jatom) = 1;
864                 end
865             end
866
867         end
868
869         function update_dq(self, iatom)
870
871             if (self.dqmax(iatom) - self.dqmin(iatom)) < eps
872                 return
873             end
874
875             self.dq(iatom) = min(self.dqmax(iatom), max(self.dqmin(iatom), ...
876                                  abs(self.dqerr(iatom) * self.q(iatom, 2))));
877
878             self.qlo(iatom) = self.q(iatom) - self.dq(iatom);
879             self.qhi(iatom) = self.q(iatom) + self.dq(iatom);
880
881         end
882
883         function build_ss(self)
884
885             n = self.n;
886             nn = self.nnode;
887             nb = self.nbranch;
888
889             Ann = diag(self.Cinv) * (self.B - diag(self.G));
890             Anb = diag(self.Cinv) * (self.S - self.A);
891             Abn = diag(self.Linv) * (self.T + self.A');
892             Abb = diag(self.Linv) * (self.Z - diag(self.R));
893
894             self.Ass = [Ann, Anb; Abn, Abb];
895
```

```
            self.Bss = diag(cat(1, self.Cinv, self.Linv));

            self.Uss = [self.H; self.E];

        end

        function [t, x] = run_ss_to(self, dt, tstop)

            self.build_ss();

            self.dtss = dt;

            t = self.time : dt : tstop;
            npt = length(t);

            x = zeros(self.n, npt);

            x(:, 1) = self.x;

            self.Apr = inv(eye(self.n) - dt * self.Ass);
            self.Bpr = self.Apr * self.Bss * dt;

            for k = 2:npt
                x(:,k) = self.Apr * x(:,k-1) + self.Bpr * self.Uss;
            end

        end

        function init_ss(self, dt)

            self.build_ss();

            self.dtss = dt;

            if self.tss == 0
                self.xss = self.x0(:);
            end

            self.Apr = inv(eye(self.n) - dt * self.Ass);
            self.Bpr = self.Apr * self.Bss * dt;

        end

        function step_ss(self)

            self.tss = self.tss + self.dtss;
            self.xss = self.Apr * self.xss + self.Bpr * self.Uss;

```

```
944         end
945
946         function plot(self, atom, dots, lines, upd, cumm_upd, bins, xlbl, tss, xss, ymax)
947
948             k = atom.index;
949
950             ss = length(tss) > 1;
951
952             if upd
953                 yyaxis left
954             end
955
956             if ss
957                 plot(tss, xss, 'c--', ...
958                     'DisplayName','ss'); hold on;
959             end
960
961             if dots
962                 plot(self.tout(k,1:1:self.iout(k)), ...
963                     self.qout(k,1:1:self.iout(k)), 'k.', ...
964                     'DisplayName','qss'); hold on;
965             end
966
967             if lines
968                 plot(self.tout(k,1:self.iout(k)), ...
969                     self.qout(k,1:self.iout(k)), 'b-', ...
970                     'DisplayName', 'qss'); hold on;
971             end
972
973             ylabel('atom state');
974
975             if upd
976                 yyaxis right
977
978                 updstr = strcat('updates per: ', num2str(self.time/bins), ' s');
979
980                 if cumm_upd
981                     plot(self.tupd(k,1:1:self.iupd(k)), cumsum(self.nupd(k,1:1:self.iupd(k)))/2, '
    r-',...
982                         'DisplayName', 'updates');
983                 else
984                     histogram(self.tupd(k,1:1:self.iupd(k)), bins, 'EdgeColor', 'none', 'FaceAlpha
    ', 0.3, ...
985                         'DisplayName', updstr);
986                 end
987                 title(atom.name);
```

```
            ylabel('updates');
            if ymax > 0
                ylim([0, ymax]);
            end
        end

        xlim([-1.0, self.time]);

        loc = 'southeast';

        %leg = legend();
        %leg.Location = loc;

        if xlbl
           xlabel('t (s)');
        end

    end

end

methods(Static)

    function result = isnear(a, b)

        result = abs(a-b) < 1e4*eps(min(abs(a), abs(b)));

    end

    function [q, tnext] = update_sine(x0, xa, f, phi, t, dq)

        T = 1/f;                % period
        w = mod(t, T);          % cycle time
        t0 = t - w;             % cycle base time
        omega = 2*pi*f;         % angular velocity
        theta = omega*w + phi;  % wrapped angular position
        x = xa * sin(2*pi*f*t);  % magnitude shifted by x0

        % determine next transition time. Saturate at +/- xa:

        if theta < pi/2        % quadrant I
            tnext = t0 + (asin(min(1, (x + dq)/xa))) / omega;
        elseif theta < pi      % quadrant II
            tnext = t0 + T/2 - (asin(max(0, (x - dq)/xa))) / omega;
        elseif theta < 3*pi/2  % quadrant III
            tnext = t0 + T/2 - (asin(max(-1, (x - dq)/xa))) / omega;
```

```
            else                        % quadrant IV
                tnext = t0 + T + (asin(min(0, (x + dq)/xa))) / omega;
            end

            % update state for current time:
            q = x0 + xa * sin(2*pi*f*t + phi);

        end

    end

end
```

## A.2 MATLAB QDL Model Source

```
classdef QdlDevice < handle

    properties

        name
        dqmax
        dqmin
        dqerr
        index
        freq
        duty
        phi

    end

    methods

        function self = QdlDevice(name)

            self.name  = name;
            self.dqmax = 0;
            self.dqmin = 0;
            self.dqerr = 0;
            self.index = 0;
            self.freq  = 0;
            self.duty  = 0;
            self.phi   = 0;

        end
```

```
30
31      end
32
33  end
34
35
36  classdef QdlNode < QdlDevice
37
38      properties
39
40          C
41          G
42          H
43          B
44          S
45          bnodes
46          sbranches
47          nbnode
48          nsbranch
49          v0
50          vdc
51          va
52          v1
53          v2
54          stim_type
55          source_type
56          signal_type
57
58      end
59
60      methods
61
62          function self = QdlNode(name, C, G, H)
63
64              self@QdlDevice(name);
65
66              self.C = C;
67              self.G = G;
68              self.H = H;
69
70              self.B = double.empty(0);
71              self.S = double.empty(0);
72              self.bnodes = QdlNode.empty(0);
73              self.sbranches = QdlBranch.empty(0);
74              self.nbnode = 0;
75              self.nsbranch = 0;
76
77              self.stim_type = QdlSystem.StimNone;
78              self.source_type = QdlSystem.SourceNone;
```

```
79              self.signal_type = QdlSystem.SignalNone;
80              self.v0 = 0;
81              self.vdc = 0;
82              self.va = 0;
83              self.v1 = 0;
84              self.v2 = 0;
85
86          end
87
88          function add_bnode(self, node, gain)
89
90              self.nbnode = self.nbnode + 1;
91              self.B(self.nbnode) = gain;
92              self.nbnodes(self.nbnode) = node;
93
94          end
95
96          function add_sbranch(self, branch, gain)
97
98              self.nsbranch = self.nsbranch + 1;
99              self.S(self.nsbranch) = gain;
100             self.sbranches(self.nsbranch) = branch;
101
102         end
103
104     end
105
106 end
107
108
109 classdef QdlBranch < QdlDevice
110
111     properties
112
113         inode
114         jnode
115         bindex
116         L
117         R
118         E
119         T
120         Z
121         tnodes
122         zbranches
123         ntnode
124         nzbranch
125         i0
126         idc
127         ia
```

```
        i1
        i2
        stim_type
        source_type
        signal_type

    end

    methods

        function self = QdlBranch(name, L, R, E)

            self@QdlDevice(name);

            self.L = L;
            self.R = R;
            self.E = E;

            self.T = double.empty(0);
            self.Z = double.empty(0);
            self.tnodes = QdlNode.empty(0);
            self.zbranches = QdlBranch.empty(0);
            self.ntnode = 0;
            self.nzbranch = 0;

            self.stim_type = QdlSystem.StimNone;
            self.source_type = QdlSystem.SourceNone;
            self.signal_type = QdlSystem.SignalNone;
            self.i0 = 0;
            self.idc = 0;
            self.ia = 0;
            self.i1 = 0;
            self.i2 = 0;

        end

        function connect(self, inode, jnode)

            self.inode = inode;
            self.jnode = jnode;

        end

        function add_tnode(self, node, gain)

            self.ntnode = self.ntnode + 1;
            self.T(self.ntnode) = gain;
            self.tnodes(self.ntnode) = node;
```

```
176
177         end
178
179         function add_zbranch(self, branch, gain)
180
181             self.nzbranch = self.nzbranch + 1;
182             self.Z(self.nzbranch) = gain;
183             self.zbranches(self.nzbranch) = branch;
184
185         end
186
187     end
188
189 end
```

# Appendix B

# Python Source Code

## B.1 Python QDL Simulator Source

```
"""Quantized DEVS-LIM modeling and simulation framework.
"""

import os
import time
import pickle
import queue
import threading as th

import pandas as pd
import numpy as np
import numpy.linalg as la

from glob import glob
from math import pi, sin, cos, acos, tan, acos, atan2, sqrt, floor as FLOOR
from cmath import sqrt as csqrt
from collections import deque
from collections import OrderedDict as odict
from array import array

from mpl_toolkits import mplot3d
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes.formatter', useoffset=False)

from scipy.integrate import solve_ivp
from scipy.optimize import fsolve
from scipy.interpolate import interp1d
```

```
from scipy.stats import gaussian_kde
from scipy.linalg import eig, eigh

# ============================ Private Constants ==============================

_EPS = 1.0e-12
_INF = float('inf')
_MAXITER = 1000

# ============================ Public Constants ===============================

DEF_DTMIN = 1e-12      # default minimum time step
DEF_DMAX = 1.0e5       # default maximum derivative (slew-rate)
MIN_DT_SAVE = 1.0e-9   # min time between saves (defines max time resolution)

PI_4 = float(pi / 4.0)
PI_3 = float(pi / 3.0)
PI5_6 = float(5.0 * pi / 6.0)
PI7_6 = float(7.0 * pi / 6.0)

# ============================ Enumerations ===================================

class StorageType:

    LIST = "LIST"
    ARRAY = "ARRAY"
    DEQUE = "DEQUE"

class SourceType:

    NONE = "NONE"
    CONSTANT = "CONSTANT"
    STEP = "STEP"
    SINE = "SINE"
    PWM = "PWM"
    RAMP = "RAMP"
    FUNCTION = "FUNCTION"

class OdeMethod:

    RK45 = "RK45"
    RK23 = "RK23"
    DOP853 = "DOP853"
    RADAU = "Radau"
    BDF = "BDF"
    LSODA = "LSODA"

class QssMethod:

```

```
    QSS1 = "QSS1"
    QSS2 = "QSS2"
    LIQSS1 = "LIQSS1"
    LIQSS2 = "LIQSS2"
    mLIQSS1 = "mLIQSS1"
    mLIQSS2 = "mLIQSS2"

# ================================ GLOBALS ===================================

sys = None  # set by qdl.System constructor for visibility from fode function.
simtime = 0.0
simlock = th.Lock()

# ============================== QDL MODEL ===================================

class Atom(object):

    def __init__(self, name, x0=0.0, dq=None, dqmin=None, dqmax=None,
                 dqerr=None, dtmin=None, dmax=1e10, units="", qss_method=None):

        # params:

        self.name = name
        self.x0 = x0  # initial value of state
        self.dq = dq  # quantization step size
        self.dqmin = dqmin
        self.dqmax = dqmax
        self.dqerr = dqerr
        self.dtmin = dtmin
        self.dmax = dmax
        self.units = units

        self.qss_method = qss_method  # type QssMethod

        # simulation variables:

        # quantization step:

        self.dq0 = self.dq
        self.qlo = 0.0
        self.qhi = 0.0
        self.qdlo = 0.0
        self.qdhi = 0.0

        # time:

        self.tlast = 0.0    # last update time of internal state x
        self.tlastq = 0.0   # last update time of quantized state q
```

```
126         self.tnext = 0.0    # next time to quantized state change
127
128         # internal state:
129
130         self.x = x0
131         self.xlast = x0
132
133         # internal state derivatives:
134
135         self.dx = 0.0
136         self.dxlast = 0.0
137         self.ddx = 0.0
138         self.ddxlast = 0.0
139
140         # quantized state:
141
142         self.q = x0
143         self.qlast = x0
144         self.qd = x0
145         self.qdlast = x0
146
147         self.a = 0.0
148         self.u = 0.0
149         self.du = 0.0
150
151         self.triggered = False
152         self.savetime = 0.0
153
154         # results data storage:
155
156         # qss:
157         self.tout = None  # output times quantized output
158         self.qout = None  # quantized output
159         self.tzoh = None  # zero-order hold output times quantized output
160         self.qzoh = None  # zero-order hold quantized output
161         self.evals = 0    # qss evals
162         self.updates = 0  # qss updates
163
164         # state space:
165         self.tout_ss = None  # state space time output
166         self.xout_ss = None  # state space value output
167         self.updates_ss = 0  # state space update count
168
169         # non-linear ode:
170         self.tode = None  # state space time output
171         self.xode = None  # state space value output
172         self.updates_ode = 0  # state space update count
173
174         # atom connections:
```

```
175
176         self.broadcast_to = []  # push updates to
177         self.connections = []   # recieve updates from
178
179         # jacobian cell functions:
180
181         self.jacfuncs = []
182         self.derargfunc = None
183
184         # parent object references:
185
186         self.sys = None
187         self.device = None
188
189         # other:
190
191         self.implicit = True
192
193         self.tocsv = False
194         self.outdir = None
195         self.csv = None
196
197     def add_connection(self, other, coefficient=1.0, coeffunc=None):
198
199         connection = Connection(self, other, coefficient=coefficient,
200                                 coeffunc=coeffunc)
201
202         connection.device = self.device
203         self.connections.append(connection)
204
205         return connection
206
207     def add_jacfunc(self, other, func):
208
209         self.jacfuncs.append((other, func))
210
211     def set_state(self, value, quantize=False):
212
213         self.x = float(value)
214
215         if quantize:
216
217             self.quantize(implicit=False)
218
219         else:
220             self.q = value
221             self.qhi = self.q + self.dq
222             self.qlo = self.q - self.dq
```

```
223
224     def initialize(self, t0):
225
226         self.tlast = t0
227         self.tlastq = t0
228         self.tnext = _INF
229         self.savetime = t0
230
231         # init state:
232
233         self.x = self.x0
234         self.q = self.x
235         self.qlast = self.x
236         self.qd = 0.0
237         self.qdlast = 0.0
238         self.a = 0.0
239         self.u = 0.0
240         self.du = 0.0
241         self.qsave = self.x
242         self.xsave = self.x
243
244         # init quantizer values:
245
246         self.qhi = self.q + self.dq
247         self.qlo = self.q - self.dq
248         self.qdhi = self.qd + self.dq
249         self.qdlo = self.qd - self.dq
250
251         # init output:
252
253         self.clear_data_arrays()
254
255         self.updates = 0
256         self.evals = 0
257         self.updates_ss = 0
258         self.updates_ode = 0
259
260         self.tout.append(self.tlast)
261         self.qout.append(self.qlast)
262         self.nupd.append(0)
263         self.tzoh.append(self.tlast)
264         self.qzoh.append(self.qlast)
265
266         self.tout_ss.append(self.tlast)
267         self.xout_ss.append(self.qlast)
268         self.nupd_ss.append(0)
269
```

```
        self.tode.append(self.tlast)
        self.xode.append(self.qlast)
        self.uode.append(0)

        self.tocsv = self.sys.tocsv
        self.outdir = self.sys.outdir

        if self.tocsv and self.outdir:

            self.csv = os.path.join(self.outdir, "{}_{}.csv".format(self.device.name, self.name))

            with open(self.csv, "w") as f:
                f.write("t,q,e,n\n")
                f.write("{},{},{}\n".format(self.tlast, self.qlast, 0, 0))

    def state_to_dict(self):

        state = {}

        state["tlast"]      = self.tlast
        state["tlastq"]     = self.tlastq
        state["tnext"]      = self.tnext
        state["savetime"]   = self.savetime
        state["x0"]         = self.x0
        state["x"]          = self.x
        state["xlast"]      = self.xlast
        state["dx"]         = self.dx
        state["dxlast"]     = self.dxlast
        state["ddx"]        = self.ddx
        state["ddxlast"]    = self.ddxlast
        state["q"]          = self.q
        state["qlast"]      = self.qlast
        state["qd"]         = self.qd
        state["dq0"]        = self.dq0
        state["dq"]         = self.dq
        state["qdlast"]     = self.qdlast
        state["a"]          = self.a
        state["u"]          = self.u
        state["du"]         = self.du
        state["qsave"]      = self.qsave
        state["xsave"]      = self.xsave
        state["qhi"]        = self.qhi
        state["qlo"]        = self.qlo
        state["qdhi"]       = self.qdhi
```

```
        state["qdlo"]       = self.qdlo

        return state

    def state_from_dict(self, state):

        self.tlast    = state["tlast"]
        self.tlastq   = state["tlastq"]
        self.tnext    = state["tnext"]
        self.savetime = state["savetime"]
        self.x0       = state["x0"]
        self.x        = state["x"]
        self.xlast    = state["xlast"]
        self.dx       = state["dx"]
        self.dxlast   = state["dxlast"]
        self.ddx      = state["ddx"]
        self.ddxlast  = state["ddxlast"]
        self.q        = state["q"]
        self.qlast    = state["qlast"]
        self.qd       = state["qd"]
        self.dq0      = state["dq0"]
        self.dq       = state["dq"]
        self.qdlast   = state["qdlast"]
        self.a        = state["a"]
        self.u        = state["u"]
        self.du       = state["du"]
        self.qsave    = state["qsave"]
        self.xsave    = state["xsave"]
        self.qhi      = state["qhi"]
        self.qlo      = state["qlo"]
        self.qdhi     = state["qdhi"]
        self.qdlo     = state["qdlo"]

    def step(self, t):

        self.tlast = t
        self.evals += 1
        self.updates += 1
        self.dx = self.f()
        self.dint(t)
        self.q = self.x
        self.save_qss(t, force=True)
        self.qlast = self.q

    def dint(self, t):
```

```
359
360         raise NotImplementedError()
361
362     def quantize(self):
363
364         raise NotImplementedError()
365
366     def ta(self, t):
367
368         raise NotImplementedError()
369
370     def f(self, q, t=0.0):
371
372         raise NotImplementedError()
373
374     def df(self, q, d, t=0.0):
375
376         raise NotImplementedError()
377
378     def broadcast(self):
379
380         for atom in self.broadcast_to:
381             if atom is not self:
382                 atom.triggered = True
383
384     def update_dq(self):
385
386         if not self.dqerr:
387             return
388         else:
389             if self.dqerr <= 0.0:
390                 return
391
392         if not (self.dqmin or self.dqmax):
393             return
394
395         if (self.dqmax - self.dqmin) < _EPS:
396             return
397
398         self.dq = min(self.dqmax, max(self.dqmin, abs(self.dqerr * self.q)))
399
400         self.qlo = self.q - self.dq
401         self.qhi = self.q + self.dq
402
403     def clear_data_arrays(self):
404
405         if self.sys.storage_type == StorageType.LIST:
406
```

```
            self.tout = []
            self.qout = []
            self.nupd = []
            self.tzoh = []
            self.qzoh = []

            self.tout_ss = []
            self.xout_ss = []
            self.nupd_ss = []

            self.tode = []
            self.xode = []
            self.uode = []

        elif self.sys.storage_type == StorageType.ARRAY:

            typecode = "d"

            self.tout = array(typecode)
            self.qout = array(typecode)
            self.nupd = array(typecode)

            if len(self.tzoh > 0):
                tzoh_last = self.tzoh[-1]
                self.tzoh = array(typecode)
                self.tzoj.append(tzoh_last)
            else:
                self.tzoh = array(typecode)

            if len(self.qzoh > 0):
                qzoh_last = self.qzoh[-1]
                self.qzoh = array(typecode)
                self.qzoh.append(qzoh_last)
            else:
                self.qzoh = array(typecode)

            self.tout_ss = array(typecode)
            self.xout_ss = array(typecode)
            self.nupd_ss = array(typecode)

            self.tode = array(typecode)
            self.xode = array(typecode)
            self.uode = array(typecode)

        elif self.sys.storage_type == StorageType.DEQUE:
```

```
452
453             self.tout = deque()
454             self.qout = deque()
455             self.nupd = deque()
456
457             if self.tzoh:
458                 if len(self.tzoh) > 0:
459                     tzoh_last = self.tzoh[-1]
460                     self.tzoh = deque()
461                     self.tzoh.append(tzoh_last)
462                 else:
463                     self.tzoh = deque()
464             else:
465                 self.tzoh = deque()
466
467             if self.qzoh:
468                 if len(self.qzoh) > 0:
469                     qzoh_last = self.qzoh[-1]
470                     self.qzoh = deque()
471                     self.qzoh.append(qzoh_last)
472                 else:
473                     self.qzoh = deque()
474             else:
475                 self.qzoh = deque()
476
477             self.tout_ss = deque()
478             self.xout_ss = deque()
479             self.nupd_ss = deque()
480
481             self.tode = deque()
482             self.xode = deque()
483             self.uode = deque()
484
485     def save_qss(self, t, force=False):
486
487         self.updates += 1
488
489         if t < self.savetime:
490             return
491         else:
492             self.savetime = t + MIN_DT_SAVE
493
494         if self.q != self.qlast or force:
495
496             self.tout.append(t)
497             self.qout.append(self.q)
```

```
            self.nupd.append(self.updates)

            try:
               self.qzoh.append(self.qzoh[-1])
               self.tzoh.append(t)
            except:
                pass

            self.tzoh.append(t)
            self.qzoh.append(self.q)

    def save_ss(self, t, x):

        self.tout_ss.append(t)
        self.xout_ss.append(x)
        self.nupd_ss.append(self.updates_ss)
        self.updates_ss += 1

    def save_ode(self, t, x):

        self.tode.append(t)
        self.xode.append(x)
        self.uode.append(self.updates_ss)
        self.updates_ode += 1

    def get_error(self, typ="l2"):

        # interpolate qss to ss time vector:
        # this function can only be called after state space AND qdl simualtions
        # are complete

        qout_interp = numpy.interp(self.tout2, self.tout, self.qout)

        if typ.lower().strip() == "l2":

            # calculate the L**2 relative error:
            #      ________________
            #     / sum((y - q)**2)
            #    /  --------------
            #  \/      sum(y**2)

            dy_sqrd_sum = 0.0
            y_sqrd_sum = 0.0

            for q, y in zip(qout_interp, self.qout2):
                dy_sqrd_sum += (y - q)**2
```

```
            y_sqrd_sum += y**2

        return sqrt(dy_sqrd_sum / y_sqrd_sum)

    elif typ.lower().strip() == "nrmsd":

        # calculate the normalized relative root mean squared error:
        #      _________________
        #     / sum((y - q)**2)
        #    /  ---------------
        #  \/          N
        # ----------------------
        #       max(y) - min(y)

        dy_sqrd_sum = 0.0
        y_sqrd_sum = 0.0

        for q, y in zip(qout_interp, self.qout2):
            dy_sqrd_sum += (y - q)**2
            y_sqrd_sum += y**2

        return sqrt(dy_sqrd_sum / len(qout_interp)) / (max(self.qout2) - min(self.qout2))

    elif typ.lower().strip() == "re":

        # Pointwise relative error
        # e = [|(y - q)| / |y|]

        e = []

        for q, y in zip(qout_interp, self.qout2):
            e.append(abs(y-q) / abs(y))

        return e

    elif typ.lower().strip() == "rpd":

        # Pointwise relative percent difference
        # e = [ 100% * 2 * |y - q| / (|y| + |q|)]

        e = []

        for q, y in zip(qout_interp, self.qout2):
            den = abs(y) + abs(q)
            if den >= _EPS:
                e.append(100 * 2 * abs(y-q) / (abs(y) + abs(q)))
            else:
```

```
                    e.append(0)

            return e

        return None

    def get_previous_state(self):

        if self.qout:
            if len(self.qout) >= 2:
                return self.qout[-2]
            else:
                return self.xlast
        else:
            return self.xlast

    def full_name(self):

        return self.device.name + "." + self.name

    def __repr__(self):

        return self.full_name()

    def __str__(self):

        return __repr__(self)


class SourceAtom(Atom):

    def __init__(self, name, source_type=SourceType.CONSTANT, x0=0.0, x1=0.0,
                 x2=0.0, xa=0.0, freq=0.0, phi=0.0, duty=0.0, t1=0.0, t2=0.0,
                 srcfunc=None, gainfunc=None, dq=None, dqmin=None, dqmax=None, dqerr=None,
                 dtmin=None, dmax=1e10, units=""):

        Atom.__init__(self, name=name, x0=x0, dq=dq, dqmin=dqmin, dqmax=dqmax,
                      dqerr=dqerr, dtmin=dtmin, dmax=dmax, units=units)

        self.source_type = source_type
        self.x0 = x0
        self.x1 = x1
        self.x2 = x2
        self.xa = xa
        self.freq = freq
        self.phi = phi
        self.duty = duty
        self.t1 = t1
```

```
        self.t2 = t2
        self.srcfunc = srcfunc
        self.gainfunc = gainfunc

        # source derived quantities:

        self.x = self.x0

        self.omega = 2.0 * pi * self.freq

        self.period = _INF
        if self.freq:
            self.period = 1.0 / self.freq

        if self.freq:
            self.T = 1.0 / self.freq

        if self.source_type == SourceType.RAMP:
            self.x0 = self.x1

        self.ramp_slope = 0.0
        if (self.t2 - self.t1) > 0:
            self.ramp_slope = (self.x2 - self.x1) / (self.t2 - self.t1)

    def dint(self, t):

        self.xprev = self.x

        if self.source_type == SourceType.FUNCTION:

            x = self.srcfunc(self.device, self.tlast)

        elif self.source_type == SourceType.CONSTANT:

            x = self.x0

        elif self.source_type == SourceType.STEP:

            if t < self.t1:
                x = self.x0
            else:
                x = self.x1

        elif self.source_type == SourceType.SINE:

            if t >= self.t1:
                x = self.x0 + self.xa * sin(self.omega * t + self.phi)
            else:
```

```
                x = self.x0

        elif self.source_type == SourceType.PWM:

            if self.duty <= 0.0:
                x = self.x2

            elif self.duty >= 1.0:
                x = self.x1

            else:

                w = t % self.period

                if ISCLOSE(w, self.period):
                    x = self.x1

                elif ISCLOSE(w, self.period * self.duty):
                    x = self.x2

                elif w < self.period * self.duty:
                    x = self.x1

                else:
                    x = self.x2

        elif self.source_type == SourceType.RAMP:

            if t <= self.t1:
                x = self.x1
            elif t <= self.t2:
                x = self.x1 + (t - self.t1) * self.dx
            else:
                x = self.x2

        elif self.source_type == SourceType.FUNCTION:

            x = self.srcfunc()

        if self.sys.enable_slewrate:
            if x > self.xprev:
                self.x = min(x, self.dmax * self.dq * (t - self.tlast) + self.xprev)
            elif x < self.u_prev:
                self.x = max(x, -self.dmax * self.dq * (t - self.tlast) + self.xprev)
        else:
            self.x = x

        if self.gainfunc:
```

```
735
736             k = self.gainfunc(self.device)
737             self.x *= k
738
739         return self.x
740
741     def quantize(self):
742
743         self.q = self.x
744         return False
745
746     def ta(self, t):
747
748         self.tnext = _INF
749
750         if self.source_type == SourceType.FUNCTION:
751
752             pass
753
754         if self.source_type == SourceType.RAMP:
755
756             if t < self.t1:
757                 self.tnext = self.t1
758
759             elif t < self.t2:
760                 if self.dx > 0.0:
761                     self.tnext = t + (self.q + self.dq - self.x) / self.dx
762                 elif self.dx < 0.0:
763                     self.tnext = t + (self.q - self.dq - self.x) / self.dx
764                 else:
765                     self.tnext = _INF
766
767             else:
768                 self.tnext = _INF
769
770         elif self.source_type == SourceType.STEP:
771
772             if t < self.t1:
773                 self.tnext = self.t1
774             else:
775                 self.tnext = _INF
776
777         elif self.source_type == SourceType.SINE:
778
779             if t < self.t1:
780
781                 self.tnext = self.t1
782
783             else:
```

```
784
785                 w = t % self.T              # cycle time
786                 t0 = t - w                  # cycle start time
787                 theta = self.omega * w + self.phi  # wrapped angular position
788
789                 # value at current time w/o dc offset:
790                 x = self.xa * sin(2.0 * pi * self.freq * t)
791
792                 # determine next transition time. Saturate at +/- xa:
793
794                 # quadrant I
795                 if theta < pi/2.0:
796                     self.tnext = (t0 + (asin(min(1.0, (x + self.dq) / self.xa)))
797                                   / self.omega)
798
799                 # quadrant II
800                 elif theta < pi:
801                     self.tnext = (t0 + self.T/2.0
802                                   - (asin(max(0.0, (x - self.dq) / self.xa)))
803                                   / self.omega)
804
805                 # quadrant III
806                 elif theta < 3.0*pi/2:
807                     self.tnext = (t0 + self.T/2.0
808                                   - (asin(max(-1.0, (x - self.dq) / self.xa)))
809                                   / self.omega)
810
811                 # quadrant IV
812                 else:
813                     self.tnext = (t0 + self.T
814                                   + (asin(min(0.0, (x + self.dq) / self.xa)))
815                                   / self.omega)
816
817         elif self.source_type == SourceType.PWM:
818
819             if self.duty <= 0.0 or self.duty >= 1.0:
820
821                 self.tnext = _INF
822
823             else:
824
825                 w = t % self.period
826
827                 if ISCLOSE(w, self.period):
828                     self.tnext = t + self.period * self.duty
829
```

```
                elif ISCLOSE(w, self.period * self.duty):
                    self.tnext = t + self.period - w

                elif w < self.period * self.duty:
                    self.tnext = t + self.period * self.duty - w

                else:
                    self.tnext = t + self.period - w


        elif self.source_type == SourceType.FUNCTION:

            pass
            #self.tnext = self.tlast + self.srcdt # <-- should we do this?

        #self.tnext = max(self.tnext, self.tlast + self.dtmin)

    def f(self, q, t):

        self.dx = 0.0

        if self.source_type == SourceType.RAMP:

            self.dx = self.ramp_slope

        elif self.source_type == SourceType.SINE:

            self.dx = self.omega * self.xa * cos(self.omega * t + self.phi)

        elif self.source_type == SourceType.STEP:

            pass  # todo: sigmoid approx?

        elif self.source_type == SourceType.PWM:

            pass  # todo: sigmoid approx?

        elif self.source_type == SourceType.FUNCTION:

            self.dx = 0.0  # todo: add a time derivative function delegate

        return self.dx

    def df(self, u, du, t):

        self.ddx = 0.0

        if self.source_type == SourceType.RAMP:

```

```
            self.ddx = 0.0

        elif self.source_type == SourceType.SINE:

            self.ddx = -self.omega**2 * self.xa * sin(self.omega * t + self.phi)

        elif self.source_type == SourceType.STEP:

            pass  # todo: sigmoid approx?

        elif self.source_type == SourceType.PWM:

            pass  # todo: sigmoid approx?

        elif self.source_type == SourceType.FUNCTION:

            self.ddx = 0.0  # todo: add a 2nd time derivative function delegate

        return self.ddx


class StateAtom(Atom):

    """ Qdl State Atom.
    """

    def __init__(self, name, x0=0.0, coefficient=0.0, coeffunc=None,
                 derfunc=None, der2func=None, dq=None, dqmin=None, dqmax=None, dqerr=None,
                 dtmin=None, dmax=1e10, units=""):

        Atom.__init__(self, name=name, x0=x0, dq=dq, dqmin=dqmin, dqmax=dqmax,
                      dqerr=dqerr, dtmin=dtmin, dmax=dmax, units=units)

        self.coefficient = coefficient
        self.coeffunc = coeffunc
        self.derfunc = derfunc
        self.der2func = der2func

    def dint(self, t):

        self.x += self.dx * (t - self.tlast)

        return self.x

    def quantize(self, t=0.0, implicit=True):

        interp = False
        change = False

```

```
        self.qlast = self.q

        if self.qss_method == QssMethod.QSS1:

            if self.x >= self.q + self.dq:

                self.q += self.dq

            elif self.x <= self.q - self.dq:

                self.q -= self.dq

        elif self.qss_method == QssMethod.QSS2:

            if self.x >= self.q + self.dq:

                self.q += self.dq

            elif self.x <= self.q - self.dq:

                self.q -= self.dq

            if self.dx >= self.qd + self.dq:

                self.qd += self.dq

            elif self.dx <= self.qd - self.dq:

                self.qd -= self.dq

        elif self.qss_method == QssMethod.LIQSS1:

            # save previous derivative so we can see if the sign has changed:

            self.dxlast = self.dx

            # determine if the current internal state x is outside of the band:

            if self.x >= self.qhi:

                self.q = self.qhi
                self.qlo += self.dq
                change = True

            elif self.x <= self.qlo:

                self.q = self.qlo
                self.qlo -= self.dq
                change = True
```

```
977
978             self.qhi = self.qlo + 2.0 * self.dq
979
980             if change and self.implicit and implicit:
981
982                 # we've ventured out of (qlo, qhi) bounds:
983
984                 self.dx = self.f(self.q)
985
986                 # if the derivative has changed signs, then we know
987                 # we are in a potential oscillating situation, so
988                 # we will set the q such that the derivative ~= 0:
989
990                 if (self.dx * self.dxlast) < 0:
991
992                     # derivative has changed sign:
993
994                     flo = self.f(self.qlo)
995                     fhi = self.f(self.qhi)
996                     if flo != fhi:
997                         a = (2.0 * self.dq) / (fhi - flo)
998                         self.q = self.qhi - a * fhi
999                         interp = True
1000
1001            return interp
1002
1003        elif self.qss_method == QssMethod.LIQSS2:
1004
1005            """
1006            Line number reference paper:
1007
1008            'Improving Linearly Implicit Quantized State System Methods'
1009
1010            (Algorithm 5 listing)
1011
1012            """
1013
1014            ex = t - self.tlast
1015
1016            # elapsed time since last qi update (10):
1017
1018            eq = t - self.tlastq
1019
1020            # store previous value of dxi/dt (8):
1021
1022            self.dxlast = self.dx
1023
1024            # affine coefficient projection (9):
```

```
            self.u += ex * self.du

            # store previous value of qi projected (11):

            self.qlast = self.q + eq * self.qd

            # h = MAX_2ND_ORDER_STEP_SIZE(xi) (12):

            h = self.max_2nd_order_stepsize()

            # 2ND_ORDER_STEP(x, h):

            qd = self.dx + h * self.ddx

            q = self.x + h * self.dx + h * h * self.ddx - h * self.q

            if self.q >= self.qhi:

                self.q = self.qhi
                self.qlo += self.dq

            elif self.q <= self.qlo:

                self.q = self.qlo
                self.qlo -= self.dq

            if self.qd >= self.qdhi:

                self.qd = self.qdhi
                self.qdlo += self.dq

            elif self.qd <= self.qdlo:

                self.qd = self.d
                self.qdlo -= self.dq

            self.qhi = self.qlo + 2.0 * self.dq
            self.qdhi = self.qdlo + 2.0 * self.dq

            return True

    def max_2nd_order_stepsize(self):

        h1, h2, h3, h4 = 0, 0, 0, 0

        den = self.a * self.qd + self.du - self.ddx

        if den == 0.0:
```

```
            return 0.0

        h1 = -(self.a * self.x + self.u - self.dx + self.a * self.dq) / den

        h2 = -(self.a * self.x + self.u - self.dx - self.a * self.dq) / den

        h3 = -(2 * self.a * self.x + 2 * self.u - 2 * self.dx + 2 * self.a * self.dq) / den

        h4 = -(2 * self.a * self.x + 2 * self.u - 2 * self.dx - 2 * self.a * self.dq) / den

        return min([h1, h2, h3, h4])

    def ta(self, t):

        if self.qss_method == QssMethod.QSS1:

            if self.dx > _EPS:
                self.tnext = t + (self.q + self.dq - self.x) / self.dx
            elif self.dx < -_EPS:
                self.tnext = t + (self.q - self.dq - self.x) / self.dx
            else:
                self.tnext = _INF

        elif self.qss_method == QssMethod.QSS2:

            ta1 = _INF
            ta2 = _INF

            if self.dx > _EPS:
                ta1 = t + (self.q + self.dq - self.x) / self.dx
            elif self.dx < -_EPS:
                ta1 = t + (self.q - self.dq - self.x) / self.dx

            if self.ddx > _EPS:
                ta2 = t + (self.qd + self.dq - self.dx) / self.ddx
            elif self.ddx < -_EPS:
                ta2 = t + (self.qd - self.dq - self.dx) / self.ddx

            self.tnext = min(ta1, ta2)

        elif self.qss_method == QssMethod.LIQSS1:

            if self.dx > _EPS:
                self.tnext = t + (self.qhi - self.x) / self.dx
            elif self.dx < -_EPS:
                self.tnext = t + (self.qlo - self.x) / self.dx
```

```
            else:
                self.tnext = _INF

        elif self.qss_method == QssMethod.LIQSS2:

            ta1 = _INF
            ta2 = _INF

            if self.dx > _EPS:
                ta1 = t + (self.qhi - self.x) / self.dx
            elif self.dx < -_EPS:
                ta1 = t + (self.qlo - self.x) / self.dx
            else:
                self.tnext = _INF

            if self.ddx > _EPS:
                ta2 = t + (self.qdhi - self.dx) / self.ddx
            elif self.ddx < -_EPS:
                ta2 = t + (self.qdlo - self.dx) / self.ddx

            self.tnext = min(ta1, ta2)

        self.tnext = max(self.tnext, t + self.dtmin)

    def compute_coefficient(self):

        if self.coeffunc:
            return self.coeffunc(self.device)
        else:
            return self.coefficient

    def f(self, q, t=0.0):

        if self.derfunc:
            if self.derargfunc:
                args = self.derargfunc(self.device)
                return self.derfunc(*args)
            else:
                return self.derfunc(self.device, q)

        d = self.compute_coefficient() * q

        for connection in self.connections:
            d += connection.value()

        return d

    def df(self, x, d, t=0.0):
```

```
1169
1170         if self.derfunc:
1171             return self.der2func(self.device, x, d)
1172
1173         d2 = self.compute_coefficient() * d
1174
1175         for connection in self.connections:
1176             d2 += connection.value()
1177
1178         return d2
1179
1180     def set_state(self, x):
1181
1182         self.x = x
1183         self.q = x
1184         self.qlo = x - self.dq
1185         self.qhi = x + self.dq
1186
1187
1188 class System(object):
1189
1190     def __init__(self, name="sys", qss_method=QssMethod.LIQSS1, dq=None, dqmin=None,
1191                  dqmax=None, dqerr=None, dtmin=None, dmax=None):
1192
1193         global sys
1194         sys = self
1195
1196         self.name = name
1197         self.qss_method = qss_method
1198
1199         self.dtmin = DEF_DTMIN
1200         if dtmin:
1201             self.dtmin = dtmin
1202
1203         self.dmax = DEF_DMAX
1204         if dmax:
1205             self.dmax = dmax
1206
1207         # child elements:
1208
1209         self.devices = []
1210         self.atoms = []
1211         self.state_atoms = []
1212         self.source_atoms = []
1213         self.n = 0
1214         self.m = 0
1215
1216         # simulation variables:
```

```
1217
1218            self.tstart = 0.0  # start simulation time
1219            self.tstop = 0.0  # end simulation time
1220            self.time = 0.0   # current simulation time
1221            self.tsave = 0.0  # saved time for state restore
1222            self.iprint = 0   # for runtime updates
1223            self.dt = 1e-4
1224            self.enable_slewrate = False
1225            self.jacobian = None
1226            self.Km = 2.0
1227
1228            self.savedt = 0.0   # max dt for saving data. Save all if zero
1229
1230            self.tocsv = False  # send data to csv files
1231            self.outdir = None  # send data to csv files
1232
1233            # events:
1234
1235            self.events = {}
1236
1237            # memory management:
1238            self.storage_type = StorageType.LIST
1239
1240            self.show_time = False
1241            self.time_queue = queue.Queue()
1242
1243        def schedule(self, func, t):
1244
1245            if not t in self.events:
1246                self.events[t] = []
1247
1248            self.events[t].append(func)
1249
1250        def add_device(self, device):
1251
1252            self.devices.append(device)
1253
1254            for atom in device.atoms:
1255
1256                if not atom.dq:
1257                    atom.dq = self.dq
1258
1259                #if not atom.dqmin:
1260                #    atom.dqmin = self.dqmin
1261                #
1262                #if not atom.dqmax:
1263                #    atom.dqmax = self.dqmax
1264                #
```

```
            #if not atom.dqerr:
            #    atom.dqerr = self.dqerr

            if not atom.dtmin:
                atom.dtmin = self.dtmin

            if not atom.dmax:
                atom.dmax = self.dmax

            if not atom.qss_method:
                atom.qss_method = self.qss_method

            atom.device = device
            atom.sys = self

            self.atoms.append(atom)

            if isinstance(atom, StateAtom):
                atom.index = self.n
                self.state_atoms.append(atom)
                self.n += 1

            elif isinstance(atom, SourceAtom):
                atom.index = self.m
                self.source_atoms.append(atom)
                self.m += 1

        setattr(self, device.name, device)

    def add_devices(self, *devices):

        for device in devices:
            device.setup_connections()

        for device in devices:
            device.setup_functions()

        for device in devices:
            self.add_device(device)

    def save_state(self):

        self.tsave = self.time

        for atom in self.atoms:
            atom.qsave = atom.q
            atom.xsave = atom.x
```

```
    def clear_data_arrays(self):

        for atom in self.atoms:
            atom.clear_data_arrays()

    def state_to_file(self, path):

        state = {}

        state["time"] = self.time
        state["tsave"] = self.tsave
        state["state_atoms"] = {}
        state["source_atoms"] = {}

        for atom in self.state_atoms:
            state["state_atoms"][atom.index] = atom.state_to_dict()

        for atom in self.source_atoms:
            state["source_atoms"][atom.index] = atom.state_to_dict()

        with open(path, "wb") as f:
            pickle.dump(state, f)

    def state_from_file(self, path):

        with open(path, "rb") as f:
            state = pickle.load(f)

        self.time = state["time"]
        self.tsave = state["tsave"]

        for atom in self.state_atoms:
            atom.state_from_dict(state["state_atoms"][atom.index])

        for atom in self.source_atoms:
            atom.state_from_dict(state["source_atoms"][atom.index])

    def connect(self, from_port, to_port):

        from_port.connect(to_port)

    def restore_state(self):

        self.time = self.tsave

        for atom in self.atoms:
            atom.q = atom.qsave
```

```
            atom.x = atom.xsave

            atom.qhi = atom.q + atom.dq
            atom.qlo = atom.q - atom.dq

    def get_jacobian(self):

        jacobian = np.zeros((self.n, self.n))

        for atom in self.state_atoms:
            for other, func in atom.jacfuncs:
                if atom.derargfunc:
                    args = atom.derargfunc(atom.device)
                    jacobian[atom.index, other.index] = func(*args)
                else:
                    if atom is other:
                        jacobian[atom.index, other.index] = func(atom.device, atom.q)
                    else:
                        jacobian[atom.index, other.index] = func(atom.device, atom.q, other.index)

        return jacobian

    @staticmethod
    def fode(t, x, sys):

        """Returns array of derivatives from state atoms. This function must be
        a static method in order to be passed as a delgate to the
        scipy ode integrator function. Note that sys is a global module variable.
        """

        dx_dt = [0.0] * sys.n

        for atom in sys.state_atoms:
            atom.q = x[atom.index]

        for atom in sys.state_atoms:
            dx_dt[atom.index] = atom.f(atom.q, t)

        return dx_dt

    @staticmethod
    def fode2(x, t=0.0, sys=None):

        """Returns array of derivatives from state atoms. This function must be
        a static method in order to be passed as a delgate to the
        scipy ode integrator function. Note that sys is a global module variable.
        (not that this function differs from self.fode in the argument order).
```

```
        """

        dx_dt = [0.0] * sys.n

        for atom in sys.state_atoms:
            atom.q = x[atom.index]

        for atom in sys.state_atoms:
            dx_dt[atom.index] = atom.f(atom.q, t)

        return dx_dt

    def solve_dc(self, init=True, set=True):

        xi = [0.0]*self.n

        for atom in self.state_atoms:
            if init:
                xi[atom.index] = atom.x0
            else:
                xi[atom.index] = atom.x

        xdc = fsolve(self.fode2, xi, args=(0, sys), xtol=1e-12)

        for atom in self.state_atoms:
            if init:
                atom.x0 = xdc[atom.index]
            elif set:
                atom.x = xdc[atom.index]
                atom.q = atom.x

        return xdc

    def initialize(self, t0=0.0, dt=1e-4, dc=False, savedt=0.0, tocsv=False, outdir=None):

        self.time = t0
        self.dt = dt

        self.savedt = savedt

        self.tocsv = tocsv
        self.outdir = outdir

        self.dq0 = np.zeros((self.n, 1))

        for atom in self.state_atoms:
            self.dq0[atom.index] = atom.dq0

```

```
        if dc:
            self.solve_dc()

        for atom in self.state_atoms:
            atom.initialize(self.time)

        for atom in self.source_atoms:
            atom.initialize(self.time)

    def clear_data_arrays(self):

        for atom in self.atoms:
            atom.clear_data_arrays()

    def print_time(self):

        while self.show_time:
            time.sleep(1.0)
            print(simtime)

    def run(self, tstop, ode=True, qss=True, verbose=2, qss_fixed_dt=None,
            ode_method="RK45", optimize_dq=False, chk_ss_delay=None):

        self.tstart = self.time
        self.tstop = tstop

        self.verbose = verbose
        self.calc_ss = False
        self.chk_ss_delay = chk_ss_delay

        #self.show_time = True
        #self.print_thread = th.Thread(target=self.print_time)
        #self.print_thread.start()

        if optimize_dq or self.chk_ss_delay:
            self.calc_ss = True
            self.update_steadystate_distance()

        self.ode_method = ode_method

        # add the 'tstop' or end of simulation event to the list:

        self.events[self.tstop] = None  # no function to call at tstop event

        ran_events = []

        # get the event times and event function lists, sorted by time:

```

```
        sorted_events = sorted(self.events.items())

        # loop through the event times and solve:

        print("Simulation started......")

        start_time = time.time()

        for event_time, events in sorted_events:

            if self.calc_ss:
                self.calc_steadystate()

            if optimize_dq:
                self.optimize_dq()
                self.update_steadystate_distance()

            if ode:

                print("ODE solution started to next event...")

                if qss: self.save_state()

                xi = [0.0]*self.n
                for atom in self.state_atoms:
                    xi[atom.index] = atom.x

                tspan = (self.time, event_time)

                soln = solve_ivp(self.fode, tspan, xi, ode_method, args=(sys,),
                                 max_step=self.dt)
                t = soln.t
                x = soln.y

                for i in range(len(t)):

                    for atom in self.state_atoms:
                        atom.q = x[atom.index, i]
                        atom.save_ode(t[i], atom.q)

                    for atom in self.source_atoms:
                        atom.save_ode(t[i], atom.q)
                        atom.dint(t[i])

                for atom in self.state_atoms:
                    xf = x[atom.index, -1]
                    atom.x = xf
```

```
                    atom.q = xf

                print("ODE solution completed to next event.")

            if qss:

                print("QSS solution started to next event...")

                if ode: self.restore_state()

                if self.qss_method == QssMethod.QSS1:

                    self.run_qss1()

                elif self.qss_method == QssMethod.QSS2:

                    self.run_qss2()

                elif self.qss_method == QssMethod.LIQSS1:

                    self.run_liqss1(event_time)

                elif self.qss_method == QssMethod.LIQSS2:

                    self.run_liqss2()

                print("QSS solution completed to next event.")

            if events:

                for event in events:
                    event(self)

                ran_events.append(event_time)

            self.time = event_time

            if self.time >= self.tstop:
                break  # (do not go to any more event, tstop has been reached)

        # remove run events:

        for eventtime in ran_events:
            del self.events[eventtime]

        del self.events[self.tstop]

        #self.show_time = False
```

```
        #self.print_thread.join()

        print(f"Simulation complete. Total run time = {time.time() - start_time} s.")

    def run_qss1(self):

        t = self.time

        for atom in self.atoms:
            atom.dx = atom.f(atom.q, t)
            atom.ta(t)
            atom.tlast = t
            atom.save_qss(t)

        while(t < self.tstop):                      # 1

            if self.verbose == 2: print(t)

            atomi = None
            t = self.tstop

            for atom in self.atoms:
                if atom.tnext <= t:
                    t = atom.tnext                  # 2
                    atomi = atom                    # 3

            if not atomi: break

            e = t - atomi.tlast                     # 4
            atomi.x = atomi.x + atomi.dx * e        # 5
            atomi.quantize()                        # 6
            atomi.ta(t)                             # 7

            for atomj in atomi.broadcast_to:        # 8

                e = t - atomj.tlast                 # 9
                atomj.x = atomj.x + atomj.dx * e    # 10

                atomj.dx = atomj.f(atomj.q, t)      # 12
                atomj.ta(t)                         # 13

                if atomj is not atomi:
                    atomj.tlast = t                 # 11

            atomi.tlast = t                         # 13
            atomi.save_qss(t)

        for atom in self.atoms:
```

```
            atom.tlast = t
            atom.save_qss(t, force=True)

        self.time = self.tstop

    def run_qss2(self):

        t = self.time

        if t == self.tstart:
            for atom in self.atoms:
                atom.dx = atom.f(atom.q, t)
                atom.ddx = atom.df(atom.q, atom.dx, t)
                atom.ta(t)
                atom.tlast = t
                atom.save_qss(t)

        while(t < self.tstop):                                  # 1

            if self.verbose == 2: print(t)

            atomi = None
            t = self.tstop

            for atom in self.atoms:
                if atom.tnext <= t:
                    t = atom.tnext                              # 2
                    atomi = atom                                # 3

            if not atomi: break

            e = t - atomi.tlast                                 # 4
            ee = e**2

            atomi.x += atomi.dx * e + 0.5 * atomi.ddx * ee      # 6
            atomi.dx += atomi.ddx * e                           # 7

            if atomi.q >= atomi.q + atomi.dq:
                atomi.q += atomi.dq

            elif atomi.q <= atomi.q - atomi.dq:
                atomi.q -= atomi.dq

            atomi.qd = atomi.dx

            # 11:
```

```
            ta1 = _INF
            ta2 = _INF

            if atomi.dx > _EPS:
                ta1 = t + (atomi.q + atomi.dq - atomi.x) / atomi.dx
            elif atomi.dx < -_EPS:
                ta1 = t + (atomi.q - atomi.dq - atomi.x) / atomi.dx

            if atomi.ddx > _EPS:
                ta2 = t + (atomi.qd + atomi.dq - atomi.dx) / atomi.ddx
            elif atomi.ddx < -_EPS:
                ta2 = t + (atomi.qd - atomi.dq - atomi.dx) / atomi.ddx

            self.tnext = max(t, min(ta1, ta2))

            for atomj in atomi.broadcast_to:                    # 12

                e = t - atomj.tlast                             # 13
                ee = e**2
                atomj.x += atomj.dx * e + 0.5 * atomj.ddx * ee # 15

                atomj.dx = atomj.f(atomj.q, t)                  # 16
                atomj.ddx = atomj.df(atomj.q, atomj.dx, t)      # 17

                # 18:

                ta1 = _INF
                ta2 = _INF

                if atomj.dx > _EPS:
                    ta1 = t + (atomj.q + atomj.dq - atomj.x) / atomj.dx
                elif atomj.dx < -_EPS:
                    ta1 = t + (atomj.q - atomj.dq - atomj.x) / atomj.dx

                if atomi.ddx > _EPS:
                    ta2 = t + (atomj.qd + atomj.dq - atomj.dx) / atomj.ddx
                elif atomi.ddx < -_EPS:
                    ta2 = t + (atomj.qd - atomj.dq - atomj.dx) / atomj.ddx

                atomj.tnext = max(t, min(ta1, ta2))

                if atomj is not atomi:
                    atomj.tlast = t                             # 19

            atomi.tlast = t                                     # 21
            atomi.save_qss(t)
```

```
        for atom in self.atoms:

            atom.tlast = t
            atom.save_qss(t, force=True)

        self.time = self.tstop

    def run_liqss1(self, tnext):

        #global simtime

        t = self.time

        i = 0
        n = 1e6

        ss_clock = 0.0

        for atom in self.atoms:
            atom.dx = atom.f(atom.q, t)
            atom.ta(t)
            atom.tlast = t
            atom.save_qss(t)

        while(t < tnext and t < self.tstop):        # 1

            if self.verbose == 2: print(t)

            atomi = None
            t = tnext

            for atom in self.atoms:
                if atom.tnext <= t:
                    t = atom.tnext                  # 2
                    atomi = atom                    # 3

            if not atomi: break

            e = t - atomi.tlast                     # 4
            atomi.x = atomi.x + atomi.dx * e        # 5
            atomi.quantize()                        # 6
            atomi.ta(t)                             # 7

            for atomj in atomi.broadcast_to:        # 8

                e = t - atomj.tlast                 # 9
                atomj.x = atomj.x + atomj.dx * e    # 10
```

```
                if atomj is not atomi:
                    atomj.tlast = t                 # 11

                atomj.dx = atomj.f(atomj.q, t)      # 12
                atomj.ta(t)                         # 13

            atomi.tlast = t                         # 13
            atomi.save_qss(t)

            #simtime = t

            i = i + 1
            if i > n:
                i = 0
                print(t)

            if self.chk_ss_delay:
                is_ss = self.check_steadystate(self.time)
                if is_ss:
                    tnext = self.tstop


        self.time = tnext

        for atom in self.atoms:
            atom.tlast = t
            atom.save_qss(t, force=True)

    def run_liqss1_test(self):

        t = self.time

        for atom in self.atoms:

            atom.dxlast = atom.dx
            atom.dx = atom.f(atom.q, t)
            atom.a = (atom.dx - atom.dxlast) / (2 * atom.dq)
            atom.u = atom.dx - atom.a * atom.q
            atom.tlast = t
            atom.ta(t)
            atom.save_qss(t)

        while(t < self.tstop):                      # 1

            if self.verbose == 2: print(t)

            atomi = None
            t = self.tstop
```

```
            for atom in self.atoms:
                if atom.tnext <= t:
                    t = atom.tnext                  # 2
                    atomi = atom                    # 3

            if not atomi: break

            e = t - atomi.tlast                     # 4

            atomi.x = atomi.x + atomi.dx * e        # 5

            atomi.qlast = atomi.q                   # 6

            atomi.dxlast = atomi.dx                 # 7

            dx_sign = 0
            if atomi.dx > 0: dx_sign = 1
            elif atomi.dx < 0: dx_sign = -1

            dx_plus = atomi.a * (atomi.x + dx_sign * atomi.dq) + atomi.u   # 8

            if atomi.x >= atomi.qhi:

                atomi.q = atomi.qhi
                atomi.qlo += atomi.dq

            elif atomi.x <= atomi.qlo:

                atomi.q = atomi.qlo
                atomi.qlo -= atomi.dq

            if atomi.dx * dx_sign < 0:

                atomi.q = -atomi.u / atomi.a

            atomi.qhi = atomi.qlo + 2.0 * atomi.dq

            # 7:

            if atomi.dx > _EPS:
                ta = t + (atomi.qhi - atomi.x) / atomi.dx
            elif atomi.dx < -_EPS:
                ta = t + (atomi.qlo - atomi.x) / atomi.dx
            else:
                ta = _INF

            atomi.tnext = max(t, ta)
```

```
            for atomj in atomi.broadcast_to:        # 8

                e = t - atomj.tlast                 # 9
                atomj.x = atomj.x + atomj.dx * e    # 10

                if atomj is not atomi:
                    atomj.tlast = t                 # 11

                atomj.dx = atomj.f(atomj.q, t)      # 12

                # 13:

                if atomj.dx > _EPS:
                    ta = t + (atomj.qhi - atomj.x) / atomj.dx
                elif atomj.dx < -_EPS:
                    ta = t + (atomj.qlo - atomj.x) / atomj.dx
                else:
                    ta = _INF

                atomj.tnext = max(t, ta)

            atomi.a = (atomi.dx - atomi.dxlast) / (2 * atomi.dq)  # 23

            atomi.u = atomi.dx - atomi.a * atomi.q  # 24

            atomi.tlast = t                         # 25

            atomi.save_qss(t)

        for atom in self.atoms:

            atom.tlast = t
            atom.save_qss(t, force=True)

        self.time = self.tstop

    def run_liqss2(self):

        t = self.time

        for atom in self.atoms:

            atom.dx = atom.f(atom.q, t)
            atom.ddx = atom.df(atom.dx, atom.q, t)

            atom.a = 0.0
            atom.u = atom.dx
            atom.du = atom.ddx
```

```
            atom.ta(t)
            atom.tlast = t
            atom.tlastq = t

            atom.save_qss(t)

        while(t < self.tstop):                      # 1

            if self.verbose == 2: print(t)

            atomi = None
            t = self.tstop

            for atom in self.atoms:
                if atom.tnext <= t:
                    t = atom.tnext                  # 2
                    atomi = atom                    # 3

            if not atomi: break

            e = t - atomi.tlast                     # 4
            ee = e**2

            atomi.x += atomi.dx * e + 0.5 * atomi.ddx * ee  # 6
            atomi.dx += atomi.ddx * e                       # 7

            atomi.quantize(t)                       # 8-13

            atomi.tlastq = t                        # 14

            atomi.ta(t)                             # 15

            for atomj in atomi.broadcast_to:        # 16

                e = t - atomj.tlast                 # 17-18

                atomj.x += atomj.dx * e + 0.5 * atomj.ddx * ee  # 18

                atomj.dx = atomj.f(atomj.q, t)               # 16

                atomj.ddx = atomj.df(atomj.q, atomj.dx, t)   # 17

                atomj.ta(t)                                  # 18

                if atomj is not atomi:
                    atomj.tlast = t                 # 23-25

            if atomi.q != atomi.qlast:
```

```
                atomi.a = (atomi.dx - atomi.dxlast) / (atomi.q - atomi.qlast)  # 26

                atomi.u = atomi.dx - atomi.a * atomi.q        # 27

                atomi.du = atomi.ddx - atomi.a * atomi.qd     # 28

            atomi.tlast = t

            atomi.save_qss(t)

        for atom in self.atoms:

            atom.tlast = t
            atom.save_qss(t, force=True)

        self.time = self.tstop

    def get_next(self):

        # Get next time and flag atoms to solve:

        tnext = _INF
        anext = None

        for atom in self.atoms:
            tnext = min(atom.tnext, tnext)
            anext = atom

        # limit by minimum time step:

        #tnext = max(tnext, self.time + _EPS)

        # limit by end of simulation section (to next scheduled event):

        tnext = min(tnext, self.tstop)

        return anext, tnext

    def advance(self):

        tnext = _INF

        for atom in self.atoms:
            tnext = min(atom.tnext, tnext)

        self.time = max(tnext, self.time + _EPS)
        self.time = min(self.time, self.tstop)
```

```
        if 0:  # method 1: all < tnext atoms w/ nested triggered (one iter)

            for atom in self.atoms:

                if atom.tnext <= self.time:

                    atom.update(self.time)

                    for atom in self.atoms:

                        if atom.triggered:

                            atom.update(self.time)

        if 0:  # method 2: all < tnext atoms then triggered (one iter)

            for atom in self.atoms:

                if atom.tnext <= self.time:

                    atom.update(self.time)

            for atom in self.atoms:

                if atom.triggered:

                    atom.update(self.time)


        if 0:  # method 3: all < tnext atoms w/ nested triggered (multi-iter)

            for atom in self.atoms:

                if atom.tnext <= self.time:

                    atom.update(self.time)

                    i = 0
                    while i < _MAXITER:
                        triggered = False
                        for atom in self.atoms:
                            if atom.triggered:
                                triggered = True
                                atom.update(self.time)
                        if not triggered:
                            break
                        i += 1

        if 0:  # method 4: first tnext atoms, then iterate over triggered:
```

```
2081
2082                for atom in self.atoms:
2083                    if atom.tnext <= self.time:
2084                        atom.update(self.time)
2085
2086                i = 0
2087                while i < _MAXITER:
2088                    triggered = False
2089                    for atom in self.atoms:
2090                        if atom.triggered:
2091                            triggered = True
2092                            atom.update(self.time)
2093                    if not triggered:
2094                        break
2095                    i += 1
2096
2097            if 1:  # method 5: full outer loop iterations with tnext triggered:
2098
2099                i = 0
2100                while i < _MAXITER:
2101                    triggered1 = False
2102                    for atom in self.atoms:
2103                        if atom.tnext <= self.time:
2104                            atom.update(self.time)
2105                            triggered1 = True
2106                        j = 0
2107                        while j < _MAXITER:
2108                            triggered2 = False
2109                            for atom in self.atoms:
2110                                if atom.triggered:
2111                                    triggered2 = True
2112                                    atom.update(self.time)
2113                            if not triggered2:
2114                                break
2115                            j += 1
2116                    i += 1
2117
2118        def calc_steadystate(self):
2119
2120            self.jac1 = self.get_jacobian()
2121
2122            self.save_state()
2123
2124            self.xf = self.solve_dc(init=False, set=False)
2125
2126            for atom in self.state_atoms:
2127                atom.xf = self.xf[atom.index]
```

```
2128
2129        self.jac2 = self.get_jacobian()
2130
2131        self.restore_state()
2132
2133    def print_natural_frequencies(self):
2134
2135        eigvals, eigvecs = eig(self.get_jacobian())
2136
2137        for eigval in eigvals:
2138            print(f"{csqrt(eigval).real/(2*pi):10.6f} Hz")
2139
2140        for eigvec in eigvecs:
2141            print(eigvec)
2142
2143    def print_states(self):
2144
2145        for atom in self.atoms:
2146            print(atom.x)
2147        print()
2148
2149    def update_steadystate_distance(self):
2150
2151       dq0 = [0.0]*self.n
2152       for atom in self.state_atoms:
2153           dq0[atom.index] = atom.dq0
2154
2155       self.steadystate_distance = la.norm(dq0) * self.Km
2156
2157    def optimize_dq(self):
2158
2159        if self.verbose:
2160            print("dq0 = {}\n".format(self.dq0))
2161            print("jac1 = {}\n".format(self.jac1))
2162
2163        if 1:
2164
2165            QQ0 = np.square(self.dq0)
2166
2167            JTJ = self.jac1.transpose().dot(self.jac1)
2168            QQ = la.solve(JTJ, QQ0)
2169            dq1 = np.sqrt(np.abs(QQ))
2170
2171            JTJ = self.jac2.transpose().dot(self.jac1)
2172            QQ = la.solve(JTJ, QQ0)
2173            dq2 = np.sqrt(np.abs(QQ))
```

```
        if 0:

            E = np.zeros((self.n, self.n))

            dq1 = np.zeros((self.n, 1))
            dq2 = np.zeros((self.n, 1))

            for atom in self.state_atoms:
                for j in range(self.n):
                    if atom.index == j:
                        E[atom.index, atom.index] = (atom.dq0*factor)**2
                    else:
                        pass
                        E[atom.index, j] = (atom.dq0*factor)

            JTJ = self.jac1.transpose().dot(self.jac1)
            Q = la.solve(JTJ, E)

            for atom in self.state_atoms:
                dq = 999999.9
                for j in range(self.n):
                    if atom.index == j:
                       dqii = sqrt(abs(Q[atom.index, j]))
                       dqii = abs(Q[atom.index, j])
                       if dqii < dq:
                           dq = dqii
                    else:
                       dqij = abs(Q[atom.index, j])
                       if dqij < dq:
                           dq = dqij
                dq1[atom.index, 0] = dq

            JTJ = self.jac2.transpose().dot(self.jac2)
            Q = la.solve(JTJ, E)

            for atom in self.state_atoms:
                dq = 999999.9
                for j in range(self.n):
                    if atom.index == j:
                        dqii = sqrt(abs(Q[atom.index, j]))
                        dqii = abs(Q[atom.index, j])
                        if dqii < dq:
                            dq = dqii
                    else:
                        dqij = abs(Q[atom.index, j])
```

```
                        if dqij < dq:
                            dq = dqij
                dq2[atom.index, 0] = dq

        if self.verbose:
            print("at t=inf:")
            print("dq1 = {}\n".format(dq1))
            print("at t=0+:")
            print("dq2 = {}\n".format(dq2))

        for atom in self.state_atoms:

            atom.dq = min(atom.dq0, dq1[atom.index, 0], dq2[atom.index, 0])
            #atom.dq = min(dq1[atom.index, 0], dq2[atom.index, 0]) * 0.5

            atom.qhi = atom.q + atom.dq
            atom.qlo = atom.q - atom.dq

            if self.verbose:
                print("dq_{} = {} ({})\n".format(atom.full_name(), atom.dq, atom.units))

    def check_steadystate(self, t, apply_if_true=True):

        is_ss = False

        q = [0.0]*self.n
        for atom in self.state_atoms:
            q[atom.index] = atom.q

        qe = la.norm(np.add(q, -self.xf))

        if (qe < self.steadystate_distance):
            is_ss = True

        if is_ss and apply_if_true:

            for atom in self.state_atoms:
                atom.set_state(self.xf[atom.index])

            for atom in self.source_atoms:
                atom.dint(t)
                atom.q = atom.x

        return is_ss


class Device(object):
```

```
2267
2268     """Collection of Atoms and Connections that comprise a device
2269     """
2270
2271     def __init__(self, name):
2272
2273         self.name = name
2274         self.atoms = []
2275         self.ports = []
2276
2277     def add_atom(self, atom):
2278
2279         self.atoms.append(atom)
2280         atom.device = self
2281         setattr(self, atom.name, atom)
2282
2283     def add_atoms(self, *atoms):
2284
2285         for atom in atoms:
2286             self.add_atom(atom)
2287
2288     def add_port(self, name, typ, *connections):
2289
2290         port = Port(name, typ, *connections)
2291         self.ports.append(port)
2292         port.device = self
2293         setattr(self, name, port)
2294
2295     def setup_connections(self):
2296
2297         pass
2298
2299     def setup_functions(self):
2300
2301         pass
2302
2303     def __repr__(self):
2304
2305         return self.name
2306
2307     def __str__(self):
2308
2309         return __repr__(self)
2310
2311
2312 class Connection(object):
2313
2314     """Connection between atoms.
```

```
    """

    def __init__(self, atom=None, other=None, coefficient=1.0, coeffunc=None,
                 valfunc=None, dvalfunc=None):

        self.atom = atom
        self.other = other
        self.coefficient = coefficient
        self.coeffunc = coeffunc
        self.valfunc = valfunc
        self.dvalfunc = dvalfunc

        self.device = None

        if atom and other:
            self.reset_atoms(atom, other)

    def reset_atoms(self, atom, other):

        self.atom = atom
        self.other = other

        self.other.broadcast_to.append(self.atom)

    def compute_coefficient(self):

        if self.coeffunc:
            return self.coeffunc(self.device)
        else:
            return self.coefficient

    def value(self):

        if self.other:

            if self.valfunc:

                return self.valfunc(self.other)

            else:

                return self.compute_coefficient() * self.other.q

                #if isinstance(self.other, StateAtom):
                #    return self.compute_coefficient() * self.other.q
                #
                #elif isinstance(self.other, SourceAtom):
                #    return self.compute_coefficient() * self.other.dint()
```

```
        else:

            return 0.0

    def dvalue(self):

        if self.other:

            if self.dvalfunc:

                return self.dvalfunc(self.other)

            else:

                return self.compute_coefficient() * self.other.dx

                #if isinstance(self.other, StateAtom):
                #    return self.compute_coefficient() * self.other.dx
                #
                #elif isinstance(self.other, SourceAtom):
                #    return self.compute_coefficient() * self.other.f(self.other.q)
        else:

            return 0.0


class PortConnection(object):

    def __init__(self, invar, outvar, state=None, sign=1, expr=""):

        self.invar = invar
        self.outvar = outvar
        self.state = state
        self.sign = sign
        self.expr = expr
        self.from_connections = []
        self.port = None


class Port(object):

    def __init__(self, name, typ="in", *connections):

        self.name = name
        self.typ = typ
        self.device = None

        if connections:
```

```
            self.connections = connections
            for connection in self.connections:
                connection.port = self
        else:
            self.connections = []

    def connect(self, other):

        if self.typ == "in":
            self.connections[0].from_connections.append(other.connections[0])

        elif self.typ == "out":
            other.connections[0].from_connections.append(self.connections[0])

        elif self.typ in ("inout"):
            self.connections[0].from_connections.append(other.connections[0])
            other.connections[0].from_connections.append(self.connections[0])

        elif self.typ in ("dq"):
            self.connections[0].from_connections.append(other.connections[0])
            other.connections[0].from_connections.append(self.connections[0])
            self.connections[1].from_connections.append(other.connections[1])
            other.connections[1].from_connections.append(self.connections[1])

        elif self.typ in ("abc"):
            self.connections[0].from_connections.append(other.connections[0])
            other.connections[0].from_connections.append(self.connections[0])
            self.connections[1].from_connections.append(other.connections[1])
            other.connections[1].from_connections.append(self.connections[1])
            self.connections[2].from_connections.append(other.connections[2])
            other.connections[2].from_connections.append(self.connections[2])


class SymbolicDevice(Device):

    def __init__(self, name):

        Device.__init__(self, name)

        self.states = odict()
        self.constants = odict()
        self.parameters = odict()
        self.algebraic = odict()
        self.diffeq = []
        self.dermap = odict()
```

```
        self.jacobian = odict()

    def add_state(self, name, dername, desc="", units="", x0=0.0, dq=1e-3):

        self.states[name] = odict()

        self.states[name]["name"] = name
        self.states[name]["dername"] = dername
        self.states[name]["desc"] = desc
        self.states[name]["units"] = units
        self.states[name]["x0"] = x0
        self.states[name]["dq"] = dq
        self.states[name]["device"] = self

        self.states[name]["sym"] = None
        self.states[name]["dersym"] = None
        self.states[name]["expr"] = None
        self.states[name]["atom"] = None

        self.dermap[dername] = name

    def add_constant(self, name, desc="", units="", value=None):

        self.constants[name] = odict()

        self.constants[name]["name"] = name
        self.constants[name]["desc"] = desc
        self.constants[name]["units"] = units
        self.constants[name]["value"] = value

        self.constants[name]["sym"] = None

    def add_parameter(self, name, desc="", units="", value=None):

        self.parameters[name] = odict()

        self.parameters[name]["name"] = name
        self.parameters[name]["desc"] = desc
        self.parameters[name]["units"] = units
        self.parameters[name]["value"] = value

        self.parameters[name]["sym"] = None

    def add_diffeq(self, equation):

        self.diffeq.append(equation)
```

```
    def add_algebraic(self, var, rhs):

        self.algebraic[var] = rhs

    def update_parameter(self, key, value):

        self.parameters[key]["value"] = value

    def add_input_port(self, name, var, sign=1):

        connection = PortConnection(var, sign=sign)
        self.add_port(name, "in", connection)

    def add_output_port(self, name, var=None, state=None, expr=""):

        connection = PortConnection(var, state=state, sign=sign, expr=expr)
        self.add_port(name, "out", connection)

    def add_electrical_port(self, name, input, output, sign=1, expr=""):

        connection = PortConnection(input, output, sign=sign, expr=expr)
        self.add_port(name, "inout", connection)

    def add_dq_port(self, name, inputs, outputs, sign=1, exprs=None):

        expr_d = ""
        expr_q = ""

        if exprs:
            expr_d, expr_q = exprs

        connection_d = PortConnection(inputs[0], outputs[0], sign=sign, expr=expr_d)
        connection_q = PortConnection(inputs[1], outputs[1], sign=sign, expr=expr_q)

        self.add_port(name, "dq", connection_d, connection_q)

    def setup_connections(self):

        for name, state in self.states.items():

            atom = StateAtom(name, x0=state["x0"], dq=state["dq"],
                             units=state["units"])

            atom.derargfunc = self.get_args

            self.add_atom(atom)
```

```
            self.states[name]["atom"] = atom

    def setup_functions(self):

        # 1. create sympy symbols:

        x = []
        dx_dt = []

        for name, state in self.states.items():

            sym = sp.Symbol(name)
            dersym = sp.Symbol(state["dername"])

            x.append(name)
            dx_dt.append(state["dername"])

            self.states[name]["sym"] = sym
            self.states[name]["dersym"] = dersym

        for name in self.constants:
            sp.Symbol(name)

        for name in self.parameters:
            sp.Symbol(name)

        for port in self.ports:
            for connection in port.connections:
                sp.Symbol(connection.invar)

        for var in self.algebraic:
            sp.Symbol(var)

        # 2. create symbolic derivative expressions:

        # 2a. substitute algebraic equations:

        n = len(self.algebraic)
        m = len(self.diffeq)

        algebraic = [[sp.Symbol(var), sp.sympify(expr)] for var, expr in self.algebraic.items()]


        for i in range(n-1):
             for j in range(i+1, n):
                 algebraic[j][1] = algebraic[j][1].subs(algebraic[i][0], algebraic[i][1])
```

```
        diffeq = self.diffeq.copy()

        for i in range(m):
            diffeq[i] = sp.sympify(diffeq[i])
            for var, expr in algebraic:
                diffeq[i] = diffeq[i].subs(var, expr)

        # 3. solve for derivatives:

        derexprs = solve(diffeq, *dx_dt, dict=True)

        for lhs, rhs in derexprs[0].items():

            dername = str(lhs)
            statename = self.dermap[dername]
            self.states[statename]["expr"] = rhs

        # 4. create atoms:

        ext_varnames = []

        ext_varsubs = {}

        external_vars = []

        # 4.a. set up ports:

        for port in self.ports:     # todo: other port types

            if port.typ == "inout":

                sign = 1

                for connection in port.connections:

                    varname = connection.invar
                    sign = connection.sign

                    for from_connection in connection.from_connections:

                        devicename = from_connection.port.device.name
                        varname = from_connection.outvar

                        mangeld_name = "{}_{}".format(devicename, varname)
                        ext_varnames.append(mangeld_name)

                        external_vars.append(varname)
```

```
            # make a sum expression for all input symbols:

            ext_varsubs[varname] = "(" + " + ".join(ext_varnames) + ")"

            if sign == -1:
                ext_varsubs[varname] = "-" + ext_varsubs[varname]

        # 4.b.

        argstrs = (list(self.constants.keys()) +
                   list(self.parameters.keys()) +
                   list(self.states.keys()) +
                   ext_varnames)

        argstr = " ".join(argstrs)
        argsyms = sp.var(argstr)

        for name, state in self.states.items():

            expr = state["expr"]

            for var, substr in ext_varsubs.items():
                subexpr = sp.sympify(substr)
                expr = expr.subs(var, subexpr)

            state["expr"] = expr

            func = lambdify(argsyms, expr, dummify=False)

            self.states[name]["atom"].derfunc = func

        for name in self.output_ports:

            statename = self.output_ports[name]["state"]
            state = self.states[statename]
            self.output_ports[name]["atom"] = state["atom"]

        # 5. connect atoms:

        for statex in self.states.values():

            for statey in self.states.values():

                f = statex["expr"]

                if statey["sym"] in f.free_symbols:
```

```
                    # connect:
                    statex["atom"].add_connection(statey["atom"])

                    # add jacobian expr:
                    df_dy = sp.diff(f, statey["sym"])

                    func = lambdify(argsyms, df_dy, dummify=False)

                    statex["atom"].add_jacfunc(statey["atom"], func)

            for var in external_vars:

                statey = self.states[var]

                f = statex["expr"]

                mangled_name = "{}_{}".format(statey["device"].name, statey["name"])
                mangled_symbol = sp.Symbol(mangled_name)

                if mangled_symbol in f.free_symbols:

                    # connect:
                    statex["atom"].add_connection(statey["atom"])

                    # jacobian expr:
                    df_dy = sp.diff(f, mangled_symbol)

                    func = lambdify(argsyms, df_dy, dummify=False)

                    statex["atom"].add_jacfunc(statey["atom"], func)

    @staticmethod
    def get_args(self):

        args = []

        for name, constant in self.constants.items():
            args.append(float(constant["value"]))

        for name, parameter in self.parameters.items():
            args.append(float(parameter["value"]))

        for name, state in self.states.items():
            args.append(float(state["atom"].q))

        for name, port in self.input_ports.items():
```

```
            for port2 in port["ports"]:
                args.append(port2["atom"].q)

        return args
```

## B.2 Python QDL Model Source

```
"""QDL Model Library
"""

from qdl import *


# ============================ Basic Devices ===================================


class GroundNode(Device):

    def __init__(self, name="ground"):

        Device.__init__(self, name)

        self.voltage = SourceAtom(name="voltage", source_type=SourceType.CONSTANT,
                                  x0=0.0, units="V", dq=1.0)

        self.add_atom(self.voltage)


class ConstantSourceNode(Device):

    def __init__(self, name="source", v0=0.0):

        Device.__init__(self, name)

        self.voltage = SourceAtom(name="voltage", source_type=SourceType.CONSTANT,
                                  x0=v0, units="V", dq=1.0)

        self.add_atom(self.voltage)


class PwmSourceNode(Device):

    def __init__(self, name="source", vlo=0.0, vhi=1.0, freq=1e3, duty=0.5):

```

```
        Device.__init__(self, name)

        self.voltage = SourceAtom(name="voltage", source_type=SourceType.PWM, x0=vlo, x1=vhi,
                                  x2=vlo, freq=freq, duty=duty, dq=1.0)

        self.add_atom(self.voltage)


class LimNode(Device):

    """Generic LIM Lantency Node with G, C, I, B and S components.

                          \
                i_i2(t)  ^   ...
                          \
                  i_i1(t)  \      i_ik(t)
                 ----<------o------>----
                            |
                            |   ^
                            |   | i_i(t)
            .--------.------+-----------.--------------.
            |        |      |    i_b =  |      i_s =   |       +
           ,-.      <.     _|_   b_k *  ,^.     s_pq * ,^.
    H(t) ( ^ )   G <.   C ___  v_k(t) < ^ >  i_pq(t) < ^ >   v(t)
           `-'      <.      |            `.' ^           `.' ^
            |        |      |             |   \           |   \   -
            '--------'------+-------------'----\----------'    \
                          _|_                   \                '---- port_s
                           -                     '---- port_b

    isum = -(h + i_s + i_b) + v*G + v'*C

    v'= 1/C * (isum + h + i_s + i_b - v*G)

    """

    def __init__(self, name, c, g=0.0, h0=0.0, v0=0.0,
                 source_type=SourceType.CONSTANT, h1=0.0, h2=0.0, ha=0.0,
                 freq=0.0, phi=0.0, duty=0.0, t1=0.0, t2=0.0, dq=None):

        Device.__init__(self, name)

        self.c = c
        self.g = g

        self.h = SourceAtom("h", source_type=source_type, x0=h0,
                                 x1=h1, x2=h2, xa=ha, freq=freq, phi=phi,
```

```
                                    duty=duty, t1=t1, t2=t2, dq=dq, units="A")

        self.voltage = StateAtom("voltage", x0=0.0, coeffunc=self.aii, dq=dq,
                                 units="V")

        self.add_atoms(self.h, self.voltage)

        self.voltage.add_connection(self.h, coeffunc=self.bii)

        self.voltage.add_jacfunc(self.voltage, self.aii)

    def connect(self, branch, terminal="i"):

        if terminal == "i":
            self.voltage.add_connection(branch.current, coeffunc=self.aij)
            self.voltage.add_jacfunc(branch.current, self.aij)

        elif terminal == "j":
            self.voltage.add_connection(branch.current, coeffunc=self.aji)
            self.voltage.add_jacfunc(branch.current, self.aji)

    @staticmethod
    def aii(self, v=0):
        return -self.g / self.c

    @staticmethod
    def bii(self):
        return 1.0 / self.c

    @staticmethod
    def aij(self, v=0, i=0):
        return -1.0 / self.c

    @staticmethod
    def aji(self, i=0, v=0):
        return 1.0 / self.c


class LimBranch(Device):

    """Generic LIM Lantency Branch with R, L, V, T and Z components.

               +                         v_ij(t)                       -

                                          i(t) -->

                                        v_t(t) =         v_z(t) =
```

```
132                v(t)  +   -   +   -  T_ijk * v_k(t)  Z_ijpq * i_pq(t)
133     positive    ,-.     R       L         ,^.           ,^.       negative
134         o-----(- +)---VVV-----UUU-------<- +>---------<- +>-------o
135                `_'                       `.'           `.'
136                                           ^             ^
137                                           |             |
138                                       port_t         port_z
139
140     vij = -(v + vt + vz) + i*R + i'*L
141
142     i'= 1/L * (vij + v + vt + vz - i*R)
143
144     """
145
146     def __init__(self, name, l, r=0.0, e0=0.0, i0=0.0,
147                  source_type=SourceType.CONSTANT, e1=0.0, e2=0.0, ea=0.0,
148                  freq=0.0, phi=0.0, duty=0.0, t1=0.0, t2=0.0, dq=None):
149
150         Device.__init__(self, name)
151
152         self.l = l
153         self.r = r
154
155         self.e = SourceAtom("e", source_type=source_type, x0=e0,
156                                  x1=e1, x2=e2, xa=ea, freq=freq, phi=phi,
157                                  duty=duty, t1=t1, t2=t2, dq=dq, units="V")
158
159         self.current = StateAtom("current", x0=0.0, coeffunc=self.aii, dq=dq,
160                                  units="A")
161
162         self.add_atoms(self.e, self.current)
163
164         self.current.add_connection(self.e, coeffunc=self.bii)
165
166         self.current.add_jacfunc(self.current, self.aii)
167
168     def connect(self, inode, jnode):
169
170         self.current.add_connection(inode.voltage, coeffunc=self.aij)
171         self.current.add_connection(jnode.voltage, coeffunc=self.aji)
172
173         self.current.add_jacfunc(inode.voltage, self.aij)
174         self.current.add_jacfunc(jnode.voltage, self.aji)
175
176     @staticmethod
177     def aii(self, i=0):
178         return -self.r / self.l
```

```
179
180     @staticmethod
181     def bii(self):
182         return 1.0 / self.l
183
184     @staticmethod
185     def aij(self, i=0, v=0):
186         return 1.0 / self.l
187
188     @staticmethod
189     def aji(self, v=0, i=0):
190         return -1.0 / self.l
191
192
193 # ============================== DQ Devices ====================================
194
195
196 class GroundNodeDQ(Device):
197
198     """
199           noded     nodeq
200             o         o
201             |         |
202             '----+----'
203                  _|_
204     """
205
206
207     def __init__(self, name="ground"):
208
209         Device.__init__(self, name)
210
211         self.vd = SourceAtom(name="vd", source_type=SourceType.CONSTANT,
212                              x0=0.0, units="V", dq=1.0)
213
214         self.vq = SourceAtom(name="vq", source_type=SourceType.CONSTANT,
215                              x0=0.0, units="V", dq=1.0)
216
217         self.add_atoms(self.vd, self.vq)
218
219
220 class SourceNodeDQ(Device):
221
222     """
223           noded       nodeq
224             o           o
225             |           |
226
```

```
227             +   ,-.       +   ,-.
228            Vd (    )     Vq (    )
229             -   `-'       -   `-'
230                 |             |
231                 '-------+-----'
232                       -|-
233     """
234
235     def __init__(self, name, voltage, th0=0.0):
236
237         Device.__init__(self, name)
238
239         self.voltage = voltage
240
241         self.vd0 = voltage * sin(th0)
242         self.vq0 = voltage * cos(th0)
243
244         self.vd = SourceAtom(name="vd", source_type=SourceType.FUNCTION,
245                              srcfunc=self.get_vd, x0=self.vd0, units="V", dq=1.0)
246
247         self.vq = SourceAtom(name="vq", source_type=SourceType.FUNCTION,
248                              srcfunc=self.get_vq, x0=self.vq0, units="V", dq=1.0)
249
250         self.add_atoms(self.vd, self.vq)
251
252         self.theta = None
253
254     def connect_theta(self, theta):
255
256         self.theta = theta
257
258         self.vd.add_connection(self.theta)
259         self.vq.add_connection(self.theta)
260
261     @staticmethod
262     def get_vd(self, t):
263
264         if self.th_atom:
265             return self.voltage * cos(self.theta.q)
266         else:
267             return self.vd0
268
269     @staticmethod
270     def get_vq(self, t):
271
272         if self.th_atom:
273
```

```
            return self.voltage * sin(self.th_atom.q)
        else:
            return self.vq0


class LimBranchDQ(Device):

    """RLV Branch DQ Dynamic Phasor Model.

                  .--------------------------------------.
                  |  vd(t)    id*(r + w*L)      id'*L    |
                  |   ,-.    +            -    +     -  |
     inodep  o---|--(- +)---------VVV-----------UUU----|---o  jnodep
                  |   `-'             id -->             |
                  |                                      |
                  |  vq(t)    iq*(r + w*L)      iq'*L    |
                  |   ,-.    +            -    +     -  |
     inodeq  o---|--(- +)---------VVV-----------UUU----|---o  jnodeq
                  |   `-'             iq -->             |
                  |                                      |
                  '--------------------------------------'
    """

    def __init__(self, name, l, r=0.0, vd0=0.0, vq0=0.0, w=60.0*PI, id0=0.0,
                 iq0=0.0, source_type=SourceType.CONSTANT,
                 vd1=0.0, vd2=0.0, vda=0.0, freqd=0.0, phid=0.0, dutyd=0.0,
                 td1=0.0, td2=0.0, vq1=0.0, vq2=0.0, vqa=0.0,
                 freqq=0.0, phiq=0.0, dutyq=0.0, tq1=0.0, tq2=0.0, dq=1e0):

        Device.__init__(self, name)

        self.l = l
        self.r = r
        self.w = w
        self.id0 = id0
        self.iq0 = iq0

        dmax = 1e8

        self.ed = SourceAtom("ed", source_type=source_type, x0=vd0, x1=vd1,
                             x2=vd2, xa=vda, freq=freqd, phi=phid, dmax=dmax,
                             duty=dutyd, t1=td1, t2=td2, dq=dq, units="V")

        self.eq = SourceAtom("eq", source_type=source_type, x0=vq0, x1=vq1,
                             x2=vq2, xa=vqa, freq=freqq, phi=phiq, dmax=dmax,
                             duty=dutyq, t1=tq1, t2=tq2, dq=dq, units="V")

```

```
        self.id = StateAtom("id", x0=id0, coeffunc=self.aii, dq=dq, dmax=dmax,
                            units="A")

        self.iq = StateAtom("iq", x0=iq0, coeffunc=self.aii, dq=dq, dmax=dmax,
                            units="A")

        self.add_atoms(self.ed, self.eq, self.id, self.iq)

        self.id.add_connection(self.ed, coeffunc=self.bii)
        self.iq.add_connection(self.eq, coeffunc=self.bii)

        self.id.add_jacfunc(self.id, self.aii)
        self.iq.add_jacfunc(self.iq, self.aii)

    def connect(self, inodedq, jnodedq):

        self.id.add_connection(inodedq.vd, coeffunc=self.aij)
        self.id.add_connection(jnodedq.vd, coeffunc=self.aji)

        self.iq.add_connection(inodedq.vq, coeffunc=self.aij)
        self.iq.add_connection(jnodedq.vq, coeffunc=self.aji)

        self.id.add_jacfunc(inodedq.vd, self.aij)
        self.id.add_jacfunc(jnodedq.vd, self.aji)

        self.iq.add_jacfunc(inodedq.vq, self.aij)
        self.iq.add_jacfunc(jnodedq.vq, self.aji)

    @staticmethod
    def aii(self, idq=0):
        return -(self.r + self.w * self.l) / self.l

    @staticmethod
    def bii(self):
        return 1.0 / self.l

    @staticmethod
    def aij(self, idq=0, vdq=0):
        return 1.0 / self.l

    @staticmethod
    def aji(self, idq=0, vdq=0):
        return -1.0 / self.l


class LimNodeDQ(Device):
```

```
    """
                         inoded                                    inodeq
                            o                                         o
                            |   | isumd                               |   | isumq
                            |   v                                     |   v
    .-----------------------+-----------------------------------------+-------------.
    |                       |                                         |             |
    |           .-----------+---------.                     .---------+---------.   |
    |           |           |         |                     |         |         |   |
    |  vd *  ^ <.       ^ _|_       ,-.      vq *  ^ <.       ^ _|_       ,-.  |   |
    |(g+w*C) | <. vd'*C | ___ id(t)( ^ )   (g+w*C) | <. vq'*C | ___ iq(t)( ^ ) |   |
    |        | <.       |   |         `-'             | <.       |   |         `-'  |   |
    |          |         |   |                       |          |   |           |   |
    |          '---------+---'                       '----------+---'           |   |
    |                      -|-                                   -|-             |   |
    |                       -                                     -              |   |
    '-----------------------------------------------------------------------------'
    """

    def __init__(self, name, c, g=0.0, id0=0.0, iq0=0.0, w=60.0*pi, vd0=0.0,
                 vq0=0.0, source_type=SourceType.CONSTANT, id1=0.0, id2=0.0,
                 ida=0.0, freqd=0.0, phid=0.0, dutyd=0.0, td1=0.0, td2=0.0,
                 iq1=0.0, iq2=0.0, iqa=0.0, freqq=0.0, phiq=0.0, dutyq=0.0,
                 tq1=0.0, tq2=0.0, dq=None):

        Device.__init__(self, name)

        self.c = c
        self.g = g
        self.w = w
        self.vd0 = vd0
        self.vq0 = vq0

        self.hd = SourceAtom("hd", source_type=source_type, x0=id0, x1=id1,
                                   x2=id2, xa=ida, freq=freqd, phi=phid,
                                   duty=dutyd, t1=td1, t2=td2, dq=dq, units="A")

        self.hq = SourceAtom("hq", source_type=source_type, x0=iq0, x1=iq1,
                                   x2=iq2, xa=iqa, freq=freqq, phi=phiq,
                                   duty=dutyq, t1=tq1, t2=tq2, dq=dq, units="A")

        self.vd = StateAtom("vd", x0=vd0, coeffunc=self.aii, dq=dq, units="V")

        self.vq = StateAtom("vq", x0=vq0, coeffunc=self.aii, dq=dq, units="V")

```

```
413         self.add_atoms(self.hd, self.hq, self.vd, self.vq)
414
415         self.vd.add_connection(self.hd, coeffunc=self.bii)
416         self.vq.add_connection(self.hq, coeffunc=self.bii)
417
418         self.vd.add_jacfunc(self.vd, self.aii)
419         self.vq.add_jacfunc(self.vq, self.aii)
420
421     def connect(self, device, terminal="i"):
422
423         if terminal == "i":
424
425             self.vd.add_connection(device.id, coeffunc=self.aij)
426             self.vq.add_connection(device.iq, coeffunc=self.aij)
427
428             self.vd.add_jacfunc(device.id, self.aij)
429             self.vq.add_jacfunc(device.iq, self.aij)
430
431         elif terminal == "j":
432
433             self.vd.add_connection(device.id, coeffunc=self.aji)
434             self.vq.add_connection(device.iq, coeffunc=self.aji)
435
436             self.vd.add_jacfunc(device.id, self.aji)
437             self.vq.add_jacfunc(device.iq, self.aji)
438
439     @staticmethod
440     def aii(self, vdq=0):
441         return -(self.g + self.w * self.c) / self.c
442
443     @staticmethod
444     def bii(self):
445         return 1.0 / self.c
446
447     @staticmethod
448     def aij(self, vdq=0, idq=0):
449         return -1.0 / self.c
450
451     @staticmethod
452     def aji(self, vdq=0, idq=0):
453         return 1.0 / self.c
454
455
456 class SyncMachineDQ(Device):
457
458     """Synchronous Machine Reduced DQ Model as Voltage source behind a
459     Lim Latency Branch.
```

```
460
461     """
462
463     def __init__(self, name, Psm=25.0e6, VLL=4160.0, ws=60.0*PI,
464                  P=4, pf=0.80, rs=3.00e-3, Lls=0.20e-3, Lmq=2.00e-3,
465                  Lmd=2.00e-3, rkq=5.00e-3, Llkq=0.04e-3, rkd=5.00e-3,
466                  Llkd=0.04e-3, rfd=20.0e-3, Llfd=0.15e-3, vfdb=90.1, Kp=10.0e4,
467                  Ki=10.0e4, J=4221.7, fkq0=0.0, fkd0=0.0, ffd0=0.0, wr0=60.0*PI,
468                  th0=0.0, iqs0=0.0, ids0=0.0, dq_i=1e-2, dq_f=1e-2, dq_wr=1e-1,
469                  dq_th=1e-3, dq_v=1e0):
470
471         self.name = name
472
473         # sm params:
474
475         self.Psm  = Psm
476         self.VLL  = VLL
477         self.ws   = ws
478         self.P    = P
479         self.pf   = pf
480         self.rs   = rs
481         self.Lls  = Lls
482         self.Lmq  = Lmq
483         self.Lmd  = Lmd
484         self.rkq  = rkq
485         self.Llkq = Llkq
486         self.rkd  = rkd
487         self.Llkd = Llkd
488         self.rfd  = rfd
489         self.Llfd = Llfd
490         self.vfdb = vfdb
491
492         # turbine/governor params:
493
494         self.Kp = Kp
495         self.Ki = Ki
496         self.J  = J
497
498         # intial conditions:
499
500         self.iqs0 = iqs0
501         self.ids0 = ids0
502         self.fkq0 = fkq0
503         self.fkd0 = fkd0
504         self.ffd0 = ffd0
505         self.wr0 = wr0
506         self.th0  = th0
507
```

```
        dmax = 1e8

        # call super:

        Device.__init__(self, name)

        # derived:

        self.Lq = Lls + (Lmq * Llkq) / (Llkq + Lmq)
        self.Ld = (Lls + (Lmd * Llfd * Llkd) / (Lmd * Llfd + Lmd * Llkd + Llfd * Llkd))

        # state atoms:

        self.ids = StateAtom("ids", x0=ids0, derfunc=self.dids, units="A",     dq=dq_i)
        self.iqs = StateAtom("iqs", x0=iqs0, derfunc=self.diqs, units="A",     dq=dq_i)
        self.fkq = StateAtom("fkq", x0=fkq0, derfunc=self.dfkq, units="Wb",    dq=dq_f)
        self.fkd = StateAtom("fkd", x0=fkd0, derfunc=self.dfkd, units="Wb",    dq=dq_f)
        self.ffd = StateAtom("ffd", x0=ffd0, derfunc=self.dffd, units="Wb",    dq=dq_f)
        self.wr  = StateAtom("wr",  x0=wr0,  derfunc=self.dwr,  units="rad/s", dq=dq_wr)
        self.th  = StateAtom("th",  x0=th0,  derfunc=self.dth,  units="rad",   dq=dq_th)

        self.add_atoms(self.ids, self.iqs, self.fkq, self.fkd, self.ffd, self.wr, self.th)

        # atom connections:

        self.ids.add_connection(self.iqs)
        self.ids.add_connection(self.wr)
        self.ids.add_connection(self.fkq)

        self.iqs.add_connection(self.ids)
        self.iqs.add_connection(self.wr)
        self.iqs.add_connection(self.fkd)
        self.iqs.add_connection(self.ffd)

        self.fkd.add_connection(self.ffd)
        self.fkd.add_connection(self.ids)

        self.fkq.add_connection(self.iqs)

        self.ffd.add_connection(self.ids)
        self.ffd.add_connection(self.fkd)

        self.wr.add_connection(self.ids)
        self.wr.add_connection(self.iqs)
        self.wr.add_connection(self.fkq)
        self.wr.add_connection(self.fkd)
```

```
554         self.wr.add_connection(self.ffd)
555         self.wr.add_connection(self.th)
556
557         self.th.add_connection(self.wr)
558
559         # jacobian:
560
561         self.ids.add_jacfunc(self.ids, self.jids_ids)
562         self.ids.add_jacfunc(self.iqs, self.jids_iqs)
563         self.ids.add_jacfunc(self.fkq, self.jids_fkq)
564         self.ids.add_jacfunc(self.wr , self.jids_wr )
565
566         self.iqs.add_jacfunc(self.ids, self.jiqs_ids)
567         self.iqs.add_jacfunc(self.iqs, self.jiqs_iqs)
568         self.iqs.add_jacfunc(self.fkd, self.jiqs_fkd)
569         self.iqs.add_jacfunc(self.ffd, self.jiqs_ffd)
570         self.iqs.add_jacfunc(self.wr , self.jiqs_wr )
571
572         self.fkq.add_jacfunc(self.iqs, self.jfkq_iqs)
573         self.fkq.add_jacfunc(self.fkq, self.jfkq_fkq)
574
575         self.fkd.add_jacfunc(self.ids, self.jfkd_ids)
576         self.fkd.add_jacfunc(self.fkd, self.jfkd_fkd)
577         self.fkd.add_jacfunc(self.ffd, self.jfkd_ffd)
578
579         self.ffd.add_jacfunc(self.ids, self.jffd_ids)
580         self.ffd.add_jacfunc(self.fkd, self.jffd_fkd)
581         self.ffd.add_jacfunc(self.ffd, self.jffd_ffd)
582
583         self.wr.add_jacfunc(self.wr  , self.jwr_wr )
584         self.wr.add_jacfunc(self.ids , self.jwr_ids )
585         self.wr.add_jacfunc(self.iqs , self.jwr_iqs )
586         self.wr.add_jacfunc(self.fkq , self.jwr_fkq )
587         self.wr.add_jacfunc(self.fkd , self.jwr_fkd )
588         self.wr.add_jacfunc(self.ffd , self.jwr_ffd )
589         self.wr.add_jacfunc(self.th  , self.jwr_th  )
590
591         self.th.add_jacfunc(self.wr  ,  self.jth_wr )
592
593         # ports:
594
595         self.id = self.iqs
596         self.iq = self.ids
597
598         self.vd = None  # terminal voltage d connection
```

```
        self.vq = None  # terminal voltage q connection

        self.input = None  # avr

    def connect(self, bus, avr=None):

        self.vd = self.iqs.add_connection(bus.vq)
        self.vq = self.ids.add_connection(bus.vd)

        self.iqs.add_jacfunc(bus.vq, self.jiqs_vq)
        self.ids.add_jacfunc(bus.vd, self.jids_vd)

        if avr:
            self.input = self.ffd.add_connection(avr.vfd, self.vfdb)
            self.ffd.add_jacfunc(avr.vfd, self.jffd_vfd)

    def vtermd(self):
        return self.vd.value()

    def vtermq(self):
        return self.vq.value()

    def vfd(self):
        if self.input:
            return self.input.value()
        else:
           return self.vfdb # no feedback control

    @staticmethod
    def jffd_vfd(self, ffd, vfd):
        return self.vfdb

    @staticmethod
    def jids_vd(self, ids, vd):
        return -1 / self.Lls

    @staticmethod
    def jiqs_vq(self, iqs, vq):
        return -1 / self.Lls

    @staticmethod
    def jids_ids(self, ids):
        return -self.rs/self.Lls

    @staticmethod
    def jids_iqs(self, ids, iqs):
```

```
        return -(self.Lq * self.wr.q) / self.Lls

    @staticmethod
    def jids_fkq(self, ids, fkq):
        return (self.Lmq * self.wr.q) / (self.Lls * (self.Lmq + self.Llkq))

    @staticmethod
    def jids_wr(self, ids, wr):
        return (((self.Lmq * self.fkq.q) / (self.Lmq + self.Llkq)
                - self.Lq * self.iqs.q) / self.Lls)

    @staticmethod
    def jiqs_ids(self, iqs, ids):
        return (self.Ld * self.wr.q) / self.Lls

    @staticmethod
    def jiqs_iqs(self, iqs):
        return -self.rs / self.Lls

    @staticmethod
    def jiqs_fkd(self, iqs, fkd):
        return ((self.Lmd * self.wr.q) / (self.Llkd * self.Lls
                * (self.Lmd / self.Llkd + self.Lmd / self.Llfd + 1)))

    @staticmethod
    def jiqs_ffd(self, iqs, ffd):
        return ((self.Lmd * self.wr.q) / (self.Llfd * self.Lls
                * (self.Lmd / self.Llkd + self.Lmd / self.Llfd + 1)))

    @staticmethod
    def jiqs_wr(self, iqs, wr):
        return ((self.Ld * self.ids.q + (self.Lmd * (self.fkd.q / self.Llkd
                + self.ffd.q / self.Llfd)) / (self.Lmd / self.Llkd
                + self.Lmd / self.Llfd + 1)) / self.Lls)

    @staticmethod
    def jfkq_iqs(self, fkq, iqs):
        return -((self.Lls - self.Lq) * self.rkq) / self.Llkq

    @staticmethod
    def jfkq_fkq(self, fkq):
        return -((1 - self.Lmq / (self.Lmq + self.Llkq)) * self.rkq) / self.Llkq

    @staticmethod
    def jfkd_ids(self, fkd, ids):
        return -((-self.Lls - self.Ld) * self.rkd) / self.Llkd
```

```
691
692     @staticmethod
693     def jfkd_fkd(self, fkd):
694         return (-((self.Lmd / (self.Llkd * (self.Lmd / self.Llkd + self.Lmd
695                 / self.Llfd + 1)) + 1) * self.rkd) / self.Llkd)
696
697     @staticmethod
698     def jfkd_ffd(self, fkd, ffd):
699         return -(self.Lmd * self.rkd) / (self.Llfd * self.Llkd
700                * (self.Lmd / self.Llkd + self.Lmd / self.Llfd + 1))
701
702     @staticmethod
703     def jffd_ids(self, ffd, ids):
704         return -((-self.Lls - self.Ld) * self.rfd) / self.Llfd
705
706     @staticmethod
707     def jffd_fkd(self, ffd, fkd):
708         return (-(self.Lmd * self.rfd) / (self.Llfd * self.Llkd
709                * (self.Lmd / self.Llkd + self.Lmd/self.Llfd + 1)))
710
711     @staticmethod
712     def jffd_ffd(self, ffd):
713         return (-((self.Lmd / (self.Llfd * (self.Lmd / self.Llkd + self.Lmd
714                 / self.Llfd + 1)) + 1) * self.rfd) / self.Llfd)
715
716     @staticmethod
717     def jwr_ids(self, wr, ids):
718         return (0.75*self.P*(-self.Lq*self.iqs.q+self.Ld*self.iqs.q-(self.Lmq*self.fkq.q)/(self.
    Lmq+self.Llkq)))/self.J
719
720     @staticmethod
721     def jwr_iqs(self, wr, iqs):
722         return (0.75*self.P*(-self.Lq*self.ids.q+self.Ld*self.ids.q+(self.Lmd*(self.fkd.q/self.
    Llkd+self.ffd.q/self.Llfd))/(self.Lmd/self.Llkd+self.Lmd/self.Llfd+1.0)))/self.J
723
724     @staticmethod
725     def jwr_fkq(self, wr, fkq):
726         return (-(0.75 * self.Lmq * self.P * self.ids.q)
727                 / (self.J * (self.Lmq + self.Llkq)))
728
729     @staticmethod
730     def jwr_fkd(self, wr, fkd):
731         return ((0.75 * self.Lmd * self.P * self.iqs.q) / (self.J * self.Llkd
732                * (self.Lmd / self.Llkd + self.Lmd / self.Llfd + 1)))
733
734     @staticmethod
```

```
    def jwr_ffd(self, wr, ffd):
        return ((0.75 * self.Lmd * self.P * self.iqs.q) / (self.J * self.Llfd
                * (self.Lmd / self.Llkd + self.Lmd / self.Llfd + 1)))

    @staticmethod
    def jwr_wr(self, wr):
        return -self.Kp / self.J

    @staticmethod
    def jwr_th(self, wr, fthkq):
        return self.Ki / self.J

    @staticmethod
    def jth_wr(self, th, wr):
        return -1.0

    @staticmethod
    def dids(self, ids):
        return ((-self.wr.q * self.Lq * self.iqs.q + self.wr.q
                * (self.Lmq / (self.Lmq + self.Llkq) * self.fkq.q)
                - self.vtermd() - self.rs * ids) / self.Lls)

    @staticmethod
    def diqs(self, iqs):
        return ((self.wr.q * self.Ld * self.ids.q + self.wr.q
                * (self.Lmd * (self.fkd.q / self.Llkd + self.ffd.q / self.Llfd)
                /  (1.0 + self.Lmd / self.Llfd + self.Lmd / self.Llkd))
                - self.vtermq() - self.rs * iqs) / self.Lls)

    @staticmethod
    def dfkq(self, fkq):
        return (-self.rkq / self.Llkq * (fkq - self.Lq * self.iqs.q
                - (self.Lmq / (self.Lmq + self.Llkq) * fkq) + self.Lls * self.iqs.q))

    @staticmethod
    def dfkd(self, fkd):
        return (-self.rkd / self.Llkd * (fkd - self.Ld * self.ids.q
                + (self.Lmd * (fkd / self.Llkd + self.ffd.q / self.Llfd)
                / (1.0 + self.Lmd / self.Llfd + self.Lmd / self.Llkd))
                - self.Lls * self.ids.q))

    @staticmethod
    def dffd(self, ffd):
        return (self.vfd() - self.rfd / self.Llfd
                * (ffd - self.Ld * self.ids.q
```

```
                + (self.Lmd * (self.fkd.q / self.Llkd + ffd / self.Llfd)
                /  (1.0 + self.Lmd / self.Llfd + self.Lmd / self.Llkd))
                - self.Lls * self.ids.q))

    @staticmethod
    def dwr(self, wr):
        return ((3.0 * self.P / 4.0 * ((self.Ld * self.ids.q
                + ((self.Lmd * (self.fkd.q / self.Llkd + self.ffd.q / self.Llfd)
                /  (1.0 + self.Lmd / self.Llfd + self.Lmd / self.Llkd)))) * self.iqs.q
                - (self.Lq * self.iqs.q + (self.Lmq / (self.Lmq + self.Llkq) * self.fkq.q)) * self
    .ids.q))
                + (self.Kp * (self.ws - wr) + self.Ki * self.th.q)) / self.J

    @staticmethod
    def dth(self, th):
        return (self.ws - self.wr.q)


class AC8B(Device):

    """IEEE AC8B Exciter

                .--------.
    vref     .->|  Kpr   |----.                Vrmax
      |      |  '--------'    |              ,---              (vfd)
    + v      |                + v       .-------'-.                    .------.
     ,-.  e1 |  .--------. + ,-.  pid |   Ka    |    + ,-.  e2  |  1   |
    ( S )----+->| Kir/s  |->( S )----->| ------- |---->( S )---->| ---- |---+--> vfd
     '-'     |  '--------'   '-'       | 1+s*Ta  |  vr  '-'      | s*Te |   |
    - ^      |  .--------.  + ^        '-,-------'      - ^       '------'   |
      |      |  | s*Kdr  |    |     ---'  (x3)            | vse              |
     vt      '->| ------ |----'    Vrmin                  ,-.    .--------.   |
              | 1+s*Tr |                               ( S )<--| Se(Ve) |<--+
              '--------'                                '-' +  '--------'   |
                                                      + ^      .----.       |
                (x1, x2)                                '------| Ke |<-----'
                                                              '----'
    """

    def __init__(self, name, VLL=4160.0, vref=1.0, Kpr=200.0, Kir=0.8,
                 Kdr=1e-3, Tdr=1e-3, Ka=1.0, Ta=1e-4, Vrmin=0.0,
                 Vrmax=5.0, Te=1.0, Ke=1.0, Sea=1.0119, Seb=0.0875,
                 x10=0.0, x20=0.0, x30=0.0, vfd0=None,
                 dq_x1=1e-8, dq_x2=1e-8, dq_x3=1e-5, dq_vfd=1e-2):

        Device.__init__(self, name)
```

```
        self.VLL = VLL
        self.vref = vref
        self.Kpr = Kpr
        self.Kir = Kir
        self.Kdr = Kdr
        self.Tdr = Tdr
        self.Ka = Ka
        self.Ta = Ta
        self.Vrmin = Vrmin
        self.Vrmax = Vrmax
        self.Te = Te
        self.Ke = Ke
        self.Sea = Sea
        self.Seb = Seb

        x10 = x10
        x20 = x20
        x30 = x30
        if not vfd0:
            vfd0 = self.vref

        dq_x1 = dq_x1
        dq_x2 = dq_x2
        dq_x3 = dq_x3
        dq_vfd = dq_vfd

        self.x1 = StateAtom("x1", x0=x10, derfunc=self.dx1, dq=dq_x1)
        self.x2 = StateAtom("x2", x0=x20, derfunc=self.dx2, dq=dq_x2)
        self.x3 = StateAtom("x3", x0=x30, derfunc=self.dx3, dq=dq_x3)
        self.vfd = StateAtom("vfd", x0=vfd0, derfunc=self.dvfd, dq=dq_vfd)

        self.add_atoms(self.x1, self.x2, self.x3, self.vfd)

        self.x2.add_connection(self.x1)
        self.x3.add_connection(self.x1)
        self.x3.add_connection(self.x2)
        self.vfd.add_connection(self.x3)

        self.x1.add_jacfunc(self.x1, self.jx1_x1)

        self.x2.add_jacfunc(self.x1, self.jx2_x1)

        self.x3.add_jacfunc(self.x1, self.jx3_x1)
        self.x3.add_jacfunc(self.x2, self.jx3_x2)

        self.vfd.add_jacfunc(self.x3, self.jvfd_x3)
```

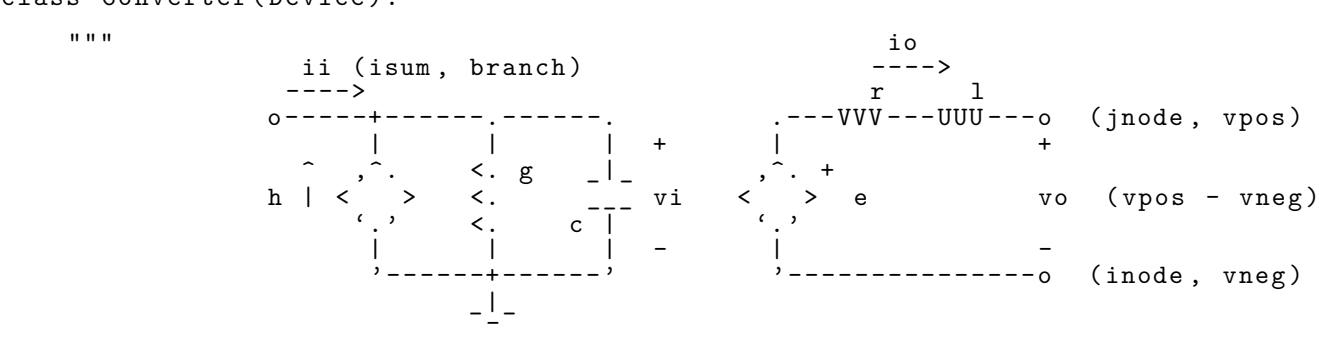
```
872         self.vfd.add_jacfunc(self.vfd, self.jvfd_vfd)
873
874         self.vd = None
875         self.vq = None
876
877     def connect(self, bus, sm):
878
879         self.vd = self.x1.add_connection(bus.vd, 1.0 / self.VLL)
880         self.vq = self.x1.add_connection(bus.vq, 1.0 / self.VLL)
881
882         self.x3.add_connection(bus.vd)
883         self.x3.add_connection(bus.vq)
884
885         self.x1.add_jacfunc(bus.vd, self.jx1_vd)
886         self.x1.add_jacfunc(bus.vq, self.jx1_vq)
887
888         self.x3.add_jacfunc(bus.vd, self.jx3_vd)
889         self.x3.add_jacfunc(bus.vq, self.jx3_vq)
890
891     @staticmethod
892     def jx1_x1(self, x1):
893         return -1.0/self.Tdr
894
895     @staticmethod
896     def jx1_vd(self, x1, vd):
897         return -self.vd.value()/sqrt(self.vq.value()**2+self.vd.value()**2)
898
899     @staticmethod
900     def jx1_vq(self, x1, vq):
901         return -self.vq.value()/sqrt(self.vq.value()**2+self.vd.value()**2)
902
903     @staticmethod
904     def jx2_x1(self, x2, x1):
905         return 1
906
907     @staticmethod
908     def jx3_x1(self, x3, x1):
909         return self.Kir*self.Tdr-self.Kdr/self.Tdr
910
911     @staticmethod
912     def jx3_x2(self, x3, x2):
913         return self.Kir
914
915     @staticmethod
916     def jx3_vd(self, x3, vd):
917         return (-((self.Kpr*self.Tdr+self.Kdr)*self.vd.value())
```

```
                /sqrt(self.vq.value()**2+self.vd.value()**2))

    @staticmethod
    def jx3_vq(self, x3, vq):
        return (-((self.Kpr*self.Tdr+self.Kdr)*self.vq.value())
                /sqrt(self.vq.value()**2+self.vd.value()**2))

    @staticmethod
    def jvfd_x3(self, vfd, x3):
        return self.Ka/(self.Ta*self.Te)

    @staticmethod
    def jvfd_vfd(self, vfd):
        return -self.Ke/self.Te

    @staticmethod
    def dx1(self, x1):
        return (-1.0 / self.Tdr * x1 + (self.vref
               - sqrt(self.vd.value()**2 + self.vq.value()**2)))

    @staticmethod
    def dx2(self, x2):
        return self.x1.q

    @staticmethod
    def dx3(self, x3):
        return -1.0 / self.Ta + ((self.Kir * self.Tdr - self.Kdr / self.Tdr)
                * self.x1.q + self.Kir * self.x2.q + (self.Kdr + self.Kpr
                * self.Tdr) * (self.vref - sqrt(self.vd.value()**2
                + self.vq.value()**2)))

    @staticmethod
    def dvfd(self, vfd):
        return (self.Ka / self.Ta * self.x3.q - vfd * self.Ke) / self.Te


class DCMotor(Device):

    """

    Jm * dwr = Kt * ia - Bm * wr
    La * dia = -ra * ia + va - Ke * wr

    dwr = (Kt * ia - Bm * wr) / Jm
    dia = (-ra * ia + va - Ke * wr) / La

               ia(t) -->
```

```
965
966                Ra    La                              (shaft)
967           o----VVV---UUU----.                  .-------o------.
968        (inode)              |                 |       |      |   +
969          +                +  ,^.    Kt * ia ,^.      <.     _|_
970          va                 <   >           < ^ >  B <.   J ___  wr
971          -                  -  `.' Ke * wr    `.'      <.      |
972        (jnode)              |                 |       |      |   -
973           o-----------------'                 '-------+------'
974                                                     _|_
975
976      """
977
978      def __init__(self, name, ra=0.1, La=0.01, Jm=0.1, Bm=0.001, Kt=0.1, Ke=0.1,
979                   ia0=0.0, wr0=0.0, dq_ia=1e-2, dq_wr=1e-1):
980
981          Device.__init__(self, name)
982
983          self.name = name
984
985          self.ra = ra
986          self.La = La
987          self.Jm = Jm
988          self.Bm = Bm
989          self.Kt = Kt
990          self.Ke = Ke
991
992          self.ia0 = ia0
993          self.wr0 = wr0
994
995          self.dq_ia = dq_ia
996          self.dq_wr = dq_wr
997
998          self.ia = StateAtom("ia", x0=ia0, derfunc=self.dia, der2func=self.d2ia,
999                              units="A", dq=dq_ia)
1000
1001         self.wr = StateAtom("wr", x0=wr0, derfunc=self.dwr, der2func=self.d2wr,
1002                             units="rad/s", dq=dq_wr)
1003
1004         self.add_atoms(self.ia, self.wr)
1005
1006         #self.ia.add_connection(self.ia)
1007         self.ia.add_connection(self.wr)
1008
1009         #self.wr.add_connection(self.wr)
1010         self.wr.add_connection(self.ia)
1011
```

```
        self.ia.add_jacfunc(self.ia, self.jia_ia)
        self.ia.add_jacfunc(self.wr, self.jia_wr)

        self.wr.add_jacfunc(self.wr, self.jwr_wr)
        self.wr.add_jacfunc(self.ia, self.jwr_ia)

        self.vi = None
        self.vj = None

        self.current = self.ia

    def connect(self, inode, jnode):

        self.ia.add_connection(inode.voltage)
        self.ia.add_connection(jnode.voltage)

        self.ia.add_jacfunc(inode.voltage, self.jia_vi)
        self.ia.add_jacfunc(jnode.voltage, self.jia_vj)

        self.vi = inode.voltage
        self.vj = jnode.voltage

    @staticmethod
    def dia(self, ia):
        return (-self.Ke * self.wr.q - self.vj.q + self.vi.q - ia * self.ra) / self.La

    @staticmethod
    def dwr(self, wr):
        return (self.Kt * self.ia.q - self.Bm * wr) / self.Jm

    @staticmethod
    def d2ia(self, ia, dia):
        return (-self.Ke * self.wr.d - self.vi.d + self.vj.d - dia * self.ra) / self.La

    @staticmethod
    def d2wr(self, wr, dwr):
        return (self.Kt * self.ia.d - self.Bm * dwr) / self.Jm

    @staticmethod
    def jia_ia(self, ia):
        return -self.ra / self.La

    @staticmethod
    def jia_wr(self, ia, wr):
        return -self.Ke / self.La

    @staticmethod
```

```
    def jia_vi(self, ia, vi):
        return 1 / self.La

    @staticmethod
    def jia_vj(self, ia, vj):
        return -1 / self.La

    @staticmethod
    def jwr_ia(self, wr, ia):
        return self.Kt / self.Jm

    @staticmethod
    def jwr_wr(self, wr):
        return -self.Bm / self.Jm


class Converter(Device):

    """
```




```
    ii = vi * g + dvi * c - h
    e  = io * r + dio * l + vo

    dvi = (-g * vi + i + h) / c
    dio = (-v - io * r + e) / l


    """

    def __init__(self, name, r=0.0, l=0.01, c=0.01, g=0.0, freq=100.0, duty=1.0,
                 io0=0.0, vi0=0.0, dq_i=None, dq_v=None):

        Device.__init__(self, name)

        self.name = name

```

```
        self.r = r
        self.l = l
        self.c = c
        self.g = g

        self.freq = freq
        self.duty = duty

        self.io0 = io0
        self.vi0 = vi0

        self.dq_i = dq_i
        self.dq_v = dq_v

        self.vi = StateAtom("vi", x0=vi0, coeffunc=self.jvi_vi, units="V", dq=dq_v)

        self.io = StateAtom("io", x0=io0, coeffunc=self.jio_io, units="A", dq=dq_i)

        self.e = SourceAtom(name="e", source_type=SourceType.PWM, x1=0.0, x2=1.0,
                            gainfunc=self.ke, freq=freq, duty=duty, dq=1.0)

        self.h = SourceAtom(name="h", source_type=SourceType.PWM, x1=0.0, x2=-1.0,
                            gainfunc=self.kh, freq=freq, duty=duty, dq=1.0)

        self.add_atoms(self.io, self.vi, self.e, self.h)

        self.vi.add_connection(self.h, coeffunc=self.jvi_h)
        self.vi.add_jacfunc(self.vi, self.jvi_vi)

        self.io.add_connection(self.e, coeffunc=self.jio_e)
        self.io.add_jacfunc(self.io, self.jio_io)

        # output connection aliases:

        self.voltage = self.vi
        self.current = self.io

    def connect(self, branch, inode, jnode, terminal="j"):

        self.io.add_connection(inode.voltage, coeffunc=self.jio_vneg)
        self.io.add_connection(jnode.voltage, coeffunc=self.jio_vpos)

        self.io.add_jacfunc(inode.voltage, self.jio_vneg)
        self.io.add_jacfunc(jnode.voltage, self.jio_vpos)

        if terminal == "i":
            self.vi.add_connection(branch.current, coeffunc=self.jvi_ii)
```

```
            self.vi.add_jacfunc(branch.current, self.jvi_ii)

        elif terminal == "j":
            self.voltage.add_connection(branch.current, coeffunc=self.jvi_ij)
            self.voltage.add_jacfunc(branch.current, self.jvi_ij)

    @staticmethod
    def ke(self):
        return self.vi.q

    @staticmethod
    def kh(self):
        return self.io.q

    @staticmethod
    def jio_io(self, *args):
        return -self.r / self.l

    @staticmethod
    def jvi_vi(self, *args):
        return -self.g / self.c

    @staticmethod
    def jio_e(self, *args):
        return 1.0 / self.l

    @staticmethod
    def jvi_h(self, *args):
        return 1.0 / self.c

    @staticmethod
    def jio_vpos(self, *args):
        return -1.0 / self.l

    @staticmethod
    def jio_vneg(self, *args):
        return 1.0 / self.l

    @staticmethod
    def jvi_ii(self, *args):
        return -1.0 / self.c

    @staticmethod
    def jvi_ij(self, *args):
        return 1.0 / self.c
```

```
class InductionMachineDQ(Device):

    """
    fqs = Lls * iqs + Lm * (iqs + iqr)
    fds = Lls * ids + Lm * (ids + idr)
    fqr = Llr * iqr + Lm * (iqr + iqs)
    fdr = Llr * idr + Lm * (idr + ids)

    vqs = Rs * iqs + wr * fds + (Lls + Lm) * diqs + Lm * diqr
    vds = Rs * ids - wr * fqs + (Lls + Lm) * dids + Lm * didr
    vqr = Rr * iqr + (ws - wr) * fdr + (Llr + Lm) * diqr + Lm * diqs
    vdr = Rr * idr - (ws - wr) * fqr + (Llr + Lm) * didr + Lm * dids
    """

    def __init__(self, name, ws=30*PI, P=4, Tb=26.53e3, Rs=31.8e-3,
                 Lls=0.653e-3, Lm=38e-3, Rr=24.1e-3, Llr=0.658e-3, J=250.0,
                 iqs0=0.0, ids0=0.0, iqr0=0.0, idr0=0.0, wr0=0.0, dq_i=1e-2,
                 dq_wr=1e-1):

        Device.__init__(self, name)

        self.name = name
        self.ws = ws
        self.P = P
        self.Tb = Tb
        self.Rs = Rs
        self.Lls = Lls
        self.Lm = Lm
        self.Rr = Rr
        self.Llr = Llr
        self.J = J

        self.iqs0 = iqs0
        self.ids0 = ids0
        self.iqr0 = iqr0
        self.idr0 = idr0
        self.wr0 = wr0

        self.dq_i = dq_i
        self.dq_wr = dq_wr

        # derived:

        self.dinv = 1.0 / (Lls * Lm + Llr * Lm + Llr * Lls)

        # atoms:

```

```
        self.iqs = StateAtom("iqs", x0=iqs0, derfunc=self.diqs, units="A",     dq=dq_i)
        self.ids = StateAtom("ids", x0=ids0, derfunc=self.dids, units="A",     dq=dq_i)
        self.iqr = StateAtom("iqr", x0=iqr0, derfunc=self.diqr, units="A",     dq=dq_i)
        self.idr = StateAtom("idr", x0=idr0, derfunc=self.didr, units="A",     dq=dq_i)
        self.wr  = StateAtom("wr",  x0=wr0,  derfunc=self.dwr,  units="rad/s", dq=dq_wr)

        self.add_atoms(self.iqs, self.ids, self.iqr, self.idr, self.wr)

        # atom connections:

        self.iqs.add_connection(self.ids)
        self.iqs.add_connection(self.iqr)
        self.iqs.add_connection(self.idr)
        self.iqs.add_connection(self.wr)

        self.ids.add_connection(self.iqs)
        self.ids.add_connection(self.iqr)
        self.ids.add_connection(self.idr)
        self.ids.add_connection(self.wr)

        self.iqr.add_connection(self.iqs)
        self.iqr.add_connection(self.ids)
        self.iqr.add_connection(self.idr)
        self.iqr.add_connection(self.wr)

        self.idr.add_connection(self.iqs)
        self.idr.add_connection(self.ids)
        self.idr.add_connection(self.iqr)
        self.idr.add_connection(self.wr)

        self.wr.add_connection(self.iqs)
        self.wr.add_connection(self.ids)
        self.wr.add_connection(self.iqr)
        self.wr.add_connection(self.idr)

        self.iqs.add_jacfunc(self.iqs, self.jiqs_iqs)
        self.iqs.add_jacfunc(self.ids, self.jiqs_ids)
        self.iqs.add_jacfunc(self.iqr, self.jiqs_iqr)
        self.iqs.add_jacfunc(self.idr, self.jiqs_idr)
        self.iqs.add_jacfunc( self.wr, self.jiqs_wr )

        self.ids.add_jacfunc(self.iqs, self.jids_iqs)
        self.ids.add_jacfunc(self.ids, self.jids_ids)
        self.ids.add_jacfunc(self.iqr, self.jids_iqr)
```

```
        self.ids.add_jacfunc(self.idr, self.jids_idr)
        self.ids.add_jacfunc( self.wr, self.jids_wr )

        self.iqr.add_jacfunc(self.iqs, self.jiqr_iqs)
        self.iqr.add_jacfunc(self.ids, self.jiqr_ids)
        self.iqr.add_jacfunc(self.iqr, self.jiqr_iqr)
        self.iqr.add_jacfunc(self.idr, self.jiqr_idr)
        self.iqr.add_jacfunc( self.wr, self.jiqr_wr )

        self.idr.add_jacfunc(self.iqs, self.jidr_iqs)
        self.idr.add_jacfunc(self.ids, self.jidr_ids)
        self.idr.add_jacfunc(self.iqr, self.jidr_iqr)
        self.idr.add_jacfunc(self.idr, self.jidr_idr)
        self.idr.add_jacfunc( self.wr, self.jidr_wr )

        self.wr.add_jacfunc(self.iqs, self.jwr_iqs)
        self.wr.add_jacfunc(self.ids, self.jwr_ids)
        self.wr.add_jacfunc(self.iqr, self.jwr_iqr)
        self.wr.add_jacfunc(self.idr, self.jwr_idr)
        self.wr.add_jacfunc( self.wr, self.jwr_wr )

        # ports:

        self.id = self.iqs
        self.iq = self.ids

        self.vds = None  # terminal voltage d atom
        self.vqs = None  # terminal voltage q atom

    def connect(self, bus):

        self.vds = bus.vq
        self.vqs = bus.vd

        self.iqs.add_connection(self.vqs)
        self.iqr.add_connection(self.vqs)

        self.ids.add_connection(self.vds)
        self.idr.add_connection(self.vds)

    def vdr(self):
        return 0.0

    def vqr(self):
        return 0.0
```

```
    def Te(self, iqs, ids, iqr, idr):

        fqs = self.Lls * iqs + self.Lm * (iqs + iqr)
        fds = self.Lls * ids + self.Lm * (ids + idr)
        return (3*self.P/4) * (fds * iqs - fqs * ids)

    def Tm(self, wr):
        return self.Tb * (wr / self.ws)**3

    @staticmethod
    def diqs(self, iqs):
        return ((self.Lm**2*self.ids.q+(self.Lm**2+self.Llr*self.Lm)*self.idr.q)*self.ws
          +((-2*self.Lm**2+(-self.Lls-self.Llr)*self.Lm-self.Llr*self.Lls)*self.ids.q
            +(-2*self.Lm**2-2*self.Llr*self.Lm)*self.idr.q)*self.wr.q+(self.Lm+self.Llr)*self.vqs.
    q-self.Lm*self.vqr()
            +(-self.Lm-self.Llr)*self.Rs*self.iqs.q+self.Lm*self.Rr*self.iqr.q)/((self.Lls+self.
    Llr)*self.Lm+self.Llr*self.Lls)

    @staticmethod
    def dids(self, ids):
        return -((self.Lm**2*self.iqs.q+(self.Lm**2+self.Llr*self.Lm)*self.iqr.q)*self.ws
          +((-2*self.Lm**2+(-self.Lls-self.Llr)*self.Lm-self.Llr*self.Lls)*self.iqs.q
        +(-2*self.Lm**2-2*self.Llr*self.Lm)*self.iqr.q)*self.wr.q+(-self.Lm-self.Llr)*self.vds.q+
    self.Lm*self.vdr()
        +(self.Lm+self.Llr)*self.Rs*self.ids.q-self.Lm*self.Rr*self.idr.q)/((self.Lls+self.Llr)*
    self.Lm+self.Llr*self.Lls)

    @staticmethod
    def diqr(self, iqr):
        return -(((self.Lm**2+self.Lls*self.Lm)*self.ids.q+(self.Lm**2+(self.Lls+self.Llr)*self.Lm
    +self.Llr*self.Lls)*self.idr.q)*self.ws
          +((-2*self.Lm**2-2*self.Lls*self.Lm)*self.ids.q+(-2*self.Lm**2+(-self.Lls-self.Llr)*self
    .Lm-self.Llr*self.Lls)*self.idr.q)*self.wr.q
          +self.Lm*self.vqs.q+(-self.Lm-self.Lls)*self.vqr()-self.Lm*self.Rs*self.iqs.q+(self.Lm+
    self.Lls)*self.Rr*self.iqr.q)/((self.Lls+self.Llr)*self.Lm+self.Llr*self.Lls)

    @staticmethod
    def didr(self, idr):
        return (((self.Lm**2+self.Lls*self.Lm)*self.iqs.q+(self.Lm**2+(self.Lls+self.Llr)*self.Lm+
    self.Llr*self.Lls)*self.iqr.q)*self.ws
          +((-2*self.Lm**2-2*self.Lls*self.Lm)*self.iqs.q+(-2*self.Lm**2+(-self.Lls-self.Llr)*self
    .Lm-self.Llr*self.Lls)*self.iqr.q)*self.wr.q
          -self.Lm*self.vds.q+(self.Lm+self.Lls)*self.vdr()+self.Lm*self.Rs*self.ids.q+(-self.Lm-
    self.Lls)*self.Rr*self.idr.q)/((self.Lls+self.Llr)*self.Lm+self.Llr*self.Lls)
```

```
1371
1372     @staticmethod
1373     def dwr(self, wr):
1374         return (self.P / (2.0 * self.J)) * (0.75 * self.P * ((self.Lls
1375                 * self.ids.q + self.Lm * (self.ids.q + self.idr.q)) * self.iqs.q
1376                 - (self.Lls * self.iqs.q + self.Lm * (self.iqs.q + self.iqr.q))
1377                 * self.ids.q) - self.Tb * (wr / self.ws)**3)
1378
1379     @staticmethod
1380     def jiqs_iqs(self, iqs):
1381         return (((-self.Lm-self.Llr)*self.Rs)/((self.Lls+self.Llr)*self.Lm
1382                 +self.Llr*self.Lls))
1383
1384     @staticmethod
1385     def jiqs_ids(self, iqs, ids):
1386         return ((self.Lm**2*self.ws+(-2*self.Lm**2+(-self.Lls-self.Llr)*self.Lm
1387                 -self.Llr*self.Lls)*self.wr.q)/((self.Lls+self.Llr)*self.Lm
1388                 +self.Llr*self.Lls))
1389
1390     @staticmethod
1391     def jiqs_iqr(self, iqs, iqr):
1392         return  ((self.Lm*self.Rr)/((self.Lls+self.Llr)*self.Lm+self.Llr*self.Lls))
1393
1394     @staticmethod
1395     def jiqs_idr(self, iqs, idr):
1396         return (((self.Lm**2+self.Llr*self.Lm)*self.ws+(-2*self.Lm**2-2*self.Llr*self.Lm)
1397                  *self.wr.q)/((self.Lls+self.Llr)*self.Lm+self.Llr*self.Lls))
1398
1399     @staticmethod
1400     def jiqs_wr(self, iqs, wr):
1401         return (((-2*self.Lm**2+(-self.Lls-self.Llr)*self.Lm-self.Llr*self.Lls)
1402                  *self.ids.q+(-2*self.Lm**2-2*self.Llr*self.Lm)*self.idr.q)
1403                 /((self.Lls+self.Llr)*self.Lm+self.Llr*self.Lls))
1404
1405     @staticmethod
1406     def jids_iqs(self, ids, iqs):
1407         return ((-self.Lm**2*self.ws-(-2*self.Lm**2+(-self.Lls-self.Llr)
1408                 *self.Lm-self.Llr*self.Lls)*self.wr.q)/((self.Lls+self.Llr)
1409                 *self.Lm+self.Llr*self.Lls))
1410
1411     @staticmethod
1412     def jids_ids(self, ids):
1413         return (-((self.Lm+self.Llr)*self.Rs)/((self.Lls+self.Llr)*self.Lm
1414                 +self.Llr*self.Lls))
1415
```

```
    @staticmethod
    def jids_iqr(self, ids, iqr):
        return ((-(self.Lm**2+self.Llr*self.Lm)*self.ws-(-2*self.Lm**2-2
                *self.Llr*self.Lm)*self.wr.q)/((self.Lls+self.Llr)*self.Lm
                +self.Llr*self.Lls))

    @staticmethod
    def jids_idr(self, ids, idr):
        return ((self.Lm*self.Rr)/((self.Lls+self.Llr)*self.Lm+self.Llr*self.Lls))

    @staticmethod
    def jids_wr(self, ids, wr):
        return ((-(-2*self.Lm**2+(-self.Lls-self.Llr)*self.Lm-self.Llr*self.Lls)
                 *self.iqs.q-(-2*self.Lm**2-2*self.Llr*self.Lm)*self.iqr.q)
                /((self.Lls+self.Llr)*self.Lm+self.Llr*self.Lls))

    @staticmethod
    def jiqr_iqs(self, iqr, iqs):
        return ((self.Lm*self.Rs)/((self.Lls+self.Llr)*self.Lm+self.Llr*self.Lls))

    @staticmethod
    def jiqr_ids(self, iqr, ids):
        return ((-(self.Lm**2+self.Lls*self.Lm)*self.ws-(-2*self.Lm**2
                -2*self.Lls*self.Lm)*self.wr.q)/((self.Lls+self.Llr)*self.Lm
                +self.Llr*self.Lls))

    @staticmethod
    def jiqr_iqr(self, iqr):
        return (-((self.Lm+self.Lls)*self.Rr)/((self.Lls+self.Llr)*self.Lm
                +self.Llr*self.Lls))

    @staticmethod
    def jiqr_idr(self, iqr, idr):
        return ((-(self.Lm**2+(self.Lls+self.Llr)*self.Lm+self.Llr*self.Lls)
                *self.ws-(-2*self.Lm**2+(-self.Lls-self.Llr)*self.Lm-self.Llr
                *self.Lls)*self.wr.q)/((self.Lls+self.Llr)*self.Lm+self.Llr*self.Lls))

    @staticmethod
    def jiqr_wr(self, iqr, wr):
        return ((-(-2*self.Lm**2-2*self.Lls*self.Lm)*self.ids.q-(-2*self.Lm**2
                +(-self.Lls-self.Llr)*self.Lm-self.Llr*self.Lls)*self.idr.q)
                /((self.Lls+self.Llr)*self.Lm+self.Llr*self.Lls))

    @staticmethod
    def jidr_iqs(self, idr, iqs):
```

```
        return (((self.Lm**2+self.Lls*self.Lm)*self.ws+(-2*self.Lm**2-2
                *self.Lls*self.Lm)*self.wr.q)/((self.Lls+self.Llr)*self.Lm
                +self.Llr*self.Lls))

    @staticmethod
    def jidr_ids(self, idr, ids):
        return ((self.Lm*self.Rs)/((self.Lls+self.Llr)*self.Lm+self.Llr*self.Lls))

    @staticmethod
    def jidr_iqr(self, idr, iqr):
        return (((self.Lm**2+(self.Lls+self.Llr)*self.Lm+self.Llr*self.Lls)*self.ws
                 +(-2*self.Lm**2+(-self.Lls-self.Llr)*self.Lm-self.Llr*self.Lls)
                 *self.wr.q)/((self.Lls+self.Llr)*self.Lm+self.Llr*self.Lls))

    @staticmethod
    def jidr_idr(self, idr):
        return (((-self.Lm-self.Lls)*self.Rr)/((self.Lls+self.Llr)*self.Lm+self.Llr*self.Lls))

    @staticmethod
    def jidr_wr(self, idr, wr):
        return (((-2*self.Lm**2-2*self.Lls*self.Lm)*self.iqs.q+(-2*self.Lm**2
                +(-self.Lls-self.Llr)*self.Lm-self.Llr*self.Lls)*self.iqr.q)
                /((self.Lls+self.Llr)*self.Lm+self.Llr*self.Lls))

    @staticmethod
    def jwr_iqs(self, wr, iqs):
        return ((0.375*self.P**2*(self.Lm*(self.ids.q+self.idr.q)-(self.Lm+self.Lls)
                *self.ids.q+self.Lls*self.ids.q))/self.J)

    @staticmethod
    def jwr_ids(self, wr, ids):
        return ((0.375*self.P**2*(-self.Lm*(self.iqs.q+self.iqr.q)+(self.Lm+self.Lls)
                *self.iqs.q-self.Lls*self.iqs.q))/self.J)

    @staticmethod
    def jwr_iqr(self, wr, iqr):
        return (-(0.375*self.Lm*self.P**2*self.ids.q)/self.J)

    @staticmethod
    def jwr_idr(self, wr, idr):
        return ((0.375*self.Lm*self.P**2*self.iqs.q)/self.J)

    @staticmethod
    def jwr_wr(self, wr):
        return (-(1.5*self.P*self.Tb*self.wr.q**2)/(self.J*self.ws**3))

```

```
class TRLoadDQ(Device):

    """Transformer rectifier load average model.


                .-----.-----.------VVVV---UUUU----.------.
               _|_   _|_   _|_      Rdc    Ldc     |      |
              /_\   /_\   /_\                      |      |
       Lc      |     |     |        ----->         |      |           +
    o--UUUU---o     |     |         idc           |      |
       Lc      |     |     |                     _|_     <.
    o--UUUU----------o     |                  C  ___     <.  R      vdc
                |     |     |                     |      <.
    o--UUUU----------------o                      |      |
       Lc     _|_   _|_   _|_                     |      |
              /_\   /_\   /_\                      |      |           -
               |     |     |                      |      |
               '-----'-----'----------------------'------'

               Average DQ model:

                     .-------------------------------------------------------.
                     |             TRLoadDQ       Rdc  Ldc                  |
    .----. vtermd    |       ,^.                .--VVV--UUU--.----.         |
    | DQ |---o-------|-----<   >--.             |   idc -->  |    |    +    |
    |Node|    id ->  |  igd '.'   |  .     + ,^.            _|_  <.         |
    |    |           |            +--||-     <   >     C    ___  <. R  vdc  |
    |    |    iq ->  |  igq ,^.   |  '    vg '.'            |    <.         |
    |    |---o-------|-----<   >--'           |             |    |    -    |
    '----' vtermq    |       '.'              '-------------'----'         |
                     '-------------------------------------------------------'

    """

    def __init__(self, name, w=2*PI*60.0, alpha_cmd=0.0, Lc=76.53e-6, Rdc=0.0,
                 Ldc=0.383e-3, R=3.16, C=0.384e-3, Nps=1.0, idc0=0.0, vdc0=0.0,
                 dq_i=1e0, dq_v=1e0):

        self.name = name

        # params:

        self.w = w
        self.alpha_cmd = alpha_cmd
        self.Lc = Lc
        self.Rdc = Rdc
```

```
        self.Ldc = Ldc
        self.R = R
        self.C = C
        self.Nps = Nps

        # intial conditions:

        self.idc0 = idc0
        self.vdc0 = vdc0

        # delta q:

        self.dq_i = dq_i
        self.dq_v = dq_v

        # call super:

        Device.__init__(self, name)

        # derived:

        self.ke = 1.0 / (sqrt(2.0) * self.Nps)

        # state atoms:

        self.id = SourceAtom("id", source_type=SourceType.FUNCTION,
                             srcfunc=self.get_id, units="A", dq=dq_i)

        self.iq = SourceAtom("iq", source_type=SourceType.FUNCTION,
                             srcfunc=self.get_iq, units="A", dq=dq_i)

        self.idc = StateAtom("idc", x0=idc0, derfunc=self.didc, units="A", dq=dq_i)
        self.vdc = StateAtom("vdc", x0=vdc0, derfunc=self.dvdc, units="V", dq=dq_v)

        self.add_atoms(self.id, self.iq, self.idc, self.vdc)

        # atom connections:

        self.idc.add_connection(self.vdc)
        self.vdc.add_connection(self.idc)

        self.id.add_connection(self.idc)
        self.iq.add_connection(self.idc)

        # ports:

        self.atomd = self.id
        self.atomq = self.iq
```

```
        self.vdg = None  # terminal voltage d atom
        self.vqg = None  # terminal voltage q atom

    def connect(self, bus):

        self.vdg = bus.atomd
        self.vqg = bus.atomq

        self.id.add_connection(bus.atomd)
        self.id.add_connection(bus.atomq)

        self.iq.add_connection(bus.atomd)
        self.iq.add_connection(bus.atomq)

        self.idc.add_connection(bus.atomd)
        self.idc.add_connection(bus.atomq)

    def E(self, vdg, vqg):
        return self.ke * sqrt(vdg**2 + vqg**2)

    def phi(self, vdg, vqg):
        return atan2(vdg, vqg)

    def get_angles(self, idc, vdg, vqg):

        e = sqrt(6) * self.E(vdg, vqg)

        if e == 0.0:
           return 0.0, 0.0

        k = cos(self.alpha_cmd) - 2 * self.Lc * self.w * idc / e
        mu = -self.alpha_cmd + acos(k)
        alpha = self.alpha_cmd

        if mu >= PI_3 or mu + alpha >= PI:
            mu = PI_3
            alpha = PI_3 - acos((2 * self.Lc * self.w * idc) / e)

        return alpha, mu

    def iqg_com(self, vdg, vqg, idc, alpha, mu):

        E = self.E(vdg, vqg)
        k1 = 2 * sqrt(3) / PI
        k2 = 3 * sqrt(3) * E / (PI * self.Lc * self.w)
```

```
        k3 = 3 * sqrt(2) * E / (4 * PI * self.Lc * self.w)

        return (k1 * idc * (sin(mu + alpha - PI5_6) - sin(alpha - PI5_6))
              * k2 * cos(alpha) * (cos(mu + alpha) - cos(alpha))
              + k3 * (cos(2*mu) - cos(2*alpha + 2*mu)))

    def idg_com(self, vdg, vqg, idc, alpha, mu):

        E = self.E(vdg, vqg)
        k1 = 2 * sqrt(3) / PI
        k2 = 3 * sqrt(2) * E / (PI * self.Lc * self.w)
        k3 = 3 * sqrt(2) * E / (4 * PI * self.Lc * self.w)
        k4 = 3 * sqrt(2) * E / (2 * PI * self.Lc * self.w)

        return (k1 * idc * (-cos(mu + alpha - PI5_6) + cos(alpha - PI5_6))
              * k2 * cos(alpha) * (sin(mu + alpha) - sin(alpha))
              + k3 * (sin(2*mu) - sin(2*alpha + 2*mu))
              - k4 * mu)

    def iqg_cond(self, idc, alpha, mu):
        return 2 * sqrt(3) / PI * idc * (sin(alpha + PI7_6) - sin(alpha + mu + PI5_6))

    def idg_cond(self, idc, alpha, mu):
        return 2 * sqrt(3) / PI * idc * (-cos(alpha + PI7_6) + cos(alpha + mu + PI5_6))

    def iqg(self, idc, vdg, vqg, alpha, mu):
        return (self.iqg_com(vdg, vqg, idc, alpha, mu) + self.iqg_cond(idc, alpha, mu))

    def idg(self, idc, vdg, vqg, alpha, mu):
        return (self.idg_com(vdg, vqg, idc, alpha, mu) + self.idg_cond(idc, alpha, mu))

    @staticmethod
    def get_iq(self, t):  # iq depends on: vd, vq, idc

        vdg = self.vdg.q
        vqg = self.vqg.q
        idc = self.idc.q
        phi = self.phi(vdg, vqg)
        alpha, mu = self.get_angles(idc, vdg, vqg)

        return (self.iqg(idc, vdg, vqg, alpha, mu) * cos(phi)
              - self.idg(idc, vdg, vqg, alpha, mu) * sin(phi))

    @staticmethod
    def get_id(self, t):  # id depends on: vd, vq, idc
```

```
1693
1694            vdg = self.vdg.q
1695            vqg = self.vqg.q
1696            idc = self.idc.q
1697            phi = self.phi(vdg, vqg)
1698            alpha, mu = self.get_angles(idc, vdg, vqg)
1699
1700            return (self.iqg(idc, vdg, vqg, alpha, mu) * sin(phi)
1701                  + self.idg(idc, vdg, vqg, alpha, mu) * cos(phi))
1702
1703        @staticmethod
1704        def didc(self, idc):  # idc depends on: vd, vq, vdc
1705
1706            vdg = self.vdg.q
1707            vqg = self.vqg.q
1708            alpha, mu = self.get_angles(idc, vdg, vqg)
1709            E = self.E(vdg, vqg)
1710            k1 = 3 * sqrt(3) * sqrt(2) / PI
1711
1712            e = k1 * E * cos(alpha)
1713            req = self.Rdc + 3/PI * self.Lc * self.w
1714
1715            return (e - req * idc - self.vdc.q) / (self.Ldc + 2*self.Lc)
1716
1717        @staticmethod
1718        def dvdc(self, vdc):  # vdc depends on: idc
1719
1720            return (self.idc.q - vdc / self.R) / self.C
1721
1722
1723    class TRLoadDQ2(Device):
1724
1725        """Transformer rectifier load average model simplified (alpha=0).
1726
1727        Sd = sqrt(3/2) * 2 * sqrt(3)/pi * cos(phi)
1728        Sq = -sqrt(3/2) * 2 * sqrt(3)/pi * sin(phi)
1729
1730        id = idc/Sd
1731        iq = idc/Sq
1732
1733        idc = Sd*id + Sq*iq
1734
1735        edc = Sd*vd + Sq*vq
1736
1737        vg - vdc = idc*Rdc + didc*Ldc
1738
1739        didc*Ldc = (edc - vdc - idc*Rdc) / Ldc
```

```
1740
1741      idc = dvdc*C + vdc/R
1742
1743      did = (pi/(3*sqrt(2)*cos(phi))) * didc
1744      diq = (-pi/(3*sqrt(2)*sin(phi))) * didc
1745
1746      dvg = (self.id.q / self.S - vdc / self.R) / self.C)
1747
1748      """
1749
1750      def __init__(self, name, w=2*PI*60.0, Lc=76.53e-6, Rdc=0.0,
1751                   Ldc=0.383e-3, R=3.16, C=0.384e-3, id0=0.0,
1752                   vdc0=0.0, dq_i=1e0, dq_v=1e0):
1753
1754          self.name = name
1755
1756          # params:
1757
1758          self.w = w
1759          self.Lc = Lc
1760          self.Rdc = Rdc
1761          self.Ldc = Ldc
1762          self.R = R
1763          self.C = C
1764
1765          # intial conditions:
1766
1767          self.id0 = id0
1768          self.vdc0 = vdc0
1769
1770          # delta q:
1771
1772          self.dq_i = dq_i
1773          self.dq_v = dq_v
1774
1775          # cached:
1776
1777          self.S = sqrt(3/2) * 2 * sqrt(3) / PI
1778          self.S2 = self.S**2
1779          self.Req = Rdc + 3 / PI * Lc * w
1780          self.Leq = Ldc + 2 * Lc
1781
1782          # call super:
1783
1784          Device.__init__(self, name)
1785
1786          # atoms:
1787
```

```
        self.iq = SourceAtom("iq", source_type=SourceType.CONSTANT, x0=0.0,
                             units="A", dq=dq_i)

        self.id = StateAtom("id", x0=id0, derfunc=self.did, units="A", dq=dq_i)

        self.vdc = StateAtom("vdc", x0=vdc0, derfunc=self.dvdc, units="V", dq=dq_v)

        self.add_atoms(self.id, self.iq, self.vdc)

        # atom connections:

        self.id.add_connection(self.vdc)
        self.vdc.add_connection(self.id)

        self.id.add_jacfunc(self.id, self.jid_id)
        self.id.add_jacfunc(self.vdc, self.jid_vdc)
        self.vdc.add_jacfunc(self.id, self.jvdc_id)
        self.vdc.add_jacfunc(self.vdc, self.jvdc_vdc)

        # ports:

        self.vd = None  # terminal voltage d atom
        self.vq = None  # terminal voltage q atom

    def connect(self, bus):

        self.vd = bus.vd
        self.id.add_connection(bus.vd)

    @staticmethod
    def did(self, id):  # id depends on: vd, vdc
        return (self.vd.q * self.S2 - self.Req * id - self.vdc.q * self.S) / self.Leq

    @staticmethod
    def dvdc(self, vdc):  # vdc depends on: id
        return (self.id.q / self.S - vdc / self.R) / self.C

    @staticmethod
    def jid_id(self, id):
       return -self.Req / self.Leq

    @staticmethod
    def jid_vdc(self, id, vdc):
       return -self.S / self.Leq

    @staticmethod
    def jvdc_id(self, vdc, id):
```

```
1835        return 1 / (self.C * self.S)
1836
1837    @staticmethod
1838    def jvdc_vdc(self, vdc):
1839        return -1 / (self.C * self.R)
1840
1841
1842 class Pendulum(Device):
1843
1844    """  Simple non-linear example.
1845    """
1846
1847    def __init__(self, name, r=1.0, l=1.0, theta0=0.0, omega0=0.0,
1848                 dq_omega=1e-3, dq_theta=1e-3):
1849
1850        Device.__init__(self, name)
1851
1852        self.l = l
1853        self.r = r
1854        self.g = 9.81
1855
1856        self.omega = StateAtom("omega", x0=omega0, derfunc=self.domega, units="rad/s", dq=dq_omega
    )
1857        self.theta = StateAtom("theta", x0=theta0, derfunc=self.dtheta, units="rad",   dq=dq_theta
    )
1858
1859        self.add_atoms(self.omega, self.theta)
1860
1861        self.omega.add_connection(self.theta)
1862        self.theta.add_connection(self.omega)
1863
1864        # jacobian:
1865        self.omega.add_jacfunc(self.theta, self.j21)
1866        self.omega.add_jacfunc(self.omega, self.j22)
1867        self.theta.add_jacfunc(self.omega, self.j12)
1868
1869    @staticmethod
1870    def domega(self, omega):
1871        return -(self.r*omega + self.g / self.l * sin(self.theta.q))
1872
1873    @staticmethod
1874    def dtheta(self, theta):
1875        return self.omega.q
1876
1877    @staticmethod
1878    def j12(self, theta, omega):
1879        return 1.0
```

```
1880
1881     @staticmethod
1882     def j21(self, omega, theta):
1883         return -self.g / self.l * cos(theta)
1884
1885     @staticmethod
1886     def j22(self, omega):
1887         return -self.r
1888
1889
1890 class LiqssTest(Device):
1891
1892     """Original Liqss test/demo stiff system from the paper
1893
1894     'Linearly Implicit Quantization Based Integration
1895     Methods for Stiff Ordinary Differential Equations'
1896
1897     dx1 = 0.01 * x2
1898     dx2 = -100 * x1 - 100 * x2 + 2020
1899
1900     lambda1 ~= -0.01 and lambda2 ~= -100.0 (stiffness ratio of 1e4)
1901
1902     """
1903
1904     def __init__(self, name, x10=0.0, x20=20.0, dq1=1.0, dq2=1.0):
1905
1906         Device.__init__(self, name)
1907
1908         self.x1 = StateAtom("x1", x0=x10, derfunc=self.dx1, der2func=self.d2x1,
1909                             units="", dq=dq1)
1910
1911
1912         self.x2 = StateAtom("x2", x0=x20, derfunc=self.dx2, der2func=self.d2x2,
1913                             units="", dq=dq2)
1914
1915         self.add_atoms(self.x1, self.x2)
1916
1917         self.x1.add_connection(self.x2)
1918         self.x2.add_connection(self.x1)
1919         self.x2.add_connection(self.x2)
1920
1921         self.x1.add_jacfunc(self.x2, self.j12)
1922         self.x2.add_jacfunc(self.x1, self.j21)
1923         self.x2.add_jacfunc(self.x2, self.j22)
1924
1925     @staticmethod
1926     def dx1(self, x1):
```

```
        return 0.01 * self.x2.q

    @staticmethod
    def dx2(self, x2):
        return -100 * self.x1.q - 100 * x2 + 2020

    @staticmethod
    def d2x1(self, x1, d1):
        return 0.01 * self.x2.dx

    @staticmethod
    def d2x2(self, x2, d2):
        return -100 * self.x1.dx - 100 * d2

    @staticmethod
    def j12(self, x1, x2):
        return 0.01

    @staticmethod
    def j21(self, x2, x1):
        return -100

    @staticmethod
    def j22(self, x2):
        return -100


class MLiqssTest(Device):

    """mLiqss test/demo system from the paper:

    'Improving Linearly Implicit Quantized State System Methods'

    dx1 = -x1 - x2 + 0.2
    dx2 = x1 - x2 + 1.2

    """

    def __init__(self, name, x10=-4.0, x20=4.0, dq1=1.0, dq2=1.0):

        Device.__init__(self, name)

        self.x1 = StateAtom("x1", x0=x10, derfunc=self.dx1, der2func=self.ddx1,
                            units="", dq=dq1)


        self.x2 = StateAtom("x2", x0=x20, derfunc=self.dx2, der2func=self.ddx2,
                            units="", dq=dq2)
```

```
1975
1976            self.add_atoms(self.x1, self.x2)
1977
1978            #self.x1.add_connection(self.x1)
1979            self.x1.add_connection(self.x2)
1980            self.x2.add_connection(self.x1)
1981            #self.x2.add_connection(self.x2)
1982
1983            self.x1.add_jacfunc(self.x1, self.j11)
1984            self.x1.add_jacfunc(self.x2, self.j12)
1985            self.x2.add_jacfunc(self.x1, self.j21)
1986            self.x2.add_jacfunc(self.x2, self.j22)
1987
1988        @staticmethod
1989        def dx1(self, x1):
1990            return -x1 - self.x2.q + 0.2
1991
1992        @staticmethod
1993        def dx2(self, x2):
1994            return self.x1.q - x2 + 1.2
1995
1996        @staticmethod
1997        def ddx1(self, x1, dx1):
1998            return -dx1 - self.x2.dx
1999
2000        @staticmethod
2001        def ddx2(self, x2, dx2):
2002            return self.x1.dx - dx2
2003
2004        @staticmethod
2005        def j11(self, x1):
2006            return -1.0
2007
2008        @staticmethod
2009        def j12(self, x1, x2):
2010            return -1.0
2011
2012        @staticmethod
2013        def j21(self, x2, x1):
2014            return 1.0
2015
2016        @staticmethod
2017        def j22(self, x2):
2018            return -1.0
2019
2020
2021    class CoupledPendulums(Device):
```

```
    """
    dw1 = (-g/l1 * sin(th1)/cos(th1) + w1*w1 * sin(th1)/cos(th1)
          + k*l2/(m1*l1) * sin(th2) / cos(th1)
          + k*l1/(m1*l1) * sin(th1) / cos(th1))

    dw2 = (-g/l2 * sin(th2)/cos(th2) + w2*w2 * sin(th2)/cos(th2)
          + k*l1/(m2*l2) * sin(th1) / cos(th2)
          + k*l2/(m2*l2) * sin(th2) / cos(th2))

    dth1 = w1
    dth2 = w2

    """

    def __init__(self, name, k=1.0, r1=1.0, r2=1.0, l1=1.0, l2=1.0, m1=1.0,
                 m2=1.0, th10=0.0, w10=0.0, th20=0.0, w20=0.0, dq_w=1e-3,
                 dq_th=1e-3):

        Device.__init__(self, name)

        self.k = k
        self.r1 = r1
        self.r2 = r2
        self.l1 = l1
        self.l2 = l2
        self.m1 = m1
        self.m2 = m2

        self.g = 9.81

        self.w1  = StateAtom("w1", x0=w10,  derfunc=self.dw1,
                             der2func=self.d2w1, units="rad/s", dq=dq_w)


        self.th1 = StateAtom("th1", x0=th10, derfunc=self.dth1,
                             der2func=self.d2w2, units="rad", dq=dq_th)


        self.w2  = StateAtom("w2", x0=w20, derfunc=self.dw2,
                             der2func=self.d2th1, units="rad/s", dq=dq_w)


        self.th2 = StateAtom("th2", x0=th20, derfunc=self.dth2,
                             der2func=self.d2th2, units="rad", dq=dq_th)


        self.add_atoms(self.w1, self.th1, self.w2, self.th2)
```

```
        """
        dw1 = (-g/l1 * sin(th1)/cos(th1) + w1*w1 * sin(th1)/cos(th1)
              + k*l2/(m1*l1) * sin(th2) / cos(th1)
              + k*l1/(m1*l1) * sin(th1) / cos(th1))

        dw2 = )-g/l2 * sin(th2)/cos(th2) + w2*w2 * sin(th2)/cos(th2)
              + k*l1/(m2*l2) * sin(th1) / cos(th2)
              + k*l2/(m2*l2) * sin(th2) / cos(th2))

        dth1 = w1
        dth2 = w2

        """

        #self.w1.add_connection(self.w1)
        self.w1.add_connection(self.th1)
        self.w1.add_connection(self.w2)

        #self.w2.add_connection(self.w2)
        self.w2.add_connection(self.th2)
        self.w2.add_connection(self.w1)

        self.th1.add_connection(self.w1)
        self.th2.add_connection(self.w2)

        # jacobian:

        self.w1.add_jacfunc(self.th1, self.jw1_th1)
        self.w1.add_jacfunc(self.th2, self.jw1_th2)
        self.w1.add_jacfunc(self.w1, self.jw1_w1)

        self.w2.add_jacfunc(self.th1, self.jw2_th1)
        self.w2.add_jacfunc(self.th2, self.jw2_th2)
        self.w2.add_jacfunc(self.w2, self.jw2_w2)

        self.th1.add_jacfunc(self.w1, self.jth1_w1)
        self.th2.add_jacfunc(self.w2, self.jth2_w2)

    @staticmethod
    def dth1(self, th1):
        return self.w1.q

    @staticmethod
    def dw1(self, w1):
        return (-self.r1 * w1 - self.g / self.l1
```

```
                * sin(self.th1.q) / cos(self.th1.q)
                + w1**2 * sin(self.th1.q) / cos(self.th1.q)
                + self.k * self.l2/(self.m1 * self.l1)
                * sin(self.th2.q) / cos(self.th1.q)
                + self.k * self.l1 / (self.m1 * self.l1)
                * sin(self.th1.q) / cos(self.th1.q))

    @staticmethod
    def dth2(self, th2):
        return self.w2.q

    @staticmethod
    def dw2(self, w2):
        return (-self.r2 * w2 - self.g / self.l2
                * sin(self.th2.q) / cos(self.th2.q)
                + w2**2 * sin(self.th2.q) / cos(self.th2.q)
                + self.k * self.l1 / (self.m2 * self.l2)
                * sin(self.th1.q) / cos(self.th2.q)
                + self.k * self.l2 / (self.m2 * self.l2)
                * sin(self.th2.q) / cos(self.th2.q))

    @staticmethod
    def d2w1(self, w1, dw1):
        return ((2 * w1 * sin(self.th1.q) * dw1) / cos(self.th1.q)
                + (self.k * self.l2 * cos(self.th2.q) * self.th2.d)
                / (self.l1 * self.m1 * cos(self.th1.q)) + (self.k * self.l2
                * sin(self.th1.q) * self.th1.d * sin(self.th2.q)) / (self.l1
                * self.m1 * cos(self.th1.q)**2) + (w1**2 * sin(self.th1.q)**2
                * self.th1.d) / cos(self.th1.q)**2 + (self.k
                * sin(self.th1.q)**2 * self.th1.d) / (self.m1
                * cos(self.th1.q)**2) - (self.g * sin(self.th1.q)**2
                * self.th1.d) / (self.l1 * cos(self.th1.q)**2) + w1**2
                * self.th1.d + (self.k * self.th1.d) / self.m1 - (self.g
                * self.th1.d) / self.l1)

    @staticmethod
    def d2w2(self, w2, dw2):
        return ((2 * w2 * sin(self.th2.q) * dw2) / cos(self.th2.q)
                + (w2**2 * sin(self.th2.q)**2 * self.th2.d) / cos(self.th2.q)**2
                + (self.k * sin(self.th2.q)**2 * self.th2.d) / (self.m2
                * cos(self.th2.q)**2) - (self.g * sin(self.th2.q)**2
                * self.th2.d) / (self.l2 * cos(self.th2.q)**2) + (self.k * self.l1
                * sin(self.th1.q) * sin(self.th2.q) * self.th2.d) / (self.l2
                * self.m2 * cos(self.th2.q)**2) + w2**2 *self.th2.d
```

```
                + (self.k * self.th2.d) / self.m2 - (self.g * self.th2.d)
                / self.l2 + (self.k * self.l1 *cos(self.th1.q) * self.th1.d)
                / (self.l2 * self.m2 * cos(self.th2.q)))

    @staticmethod
    def d2th1(self, th1, dth1):
        return self.w1.d

    @staticmethod
    def d2th2(self, th2, dth2):
        return self.w2.d

    @staticmethod
    def jw1_th1(self, w1, th1):
        return (sin(th1)**2 * w1**2
                / cos(th1)**2 + w1**2
                + self.k * self.l2 * sin(th1) * sin(self.th2.q)
                / (self.l1 * self.m1 * cos(th1)**2)
                + (self.k * sin(th1)**2)
                / (self.m1 * cos(th1)**2)
                - (self.g * sin(th1)**2)
                / (self.l1 * cos(th1)**2)
                + self.k / self.m1 - self.g / self.l1)

    @staticmethod
    def jw1_th2(self, w1, th2):
        return (self.k * self.l2 * cos(th2)
               / (self.l1 * self.m1 * cos(self.th1.q)))

    @staticmethod
    def jw1_w1(self, w1):
        return 2.0 * sin(self.th1.q) * w1 / cos(self.th1.q)

    @staticmethod
    def jw2_th1(self, w2, th1):
        return (self.k * self.l1 * cos(th1)
                / (self.l2 * self.m2 * cos(self.th2.q)))

    @staticmethod
    def jw2_th2(self, w2, th2):
        return (sin(th2)**2 * w2**2
                / cos(th2)**2 + w2**2
                + (self.k *  sin(th2)**2)
                / (self.m2 * cos(th2)**2)
                - (self.g *  sin(th2)**2)
```

```
                / (self.l2 * cos(th2)**2)
                + (self.k * self.l1 * sin(self.th1.q) * sin(th2))
                / (self.l2 * self.m2 * cos(th2)**2)
                + self.k / self.m2 - self.g / self.l2)

    @staticmethod
    def jw2_w2(self, w2):
        return 2.0 * sin(self.th2.q) * w2 / cos(self.th2.q)

    @staticmethod
    def jth1_w1(self, th, w1):
        return 1.0

    @staticmethod
    def jth2_w2(self, th2, w2):
        return 1.0


# ============================ SYMBOLIC DEVICES ================================


class Pendulum2(SymbolicDevice):

    def __init__(self, name, mu=1.0, l=1.0, g=9.81, w0=0.0, a0=0.0,
                 dq_w=1e-3, dq_a=1e-3):

        SymbolicDevice.__init__(self, name)

        self.add_constant("g", desc="Acceleration of gravity", units="m.s^-2", value=g)

        self.add_parameter("mu", desc="Viscous friction", value=mu)
        self.add_parameter("l", desc="Length", value=l)

        self.add_state("w", "dw_dt", desc="Angular velocity", units="rad/s", x0=w0, dq=dq_w)
        self.add_state("a", "da_dt", desc="Angle", units="rad", x0=a0, dq=dq_a)

        self.add_diffeq("dw_dt + (mu * w + g / l * sin(a))")
        self.add_diffeq("da_dt - w")


class LimNode2(SymbolicDevice):

    def __init__(self, name, c, g=0.0, h=0.0, v0=0.0, dq=None):

        SymbolicDevice.__init__(self, name)

        self.add_parameter("c", desc="Capacitance", units="F", value=c)
```

```
        self.add_parameter("g", desc="Conductance", units="S", value=g)
        self.add_parameter("h", desc="Source Current", units="A", value=h)

        self.add_state("v", "dv_dt", desc="Voltage", units="V", x0=v0, dq=dq)

        self.add_electrical_port("positive", output="v", input="isum")

        self.add_diffeq("c * dv_dt + g * v - h - isum")


class LimBranch2(SymbolicDevice):

    def __init__(self, name, l, r=0.0, e=0.0, i0=0.0, dq=None):

        SymbolicDevice.__init__(self, name)

        self.add_parameter("l", desc="Inductance", units="H", value=l)
        self.add_parameter("r", desc="Resistance", units="Ohm", value=r)
        self.add_parameter("e", desc="Source Voltage", units="V", value=e)

        self.add_state("i", "di_dt", desc="Current", units="A", x0=i0, dq=dq)

        self.add_electrical_port("positive", output="i", input="vpos")
        self.add_electrical_port("negative", output="i", input="vneg", sign=-1)

        self.add_diffeq("l * di_dt + r * i - e + vpos - vneg")


class LimNodeDQ2(SymbolicDevice):

    def __init__(self, name, c, g=0.0, ws=2*pi*60, theta=0.0, h=0.0, vd0=0.0,
                 vq0=0.0, dq=None):

        SymbolicDevice.__init__(self, name)

        self.add_parameter("c", desc="Capacitance", units="F", value=c)
        self.add_parameter("g", desc="Conductance", units="S", value=g)
        self.add_parameter("h", desc="Source Current", units="A", value=h)
        self.add_parameter("ws", desc="Radian frequency", units="rad/s", value=ws)
        self.add_parameter("theta", desc="DQ Angle", units="rad", value=theta)

        self.add_state("vd", "dvd_dt", desc="D-axis Voltage", units="V", x0=vd0, dq=dq)
        self.add_state("vq", "dvq_dt", desc="Q-axis Voltage", units="V", x0=vq0, dq=dq)

        self.add_dq_port("positive", inputs=("id", "iq"), outputs=("vd", "vq"))

```

```
        self.add_diffeq("c * dvd_dt + (g + ws*c) * vd - h * cos(theta) - id")
        self.add_diffeq("c * dvq_dt + (g + ws*c) * vq - h * sin(theta) - iq")


class LimBranchDQ2(SymbolicDevice):

    def __init__(self, name, l, r=0.0, ws=2*pi*60, theta=0.0, e=0.0, id0=0.0,
                 iq0=0.0, dq=None):

        SymbolicDevice.__init__(self, name)

        self.add_parameter("l", desc="Inductance", units="H", value=l)
        self.add_parameter("r", desc="Resistance", units="Ohm", value=r)
        self.add_parameter("e", desc="Source Voltage", units="V", value=e)
        self.add_parameter("ws", desc="Radian frequency", units="rad/s", value=ws)
        self.add_parameter("theta", desc="DQ Angle", units="rad", value=theta)

        self.add_state("id", "did_dt", desc="D-axis Current", units="A", x0=id0, dq=dq)
        self.add_state("iq", "diq_dt", desc="Q-axis Current", units="A", x0=iq0, dq=dq)

        self.add_dq_port("positive", inputs=("vposd", "vposq"), outputs=("id", "iq"))
        self.add_dq_port("negative", inputs=("vnegd", "vnegq"), outputs=("id", "iq"), sign=-1)

        self.add_diffeq("l * did_dt + (r + ws*l) * id - e * cos(theta) + vposd - vnegd")
        self.add_diffeq("l * diq_dt + (r + ws*l) * iq - e * sin(theta) + vposq - vnegq")


class InductionMachineDQ2(SymbolicDevice):

    def __init__(self, name, ws=30*pi, P=4, Tb=26.53e3, Rs=31.8e-3,
                 Lls=0.653e-3, Lm=38e-3, Rr=24.1e-3, Llr=0.658e-3, J=250.0,
                 iqs0=0.0, ids0=0.0, iqr0=0.0, idr0=0.0, wr0=0.0, dq_i=1e-2,
                 dq_wr=1e-1):

        SymbolicDevice.__init__(self, name)

        self.add_parameter("ws",  desc="Synchronous Speed",         units="rad/s", value=ws)
        self.add_parameter("P",   desc="Pole Pairs",                units="",      value=P)
        self.add_parameter("Tb",  desc="Base Torque",               units="N.m",   value=Tb)
        self.add_parameter("Rs",  desc="Stator Resistance",         units="Ohm",   value=Rs)
        self.add_parameter("Lls", desc="Stator Leakage Inductance", units="H",     value=Lls)
        self.add_parameter("Lm",  desc="Magnetizing Inductance",    units="H",     value=Lm)
        self.add_parameter("Rr",  desc="Rotor Resistance",          units="Ohm",   value=Rr)
        self.add_parameter("Llr", desc="Rotor Leakage Inductance",  units="H",     value=Llr)
        self.add_parameter("J",   desc="Inertia",                   units="",      value=J)
```

```
2343
2344         self.add_state("ids", "dids_dt", desc="Stator D-axis Current", units="A",     x0=ids0, dq=
     dq_i)
2345         self.add_state("iqs", "diqs_dt", desc="Stator Q-axis Current", units="A",     x0=iqs0, dq=
     dq_i)
2346         self.add_state("idr", "didr_dt", desc="Rotor D-axis Current",  units="A",     x0=idr0, dq=
     dq_i)
2347         self.add_state("iqr", "diqr_dt", desc="Rotor Q-axis Current",  units="A",     x0=iqr0, dq=
     dq_i)
2348         self.add_state("wr",  "dwr_dt",  desc="Rotor Speed",           units="rad/s", x0=wr0,  dq=
     dq_wr)
2349
2350         self.add_dq_port("terminal", inputs=("vds", "vqs"), outputs=("ids", "iqs"))
2351
2352         self.add_algebraic("fqs", "Lls * iqs + Lm * (iqs + iqr)")
2353         self.add_algebraic("fds", "Lls * ids + Lm * (ids + idr)")
2354         self.add_algebraic("fqr", "Llr * iqr + Lm * (iqr + iqs)")
2355         self.add_algebraic("fdr", "Llr * idr + Lm * (idr + ids)")
2356         self.add_algebraic("Tm", "Tb * (wr / ws)**3")
2357         self.add_algebraic("Te", "-3/2 * P/2 * (fds * iqs - fqs * ids)")
2358
2359         self.add_diffeq("Rs * iqs + wr * fds + (Lls + Lm) * diqs_dt + Lm * diqr_dt - vqs")
2360         self.add_diffeq("Rs * ids - wr * fqs + (Lls + Lm) * dids_dt + Lm * didr_dt - vds")
2361         self.add_diffeq("Rr * iqr + (ws - wr) * fdr + (Llr + Lm) * diqr_dt + Lm * diqs_dt")
2362         self.add_diffeq("Rr * idr - (ws - wr) * fqr + (Llr + Lm) * didr_dt + Lm * dids_dt")
2363         self.add_diffeq("dwr_dt - P/2 * (Te - Tm) / J")
2364
2365
2366 class SyncMachineDQ2(SymbolicDevice):
2367
2368     def __init__(self, name, VLL=4160, ws=60*pi, P=4, rs=3e-3, Lls=2e-4,
2369                  Lmq=2e-3, Lmd=2e-3, rkq=5e-3, Llkq=4e-5, rkd=5e-3, Llkd=4e-5,
2370                  rfd=2e-2, Llfd=15e-5, vfdb=90.1, Kp=1e5, Ki=1e5, J=4221.7,
2371                  fkq0=0, fkd0=0, ffd0=0, wr0=0, th0=0, iqs0=0, ids0=0,
2372                  dq_i=1e-2, dq_f=1e-2, dq_wr=1e-1, dq_th=1e-3, dq_v=1e0):
2373
2374         SymbolicDevice.__init__(self, name)
2375
2376         self.add_parameter("VLL" , value=VLL )
2377         self.add_parameter("ws"  , value=ws  )
2378         self.add_parameter("P"   , value=P   )
2379         self.add_parameter("rs"  , value=rs  )
2380         self.add_parameter("Lls" , value=Lls )
2381         self.add_parameter("Lmq" , value=Lmq )
```

```
        self.add_parameter("Lmd" , value=Lmd )
        self.add_parameter("rkq" , value=rkq )
        self.add_parameter("Llkq", value=Llkq)
        self.add_parameter("rkd" , value=rkd )
        self.add_parameter("Llkd", value=Llkd)
        self.add_parameter("rfd" , value=rfd )
        self.add_parameter("Llfd", value=Llfd)
        self.add_parameter("vfdb", value=vfdb)
        self.add_parameter("Kp"  , value=Kp  )
        self.add_parameter("Ki"  , value=Ki  )
        self.add_parameter("J"   , value=J   )
        self.add_parameter("vfd" , value=vfdb)

        self.add_state("fkq", "dfkq_dt", units="Wb"   , x0=fkq0, dq=dq_f )
        self.add_state("fkd", "dfkd_dt", units="Wb"   , x0=fkd0, dq=dq_f )
        self.add_state("ffd", "dffd_dt", units="Wb"   , x0=ffd0, dq=dq_f )
        self.add_state("wr" , "dwr_dt" , units="rad/s", x0=wr0 , dq=dq_wr)
        self.add_state("th" , "dth_dt" , units="rad"  , x0=th0 , dq=dq_th)
        self.add_state("iqs", "diqs_dt", units="A"    , x0=iqs0, dq=dq_i )
        self.add_state("ids", "dids_dt", units="A"    , x0=ids0, dq=dq_i )

        self.add_dq_port("terminal", inputs=("vds", "vqs"), outputs=("ids", "iqs"))

        self.add_output_port("vterm", output="vterm")
        self.add_input_port("vfd", input="vfd")

        self.add_algebraic("Lq", "Lls + (Lmq * Llkq) / (Llkq + Lmq)")
        self.add_algebraic("Ld", "Lls + (Lmd * Llfd * Llkd) / (Lmd * Llfd + Lmd * Llkd + Llfd *
    Llkd)")
        self.add_algebraic("fq", "Lmq / (Lmq + Llkq) * fkq")
        self.add_algebraic("fd", "Lmd * (Lmd * (fkd / Llkd + ffd / Llfd)) / (1 + Lmd / Llfd + Lmd
    / Llkd)")
        self.add_algebraic("Te", "3/2 * P/2 * (fds * iqs - fqs * ids)")
        self.add_algebraic("Tm", "Kp * (ws - wr) + th * Ki")
        self.add_algebraic("vterm", "sqrt(vds**2 + vqs**2)")

        self.add_diffeq("diqs_dt * Lls + rs * iqs + wr * Ld + wr * fd - vqs")
        self.add_diffeq("dids_dt * Lls + rs * ids - wr * Lq - wr * fq - vds")
        self.add_diffeq("dfkq_dt * Llkq + rkq * (fkq - Lq * iqs - fq + Lls * iqs)")
        self.add_diffeq("dfkd_dt * Llkd + rkd * (fkd - Ld * ids + fd + Lls * ids)")
        self.add_diffeq("dffd_dt * Llfd + rfd * (ffd - Ld * ids + fd + Lls * ids) - vfd*VLL")
        self.add_diffeq("dwr_dt * J + Tm - Te")
        self.add_diffeq("dth_dt + wr - ws")
```

```
2423
2424
2425 class Exciter(SymbolicDevice):
2426
2427     def __init__(self, name, VLL=4160, vref=1.0, Kpr=200.0, Kir=0.8, Kdr=1e-3, Tdr=1e-3,
2428                  Ka=1.0, Ta=1e-4, Vrmin=0.0, Vrmax=5.0, Te=1.0, Ke=1.0, x10=0.0,
2429                  x20=0.0, x30=0.0, vout0=0.0, dq_x1=1e-8, dq_x2=1e-8, dq_x3=1e-5,
2430                  dq_vout=1e-2):
2431
2432         SymbolicDevice.__init__(self, name)
2433
2434         self.add_parameter("VLL"  , value=VLL  )
2435         self.add_parameter("vref" , value=vref )
2436         self.add_parameter("Kpr"  , value=Kpr  )
2437         self.add_parameter("Kir"  , value=Kir  )
2438         self.add_parameter("Kdr"  , value=Kdr  )
2439         self.add_parameter("Tdr"  , value=Tdr  )
2440         self.add_parameter("Ka"   , value=Ka   )
2441         self.add_parameter("Ta"   , value=Ta   )
2442         self.add_parameter("Vrmin", value=Vrmin)
2443         self.add_parameter("Vrmax", value=Vrmax)
2444         self.add_parameter("Te"   , value=Te   )
2445         self.add_parameter("Ke"   , value=Ke   )
2446
2447         self.add_state("x1",   "dx1_dt",    x0=x10,   dq=dq_x1  )
2448         self.add_state("x2",   "dx2_dt",    x0=x20,   dq=dq_x2  )
2449         self.add_state("x3",   "dx3_dt",    x0=x30,   dq=dq_x3  )
2450         self.add_state("vout", "dvout_dt",  x0=vout0, dq=dq_vout)
2451
2452         self.add_input_port("vterm", intput="vterm")
2453         self.add_output_port("vfd", output="vout")
2454
2455         self.add_algebraic("vin", "vref - vterm / VLL")
2456
2457         self.add_diffeq("dx1_dt + 1/Tdr * x1 - vin")
2458         self.add_diffeq("dx2_dt - x1")
2459         self.add_diffeq("dx3_dt - ((-Kdr/(Tdr**2) + Kir)*x1 + Kir/Tdr * x2 - 1/Ta * x3 + (Kdr/Tdr
    + Kpr) * vin)")
2460         self.add_diffeq("dvout_dt - (Ka/Ta * x3 - 1/Te * (vout * Te / Ke))")
2461
2462
2463 class Ground(SymbolicDevice):
2464
2465     def __init__(self, name):
2466
```

```
        SymbolicDevice.__init__(self, name)

        self.add_electrical_port("positive", output="v", input="i")

        self.add_algebraic("v", "0")


class DCMotorSym(SymbolicDevice):

    def __init__(self, name, Vs=10, Gs=1e2, Cl=1e-6, Ra=0.1, La=0.001, Ke=0.1, Kt=0.1,
                 Jm=0.01, Bm=0.001, Jp=0.01, Fp=1, JL=0.5, TL=0, BL=0.1,
                 ia0=0.0, wr0=0.0, dq_ia=1e-1, dq_wr=1e-1):

        SymbolicDevice.__init__(self, name)

        self.add_parameter("Vs", desc="", units="", value=Vs)
        self.add_parameter("Gs", desc="", units="", value=Gs)
        self.add_parameter("Cl", desc="", units="", value=Cl)
        self.add_parameter("Ra", desc="", units="", value=Ra)
        self.add_parameter("La", desc="", units="", value=La)
        self.add_parameter("Ke", desc="", units="", value=Ke)
        self.add_parameter("Kt", desc="", units="", value=Kt)
        self.add_parameter("Jm", desc="", units="", value=Jm)
        self.add_parameter("Bm", desc="", units="", value=Bm)
        self.add_parameter("Jp", desc="", units="", value=Jp)
        self.add_parameter("Fp", desc="", units="", value=Fp)
        self.add_parameter("JL", desc="", units="", value=JL)
        self.add_parameter("TL", desc="", units="", value=TL)
        self.add_parameter("BL", desc="", units="", value=BL)

        self.add_state("ia", "dia_dt", desc="Armature Current", units="A",     x0=ia0, dq=dq_ia)
        self.add_state("wr", "dwr_dt", desc="Rotor Speed",      units="rad/s", x0=wr0, dq=dq_wr)

        self.add_electrical_port("positive", output="i", input="vpos")
        self.add_electrical_port("negative", output="i", input="vneg", sign=-1)

        self.add_diffeq("La * dia_dt + Ra * ia + Ke * wr - (vpos - vneg)")
        self.add_diffeq("Jm * dwr_dt + Bm * wr - Kt * ia + TL")
```