Long-Term Dynamic Simulation of Power Systems using Python, Agent Based Modeling, and Time-Sequenced Power Flows

–Software and Simulated Controls Documentation– ver. 1.0

by

Thad Haines

Montana Technological University

2020

Introduction

PSLTDSim is a power system long-term dynamic simulator that is based on time-sequenced power flows and models additional dynamics such as system frequency and turbine speed governor action. Software development represented the crux of a masters thesis available here: `https://github.com/thadhaines/Thesis-Release`.
This document was compiled from various software explanation sections lifted from said thesis to offer a kind of 'lazy user guide'. Full working codes, which may more clearly illustrate software use, are located on the associated github page at:
`https://github.com/thadhaines/PSLTDSim/tree/master/testCases` while a .bat file that actually runs the simulation tool (`runPSLTDsim.bat`) is located a folder higher. The very similarly named `runPLTD.bat` creates plots from saved system mirrors.

While efforts have been made to produce a logically written functional software package, reality can often reduce such efforts to merely good intentions. However, some may say that intentions matter.

Questions and/or comments may be sent to jhaines at mtech dot edu.
Responses may or may not be supplied.

Table of Contents

List of Tables

List of Figures

List of Equations

Glossary of Terms

Term	Definition
ABM	Agent Based Modeling
ABS	Agent Based Simulation
AC	Alternating Current
ACE	Area Control Error
AGC	Automatic Generation Control
AMQP	Advanced Message Queue Protocol
API	Application Programming Interface
BA	Balancing Authority
BES	Bulk Electrical System
CLR	Common Language Runtime
CTS	Classical Transient Stability
DACE	Distributed ACE
DTC	Definite Time Controller
EIA	United States Energy Information Administration
ERCOT	Electric Reliability Council of Texas
FERC	Federal Energy Regulatory Commission
FTL	Frequency Trigger Limit
GE	General Electric
Hz	Hertz, cycles per second
IACE	Integral of ACE
IC	Interchange
IPC	Interprocess Communication
IPY	IronPython
ISO	Independent Service Operator
J	Joule, Neton meters, Watt seconds
LFC	Load Frequency Control
LTD	Long-Term Dynamic
NERC	North American Electric Reliability Corporation

Term	Definition
ODE	Ordinary Differential Equation
P	Real Power
PI	Proportional and Integral
PMIO	PSLF Model Information Object
PSDS	PSLF Dynamic Subsystem
PSLF	Positive Sequence Load Flow
PSLTDSim	Power System Long-Term Dynamic Simulator
PSS	Power System Stabilizer
PST	Power System Toolbox
PU	Per-Unit
PY3	Python version 3.x
PyPI	Python Package Index
Q	Reactive power
RACE	Reported ACE
RTO	Regional Transmission Organization
SACE	Smoothed ACE
SI	International System of Units
TLB	Tie-Line Bias
TSPF	Time-Sequenced Power Flow
US	United States of America
VAR	Volt Amps Reactive
W	Watt, Joules per second
WECC	Western Electricity Coordinating Council

1 Software Background

1.1 Classical Transient Stability Simulation

Classical transient stability (CTS) simulation is a simulation approach commonly used to test a power system's ability to remain stable after a large step perturbance such as a generator trip. The time frame CTS focuses on is milliseconds to tens of seconds, and as such, requires time steps of milliseconds. While this is an appropriate approach for relatively short periods of time, complicated modeling issues may lead to questionable results over the course of longer simulations.

General Electric's (GE's) Positive Sequence Load Flow (PSLF) is an industry standard power-flow solver and CTS simulation program. The continued development of PSLF has produced a large library of dynamic models and components for use in CTS simulation. PSLFs dynamic library has enabled a wide variety of system models to be created. For example, multiple full WECC base cases have been modeled in PSLF over the past 20 years and are continuously being updated. Certain versions of PSLF offer a Python 3 application programming interface (API) and a .NET API. Despite each API being at various levels of development and functionality, both offer a modern way to communicate with PSLF.

1.2 Python

Python is an interpreted high-level general purpose programming language that utilizes object oriented techniques and emphasizes code readability [37]. Guido Van Rossum first implemented a version of Python in December of 1989 [62]. Python is freely available and distributable for multiple computing platforms [59].

1.2.1 Python Packages

Software modules that expand the functionality of Python are referred to as packages and are freely available from the Python Package Index (PyPI). This work utilizes multiple packages to varying degrees, though SciPy and Pika are among the most heavily used. SciPy

is a collection of packages that include NumPy for numerical computing and Matplotlib for MATLAB style plotting [66]. Additionally, scipy.signal.lsim was used to solve state-space equations that are common in electrical engineering dynamics. To avoid rewriting well known integration routines, scipy.integrate.solve_ivp was used to perform Runge-Kutta integration of the swing equation to find system frequency. The Python implementation of the advanced message queuing protocol (AMQP) used for this project was Pika [22].

1.2.2 Varieties of Python

Python has gone through numerous versions over the course of development and has been ported and adapted according to need. IronPython (IPY) is an open-source .NET compatible version of Python written in C# [1] and is able to use the common language runtime (CLR) package required for properly interacting with the GE PSLF .NET library. As such IPY, more specifically 32-bit IPY, is used to interface with the PSLF .NET library. However, the most current stable IPY release is based off of Python 2.x and can only use Python packages compatible with 2.x. Python 3 (PY3) is 'the future of Python' and has many more maintained and useful community created packages. Some packages area available for both Python version 2 and 3, but many are not. As of this writing, the current version of Python is 3.8. A majority of project code was written using Python 3.7, but should continue to be compatible with future versions of PY3.

1.2.3 Python Specific Data Types

Python has various common data types found in other programing languages such as integer, string, list, and float. Python also has some unique data types. One unique data type is called a *dictionary* which is a collection of key value pairs. Dictionary keys are strings, and the value can be any other data type, including a dictionary. A benefit of using a dictionary is that it doesn't matter 'where' in an object a certain data point is located, only that is exists somewhere in the object. The key value pair eliminates the need to focus on indexing lists or arrays as other programming languages may require. Additionally, native python methods allow for simple searching and iteration of dictionaries.

Another somewhat unique python data type is a tuple. Tuples are essentially the same as lists, except defined using parenthesis instead of brackets and the data inside can not be changed in any way. While this may seem like a hindrance, it creates peace of mind when using tuples for important data references.

A powerful characteristic of Python is that everything is an object, and variables act as references, or pointers, to data. These Pythonic points were relied upon heavily during software development to make large collections of objects and references.

1.3 Advanced Message Queueing Protocol

Advanced message queueing protocol (AMQP) is a software messaging protocol that can be used for interprocess communication (IPC). The specific application of AMQP in this case was to enable a PY3 process to communicate with an IPY process on a Windows based machine. The idea behind AMQP is that of a virtual 'broker' which receives messages from various processes and places them into specific named queues. The named queues are accessed by other processes that check for, and receive, any queued messages.

It should be noted that Erlang was required to allow use of RabbitMQ, and the PY3 and IPY Pika packages were required so that Python could use AMQP. While detailed descriptions of these softwares is beyond the scope of this research: Erlang is an open source programming language and runtime environment, RabbitMQ is an open source AMQP broker software, and Pika is a Python package that works with RabbitMQ. The correct installation of these software packages is necessary for the created research software to function.

1.4 Agent Based Modeling

Agent Based Modeling (ABM), or Agent Based Simulation (ABS), is oriented around the idea that any situation can be described by agents in an environment, and a definition of agent-to-agent and agent-to-environment interactions [60]. The ABM coding style lends itself towards modular coding and natural expandability as each agent class may have inherited

methods and variable characteristics. As an example, [3] used an ABM approach to test the efficiency of differing airplane boarding methods. Passengers were treated as agents interacting with each other and the airplane environment. Each passenger agent may have had different characteristics, but performed a similar actions each simulation time step. ABM is applied to the coding of this project in that a power system acts as the environment, and all power system objects, such as buses, loads, and generators, are treated as agents.

Each time step, agents may perform their *step* method that executes code unique to the specific agent, but generally the same among agent types. For example, a timer agent step method may include checking a specified value and incrementing an accumulator according to a logic operation. If the accumulator becomes larger than a set threshold value, the timer may raise an activation flag. Agent action is meant to be direct and generic so that agents can be reused and nested inside other agents. Continuing the example, the timer agent may be nested inside another agent that checks the timers activation flags each step and takes action depending on the returned status. Actions can be arranged into a sequence of agent steps in a discrete time simulation, and then repeated ad infinitum. The idea of sequencing agent steps can be applied to systems of nearly any size and composition as long as agents have a step function and can reference other agents inside the environment.

2 Software Tool

2.1 Time-Sequenced Power Flows

The left side of Figure 2.1 is a visual representation of a single power-flow solution. The power-flow solution can be thought of as a snapshot of steady state system operation. A period of steady state system operation could be imagined by repeating this single power-flow solution in a time sequence. Dynamic system behavior could be realized if small changes are made to the power-flow problem inputs and the ensuing power-flow solution converges. The right side of Figure 2.1 illustrates the idea of time-sequenced power flow (TSPF) as a collection of slightly changing power-flow solutions. The TSPF method for dynamic simulation involves performing dynamic modeling calculations outside of, or inbetween, each power-flow solution.

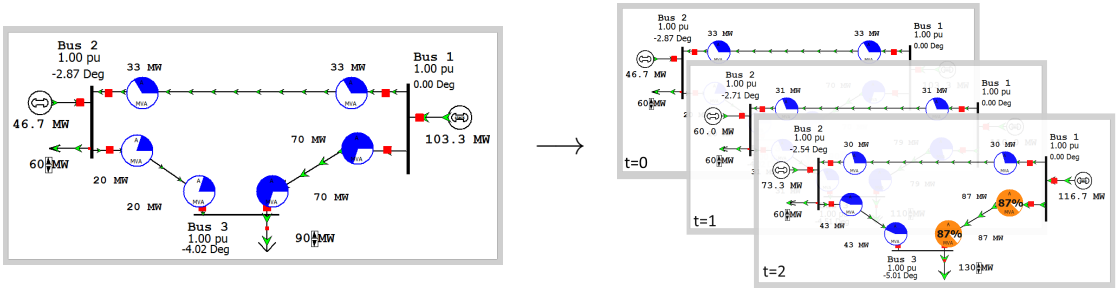

Figure 2.1: Time-sequenced power flows visualized.

Differences between TSPF and CTS simulation techniques stem from the time frame of simulation focus and differing dynamic calculations. TSPF is focused on long-term events that take place over the course of minutes to hours. CTS simulation focuses on events that are tens of seconds in duration. This difference in focus leads to each method utilizing a different appropriate time step. A TSPF time step may be 0.5 to 1 seconds while CTS uses sub-cycle time-steps in the millisecond range.

Calculations of CTS simulation include generator dynamics, exciter dynamics, governor and turbine dynamics, and load dynamics that are either simplified, aggregated, or not included in the current TSPF method used by PSLTDSim. Each simulation method

starts with a power-flow solution, but CTS simulation performs back calculations to set initial states of various dynamic models. Future CTS states, and resulting system behaviors, are dictated by dynamic model interactions. TSPF also uses dynamic models, but updated values are sent to a power-flow solver; and the power-flow solution provides new steady state system. These differences create output data that is of different resolutions and captures slightly different system characteristics. Figure 2.2 shows a comparison of CTS and TSPF data.

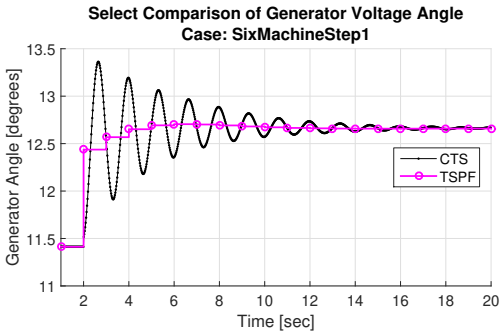

Figure 2.2: CTS and TSPF data output comparison.

The two data sets are not the same during oscillatory behavior, but match at steady state conditions. In general, TSPF data does not reflect the inter-electromechanicl dynamics of the system response, but does seem to follow the center of oscillation from the CTS simulation. This 'oscillation averaging' behavior was found to be common in TSPF results.

2.2 Simulation Assumptions and Simplifications

Numerous assumptions and simplifications were made due to fundamental differences between TSPF and CTS simulation. This section details such assumptions and simplifications and provides key TSPF equations.

2.2.1 General Assumptions and Simplifications

The focus of PSLTDSim is on the long-term operation of power systems. Since a power-flow solution is a stable steady state operating condition, it is assumed that the system of study is stable in the transient stability sense, and remains stable, for the entirety

of the simulation. Other software packages should be used if transient stability is in question.

In general, system responses to large step type contingencies are not the focus of PSLTDSim. Instead, a more suitable event type is in the form of ramps, or repeated small step perturbances. Because of the more 'gentle' system scenarios involved with ramps, power system stabilizers (PSS) were not modeled. Further, it is assumed that PSS time constants are too small to be adequately represented given the time step associated with TSPF.

Further assumptions include ideal generator exciters that can always maintain a given generator reference voltage. Modern exciters are assumed to be fast enough to hold reference voltages to small perturbances in the long-term. Despite the ideal reference voltage assumption, machine VAR limits **are** enforced according to model parameters. Should the need arise, future work can be done to incorporate any additional, or alternative modeling.

2.2.2 Time Step Assumptions and Simplifications

Multiple assumptions can be made concerning power system behavior when a time step of one second is used. The order of magnitude time step difference between CTS and TSPF allows models created for CTS simulations to be simplified for use with TSPF. Intermachine oscillations were ignored since subsynchronous resonances are sub-second and the time resolution of TSPF is not great enough to capture these phenomena. Additionally, in the long-term, these effects are minor when the system is stable. Without the need for subsynchronous characteristics, generator modeling was greatly simplified. The only details required for a machine model are MW cap, machine MVA base, and machine inertia.

To further simplify generator modeling, all machines share a single system frequency that is calculated by a combined swing equation. The following two Sections explain how the combined swing equation is used in PSLTDSim. Governor models were also simplified. Section 2.2.5 explains the how PSLTDSim models governors.

2.2.3 Combined System Frequency

Instead of a frequency being calculated for each generator or bus, a single combined swing equation is used to model only one combined system frequency. This technique requires

a known total system inertia H_{sys} and the total system acceleration power $P_{acc,sys}$. In a system with N generators, H_{sys} is calculated from each individual machine's inertia as

$$H_{sys} = \sum_{i=1}^{N} H_{PU,i} M_{base,i}. \tag{2.1}$$

In a system with N generators, total system accelerating power is calculated as

$$P_{acc,sys} = \sum_{i=1}^{N} P_{m,i} - \sum_{i=1}^{N} P_{e,i} - \sum \Delta P_{pert}, \tag{2.2}$$

where $P_{m,i}$ is mechanical power and $P_{e,i}$ is electrical power of the ith generator and any system power injections, or perturbances, are accounted for in the $\sum \Delta P_{pert}$ term.

The combined swing equation, shown in Equation 2.3, uses $P_{acc,sys}$ and H_{sys} to calculate $\dot{\omega}_{sys}$. For completeness, a damping term $D_{sys}\Delta\omega$ is included in Equation 2.3, but as Equation **??**, D_{sys} is often set to zero while $\Delta\omega_{sys}$ is calculated using Equation **??** with ω_{sys} replacing ω. Equation 2.4 shows that after integrating with time step t_s, $\dot{\omega}_{sys}$ leads to system frequency ω_{sys}.

$$\dot{\omega}_{sys} = \frac{1}{2H_{sys}} \left(\frac{P_{acc,sys}}{\omega_{sys}} - D_{sys}\Delta\omega_{sys} \right) \tag{2.3}$$

$$\omega_{sys} = \int_{t}^{t+t_s} \dot{\omega}_{sys} \, \mathrm{d}t \tag{2.4}$$

The Scipy solve_ivp function was chosen as the default numerical solver for the combined swing equation. This choice allows other integration methods to be applied that may produce more desirable results. Additionally, the solve_ivp function produces more approximations between defined time steps. It is theoretically possible to use this more detailed output for dynamic agent input. As of this writing, the ability to use this additional output is not included in the code. It may prove beneficial to explore this possibility in the future work if dynamic calculation results are shown to be unsatisfactory.

2.2.4 Distribution of Accelerating Power

While system frequency can be calculated using total system accelerating power, to properly 'seed' the next power flow, each generator participating in the system inertial response must account for a portion of accelerating power absorption. The specific amount each generator is expected to absorb is based on machine inertia. Equation 2.5 shows how the next electric power estimate $P_{e,EST,i}$ is created for generator i according to its inertia.

$$P_{e,EST,i} = P_{e,i} - P_{acc,sys}\left(\frac{H_i}{H_{sys}}\right) \tag{2.5}$$

Once all accelerating power is distributed to inertial responding generators, the new $P_{e,EST}$ value for each generator is used to seed a power flow. If the MW difference between resulting power supplied by the slack generator and estimated power output is larger than the set slack tolerance, the difference is redistributed as $P_{acc,sys}$ according to Equation 2.5 until slack tolerance is met, or a maximum number of iterations take place. Once the slack tolerance is met, the power-flow solution is accepted as the current state of the system under study.

This method of reallocation ensures the validity of a generation dispatch because in a power-flow solver such as PSLF, any difference between the dispatched MW of generation, and the aggregate MW consumption of all loads and losses is allocated to the designated slack generator. Therefore, if the slack generator responds as predicted, it is assumed all other generators have also responded as predicted.

2.2.5 Governor models

Long-term dynamic models do not require the detail of a full transient simulation model. For software validation purposes, a *tgov1* governor model was created as it appears in the PSLF documentation. This particular governor model was selected due to simplicity, and was later expanded upon to include an optional deadband and filtered input delay. The block diagram for the modified tgov1 governor is shown in Figure 2.3. Blocks with a * next

to them indicate they are optional, and only inserted into the model if user defined.

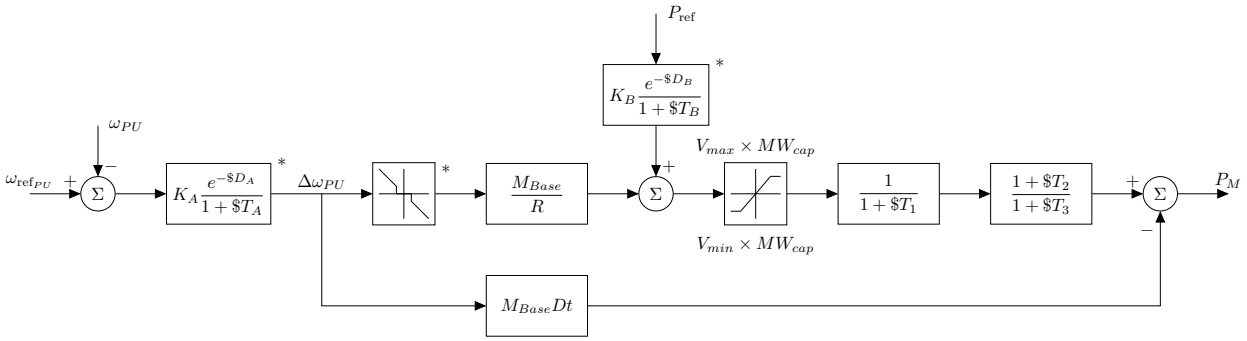

Figure 2.3: Block diagram of modified tgov1 model.

A slightly more generic governor was created based off the governor model used in Power System Toolbox (PST). This generic model, referred to as *genericGov*, is shown in Figure 2.4. The genericGov follows the time constant naming convention of the PST governor and includes the same optional blocks added to the modified tgov1 model.

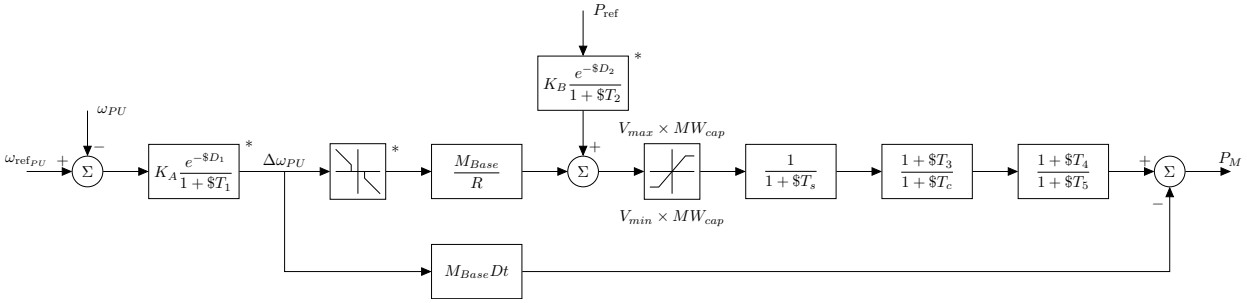

Figure 2.4: Block diagram of genericGov model.

To allow for a simpler numerical approach, the Scipy lsim function was chosen as the numerical solver for governor dynamics. Unlike solve_ivp, which requires a differential equation in the form of a function as input, lsim accepts input consisting of transfer functions or state space systems which are more common electrical engineering representations of dynamic systems. As of this writing, the above models are processed as a sequence of stages that represent a single Laplace block, or transfer function. After a full time step solution of one block, the output is then sent to the next block in the order depicted in the above diagrams. The mathematical implications of such an approach are addressed in Section

A.4.3.2.

2.2.5.1 Casting Process for genericGov

Modeling all PSDS governors would be a rather large task. Un-modeled governors encountered in the PSDS dynamic parameters file, referred to as a dyd file, were cast to a genericGov model based on assumed governor type. Table 2.1 shows the relation of PSDS model types to assumed governor setting type.

Table 2.1: Generic governor model casting between LTD and PSDS.

genericGov	Steam	Hydro	Gas
PSDS	ccbt1	g2wscc	ggov1
	gast	hyg3	ggov3
	w2301	hygov4	gpwscc
	ieeeg3	hygov	
	ieeeg1	hygovr	
		pidgov	

Universal governor settings, such as permanent droop and MW cap values are collected from the dyd file and used as R and MW_{cap} respectively. The particular time constants for each model were selected according to typical settings associated with prime mover type that a PSDS model is assumed to represent. Table 2.2 lists the time constants used for each genericGov model type.

Table 2.2: Generic governor model parameters.

Parameter	Steam	Hydro	Gas
Ts	0.04	0.40	0.50
Tc	0.20	45.00	10.00
T3	0.00	5.00	4.00
T4	1.50	−1.00	0.00
T5	5.00	0.50	1.00

2.3 General Software Explanation

This section provides an explanation of the Power System Long-Term Dynamic Simulator (PSLTDSim). A flow chart showing a general overview of simulation action is shown in Figure 2.5. A simulation begins with user input of simulation specifications to PSLTDSim. The user input is then used to initialize the required simulation environment by PSLTDSim. The general simulation loop includes performing dynamic calculations and executing any perturbances which are then transferred to PSLF. A power-flow solution is then calculated by PSLF and relevant data sent back to PSLTDSim for logging. Various simulation variables are then checked to verify if the simulation loop is to continue or end. Once the simulation is complete, data is output and plots may be generated for the user to analyze.

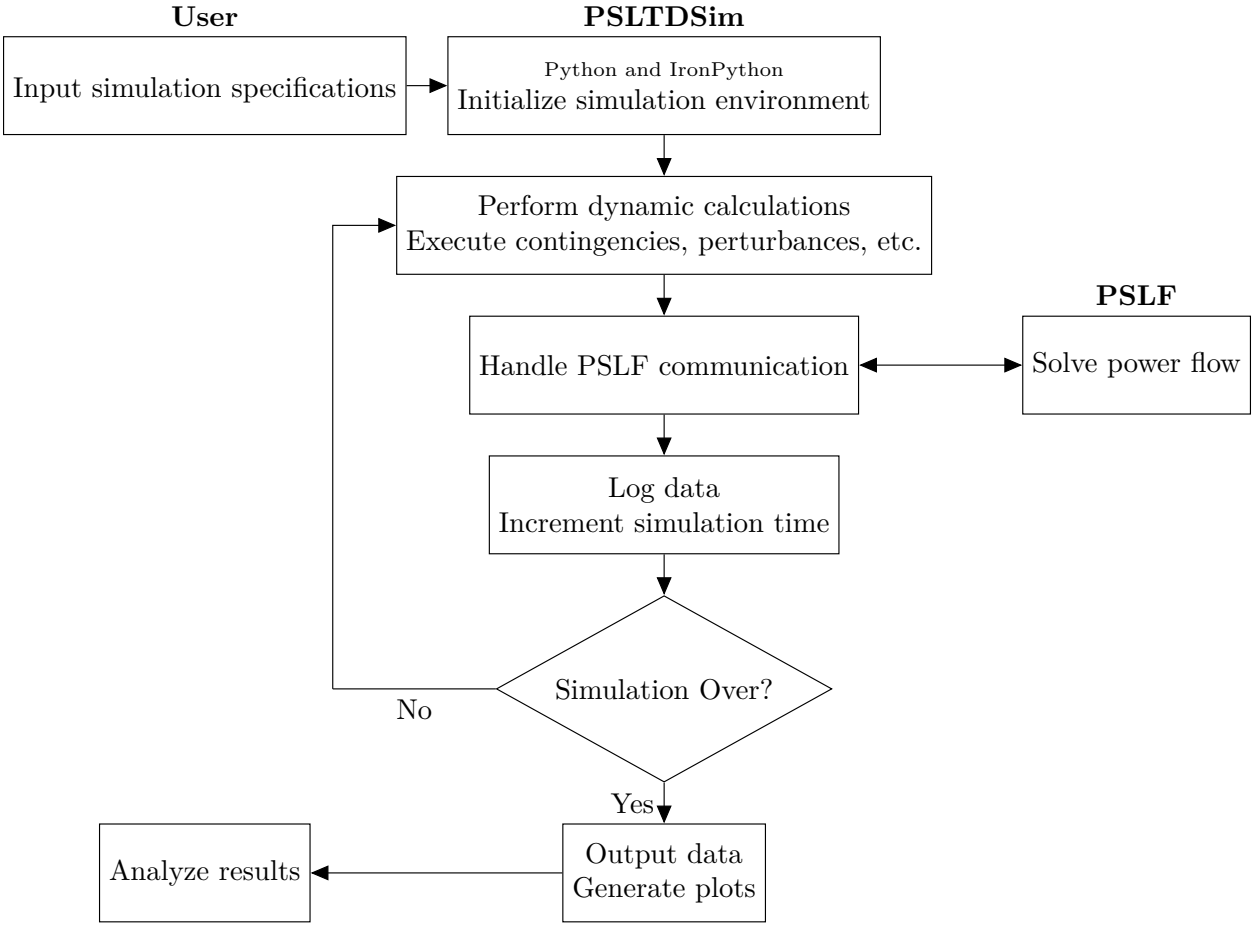

Figure 2.5: High level software flow chart.

2.3.1 Interprocess Communication

A process is another name for an instance of a running computer program. It is common for a computer program to run as a single process, however this is not always the case and not how PSLTDSim operates. Since processes are independent from each other, they do not share a memory space [24]. This effectively means that for two processes to share data, some kind of interprocess communication (IPC) must be utilized.

Due to the current state of the GE Python 3 API, Ironpython (IPY) was required for functional PSLF software communication. Unfortunately, IPY is based on Python 2 and does not have packages necessary for numerical computation that Python 3 (PY3) possesses. The solution to these issues was to create a software that has an IPY process and PY3 process that communicate to each other via AMQP. The employed method is shown in Figure 2.6. The IPY process, which has a functional PSLF API, acts as a 'middle man' between PY3 and

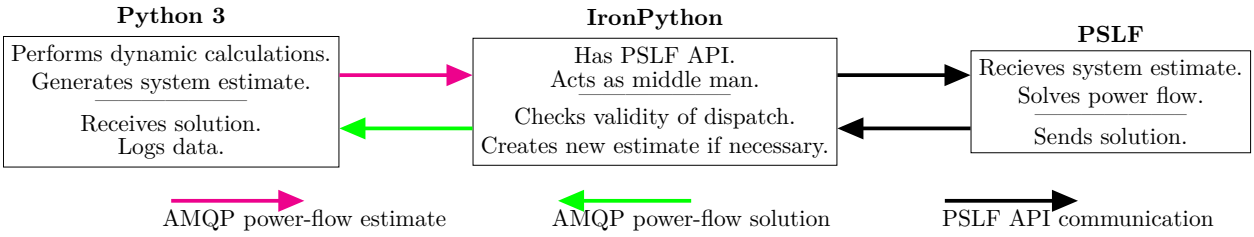

Figure 2.6: Diagram of AMQP communication.

PSLF. Newly calculated power-flow estimate from PY3 must be sent to PSLF via IPY, and after each new power-flow solution, PSLF values must be sent to PY3. This cycle continues as along as simulation is required. While the described AMQP solution has been shown to work, it is worth noting that message handling typically accounts for about one half of all simulation time, and sometimes more in larger cases.

2.3.2 Simulation Inputs

As with any simulation software, PSLTDSim requires specific inputs to operate correctly. Some required input is the same as that used by PSLF, while other input is Python based. Both types of inputs are described in this subsection.

2.3.2.1 PSLF Compatible Input

The power system model input used by PSLTDSim is the same binary file used by, and generated from, PSLF. The system model file, referred to as a sav file, contains topological and parametric data required to execute a power-flow solution. The use of this input file is due to the reliance of PSLTDSim on the power-flow solver included with PSLF. Python system dynamics are created based on the .dyd text files also used by PSLF. Information on creating .sav or .dyd files is beyond the scope of this text, but may be found in [26].

2.3.2.2 Simulation Parameter Input (.py)

Simulation parameter input is entered in a standard python .py file. Most simulation parameter input is collected in a Python dictionary named simParams. An example of the simParams dictionary defined inside the .py file is shown in Figure 2.7.

```
simParams = {
        'timeStep': 1.0, # seconds
        'endTime': 60.0*8, # seconds
        'slackTol': 1, # MW
        'PY3msgGroup' : 3, # number of Agent msgs per AMQP msg
        'IPYmsgGroup' : 60, # number of Agent msgs per AMQP msg
        'Hinput' : 0.0, # MW*sec of entire system, if !> 0.0, will be calculated in code
        'Dsys' : 0.0, # Damping
        'fBase' : 60.0, # System F base in Hertz
        'freqEffects' : True, # w in swing equation will not be assumed 1 if true
        'assumedV' : 'Vsched', # assummed voltage - either Vsched or Vinit
        # Mathematical Options
        'integrationMethod' : 'rk45',
        # Data Export Parameters
        'fileDirectory' : "\\delme\\200109-delayScenario1\\", # relative path from cwd
        'fileName' : 'SixMachineDelayStep1', # Case name in plots
        'exportFinalMirror': 1, # Export mirror with all data
        'exportMat': 1, # if IPY: requies exportDict == 1 to work
        'exportDict' : 0, # when using python 3 no need to export dicts.
        'logBranch' : True,
}
```

Figure 2.7: An example of a simParams dictionary.

The simParams dictionary contains information required for the simulation to op-

erate, such as time step, end time, base frequency, and slack tolerance. There are also parameters that alter how the simulation operates, such as integration method, inclusion of frequency effects, and how system inertia is calculated. Information related to data collection or export is also included in the simParams dictionary such as whether to log branch information or where the resultant data will be placed and what it will be named. Examples of valid dictionary keys, types of data, units of input, and a brief explanation are shown in Table B.2 located in Appendix B.

In addition to the simParams dictionary, simulation notes, absolute file paths to the desired .sav and .ltd.py file are also defined in this .py file. Dynamic input in the form of .dyd files are defined in a list to allow for using more than one `.dyd` file. The dynamic model overwriting that this feature was meant to incorporate has not been fully implemented as of this writing. An example of a valid simulation parameter input .py is shown in Appendix C as Figure C.1.

2.3.2.3 Long-Term Dynamic Input (.ltd.py)

The required .ltd.py file contains user defined objects related specifically to simulated events and control action. Further, this file is the ideal place for adding additional user input if the need should arise in future software development. Code in the .ltd.py file is standard Python and involves initializing objects attached to a *mirror* object. The mirror object is what PSLTDSim calls the total power system model. A description of the various objects that can be attached to the mirror is presented in the following sections. Most topics introduced are described completely, however, some options are more complex and described in later sections.

2.3.2.3.1 Perturbance List

The only list defined in the .ltd.py file is for entering simulation perturbances (often also referred to as contingencies or events). The list of single quoted strings describing system perturbances is defined as mirror.sysPerturbances. Common perturbances are changes in

operating state and power, however, any value in the target agents current value dictionary may be changed. The format of the string is specific to agent type and perturbance, but strings follow the general format of 'Agent Identification : Perturbance Description'. Table 2.3 shows the format for the agent identification part of the perturbance string. Optional parameters are shown in brackets. If no `ID` is specified the first agent found with matching bus values will be chosen.

Table 2.3: Perturbance agent identification options.

Agent Type	Identification Parameters		
`load`	Bus Number	[ID]	
`shunt`	Bus Number	[ID]	
`gen`	Bus Number	[ID]	
`branch`	From Bus Number	To Bus Number	[Circuit ID]

Table 2.4 describes the various parameters used to specify the action of the perturbance agent in the 'Pertrubance Description'. The three valid options for the 'Step Type' and 'Ramp Type' field are `abs`, `rel`, and `per`. To make an absolute change to the new value, the `abs` type should be selected. To alter the target parameter by a relative value, the `rel` option should be used. If a percent change is desired, the `per` type should be used.

Table 2.4: Perturbance agent action options.

Type	Settings				
`step`	Target Parameter	Action Time	New Value	Step Type	
`ramp`	Target Parameter	Start Time	Ramp Duration	New Value	Ramp Type

Figure 2.8 shows various examples of valid perturbance agent definitions. Double quoted strings may be used to clarify perturbance descriptions. It should be noted that the target parameter is case sensitive. Additionally, if stepping a governed generator, both the mechanical power and power reference variables should be changed as shown in line 8 and 9 of Figure 2.8.

```
# Perturbance Examples
mirror.sysPerturbances = [
    'gen 27 : step St 2 0',            # Set gen 27 status to 0 at t=2
    'branch 7 8 2 : step St 10 0 abs', # Trip branch between bus 7 and 8 with ckID=2
    'load 26 : ramp P 2 40 400 rel',   # ramp load 26 P up 400MW over t=2-42 seconds
    'load 9 : "Type" ramp "Target" P "startTime" 2 "RAtime" 40 "RAval" -5 "RAtype" per',
    'shunt 9 4 : step St 32 1',        # Step shunt id 4 on bus 9 on at t=32
    'gen 62 : step Pm 2 -1500 rel',    # Step gen Pm down 1500 MW at t=2
    'gen 62 : step Pref 2 -1500 rel',  # Step gen Pref down 1500 MW at t=2
    ]
```

Figure 2.8: Perturbance agent examples.

2.3.2.3.2 Noise Agent Attribute

A random noise agent was created to add random noise to all system loads. The noise agent may be useful for Monte Carlo studies, or for studies involving nonlinearities such as deadbands. The noise agent is created in the .ltd.py file by defining the mirror.NoiseAgent attribute with the associated agent. Figure 2.9 shows an example of a noise agent being attached to a system mirror. The input arguments are: system mirror reference, percent noise to be added, a boolean value dictating random walk behavior, delay before noise is added, and the random number generator seed value.

```
# Noise Agent Creation
mirror.NoiseAgent = ltd.perturbance.LoadNoiseAgent(mirror, 0.05, walk=True, delay=0,
↪  damping=0, seed=11)
```

Figure 2.9: Noise agent creation example.

If a noise agent is defined, noise is injected at every time step into each load $P_{L,i}$ in the system according to

$$P_{L,i}(t) = P_{L,i}(t-1)[1 \pm N_Z Rand_i + D_{nz}\Delta\omega] \tag{2.6}$$

where N_Z represents the maximum amount of random noise to inject as a percent, $Rand_i$ is a randomly generated number between 0 and 1 inclusive, D_{nz} is the user input damping value, and $\Delta\omega$ is calculated according to Equation **??**. The decision to add or subtract noise is chosen by a unique randomly generated number. As described in [70], Equation 2.6 creates random walk behavior in load that is representative of real power systems. If random walk behavior is not desired, the walk input argument may be set to False and the scheduled load value is always used as $P_{L,i}(t-1)$. Only mirror reference and percent of desired noise is required for noise agent creation. If no additional arguments are provided, default values for walk, delay, damping and seed are set to False, 0, 0, and 42 respectively.

2.3.2.3.3 Balancing Authority Dictionary

The mirror.sysBA dictionary is defined in the .ltd.py file and is used to configure BA agents. Individual BA dictionaries are nested inside the mirror.sysBA dictionary. The information entered in each nested dictionary describes how that particular BA acts on the specified area. Additional information on BA AGC options and action is presented in Section 3.1.3. Examples of BA parameterization are shown in Figures C.3 and C.4 of Appendix C. A description of each possible field is described in Table B.1 located in Appendix B.

2.3.2.3.4 Load Control Dictionary

To simplify changing all area loads according to a known demand schedule, a load control agent may be defined in the .ltd.py file. Similar to the balancing authority dictionary, each agent is defined as a named dictionary inside a the mirror.sysLoadControl dictionary. The load control agent requires area number, start time, time scale, and a list of tuples for demand information. Specific BA demand data can easily be acquired via the United States Energy Information Administration (EIA) website and saved as a .csv file. A script was written to parse and display demand changes over time as a relative percent change so that real load patterns could be applied to test systems of differing scale. An example of a single area load control agent is shown in Figure 2.10.

```
mirror.sysLoadControl = {
    'testSystem' : {
        'Area': 2,
        'startTime' : 2,
        'timeScale' : 10,
        'rampType' : 'per', # relative percent change
        # Data from: 12/11/2019 PACE
        'demand' : [
            #(time , Precent change from previous value)
            (0, 0.000),
            (1, 3.675),
            (2, 6.474),
            (3, 3.770),
                    ] , # end of demand tuple list
                },# end of testSystem definition
}# end of sysLoadControl dictionary
```

Figure 2.10: Load control agent dictionary definition example.

The list of tuples named 'demand' defines the desired load change over time. The first value in each tuple is assumed to be a time value and the second number is a percent change value. The entered time value is scaled by the 'timeScale' value. It is assumed that both values in the first entry are always zero since there can be no relative change from negative time.

Relative percent ramps are used to alter load value between each entry. For example, if the load control agent shown in Figure 2.10 was used in a simulation, ramps would be created for each load in area 2 that increases load by 3.675% between time 2 and 10, 6.474% between time 10 and 20, and 3.770% between time 20 and 30. In total, a 100 MW load would be 114.548 MW after all ramps executed. The relative percent is based off the value of each load at start of each ramp. This method of relative percent changing allows for other perturbances, such as noise, to be applied to the same load without control conflicts. Alternative ramp types may be employed by changing the 'rampType' parameter, but are not created as of this writing. It should be noted that the first ramp starts at 'startTime',

but all other ramps begin according to the calculated scaled time schedule. Additionally, loads that are off at system initialization (status 0 at time 0) are ignored.

2.3.2.3.5 Generation Control Dictionary

To manipulate generation in the same way a load control agent manipulates load, a generation control agent may be defined in the .ltd.py file. The definition of a generation control agent is very similar to the definition of a load control agent by design, however, differences do exist. Generation control agents are defined in the dictionary named mirror.sysGenerationControl, have a list of strings detailing control generators, and have a time value tuple list named 'forcast'. While the forecast misspelling was unintentional, and can be changed, time has proven it to be a minor detail. Other parameters inside the generation control agent definition (such as area, start time, time scale, and ramp type) function exactly the same as the load control agent. Forecast data is again collected from the EIA website and parsed in the same manner as demand data. Figure 2.11 shows an example of a generation control agent definition.

```
mirror.sysGenerationControl = {
    'testSystem' : {
        'Area': 2,
        'startTime' : 2,
        'timeScale' : 10,
        'rampType' : 'per', # relative percent change
        'CtrlGens': [
            "gen 3 : 0.25",
            "gen 4 : 0.75",
            ],
        # Data from: 12/11/2019 PACE
        'forcast' : [
            #(time , Precent change from previous value)
            (0, 0.000),
            (1, 5.137),
            (2, 6.098),
            (3, 4.471),
                    ],# end of forcast tuple list
                }, #end of testSystem def
   }# end of sysLoadControl dictionary
```

Figure 2.11: Generation control agent dictionary definition example.

The main difference between load and generation control agents is the addition of the 'CtrlGens' list of strings. While the load control agent distributes any changes to all loads equally, a generation control agent dispatches a specific portion of MW change to specific generators. The 'CtrlGens' list dictates which generators receive how much of a dispatch. Each string inside the 'CtrlGens' list is of the form: "gen BusNumber ID : Participation Factor" where ID is optional. If ID is not defined, as shown in Figure 2.11, the first generator on the given bus will be controlled. The participation factor is used to distribute the total requested MW change.

For example, if an area is generating 100 MW at time 0, the total requested area generation change by time 10 would be 5.137 MW. The generator on bus 3 would increase 1.28 MW while the generator on bus 4 would increase 3.85 MW. If a controlled generator has a governor, governor reference will be adjusted instead of mechanical power. The participation

factor for all listed generators should always sum to 1.0 or improper distribution **will** occur. It should be noted that not all generation must be controlled for proper percent change of total area output power, however, if a controlled machine hits a generation limit, excess changes are ignored.

Relative percent ramps are used to control generators in the same way as load so that a BA can also act on generators under generation control, however, this functionality is untested as of this writing.

2.3.2.3.6 Governor Input Delay and Filtering Dictionary

The inputs to modeled governors may be delayed, filtered, and gained using the Laplace domain block shown in Figure 2.12. If delay is not divisible by the simulation time step, rounding will occur.

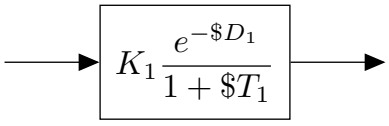

Figure 2.12: Block diagram of delay block.

To modify a governor model with a delay block, parameters may be entered in the governor delay dictionary mirror.govDelay. Like previously described dictionaries, this is located in the .ltd.py file. Figure 2.13 shows an example of a valid delay dictionary that affects the governor of the generator on bus 3. Note that while genId is optional, if not specified, the first generator found on the specified bus is used.

```
mirror.govDelay ={
        'delaygen3' : {
                'genBus' : 3,
                'genId' : None, # optional
                # (delay parameter, filter time constant, optional gain)
                'wDelay' : (40,30),
                'PrefDelay' : (10, 0),
        }, # end of 'delaygen3' definition
}# end of govDelay dictionary
```

Figure 2.13: Governor delay dictionary definition example.

Tuples are used to enter delay block parameters. Using Figure 2.3 as reference, the 'wDelay' tuple contains settings for D_A, T_A, and K_A respectively. Likewise, the 'PrefDelay' tuple contains D_B, T_B, and K_B. If the tuple contains three entries, the third is assigned to the optional gain associated with each block, otherwise K_x is one.

2.3.2.3.7 Governor Deadband Dictionary

A BA agent may be used to set area wide deadbands, but it is also possible to specify a single deadband for any governed generator. Individual governor deadband dictionaries are defined inside the mirror.govDeadBand dictionary in the .ltd.py file. Settings in each governor deadband dictionary will override any deadband settings specified by the BA dictionary. Figure 2.14 shows three examples of valid governor deadband definitions.

```
mirror.govDeadBand ={
    'gen3DB' : {
        'genBus' : 3,
        'genId' : None, # optional
        'GovDeadbandType' : 'ramp', # step, ramp, nldroop
        'GovDeadband' : 0.036, # Hz
        },
    'gen1DB' : {
        'genBus' : 1,
        'genId' : None, # optional
        'GovDeadbandType' : 'nldroop', # step, ramp, nldroop
        'GovAlpha' : 0.016, # Hz, used for nldroop
        'GovBeta' : 0.036, # Hz, used for nldroop
        },
    'gen4DB' : {
        'genBus' : 4,
        'genId' : None, # optional
        'GovDeadbandType' : 'step', # step, ramp, nldroop
        'GovDeadband' : 0.036, # Hz
        'GovAlpha' : 0.016, # Hz, used for nldroop
        'GovBeta' : 0.036, # Hz, used for nldroop
        },
}# end of govDelay dictionary
```

Figure 2.14: Governor deadband dictionary definition example.

Entering ‘step’ or ‘ramp’ as a value for the ‘GovDeadbandType’ will create a step or ramp deadband at the given ‘GovDeadband’. A non-linear droop governor deadband may be configured by setting the ‘GovDeadbandType’ to ‘NLDroop’ and entering desired ‘GovAlpha’ and ‘GovBeta’ values. Deadband types are fully explained in Section 3.1.1.

2.3.2.3.8 Definite Time Controller Dictionary

During long simulations, system loading may change from initial values by more than ±20%. Such changes can cause voltage issues that require the setting or un-setting of components contributing to available system reactive power. This can be accomplished by defining a definite time controller (DTC) agent in the mirror.DTCdict dictionary. Other general programmable logic operations may also be accomplished using a DTC agent. Figure

2.15 shows an example of a valid DTC definition where a shunt is actuated by changes in bus voltage or branch MVAR flow. It should be noted that instead of using a specific 'tarX' in a timers 'act' field, operations on any off or on target can be accomplished by using 'anyOFFtar' or 'anyONtar' respectively.

```
mirror.DTCdict = {
        'ExampleDTC' : {
                'RefAgents' : {
                        'ra1' : 'bus 8 : Vm',
                        'ra2' : 'branch 8 9 1 : Qbr', # branches defined from, to, ckID
                },# end Reference Agents
                'TarAgents' : {
                        'tar1' : 'shunt 8 2 : St',
                        'tar2' : 'shunt 8 3 : St',
                }, # end Target Agents
                'Timers' : {
                        'set' :{
                                'logic' : "(ra1 < 1.0) or (ra2 < -15)",
                                'actTime' : 30, # seconds of true logic before act
                                'act' : "tar1 = 1",
                        },# end set
                        'reset' :{
                                'logic' : "(ra1 > 1.04) or (ra2 > 15)",
                                'actTime' : 30, # seconds of true logic before act
                                'act' : "tar1 = 0",
                        },# end reset
                        'hold' : 60, # minimum time between actions
                }, # end timers
        },# end ExampleDTC definition
}# end DTCdict
```

Figure 2.15: Definite time controller dictionary definition example.

Each DTC employs a set and a reset timer and may have a hold timer if hold time is set larger than zero. Multiple references and targets can be associated with a DTC, however, as of this writing only one action can be associated with each timer. Any logic string entered in a timer uses the given key names for each reference or target and is evaluated using standard Python logic conventions.

2.3.3 Simulation Initialization

To clarify the explanation of simulation initialization, the entire process can be broken into three parts: process creation, mirror initialization, and dynamic initialization pre-simulation loop. The first part (process creation) involves creating and configuring various software processes that enable the simulation to run. The majority of part two (mirror initialization) revolves around collecting data from PSLF to create a Python duplicate, or mirror, of the power system model. The last part of simulation initialization (dynamic initialization pre-simulation loop) handles user input and creates PY3 dynamics before entering the simulation loop.

2.3.3.1 Process Creation

System initialization begins in PY3 with package imports and creation of truly global variables before user input from the .py file is handled. The .py file includes debug flags, simulation notes, simulation parameters, and file locations of the .sav, .dyd, and .ltd.py files. If all file locations are valid, PY3 initializes AMQP queues, sends appropriate initialization information to the IPY queue, starts the IPY_PSLTDSim process, and then waits for an IPY response message.

The IPY process begins by also importing required packages and setting references to certain imported packages as truly global variables. An IPY AMQP agent is created and linked to the AMQP host generated by the PY3 process. This allows the IPY process to receive initialization information from the PY3 AMQP message sent before the IPY process was evoked. IPY uses the received initialization information to load the GE Python API which is then used to load the .sav and .dyd into PSLF. Upon successful file loading in PSLF, a global reference to the object is created.

2.3.3.2 Mirror Initialization

Once the PSLF specific files are loaded into the GE software, initialization of the Python environment, or system model, may begin. The system model, referred to as the system mirror, or just mirror, is a single object that almost all other Python objects are

created inside. The mirror is a single system object with a recursive data structure that allows any object the ability to reference any other object as long as they both share a reference to the mirror, and are themselves referenced by the mirror. While the previous sentence may seem overly complicated, the use of such a linking technique eliminated the need for global variables outside of imported packages and also allowed for a single file containing **all** simulation data to be easily exported at the end of a simulation.

The system mirror begins its initialization by creating placeholder variables for simulation parameters, counters, and future agent collections. Specific case parameters, such as the number of buses or generators, are collected from PSLF to be used later in model verification processes.

Before any specific power system object data is collected, an initial power flow is performed to ensure that the loaded .sav is solvable, and to establish a steady state system operating point. System agents, representing power system objects like buses and loads, are then added to the mirror. The adding process queries PSLF for any buses in a specific area, checks each found bus for any connected system components, and creates Python agents for relevant objects. The querying and adding process continues until all system buses are accounted for. Each type of found object is added to a running tally so that it can be compared with the expected values collected earlier. Once all system buses have been found or accounted for, any created agents that are intended to log data are collected into a list for simpler group stepping.

Each area tally of found agents is checked for coherency with expected values. Any inconsistencies between the amount of found and expected objects will trigger warnings, but the simulation will not stop if this occurs. This choice is due to differences between what is counted in PSLF as a valid area object and PSLTDSim, which has the option to ignore islanded objects.

Dynamic model information from the specified .dyd file is then parsed. Collected machine or governor parameters are used to create PSLF model information objects (PMIOs)

inside the mirror. These PMIOs collect inertia H and MW cap for each machine as well as turbine type, governor MW cap, and permanent droop R from governors. Other information, such as MVA base, is also collected for both types of model. This process is required as the .dyd values for certain parameters overwrite pre-existing values that may be saved inside the .sav file.

Once the .dyd file is parsed and all PMIOs are created, the combined system inertia is calculated. For each found generator PMIO, the associated mirror agent is located and updated so that H and Mbase values match those found in the .dyd. The total system inertia is calculated according to Equation 2.1 and user input settings are interpreted so that any requested changes, such as scaling or alternative system inertia inputs, are handled correctly.

Mirror search dictionaries are then created to simplify and speed up agent searches and the global slack generator is identified. The global slack generator is important to locate as its variance from an expected value dictates accelerating power re-distribution and power-flow solution iterations during the simulation loop. Once search dictionaries are created, the IPY model is fully initialized. The mirror is then saved to disk and an AMQP message is sent to PY3 with the mirror location.

2.3.3.3 Dynamic Initialization Pre-Simulation Loop

After the handoff AMQP message is sent to PY3, IPY initializes values required for simulation and awaits an AMQP message from PY3 before entering its simulation loop.

PY3 uses the information received from IPY to load the newly created system mirror so that PY3 can perform further initializations such as creating any dynamic agents (e.g. governors), calculating area frequency characteristics and maximum capacities, executing any .ltd code, and creating the associated agents from the .ltd input.

PY3 dynamics are initialized to ensure R is on the correct PU base and any MW caps from dyd parsing are applied. Additionally, any limiting values for governor output are accounted for, and any deadbands or delays are created. Before entering the PY3 simulation loop, agents that are designed to log values initialize blank lists for expected values.

2.3.4 Simulation Loop

Once simulation initialization is complete, and the system mirror is initialized, the simulation loop is executed. Figure 2.16 shows the major actions that are processed each time step, but does not include details concerning AMQP communication. AMQP messages are sent and received during the 'Update' blocks and the system mirrors are checked for coherency at these times as well.

The simulation loop can be viewed as starting with the increment of simulation time followed by the stepping of any dynamic agents. At the time of this writing, this process includes: integrating the combined swing equation to calculate a new system frequency, stepping BA agents and any agents nested in a BA agent, stepping any definite time controllers, and finally stepping all dynamic governor agents. As mentioned in Section 2.2.3, the method used for integrating a new system frequency (solve_ivp) returns numerous values between the set integration window, but only the last result is stored and used for further calculations. All dynamic agents that use a staged dynamic calculation approach (such as governors) perform a full time step of integration before passing output from one stage to the input of the next dynamic stage. Detailed explanations of the numerical methods employed to accomplish these tasks is presented in Appendix A. After all dynamic agents have been stepped, generator electrical power is set equal to generator mechanical power. This step is the beginning of forming the next power-flow solution initial conditions.

Perturbance agents are then stepped. This means that any steps, ramps, or noise type events are performed, agents in both system mirrors are updated, and any related system value is changed accordingly. For example, the tripping of a generator requires the system inertia to change as well as the amount of power in the system. The variable ΔP_{pert} is used to keep track of system power changes. Additionally, BA action takes place at this time.

After all perturbance related actions are executed, accelerating power is calculated and distributed to the system according to generator inertia. The PSLF system is updated with any required values and a power-flow solution is attempted. If the solution diverges

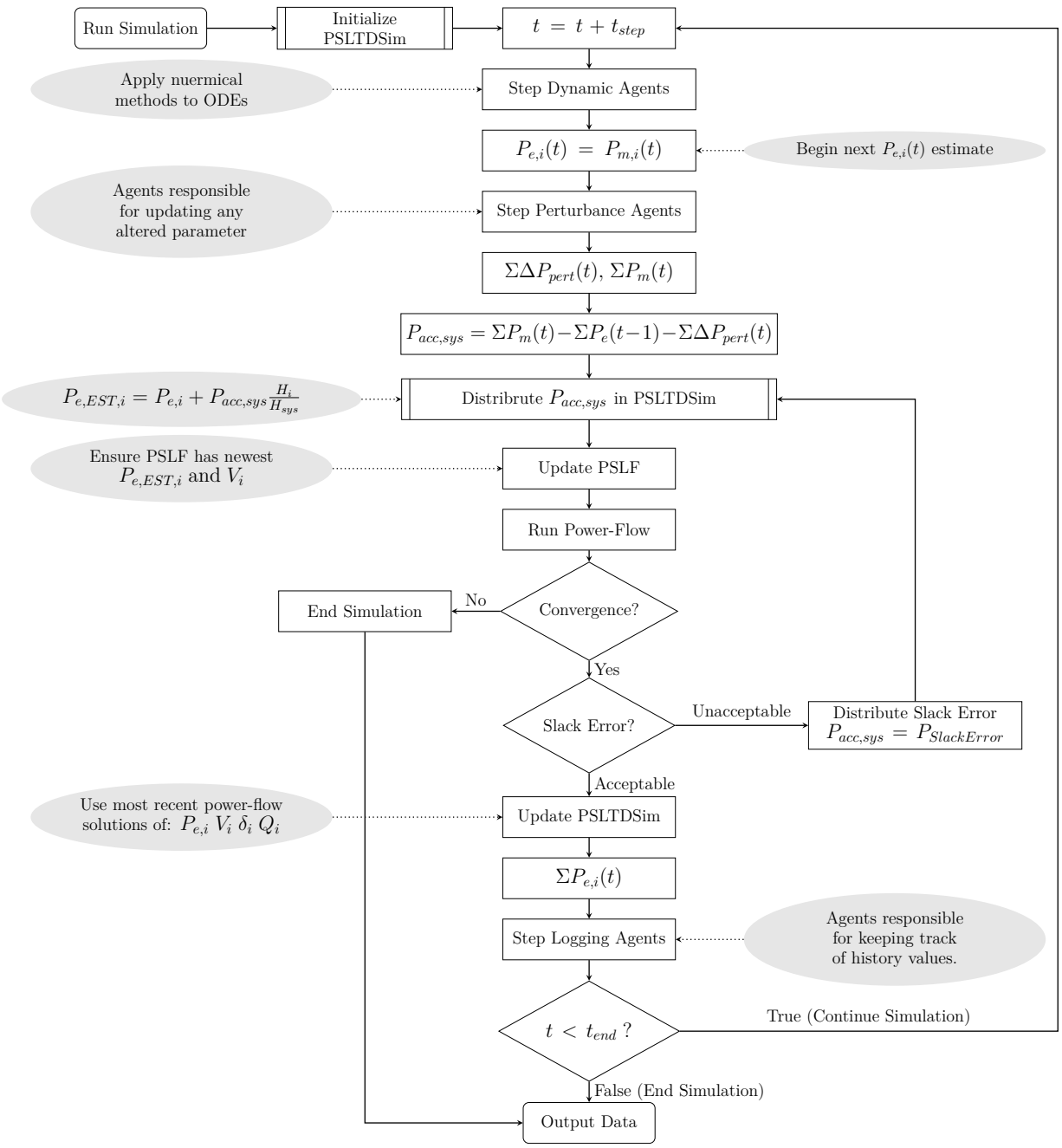

Figure 2.16: Simulation time step flowchart.

the simulation ends and any collected data is output. If the solution does not diverge, the magnitude of any slack error is checked against the slack error tolerance. If the slack error is larger than the slack tolerance, the error is redistributed to the system until the resulting error is within tolerance or the solution diverges. If the power-flow solution diverges during

accelerating power redistribution, the simulation ends and any collected data is output.

Once the system has converged to a point where the slack error is less than the slack tolerance, PSLF values for generator real and reactive power and bus voltage and angle are used to update the PY3 mirror. The electric power output of the system is summed for use in calculating system accelerating power in the next time step. Any logging agents are then stepped and data for that particular simulation time step are recorded. Finally, system time is checked and if the simulation is complete, any collected data is output. If the simulation is not complete, the simulation time is incremented by the time step and the cycle repeats itself again.

2.3.5 Simulation Outputs

Pre-defined data is collected by any agent with logging ability. Current agents with this ability are machines, loads, shunts, branches, transformers, areas, balancing authorities, and the system mirror itself. When a simulation is complete, the final system mirror is exported via the Python package `shelve` and options exist to export some data as a MATLAB .mat file. The .mat output is accomplished by combining various agent log dictionaries into a single dictionary. As such, only data deemed useful for validating the software is included.

It should be noted that numerous plot functions were created to easier visualize and validate python mirror data. Python plot functions are located in the PSLTDSim package plot folder, while the MATLAB validation plots are located in the GitHub repository only.

3 Simulated Engineering Controls

To further the investigation an engineering problem, the following simulated controls have been written and, when used in conjunction with the previously described software features, *may* prove useful.

3.1 Simulated Balancing Authority Controls

Simulated balancing authority controls affect governor response and AGC operation. These controls are defined in the sysBA dictionary in the .ltd.py file. Figure 3.1 shows an example of a BA parameter dictionary as it would appear in a .ltd.py file. A complete list of BA agent options is shown in Table B.1 in Appendix B. The following sections detail most sysBA dictionary keys.

```
# Balancing Authority Definition
mirror.sysBA = {
    'BA1':{
        'Area': 1,
        'B': "0.9 : permax", # MW/0.1 Hz
        'AGCActionTime': 30.00, # seconds
        'ACEgain' : 1.0,
        'AGCType':'TLB : 4', # Tie-Line Bias
        'UseAreaDroop' : False,
        'AreaDroop' : 0.05,
        'IncludeIACE' : True,
        'IACEconditional': True,
        'IACEwindow' : 30, # seconds - size of window - 0 for non window
        'IACEscale' : 1/5,
        'IACEdeadband' : 0, # Hz
        'ACEFiltering': 'PI : 0.04 0.0001',
        'AGCDeadband' : None, # MW? -> not implemented
        'GovDeadbandType' : 'nldroop', # step, None, ramp, nldroop
        'GovDeadband' : .036, # Hz
        'GovAlpha' : 0.016, # Hz - for nldroop
        'GovBeta' : 0.036, # Hz - for nldroop
        'CtrlGens': ['gen 1 : 0.5 : rampA',
                     'gen 2 1 : 0.5 : rampA',
                     ]
        }
```

```
26    }
```

Figure 3.1: Single sysBA dictionary definition.

3.1.1 Governor Deadbands

Modeling governor deadbands has become increasingly important to dynamic simulation [36], [51], [52]. The maximum recommended deadband is 36 mHz[20]. However, the execution of a governor deadband is not explicitly detailed and left to generator operators to configure. PSLTDSim offers a deadband agent that can apply various deadbands to the $\Delta\omega$ input of governors. Figure 3.2 graphically depicts how the various available deadband options differ.

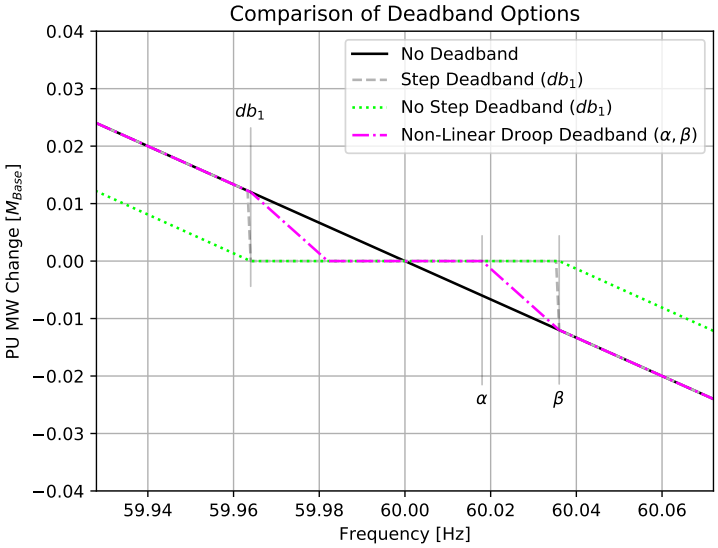

Figure 3.2: Examples of available deadband action.

A step deadband simply nullifies any $\Delta\omega$ whose absolute value is less than the prescribed deadband. A no-step, or ramp, deadband removes the step characteristic by essentially pushing the original droop curve out to deadband thresholds. While the no-step deadband has no abrupt changes, the actual governor response will always be below the ideal droop response. To eliminate this oversight, a non-linear droop option was created

that ramps from a set Hz deadband to the ideal droop curve.

Configuring a deadband is done via the sysBA parameter dictionary or the governor deadband dictionary in a .ltd.py file. The dictionary keys are the same in the sysBA dictionary as the govDeadBand dictionary. Details about creating deadbands using a governor deadband dictionary is presented in Section 2.3.2.3.7.

3.1.2 Area Wide Governor Droops

The typically recommended FERC droop is 5%[20]. Area wide governor droops can be specified in the sysBA dictionary that overwrite any droop setting previously read from a .dyd. All active governors in an area will use the specified 'AreaDroop' value if 'UseAreaDroop' is True. This setting allows for fast and easy configuration of simulations aimed at exploring droop settings.

3.1.3 Automatic Generation Control

A block diagram of the AGC model employed by PSLTDSim is shown in Figure 3.3. Simulation settings related to frequency bias, ACE integrating and filtering, conditional summing, and generation participation are explained in the following sections.

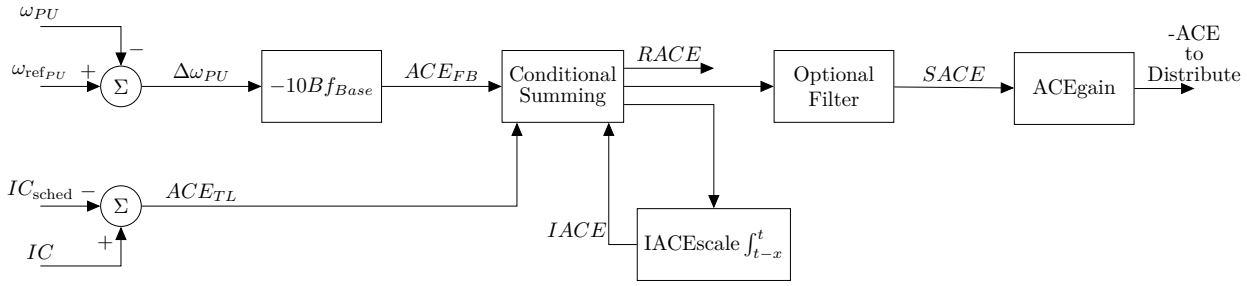

Figure 3.3: Block diagram of ACE calculation and manipulation.

3.1.3.1 Frequency Bias

Choosing a desired frequency bias B can be accomplished in a number of different ways. All methods are configured by the string entered as the 'B' value in the sysBA dictionary. The format of the 'B' string is "Float Value : B type". The available B types are scalebeta, perload, permax, and abs.

The scalebeta type will scale the automatically calculated area frequency response characteristic β by the given float value. The perload type will set B equal to the current area load times the given float value. The permax type will set B equal to the maximum area capacity times the given float value. The abs type will set B equal to the given float value. Note that the units on B are MW/0.1 Hz and, despite B being a negative number, is entered as a positive value.

A variable frequency bias may be used in distributed ACE signals. If the 'BVgain' dictionary entry is a non-zero value, a variable frequency bias B_V is calculated as

$$B_V = B_F \left(1 + K_B|\Delta\omega|\right) \tag{3.1}$$

where B_F is the fixed bias value, and K_B is the user entered value for 'BVgain'. It should be noted that $\Delta\omega$ is calculated according to Equation **??** and is a PU value. This leads to seemingly large required values of 'BVgain' before effects are noticeable. To clarify, B_V is only used in ACE calculations that are meant to be distributed to the generation fleet, RACE is always calculated using B_F.

3.1.3.2 Integral of Area Control Error

As previously shown in Figure 3.3, ACE may be integrated and fed back into the conditional summing block. The default signal sent to the integrator is RACE as AGC is meant to drive RACE to zero. Settings related to this process are configured in the sysBA dictionary. Integral of ACE (IACE) parameters are 'IACEwindow', 'IACEscale', and 'IACEdeadband'. IACEwindow defines the length in seconds of the moving window integrator. If IACEwindow is set to zero, integration will be continuous (i.e. for all time). IACEscale acts as a gain of the output integral value. IACEdeadband specifies the frequency deviation in Hz between which integration values will stop being added into the conditional summing block. IACEdeadband functionality was meant to alleviate frequency hunting.

3.1.3.3 Conditional Area Control Error Summing

Depending on the type of AGC agent chosen, ACE is calculated in different ways. The Tie-Line Bias (TLB) agent has multiple types of conditional ACE calculations. All conditionals involve comparing the sign of a value to the sign of frequency deviation. The main idea behind this conditional summing is to ensure that only events occurring inside an area will receive an AGC response. Options are available to allow for continued frequency response as well. Table 3.1 shows the conditional summations and associated TLB type.

Table 3.1: Tie-line bias AGC type ACE calculations.

TLB Type	ACE Calculation
0	$ACE_{FB} + ACE_{TL}$
1	$ACE_{FB} + ACE_{TL} * [\text{sgn}(\Delta\omega) == \text{sgn}(ACE_{TL})]$
2	$ACE_{FB} + ACE_{TL} * [\text{sgn}(\Delta\omega) == \text{sgn}(ACE_{FB} + ACE_{TL})]$
3	$ACE_{FB} * [\text{sgn}(\Delta\omega) == \text{sgn}(ACE_{FB})] + ACE_{TL} * [\text{sgn}(\Delta\omega) == \text{sgn}(ACE_{TL})]$
4	$[ACE_{FB} + ACE_{TL}] * [\text{sgn}(\Delta\omega) == \text{sgn}(ACE_{FB} + ACE_{TL})]$

3.1.3.4 Area Control Error Filtering

The calculated ACE can be put through a filter, or smoothed, to become smoothed ACE (SACE). The three basic filters created were low pass, integral, and proportional and integral (PI). Block diagrams of these filters are shown in Figure 3.4. It should be noted that these filters do not account for any integrator wind up. If wind up is predicted to be an issue, additional code must be added to the specific agent that check the output value and adjusts filter running states and output accordingly. Details about integrator wind up is presented in Section A.4.3.1.

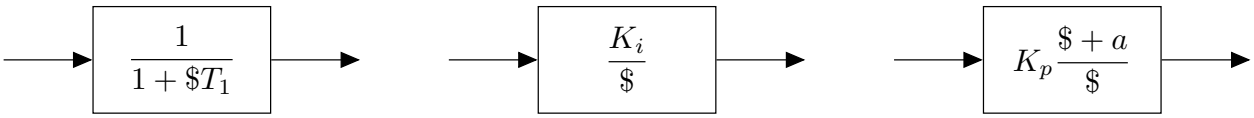

Figure 3.4: Block diagrams of optional ACE filters.

The selection and configuration of a filter is done in the BA parameter dictionary via the 'ACEFiltering' key value. The format of the string input to the 'ACEFiltering' key

is "type : val1 val2". Valid filter types are lowpass, integrator, and pi. The low pass and integrator take only one value while the PI filter takes two. In the case of the low pass filter, the passed in value is set as the low pass time constant. The value passed in with an integrator filter is simply a gain. The first PI value is used as a proportional gain value K_p and the second value describes the ratio between integral and proportional gain a.

3.1.3.5 Controlled Generators and Participation Factors

Each BA is configured with a list of controlled generators that receive AGC signals. These generators, or power plants, are given a participation factor between 1.0 and zero. The participation factor dictates the percent of ACE signal each agent receives. A check is done to ensure each BA has a total participation factor of one, however, if the sum of participation factors is not one, only a warming is issued. It is up to the user to enter reasonable values. Additionally, each list value of the 'CtrlGens' key describes if the signal should be applied as a step or a ramp. Ramps are best when single units are receiving ACE, while steps are useful for power plants that are intended to handle distribution of ACE independently.

4 Engineering Case Study Examples

Often times, examples are the best way to learn. This Chapter presents engineering case studies that may be useful starting points for future experiments.

4.1 Governor Deadband Effect on Valve Travel

Initial research goals included maximizing system stability while minimizing machine effort. The decided upon metric for machine effort was valve travel. As governor deadbands directly affect valve travel, a study into deadbands was conducted using PSLTDSim.

4.1.1 Governor Deadband Simulation Configuration

To assess long-term impacts of governor deadbands, thirty minutes of random load noise was applied to the mini WECC. All governors had identical deadband settings and PSLTDSim was used to set all governor droops to 5%. Some governors were removed from the system so that only ≈20% of generation capacity in each area had governor control. Each type of deadband shown in Figure 3.2 was simulated. No-step and non-linear droop deadbands had a threshold of 16 mHz while the step deadband used a 36 mHz deadband. Noise agent N_Z was set to 0.03 for all simulations with random walk behavior enabled. As a reminder, explicit noise agent behavior is explained in Section 2.3.2.3.2. The change in system loading caused by the noise agent is shown in Figure 4.1.

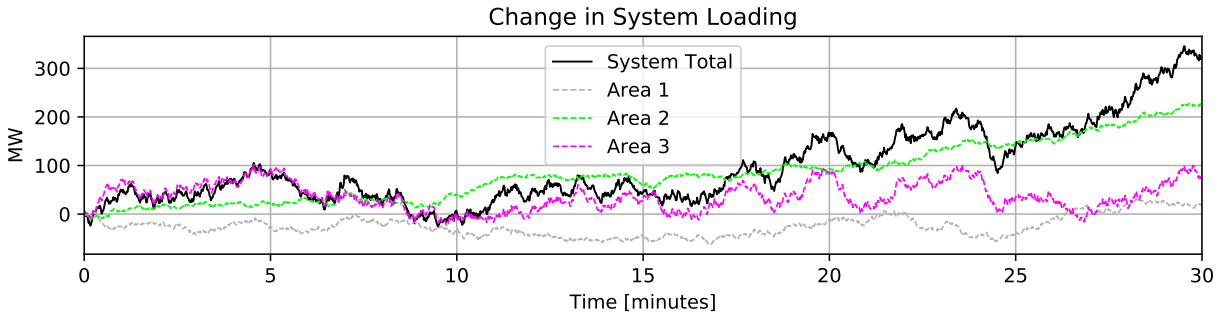

Figure 4.1: Cumulative system change in load for governor deadband simulation.

Another experiment was conducted to explore a non-homogeneous deadband scenario where all deadbands were of the no-step type, but some had different mHz deadbands.

Although PSLTDSim can model AGC, it was not enabled for these deadband simulations.

4.1.2 Governor Deadband Simulation Results

Figure 4.2 shows the resulting system frequency for each type of deadband. The step deadband holds frequency almost exactly on the set deadband except when system loading decreases during minutes 7-11. The other deadband options maintain system frequency near their respective mHz setting until loading increases beyond a point near minute 17.

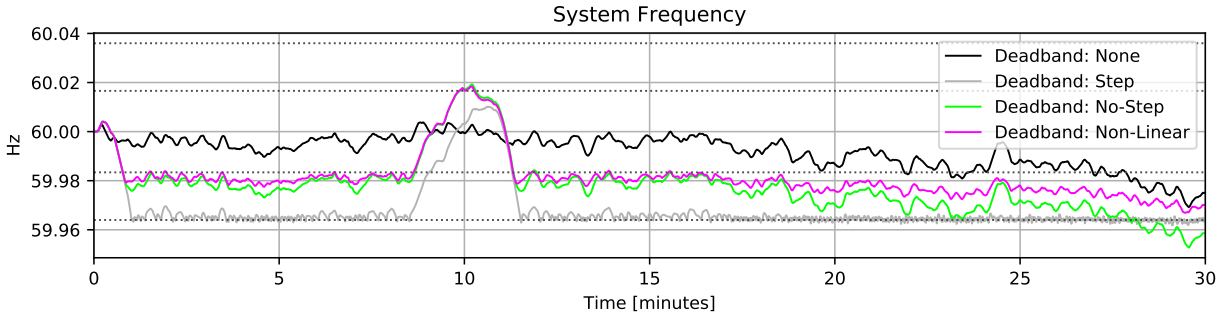

Figure 4.2: System frequency comparison of different deadband scenarios.

The first three minutes of a single generator's valve travel are shown in Figure 4.3 to compare how different deadbands affect valve movement. The step type deadband results in pulse train-esq control signals being sent when system frequency is oscillating near the deadband.

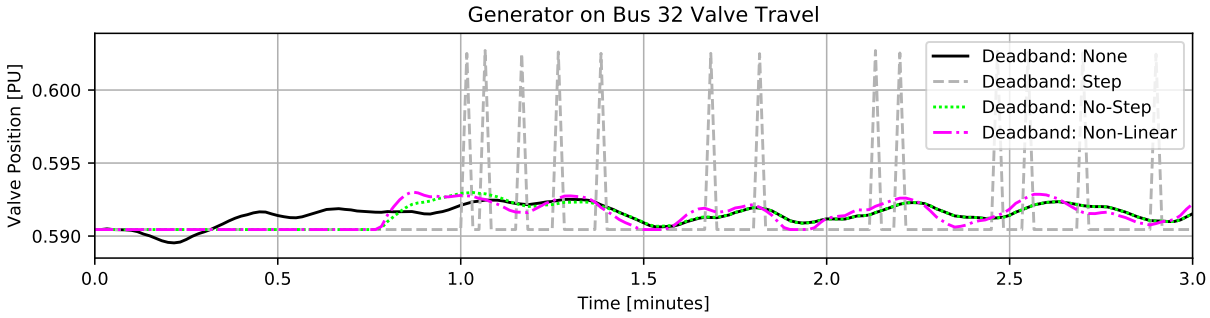

Figure 4.3: Detail comparison of initial valve movement.

With all governors using a no-step type deadband, two of the three areas mHz deadband was set to 16.6 mHz, while the third was set to 36 mHz. The left plot of figure 4.4 shows average valve travel over time in a homogeneous deadband system while the right plot

shows non-homogeneous valve travel results. In the homogeneous case, all areas have equal valve travel. In the non-homogeneous case, the larger deadband used in area 3 prevents governor response until minute 18.

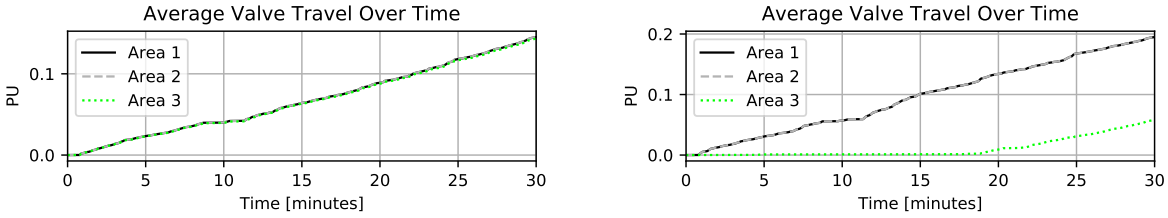

Figure 4.4: Area valve travel for homogeneous and non-homogeneous scenarios.

Complete valve travel result plots are presented in Appendix **??**. A tabular summary of valve travel is shown in Table 4.1. Repeated control pulses associated with a step type deadband greatly increase valve travel over the more linear deadband options. Simulation results reinforce NERC recommendations that step deadbands not be used [20]. When a variety of deadband thresholds are employed, total valve travel may decrease while certain individual movement increases.

Table 4.1: Total valve travel for various deadband scenarios.

	Valve Travel [PU]				
Generator	No DB	Step	No-Step	N-L Droop	No-Step Non-H
17	0.16	7.48	0.15	0.23	0.19
23	0.16	7.48	0.15	0.23	0.19
30	0.16	7.48	0.15	0.23	0.19
32	0.16	7.54	0.15	0.23	0.19
107	0.16	7.54	0.15	0.23	0.19
41	0.15	6.44	0.14	0.23	0.06
45	0.15	6.44	0.14	0.23	0.06
53	0.16	7.54	0.15	0.23	0.06
59	0.15	6.44	0.14	0.23	0.06
Total:	1.41	64.38	1.32	2.07	1.19

4.2 Automatic Generation Control Tuning

After a contingency, AGC acts to restore nominal operating conditions. Long-term simulation is required to simulate AGC action as gentle system recovery takes multiple minutes. According to [34], AGC should respond only to internal events or to correct frequency. PLSTDSim simulations using conditional AGC provide results that show conflicting AGC action extends recovery time and increases machine effort.

4.2.1 AGC Simulation Configuration

Using the two area six machine system, a 150 MW loss of generation event in each area was used to tune AGC response to a large contingency. AGC settings were manipulated until an individual BA could restore system frequency in less than 10 minutes. The effect of noise and non-linear governor deadbands was also simulated. Random noise added to each simulation is shown in Figure 4.5. Conditional ACE was used to show conflicting control effort when a BA responds to out-of-area events.

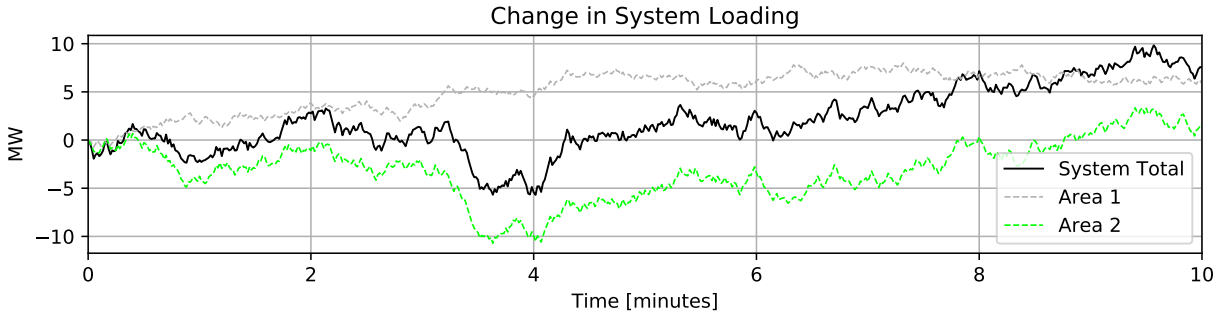

Figure 4.5: Random noise added to AGC simulations.

Full code for simulating an external area event with tuned conditional AGC is shown in Figures C.2 and C.3 in Appendix C.

4.2.2 AGC Simulation Results

4.2.2.1 Base Case Results

Using the two area six machine system, a -150 MW step in generator power was simulated. Figure 4.6 shows the system frequency response with primary control only.

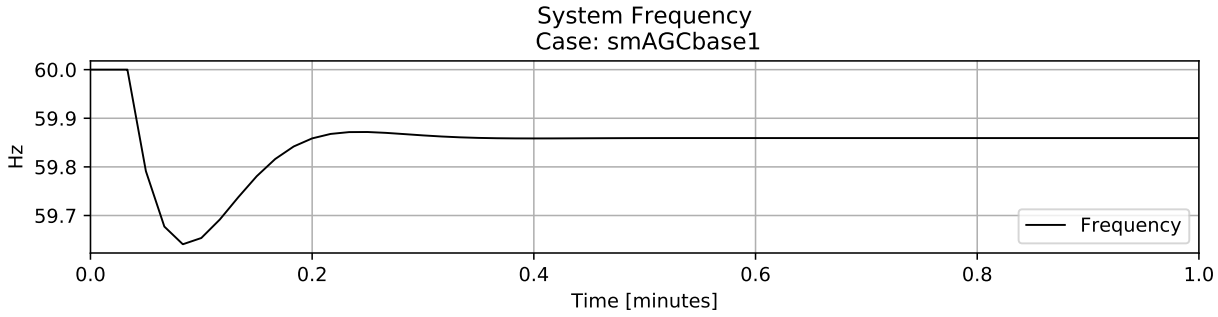

Figure 4.6: Frequency response to generation loss event in area 1.

Calculated BA values such as, reporting ACE (RACE) and interconnection (IC) error, will be different depending on where the system loss occurs. Specifically, which area the event occurs in dictates BA values. Figures 4.7 and 4.8 show BA values for the -150 MW generation step event in area 1 and area 2 respectively.

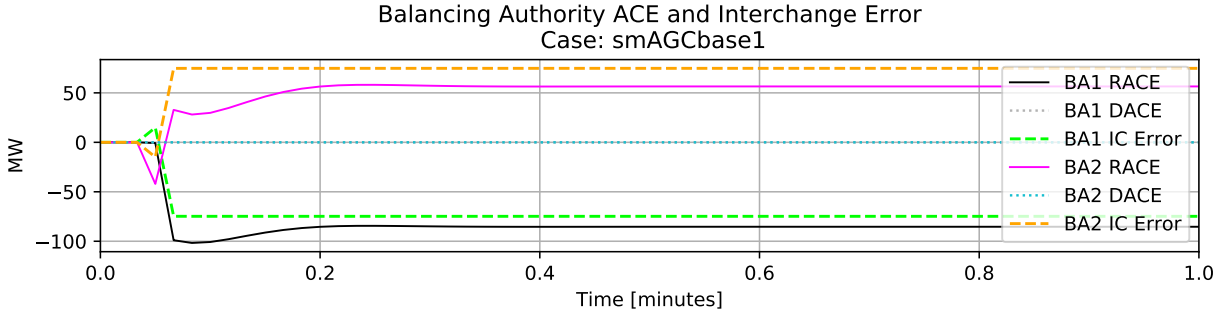

Figure 4.7: Calculated BA values during generation loss event in area 1.

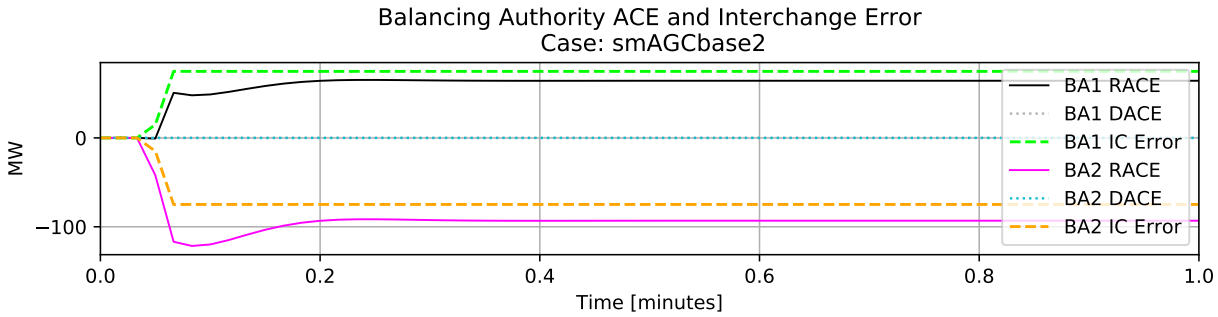

Figure 4.8: Calculated values during generation loss event in area 2.

In a two area system, IC error is symmetric and the scaling of RACE by frequency bias can be clearly seen. The initial negative RACE for BA2 shown in Figure 4.7 is not fully understood. It may be due to the multiple generators assigned on the bus where the loss of generation occurs. When a step in Pm occurs to a single bus generator, as Figure 4.8 shows, the odd RACE behavior is not replicated. To observe system response without any AGC action, AGC gain was set to zero for both areas causing distributed ACE (DACE) to also be zero.

4.2.2.2 AGC Tuning Results

The AGC tuning process and results from both areas were similar. Area 1 results and discussion are presented in this section and area 2 results are included in Appendix **??**. The BA controlling area 1 was equipped with an AGC routine that included a scaled window integrator, PI smoothed ACE, and an action time of 30 seconds. Resulting frequency response is shown in Figure 4.9. Calculated RACE, IC error, and DACE are shown in Figure 4.7.

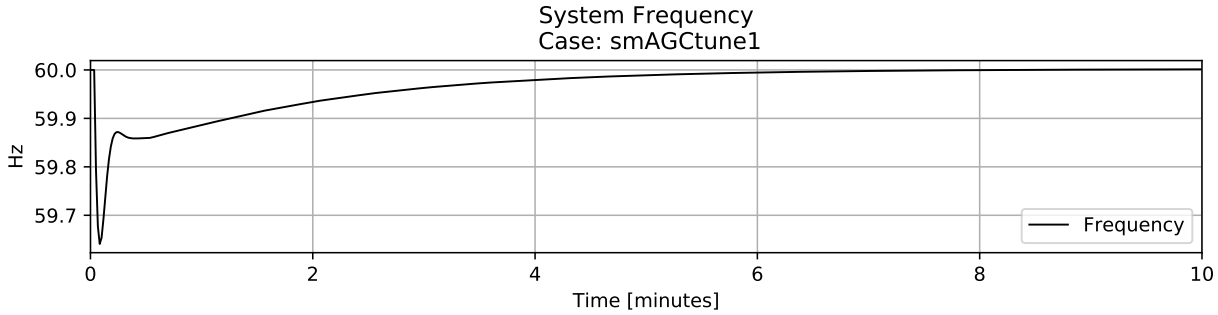

Figure 4.9: AGC Frequency response to area 1 base case scenario.

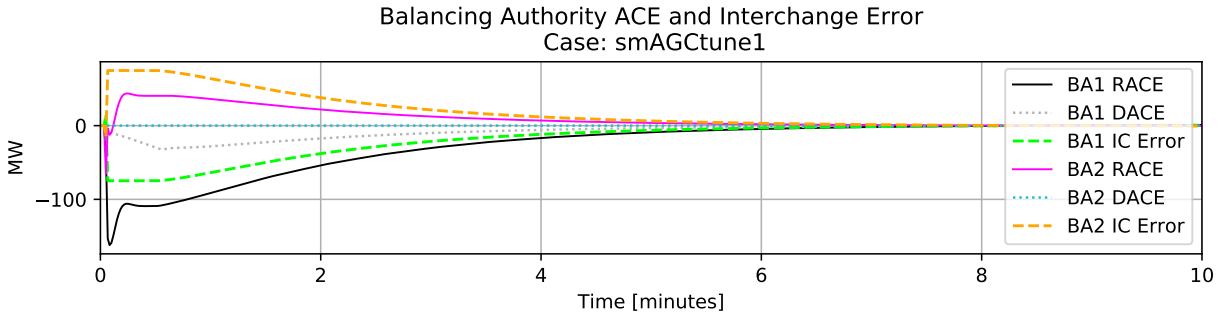

Figure 4.10: Calculated BA values during area 1 AGC tuning.

Figures 4.11 and 4.12 show governor power output and power reference set point responses for each area. During AGC tuning, ACE gain was set to zero for the BA routine not being tuned. As such, area 2 has no P_{ref} changes while area 1 adjusts its controlled governors to return RACE to zero. The effect of AGC on P_{ref} for generator 1 1 and generator 2 1 is identical.

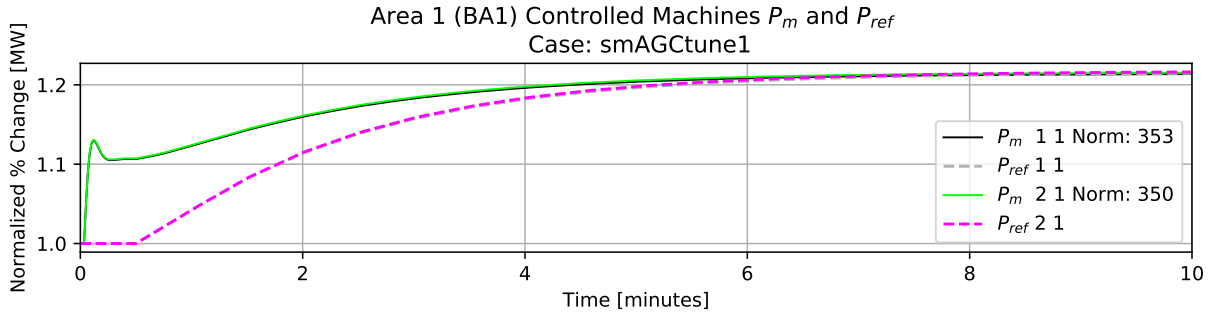

Figure 4.11: Area 1 controlled generation response during AGC tuning.

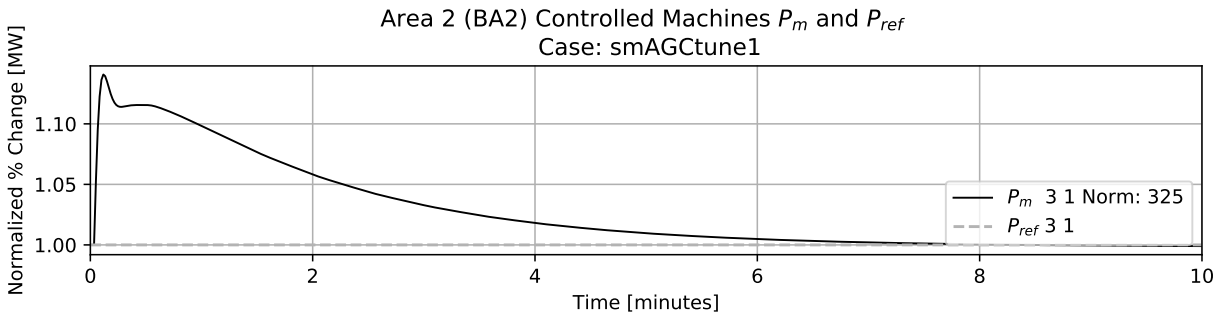

Figure 4.12: Area 2 controlled generation response during AGC tuning.

4.2.2.3 Noise and Deadband Simulation Results

To add slightly more realism to the event, noise and governor deadbands were added to the simulation. Noise was set to 0.05% with random walk enabled. All AGC controlled governors were of the non-linear droop variety with an α of 16 mHz and a β of 36 mHz. Resulting frequency is shown in Figure 4.13. System frequency oscillations between governor deadbands occurs from roughly minute 6 onwards.

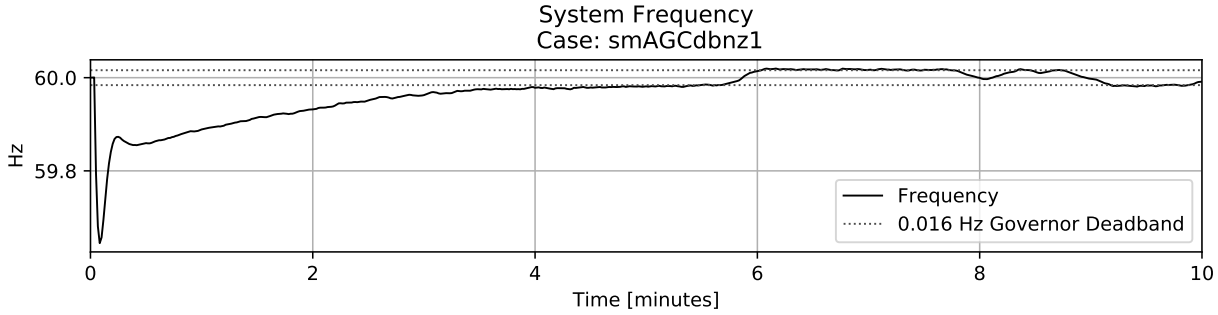

Figure 4.13: AGC frequency response with noise and deadbands.

Figures 4.14, 4.15, and 4.16 show calculated BA values and individual area controlled machine responses respectively. Despite the addition of noise and governor deadbands, system recovery is similar to ideal simulation conditions.

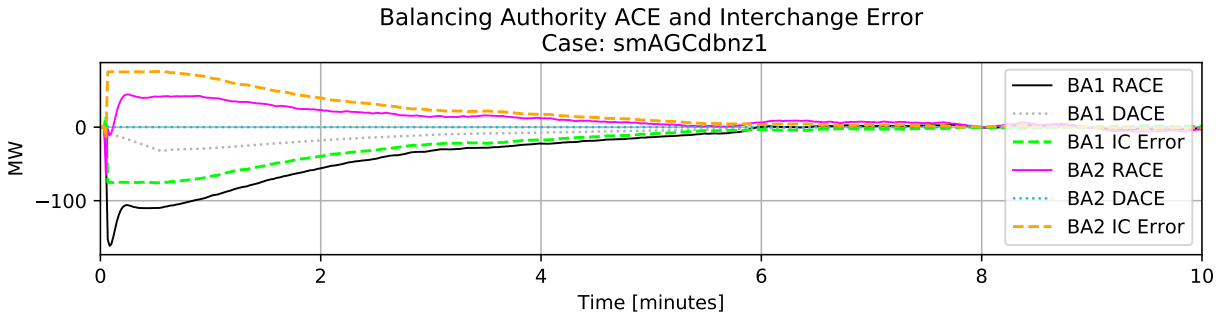

Figure 4.14: Calculated BA values with noise and deadbands.

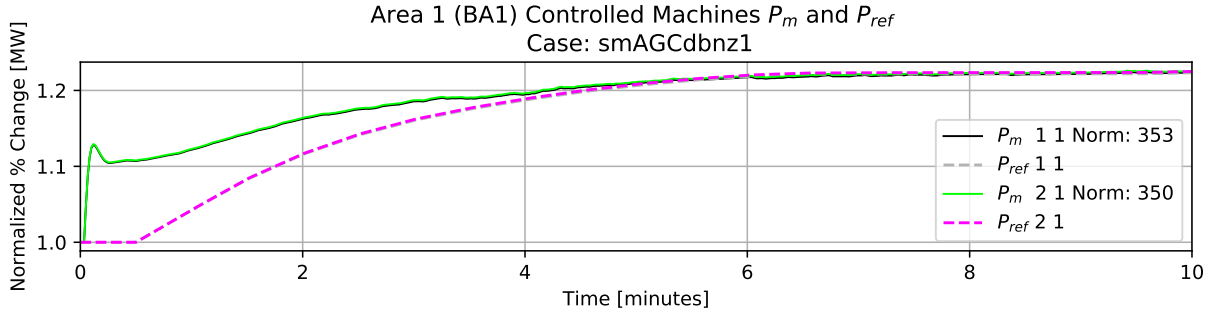

Figure 4.15: Area 1 controlled generation response to noise and deadbands.

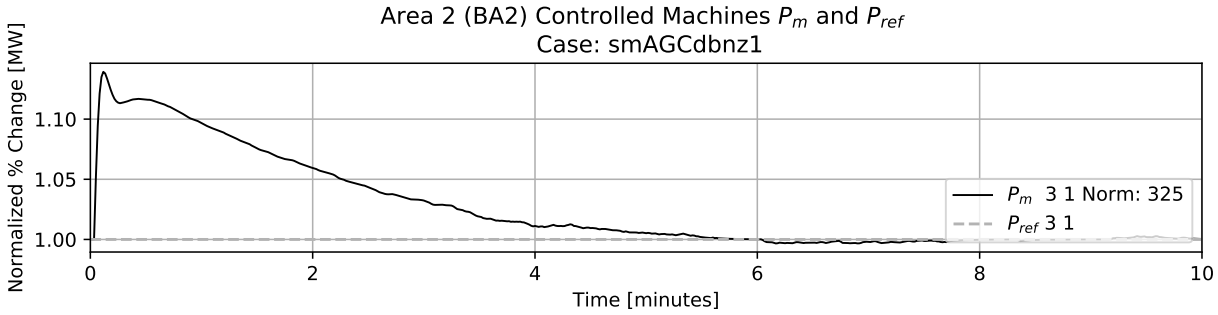

Figure 4.16: Area 2 controlled generation response to noise and deadbands.

4.2.2.4 Conditional ACE Results

After tuning for both BA AGC routines was complete, conditional ACE could be tested. Comparing TLB type 0, which is non-conditional, and conditional TLB type 4 involved enabling AGC action for both areas and simulating an in-area, and out-of-area event. Figure 4.17 shows that system frequency does not return to the nominal operating

point in 10 minutes when TLB 0 is used. Figure 4.18 shows that BA2 DACE acts opposite BA1 DACE.

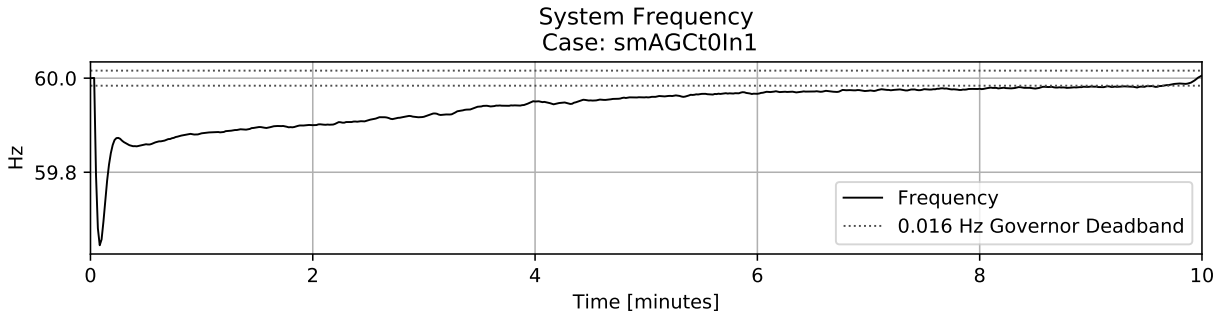

Figure 4.17: Frequency response to event using TLB 0.

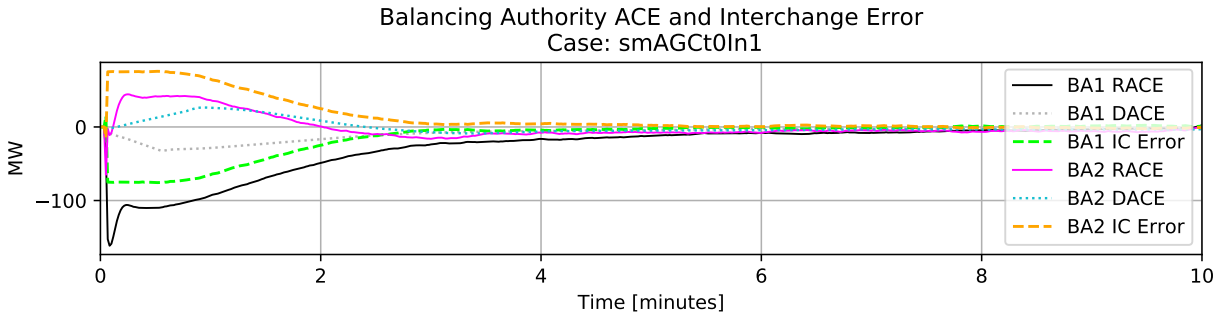

Figure 4.18: Calculated BA values during an event using TLB 0.

To more clearly show conflicting control action, Figures 4.19 and 4.20 provide individual area control responses. While BA1 acts to restore frequency by increasing generator output, BA2 acts to restore area interchange by reducing generator output. These control actions are opposite and thus prolong system recovery.

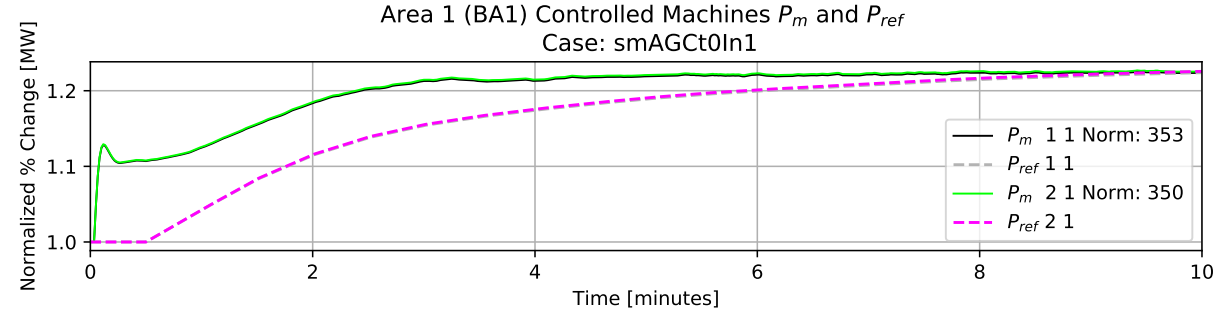

Figure 4.19: Area 1 controlled generation response to internal area event using TLB 0.

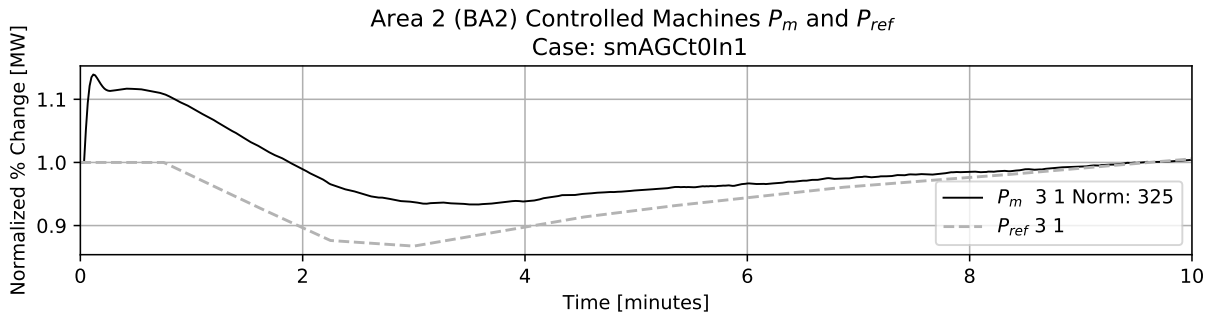

Figure 4.20: Area 2 controlled generation response to external area event using TLB 0.

Figure 4.21 shows system frequency response when each BA is set to send conditional ACE according to TLB type 4 rules. Similar behavior to the noise and deadband only scenario is reproduced. Calculated BA values in Figure 4.22 show that BA2 DACE is zero for the event. As the event is external to area 2, it does not require a response from BA2.

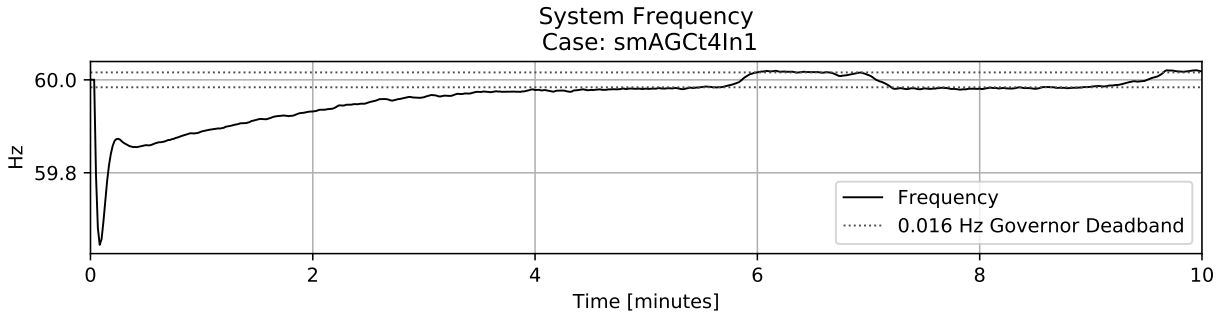

Figure 4.21: Frequency response to event using TLB 4.

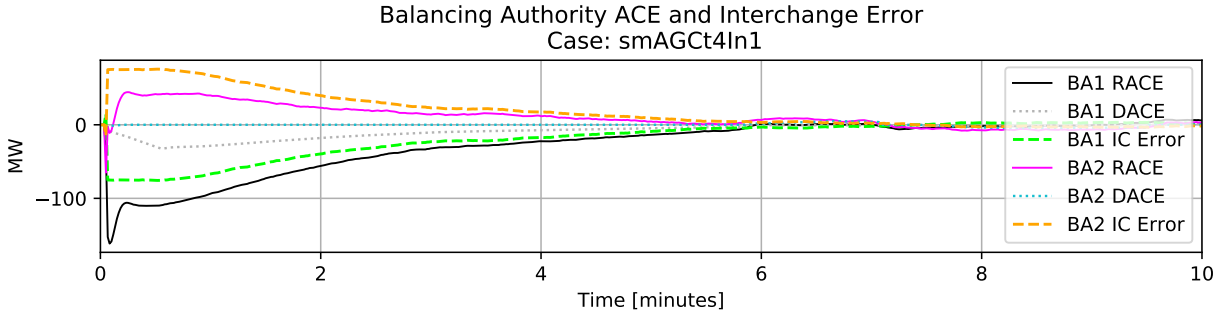

Figure 4.22: Calculated BA values during an event using TLB 4.

Figures 4.23 and 4.24 show that controlled machine response is similar to tuned

conditions when using TLB type 4. The P_{ref} AGC response in area 2 near minute 7 is believed to be due to a combination of random load changes affecting area interchange and frequency oscillations caused by governor deadbands.

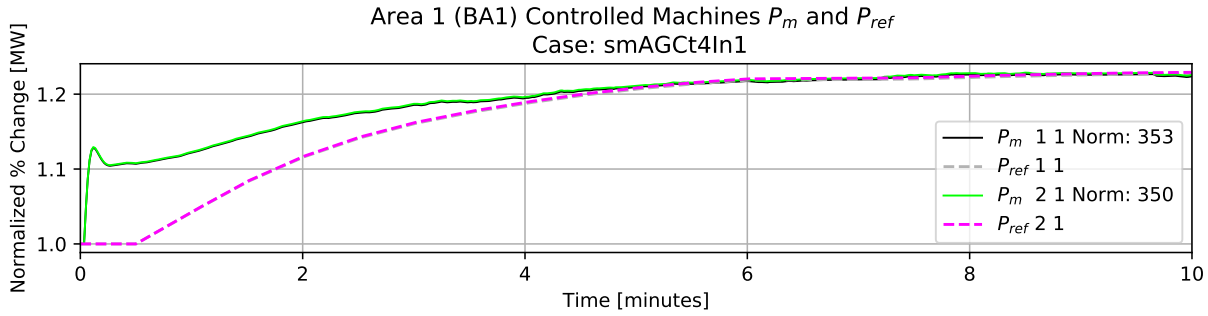

Figure 4.23: Area 1 controlled generation response to internal area event using TLB 4.

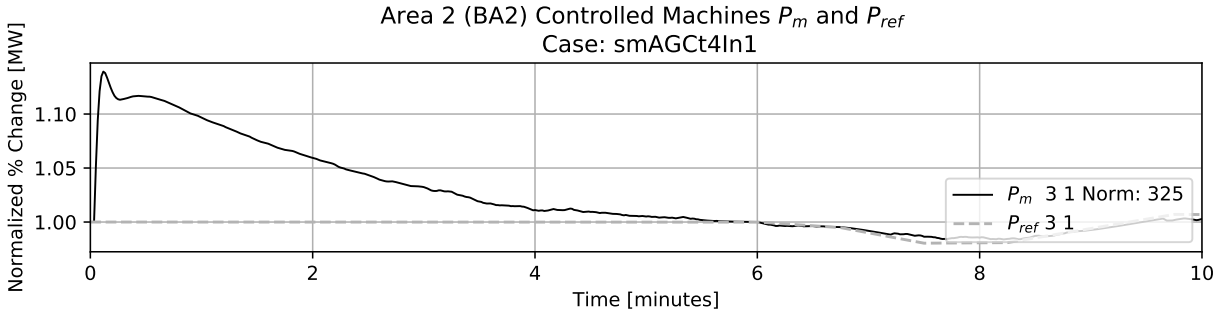

Figure 4.24: Area 1 controlled generation response to external area event using TLB 4.

While an internal event to area 1 is external to area 2, and vice versa, to throughly test conditional AGC behavior, an event was simulated in area 2 using both TLB 0 and TLB 4. Results were similar to previously conducted conditional AGC tests. For completeness, plotted results are presented in Appendix **??**.

4.2.2.5 BAAL Results

Even though this scenario is not longer than 30 minutes, an introduction to the plots used to check for control adherence to BAL-001-2 is worthwhile. As a reminder, BAL-001-2 deals with BAAL, which is calculated using Equation **??** and a minute averaged frequency.

The slowest AGC recovery case, which used non-conditional ACG, was chosen for study into BAAL. Figure 4.25 shows RACE exceeded the BAAL for nearly 4 minutes in

area 1 during the simulated event. Area 2 BAAL, shown in Figure 4.26, was exceeded immediately following the perturbance again and briefly near minute 3. In both areas, AGC action reduced RACE to acceptable levels before any NERC violations occurred.

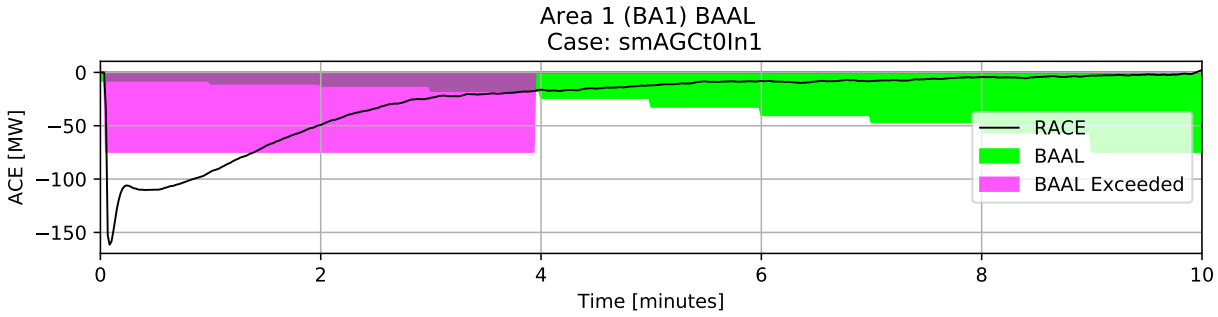

Figure 4.25: Area 1 BAAL during internal area event using TLB 0.

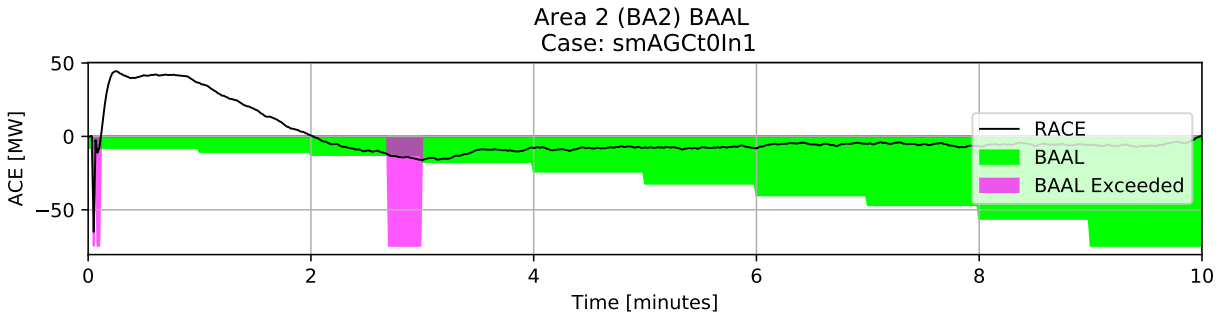

Figure 4.26: Area 2 BAAL during external area event using TLB 0.

4.2.3 AGC Result Summary

In general, simulations show governor deadbands may lead to frequency oscillation between response thresholds and conditional AGC can be used to avoid unnecessary and contradictory recovery action between areas. While BAAL was exceeded, AGC action resolved any excess before a violation occurred.

4.3 Long-Term Simulation with Shunt Control

Long-term simulations of interest required shunt control to manage voltage. Without voltage control, load changes eventually caused the power-flow solution to diverge. Scenarios chosen for simulation were a four hour morning peak and a two hour virtual wind ramp.

4.3.1 Morning Peak Forecast Demand Simulation

The same six machine two area system and tuned AGC controllers from the previous section were used for long-term simulations. Definite time controllers (DTCs) were used to switch shunts according to bus voltage. Publicly available EIA data was parsed and used to parameterize hourly forecast and demand agents. Data was selected from the morning peak on December 11, 2019 staring at 5:00 AM as reported by the Bonneville Power Administration and California ISO BAs. Normalized reported BA area power changes are shown in Figure 4.27. Simulated BA1 followed Bonneville data while BA2 adhered to California data. The selected forecast scenario was simulated under ideal conditions and then with the inclusion governor deadbands and load noise. Figure 4.28 shows the random noise added to area 2 caused load value to decrease much more than area 1. The .ltd.py file used for the forecast demand scenario with noise and deadbands is shown in Appendix C as Figure C.4.

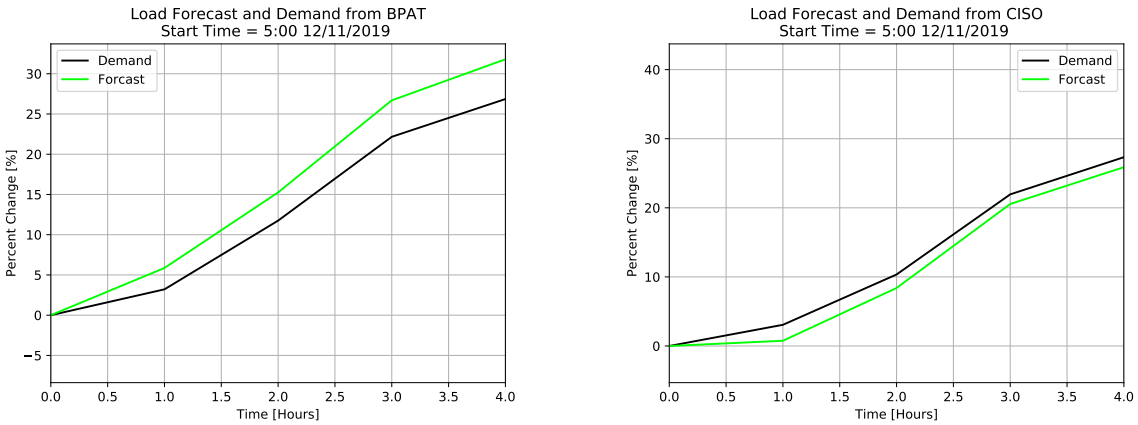

Figure 4.27: Normalized forecast and demand of parsed EIA data.

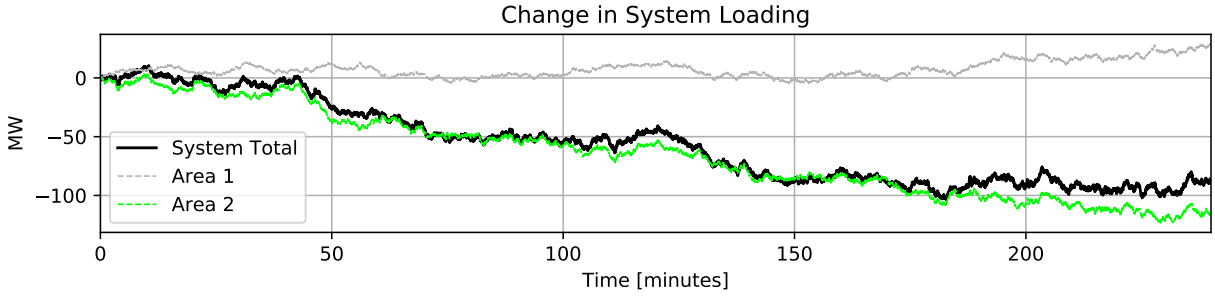

Figure 4.28: Changes in load caused by noise agent action during morning peak.

4.3.1.1 Morning Peak Forecast Demand Results

Figures 4.29 and 4.30 show the gradual increase of area real power load P, and generated electrical power P_e. Near constant differences between generation and load is maintained by AGC action throughout the entire simulation. The decreased load due to random noise in area 2 is somewhat apparent in Figure 4.30.

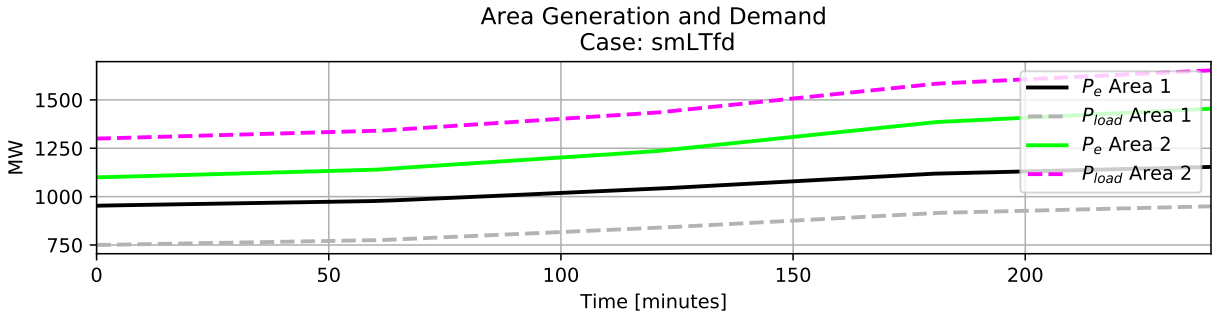

Figure 4.29: Morning peak area P_e and Load.

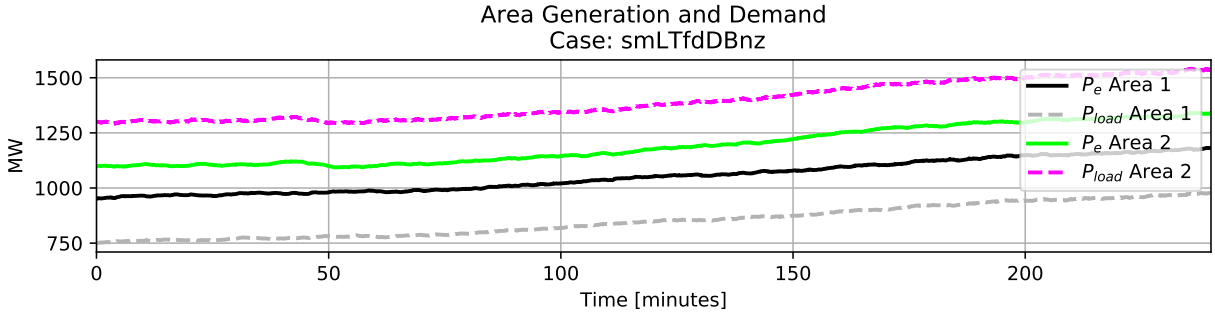

Figure 4.30: Morning peak area P_e and Load with noise and governor deadbands.

AGC action maintained system frequency near the nominal values. Figure 4.31 shows that system frequency response without the addition of deadbands or noise varied less than 1.5 mHz. Figure 4.32 shows that when deadbands and noise are included, frequency tended to oscillate between governor deadbands. A detail view of system frequency, shown in Figure 4.33, revealed the oscillation to have a rate of approximately 4 mHz.

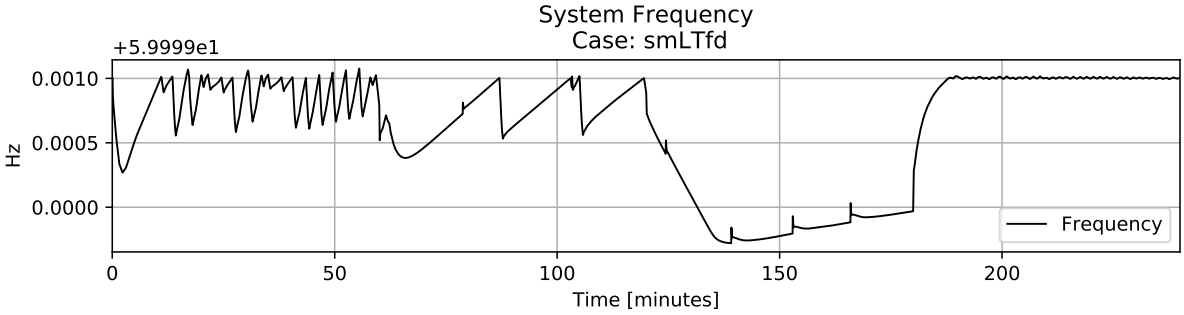

Figure 4.31: Morning peak system Frequency.

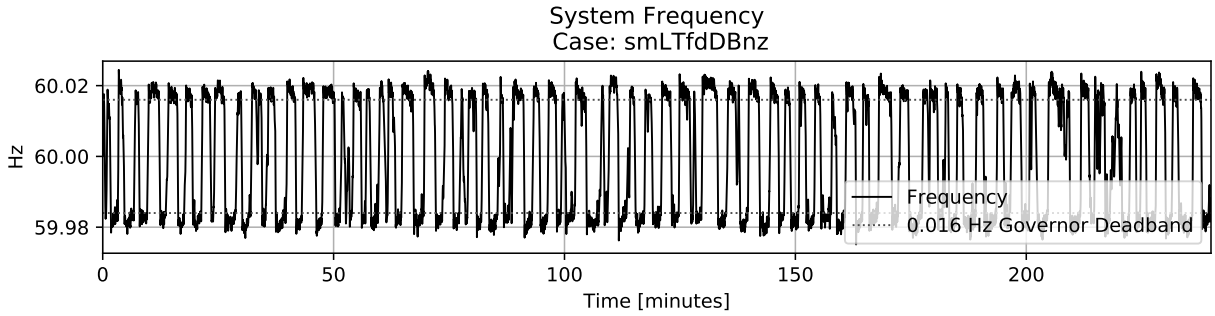

Figure 4.32: Morning peak system frequency with noise and governor deadbands.

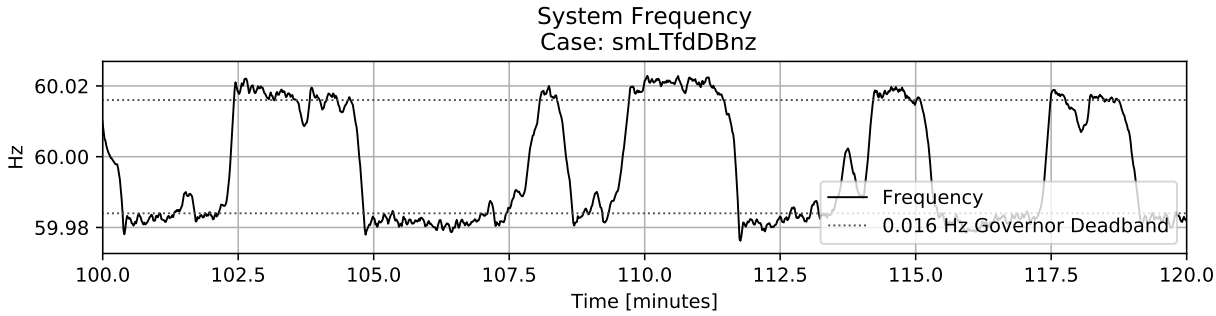

Figure 4.33: Detail morning peak system frequency with noise and governor deadbands.

Figure 4.34 shows calculated BA values during an ideal scenario never exceeded a magnitude of 1 MW. Calculated values when noise is included became more busy, but generally oscillated between ± 10 MW. Figure 4.36 shows a clearer correlation between RACE and system frequency.

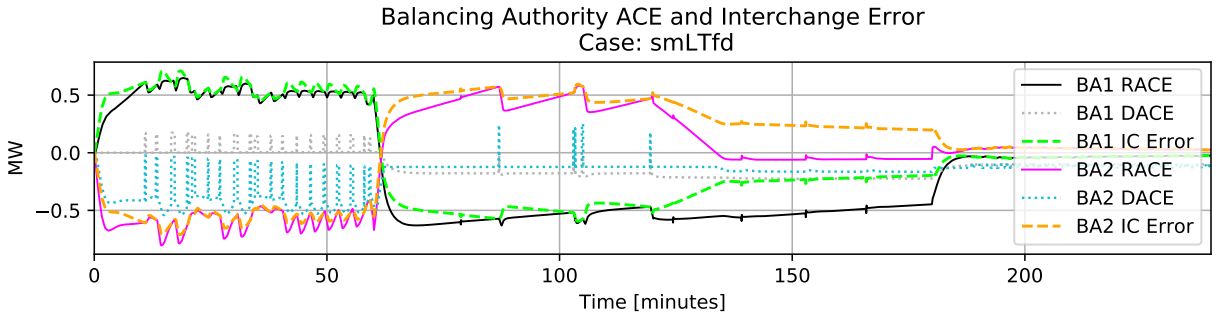

Figure 4.34: Morning peak calculated BA values.

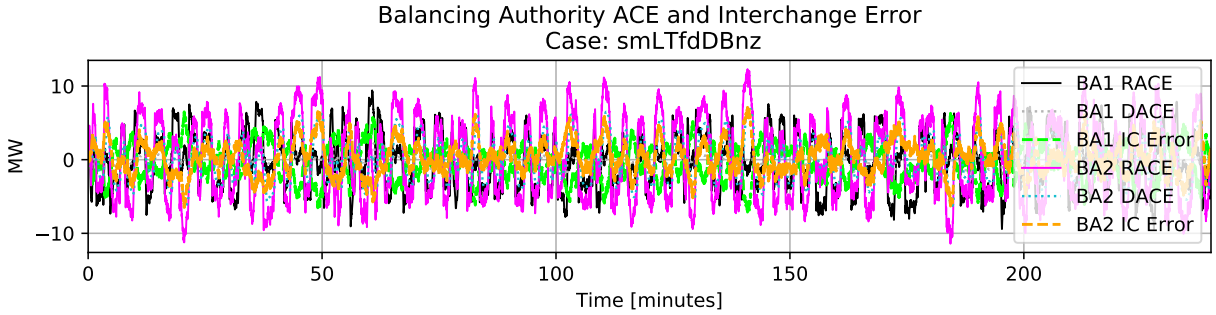

Figure 4.35: Morning peak calculated BA values with noise and governor deadbands.

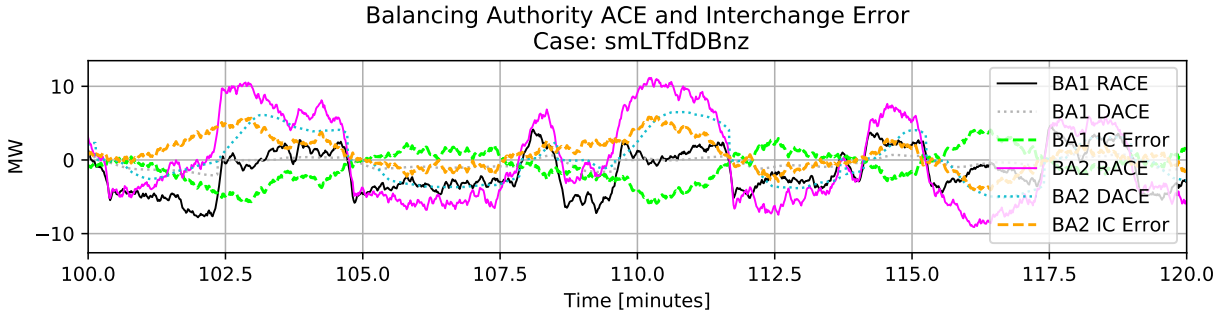

Figure 4.36: Detail morning peak calculated BA values with noise and governor deadbands.

Figures 4.37 and 4.38 show the BAAL from each scenario was never exceeded. Results from the BA in area 2 were similar and are presented in Appendix **??**. As BAAL gets very large when frequency is near its nominal value, plot y-axes were scaled to 1.25 minimum and maximum RACE values.

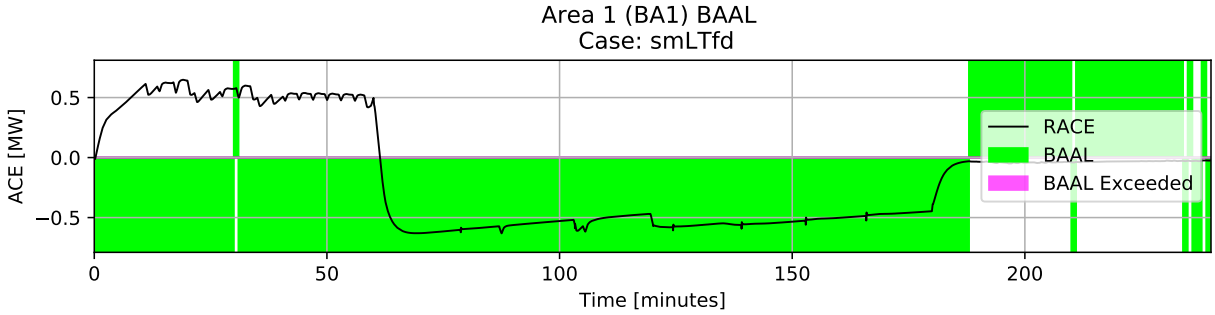

Figure 4.37: BAAL of area 1 during morning peak.

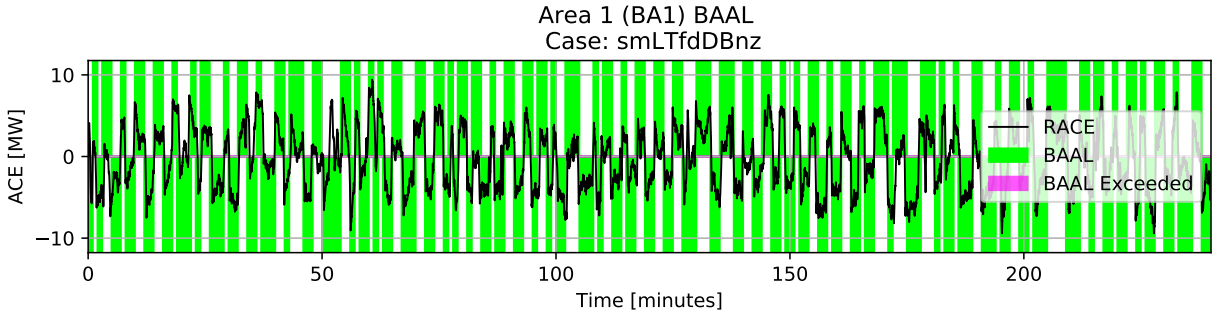

Figure 4.38: BAAL of area 2 during morning with noise and governor deadbands.

While AGC handles the balancing of generation to load, and thus frequency, voltage must be handled via DTC action. Figures 4.39 and 4.40 show shunt bus voltage under ideal conditions and with load noise and governor deadbands, respectively. As load increased, bus voltage declined. The steps in voltage were caused by capacitive shunts being switched into the system according to programmed DTC parameters. The random noise added generally reduced load, so shunt switching is slightly delayed when noise is applied compared to the ideal scenario.

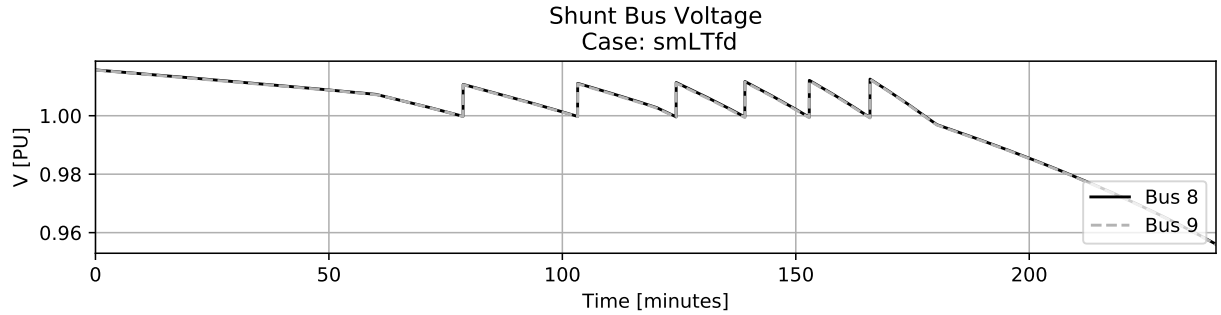

Figure 4.39: Morning peak system shunt bus voltage.

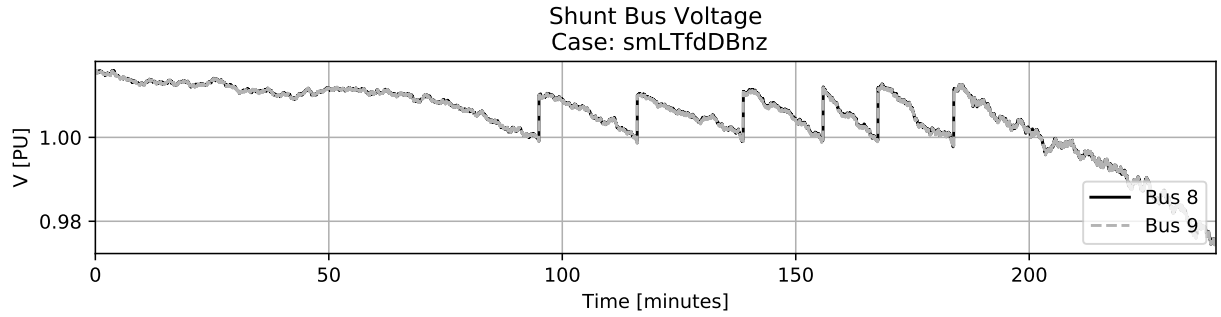

Figure 4.40: Morning peak system shunt bus voltage with noise and governor deadbands.

Figures 4.41 and 4.42 show the active capacitive load on system bus 8 and 9. DTC action switches capacitors on as voltage drops according to user input logic. Bus 8 has faster acting shunts and thus switches all available caps on before any on bus 9 can respond. Again, the random noise causes a delay in voltage drop and shunt switching as the random noise acts to generally reduce load.

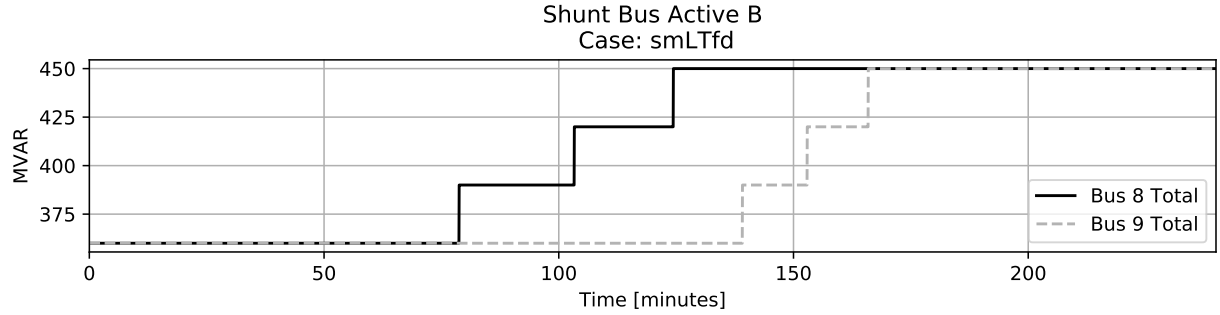

Figure 4.41: Morning peak system shunt bus MVAR.

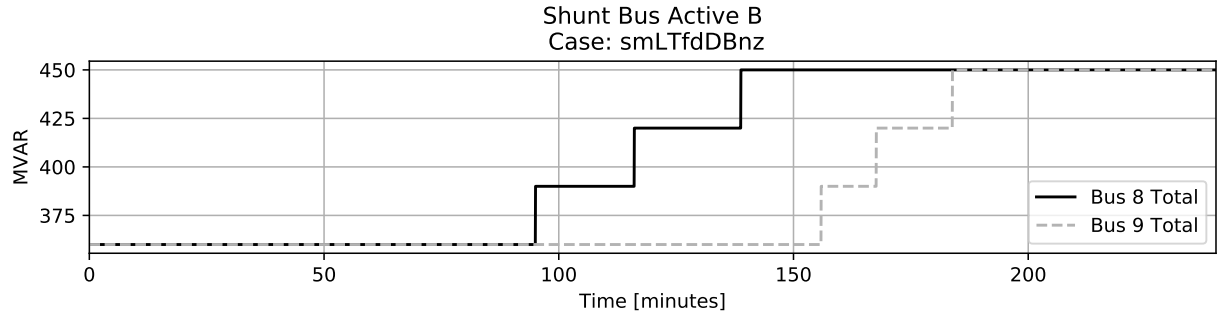

Figure 4.42: Morning peak system shunt bus MVAR with noise and governor deadbands.

4.3.2 Morning Peak Forecast Demand Result Summary

The tuned AGC routines adequately manage frequency and interchange during the event. DTC action functioned to manage bus voltage as desired. Declining voltages at the end of the simulation was due to the system not having any more shunts to switch on. Therefore, if this event were real, the addition of more switchable capacitive shunts might be suggested.

4.3.3 Virtual Wind Ramp Simulation

A virtual wind ramp similar to the one created in [32] was simulated using PSLTDSim. Unlike [32], conventional generator models were used instead of explicit wind turbine models. Two ungoverned generators were ramped up, held, and then ramped down. More specifically, using the six machine system, generator 2 2 in area 1 was altered 150 MW while generator 5 1 in area 2 was changed 300 MW. The generation ramps were 45 minute in duration. The first ramp began at $t = 300$, or 5 minutes into the simulation. A pause of 20 minutes is included between the end of the ramp up and beginning of ramp down. The wind ramp is finished by minute 115, but the simulation continued for another 20 minutes so system recovery could be observed. AGC settings for area 2 were modified from the forecast demand case to include generator 4 1 receiving 30.0% of AGC signals. This was required as AGC would force generator 3 1 to supply 0 MW otherwise. Random noise injections into the system loads shown in Figure 4.43 acted to generally reduce load in area 2.

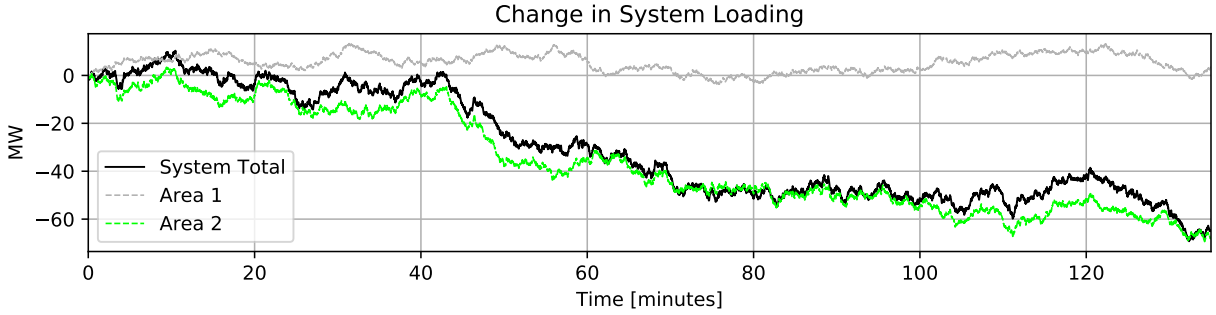

Figure 4.43: Changes in load caused by noise agent action during virtual wind ramp.

4.3.3.1 Virtual Wind Ramp Results

Figures 4.44 and 4.45 show the ramping up behavior of generators 2 2 and 5 1 as well as the AGC effect on controlled generators. General behavior observed with added load noise and governor deadbands was similar to the ideal case.

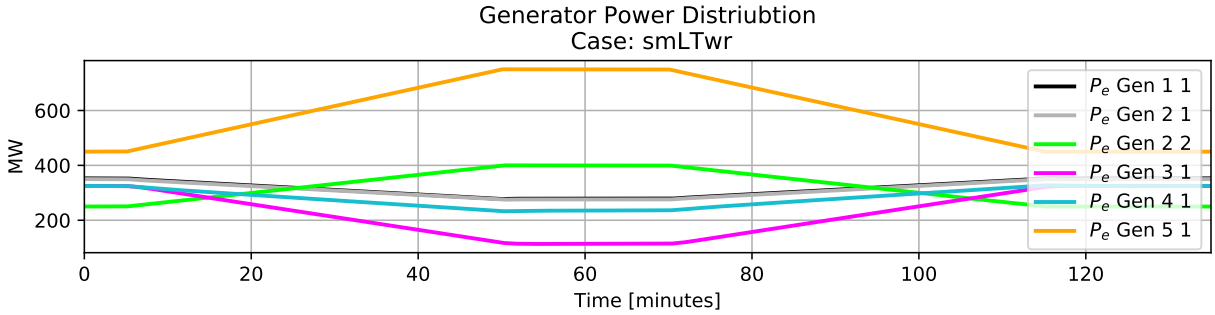

Figure 4.44: Virtual wind ramp P_e distribution under ideal conditions.

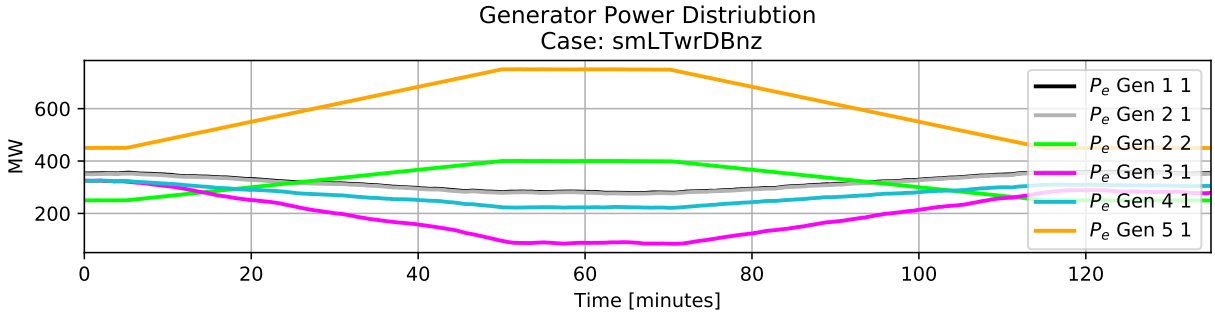

Figure 4.45: Virtual wind ramp P_e distribution with noise and governor deadbands.

Figures 4.46 and 4.47 show system frequency response to the wind ramp under ideal conditions and with governor deadbands and load noise included respectively. In both cases, AGC was not able to achieve nominal system frequency during ramp events. Because the ramps are 45 minutes, this maintained frequency deviation is in violation of interpreted NERC mandate BAL-002-3. When noise and deadbands were included, and no ramp event was taking place, system frequency oscillated between deadband thresholds.

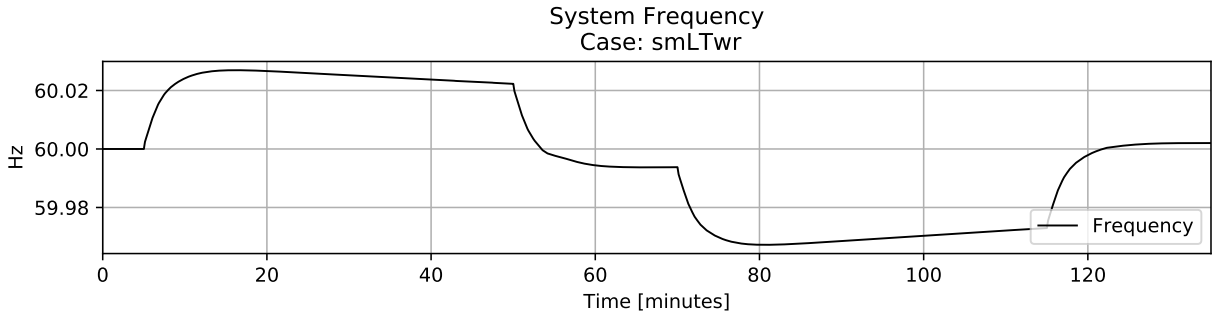

Figure 4.46: Virtual wind ramp system frequency under ideal conditions.

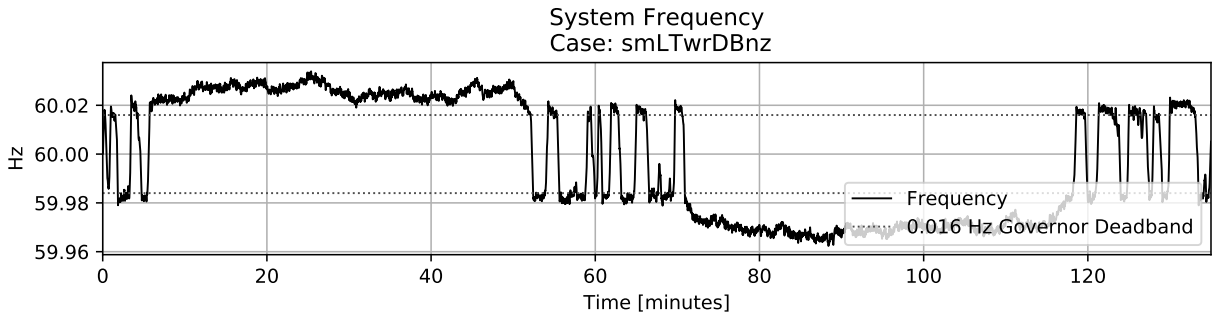

Figure 4.47: Virtual wind ramp system frequency with noise and governor deadbands.

Figure 4.48 shows calculated BA values during the virtual wind ramp under ideal conditions. It can be seen that DACE did not match RACE values during ramp events and that IC error was symmetric. Figure 4.49 shows that calculated BA values when noise and deadbands are included was slightly larger in magnitude than the ideal case. In both either case, maximum RACE does not exceed ± 15 MW.

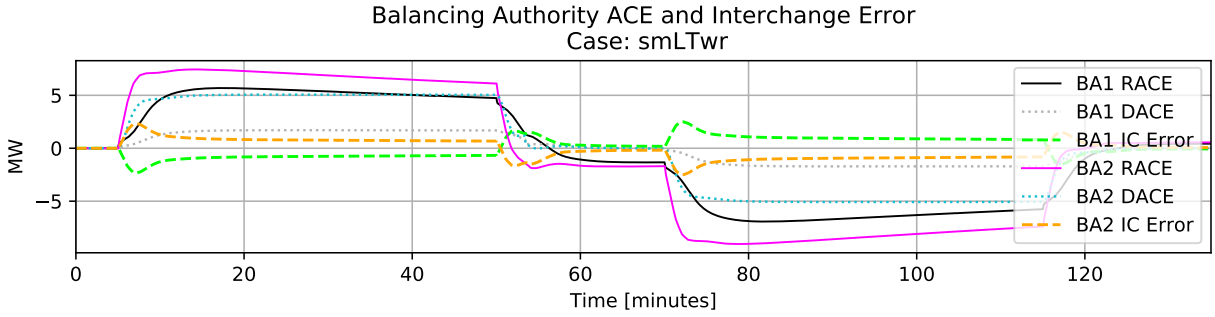

Figure 4.48: Virtual wind ramp calculated BA values under ideal conditions.

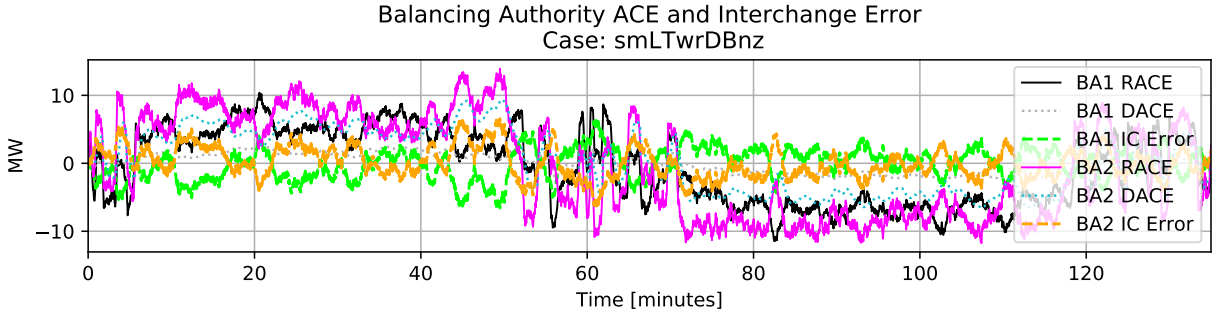

Figure 4.49: Virtual wind ramp calculated BA values with noise and governor deadbands.

Figure 4.50 shows that AGC did a good job of maintaining near constant generator output P_e and that load does not change during the ideal scenario. Figure 4.51 shows similar AGC action despite added noise decreasing area 2 loading.

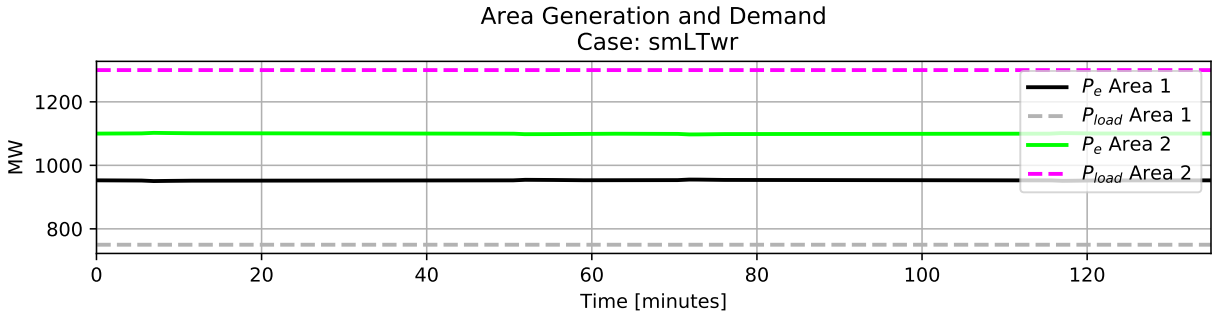

Figure 4.50: Virtual wind ramp area P_e and P_{load} under ideal conditions.

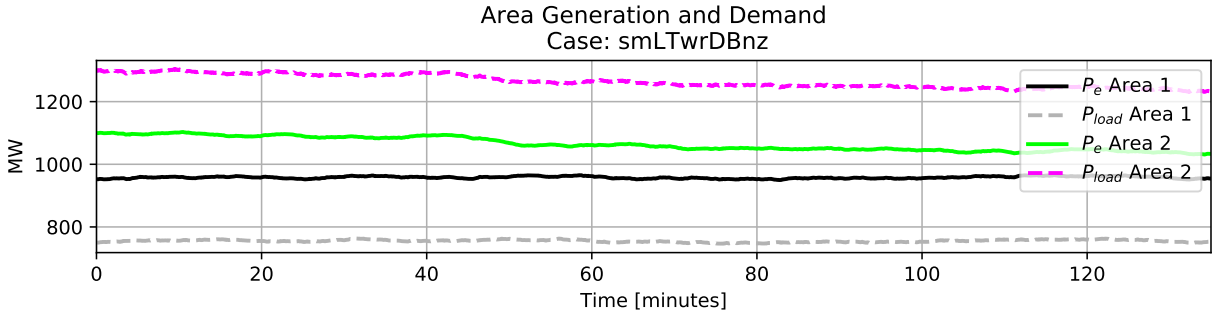

Figure 4.51: Virtual wind ramp area P_e and P_{load} with noise and governor deadbands.

Figures 4.52 and 4.52 show the BAAL from each scenario was never exceeded. Results from the BA in area 2 were to area 1 and are presented in Appendix **??**. Plot y-axes were again scaled to 1.25 minimum and maximum RACE values.

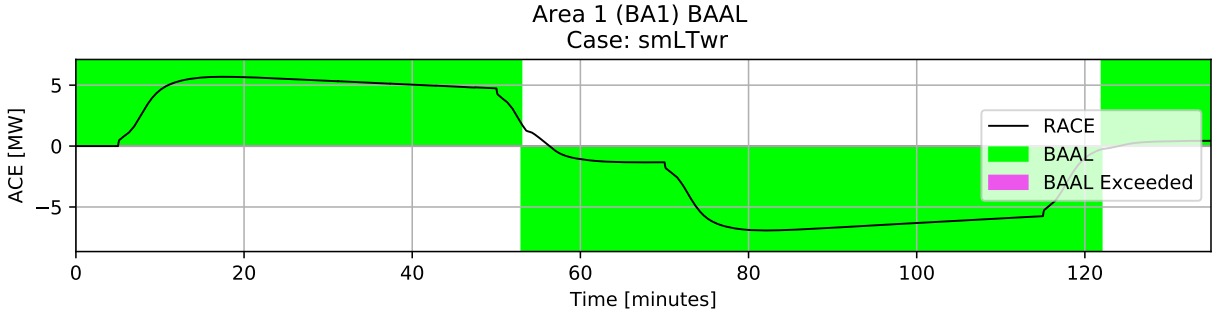

Figure 4.52: Virtual wind ramp BAAL under ideal conditions.

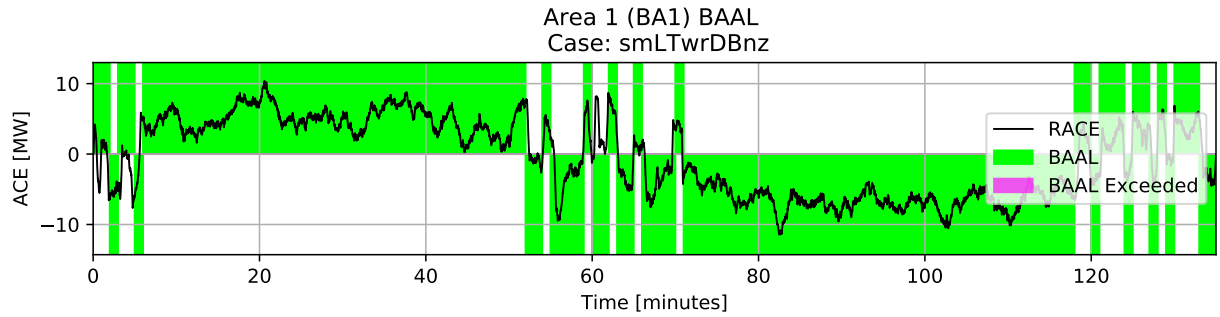

Figure 4.53: Virtual wind ramp BAAL with noise and governor deadbands.

Figures 4.54 and 4.55 show that bus voltage between the two scenarios was fairly similar. As the 'wind' ramped up, bus voltage dropped and switched shunts stepped on to increase voltage. When the wind ramped down, bus voltage increased and controlled shunts were switched off. The decreased load caused by random noise triggers an additional capacitor to be switched off near minute 115.

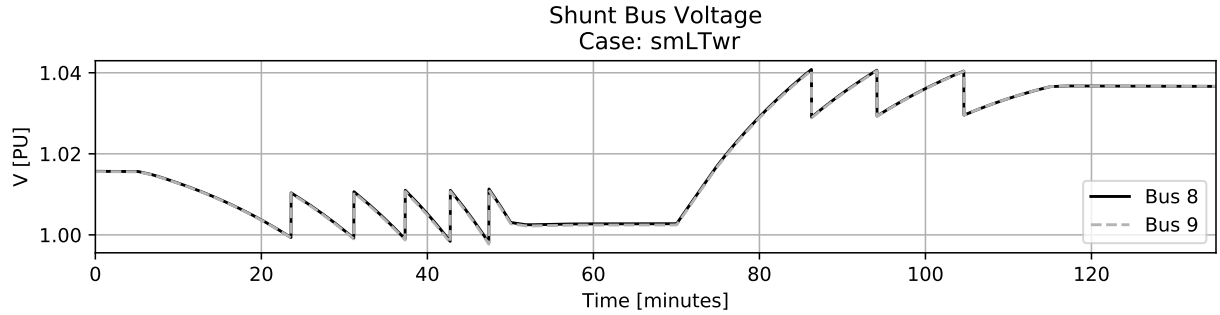

Figure 4.54: Virtual wind ramp shunt bus voltage under ideal conditions.

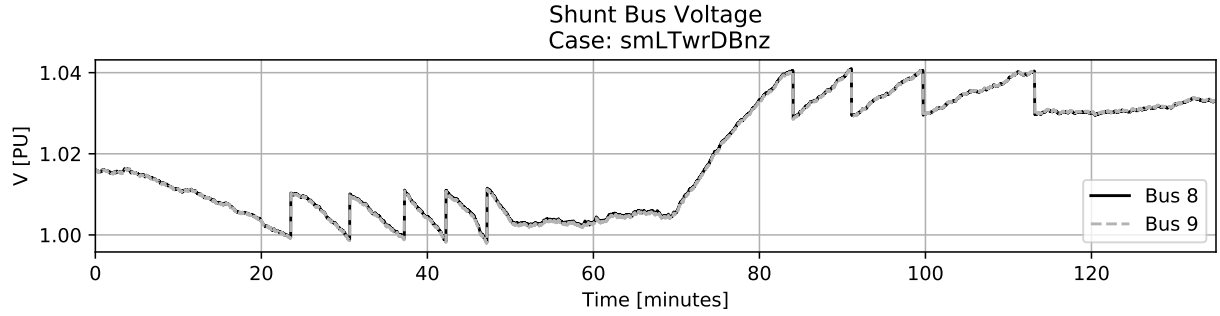

Figure 4.55: Virtual wind ramp shunt bus voltage with noise and governor deadbands.

Figure 4.56 shows the MVAR injections caused by switched shunts under ideal conditions. As bus 8 has a 'faster' DTC routine, all available capacitors are turned on before bus 9 shunts activated during the ramp up event. The same is true when the wind ramped down; which left bus 9 shunts on. Figure 4.57 shows that added noise created a lower voltage on bus 9 and enabled a bus 9 shunt to switch on before all bus 8 capacitors were deployed. The additional bus 8 cap removal near time 115 was caused by random noise, but is obscured by the plot legend.

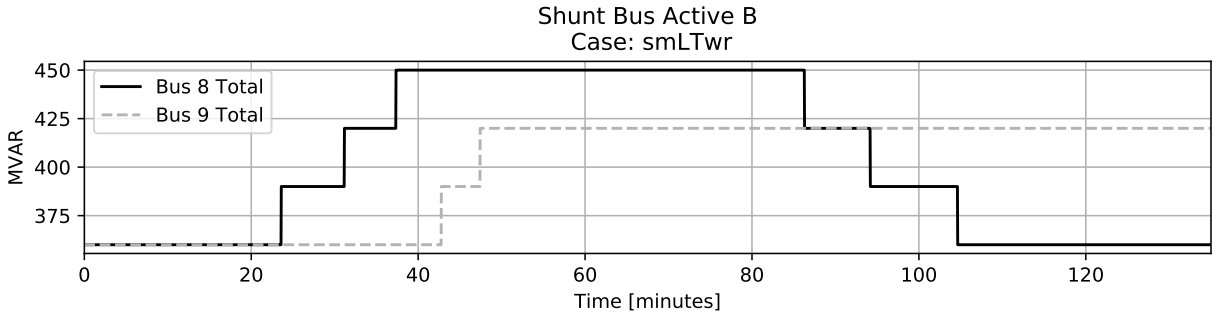

Figure 4.56: Virtual wind ramp shunt bus MVAR under ideal conditions.

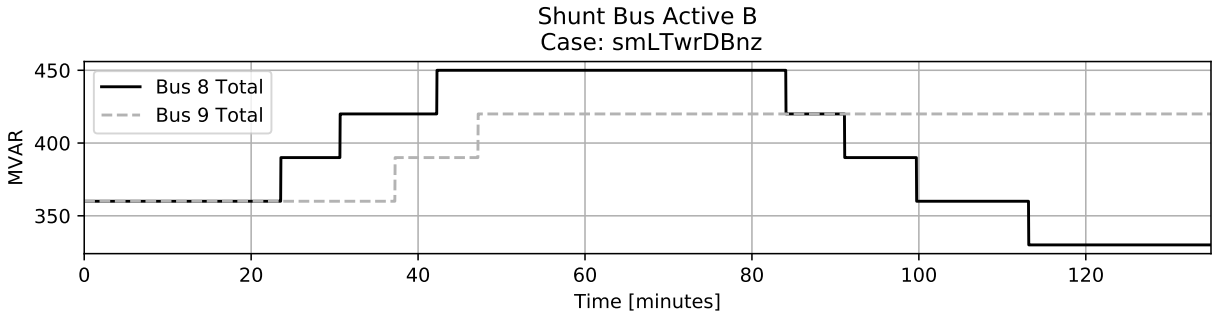

Figure 4.57: Virtual wind ramp shunt bus MVAR with noise and governor deadbands.

4.3.4 Long-Term Simulation with Shunt Control Result Summary

In general, configured AGC handled area interchange and BAAL well, but may require further tuning or additional control options to adequately automatically handle frequency during long ramps. Simulated governor deadbands introduced system frequency oscillation between deadband thresholds. Bus voltage reference signals were used to effectively manage switched shunt operation. Addition of random noise caused very slight differences that had noticeable effects on automatic system response.

4.4 Feed-Forward Governor Action

Interested parties expressed a concern with an undesirable governor response believed to be caused by feed-forward governor characteristics. The provided model of governors in question was described as a single block with no further information. Figure 4.58 is a plot of simulated and recorded generator power output that was provided as an example of undesirable behavior. A similar governor response was created using a DTC and governor input manipulation.

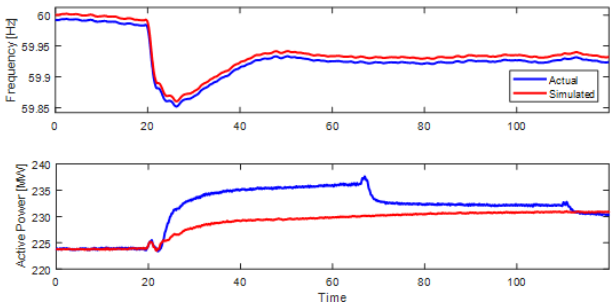

Figure 4.58: Provided information of undesired governor response.

4.4.1 Feed-Forward Governor Simulation Configuration

Using the six machine system with 0.5 second time step, a method of stepping governor P_{ref} while also gaining input $\Delta\omega_{PU}$ produced a similar undesirable response. Code used to define system generation step, governor delay, and DTC action is shown in Figure 4.59. In practice, this code would be user defined in the simulation .ltd.py file.

A block diagram showing what the DTC does is shown in Figure 4.60. The logic controlling SW1, which uses a modulo operator to act every 24 seconds, is

$$SW1 = \big((t\%24) == 0\big). \tag{4.1}$$

It should be noted that this logic, and the resulting setting of $Pref$, occurs first during dynamic computation.

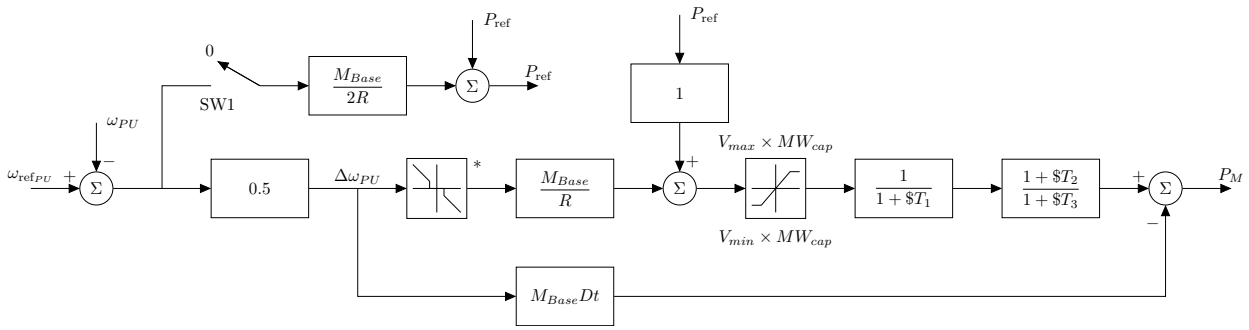

Figure 4.60: Block diagram of tgov1 model with DTC.

The perturbance list in Figure 4.59 specifies an ungoverned generator on bus 5 with a mechanical power step down of 100 MW at t=20. This was meant to simulate the tripping of a generator in an aggregated generator model. The governor delay dictionary was used to gain the $\Delta\omega$ input by 0.5. This was required so that steady state frequency did not over account for differences of $\Delta\omega$. The DTC action was defined to occur every 24 seconds and sets $P_{ref} = P_{ref0} + \frac{\Delta\omega}{R} M_{base} * 0.5$. While not truly a feed-forward control response, the stepping of P_{ref} was predicted to produce a similar result. Action time of 24 seconds was chosen so the first DTC response is near the simulated frequency nadir.

```
# Perturbances
mirror.sysPerturbances = [
    'gen 5 : step Pm 20 -100 rel', # Step no-gov generator down
    ]

# Delay block used as delta_w gain
mirror.govDelay ={
    'delaygen2' : {
        'genBus' : 2,
        'genId' : '1', # optional
        'wDelay' : (0, 0, .5), # gain of input w
        'PrefDelay' : (0, 0)
        },
    #end of defined governor delays
    }

# Definite Time Controller Definitions
mirror.DTCdict = {
    'ffGovTest' : {
        'RefAgents' : {
            'ra1' : 'mirror : f',
            'ra2' : 'gen 2 1 : R',
            'ra3' : 'gen 2 1 : Pref0',
            'ra4' : 'gen 2 1 : Mbase',
            },# end Referenc Agents
        'TarAgents' : {
            'tar1' : 'gen 2 1 : Pref',
            }, # end Target Agents
        'Timers' : {
            'set' :{ # set Pref
                'logic' : "(ra1 > 0)", # should always eval as true
                'actTime' : 24, # seconds of true logic before act
                'act' : "tar1 = ra3 + (1-ra1)/(ra2) * ra4 * 0.5 ", # step Pref
            },# end set
            'reset' :{ # not used in example
                'logic' : "0",
                'actTime' : 0, # seconds of true logic before act
                'act' : "0", # set any target On target = 0
            },# end reset
            'hold' : 0, # minimum time between actions (not used in example)
            }, # end timers
        },# end ffGovTest
    }# end DTCdict
```

Figure 4.59: Long-term dynamic settings for feed-forward governor simulation.

4.4.2 Feed-Forward Governor Simulation Results

Simulation results for frequency, and generator electrical power are shown in Figures 4.61 and 4.62 respectively. The undesired response can be seen when power output is stepped beyond the steady state value.

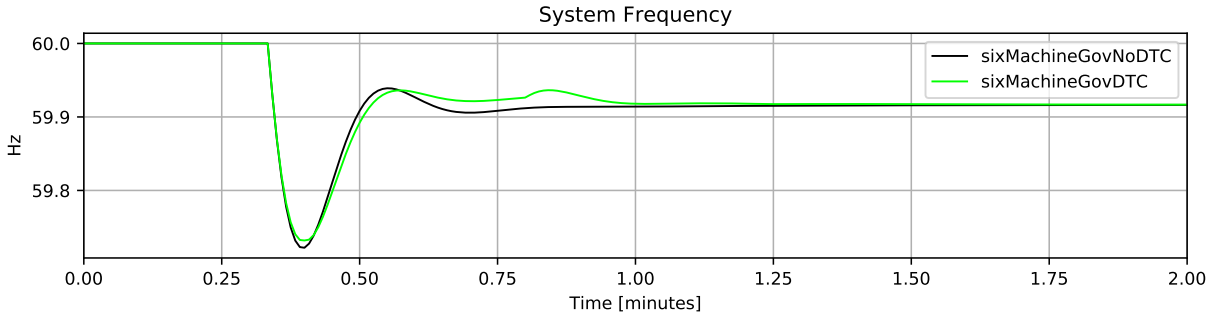

Figure 4.61: Feed-forward governor frequency comparison.

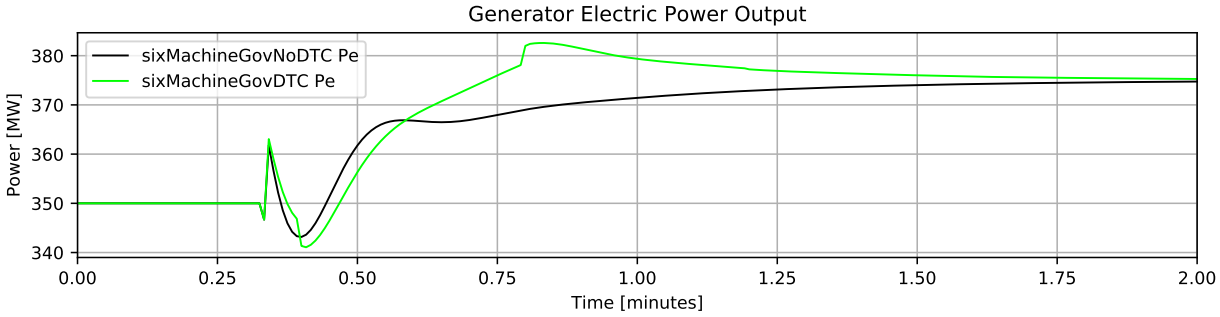

Figure 4.62: Feed-forward governor electric power comparison.

While the produced behavior does resemble observed behavior, it is not an exact match. Scaling and actual response time differences are believed to be caused by differences in model size, governor time constants, and actual feed-forward action. A new governor model with more defined feed-forward capabilities could be created if simulated results on actual system-under-study are unsatisfactory.

4.5 Variable System Damping and Inertia

PSLTDSim can be used to estimate system behavior. To adjust frequency response, system damping and inertia may be manipulated. The trend of increased inverter-based generation can lead to situations where total system inertia may not match what is expected. The single combined system inertia in PSLTDSim can be modified during simulations to enable study into variable inertia system responses.

4.5.1 Damping and Inertia Simulation Configuration

Damping effects were explored by modifying the "Dsys" value in the simParams dictionary of a simulation .py file. Modifcations of this value change the D_{sys} variable in Equation 2.3. System inertia effects were studied by altering the "Hinput" in simParams dictionary and by perturbance agent action which affect the H_{sys} value in Equation 2.3. A pulse train of load steps was created to present the ability to vary system inertia during a simulation. System inertia was altered before each positive load step. Code used to define perturbance steps is shown in Figure 4.63. In practice, this code would be user defined in the simulation .ltd.py file.

```
mirror.sysPerturbances = [
    # Initial system response
    'load 8 : step P 2 100 rel',
    'load 8 : step P 22 -100 rel',
    # decrease of H
    'mirror : step Hsys 40 -30 per',
    'load 8 : step P 42 100 rel',
    'load 8 : step P 62 -100 rel',
    # decrease of H
    'mirror : step Hsys 80 -50 per',
    'load 8 : step P 82 100 rel',
    'load 8 : step P 102 -100 rel',
    # Increase of H
    'mirror : step Hsys 120 30080 abs',
    'load 8 : step P 122 100 rel',
    'load 8 : step P 142 -100 rel',
]
```

Figure 4.63: Long-term dynamic settings for variable system inertia simulation.

The code in Figure 4.63 steps the load on bus 8 up and down 100 MW every 20 seconds starting at t=2. System inertia is reduced by 30% at t=40, 50% at t=80, and then set to 30,080 MW s at t=120.

4.5.2 Damping and Inertia Simulation Results

System damping effects to a -100 MW generator step are shown in Figure 4.64. A positive damping value increased system oscillation while a negative damping value decreased oscillations. Altering system damping affected steady state frequency response.

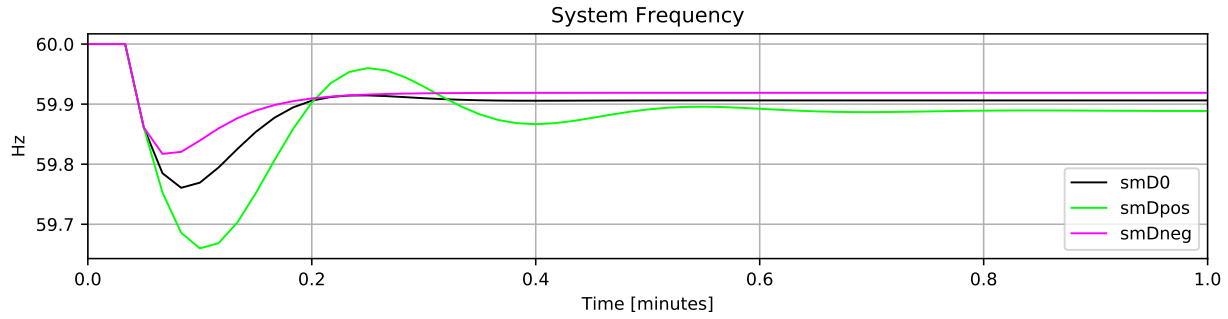

Figure 4.64: Frequency effects of system damping.

Frequency response to scalings of system inertia are shown in Figure 4.65. The smH100 case was an un-modified system inertia, while the 90, 80 and 70 case appendings reflected a 90.0%, 80.0%, and 70.0% inertia scaling respectively. A scaling of 60.0% resulted with a power-flow problem that did not converge. As system inertia was decreased, frequency response became faster and frequency nadir increased.

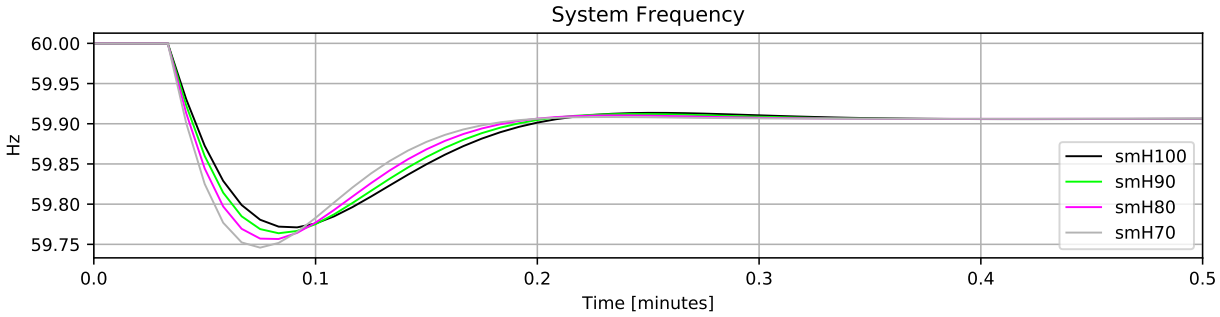

Figure 4.65: Frequency effects of system inertia.

A single machine's valve response to inertia scalings are shown in Figure 4.66. Results resemble a reflected frequency response where lower inertia cases have larger and faster valve travel.

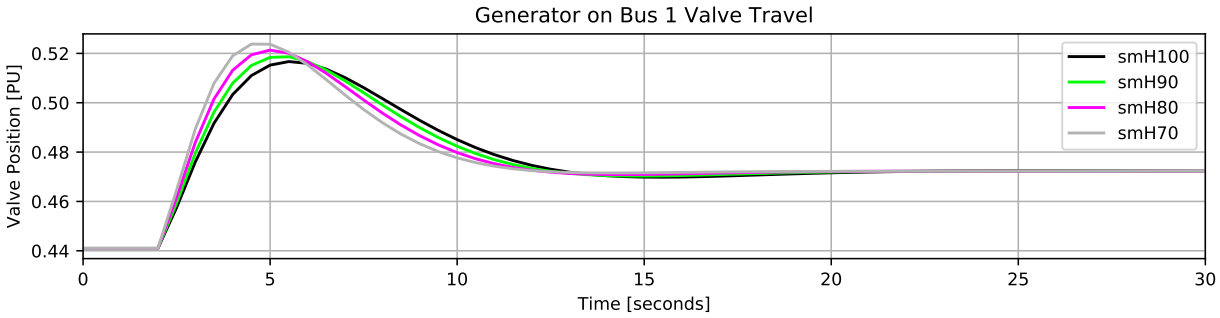

Figure 4.66: Governor response to varied system inertia.

Figure 4.67 shows the changes in system inertia caused by perturbance actions shown in Figure 4.63. Figure 4.68 shows system frequency response to a pulse train of load steps. Again, lower system inertia leads to faster frequency changes and larger frequency nadirs.

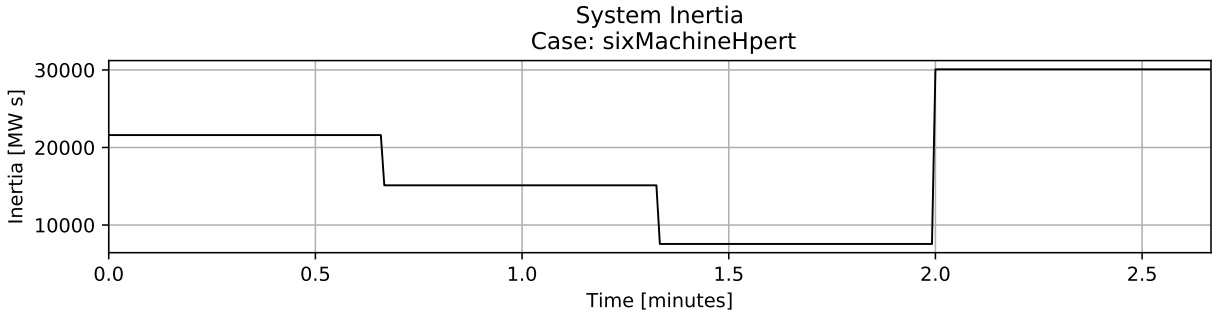

Figure 4.67: Varied system inertia during simulation.

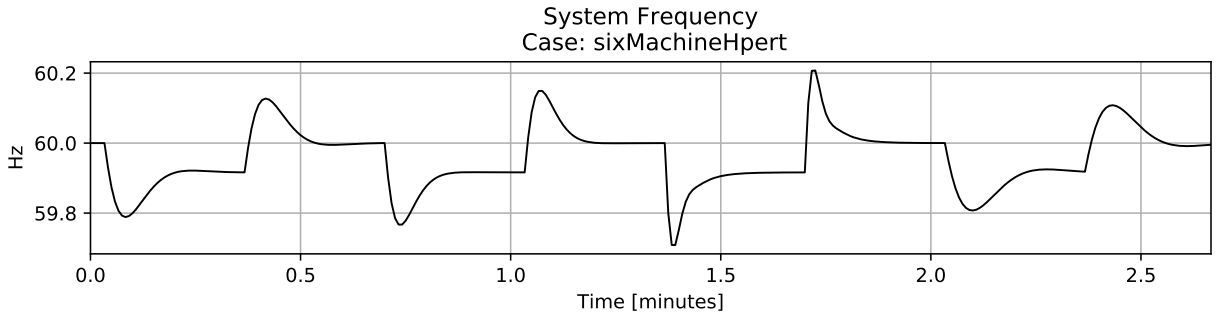

Figure 4.68: System frequency response to varying system inertia.

Damping altered system frequency dynamic response and affected steady state frequency values. System inertia may be changed before or during a simulation. Noticeable effects of varying inertia were seen during step type contingencies. PSLTDSim is not particularly well suited for large steps, however, results showed that changes in system inertia during simulation were accounted for.

5 Future Work

As with any software project, future work revolves around expansion and refinement. The number of dynamic agents, or governor models, could be expanded without bound. Alternatively, a more refined way of casting un-modeled governors could be devised. A process involving automated one machine infinite bus scenarios, step analysis, and 2nd order approximation has been suggested.

Exponential load models and under-load transformer tap changers are things that should be accounted for. Power plant agents that act as plant controllers have been conceptually modeled, but not implemented to their full extent. To better capture voltage changes, and thus reactive power, some form of exciter modeling may be desired, although with the simplification of machine models, this task may prove difficult. Voltage scheduling of generator buses via perturbance agents should be possible, but is untested as of this writing.

PSLTDSim is open-source code that relies on proprietary software for essential functions. To move away from this reliance, a method for creating system models and solving power flows should be created. Any changes would have to be incorporated into the way PSLTDSim creates a system mirror, solves a power flow, and updates the mirror. A semi-clear point to break from PSLF would be when AMQP messages are sent. If PSLTDSim did not rely on PSLF, there would also be no need for AMQP messages to be sent as all code would be PY3. This would not only enable fully open-source simulation, but speed up simulation time and create a more straight forward code flow.

To accommodate for transient, or oscillatory events, PSLTDSim could be coupled with the ideas presented in [69]. This would involve a variable time step and some way to automatically switch between TSPF and CTS simulation.

6 Bibliography

[1] .NET Foundation. (2018). Ironpython overview, [Online]. Available: https://ironpython.net/.

[2] P. M. Anderson and A. A. Fouad, *Power System Control and Stability*, Second Edition. Wiley-Interscience, 2003.

[3] J. Audenaert, K. Verbeeck, and G. V. Berghe. (2009). Mulit-agent based simulation for boarding, CODeS Research Group, [Online]. Available: https://www.semanticscholar.org/paper/Multi-Agent-Based-Simulation-for-Boarding-Audenaert-Verbeeck/24ba2c3190de5b7162c37e81581b062cda3e4d54.

[4] A. Aziz, A. Mto, and A. Stojsevski, "Automatic generation control of multigeneration power system," Journal of Power and Energy Engineering, 2014.

[5] W. E. Boyce and R. C. DiPrima, *Elementary Differential Equations and Boundary Value Problems*, 10th ed. Wiley Custom Learning Solutions, 2014.

[6] W. Briggs, L. Chochran, and B. Gillett, *Calculus Early Transcendentals*. Pearson Education, Inc., 2011.

[7] J. Carpentier, "'To be or not to be modern' that is the question for automatic generation control (point of view of a utility engineer)," International Journal of Electrical Power & Energy Systems, 1985.

[8] R. W. Cummings, W. Herbsleb, and S. Niemeyer. (2010). Generator governor and information settings webinar, North American Electric Reliability Corporation, [Online]. Available: https://www.nerc.com/files/gen-governor-info-093010.pdf.

[9] F. P. deMello and R. Mills, "Automatic generation control part II - digital control techniques," IEEE PES Summer Meeting, 1972.

[10] DEQ. (2004). Montana electric transmission grid: Operation, congestion, and issues, DEQ, [Online]. Available: https://leg.mt.gov/content/publications/Environmental/2004deq_energy_report/transmission.pdf.

[11] EIA. (2016). U.s. electric system is made up of interconnections and balancing authorities, [Online]. Available: https://www.eia.gov/todayinenergy/detail.php?id=27152.

[12] EIA. (2019). July2019map.png, U.S. Energy Information Administration, [Online]. Available: https://www.eia.gov/electricity/data/eia860m/.

[13] EIA. (2019). U.s. electric system operating data, U.S. Energy Information Administration, [Online]. Available: https://www.eia.gov/realtime_grid/.

[14] EIA. (2019). U.s. energy mapping system, U.S. Energy Information Administration, [Online]. Available: https://www.eia.gov/state/maps.php?v=Electricity.

[15] E. Ela and J. Kemper, "Wind plant ramping behavior," National Renewable Energy Laboratory, 2009.

[16] M. D. of Environmental Quality. (2020). Colstrip statistics, DEQ, [Online]. Available: http://deq.mt.gov/DEQAdmin/mfs/AllColstrip/DEQAdmin/mfs.

[17] ERCOT. (2017). Ercot-internconnection_branded.jpg, ERCOT, [Online]. Available: http://www.ercot.com/news/mediakit/maps.

[18] J. H. Eto, J. Undrill, C. Roberts, P. Mackin, and J. Ellis, "Frequency control requirements for reliable interconnection frequency response," Energy Analysis and environmental Impacts Division Lawrence Berkeley National Laboratory, 2018.

[19] D. Fabozzi and T. Van Cutsem, "Simplified time-domain simulation of detailed long-term dynamic models," IEEE Xplore, 2009.

[20] FERC, "Essential reliability services and the evolving bulk-power system–primary frequency response," Federal Energy Regulatory Commission, Docket No. RM16-6-000 Order No. 842, Feb. 2018.

[21] FERC. (2019). About ferc, [Online]. Available: https://www.ferc.gov/about/about.asp.

[22] T. Garnock-Jones and G. M. Roy. (2017). Introduction to pika, [Online]. Available: https://pika.readthedocs.io/en/stable/.

[23] GE Energy, *Mechanics of running pslf dynamics*, 2015.

[24] GeeksforGeeks. (2017). Thread in operating system, GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/thread-in-operating-system/.

[25] General Electric. (2020). Ge pslf main page, [Online]. Available: https://www.geenergyconsulting.com/practice-area/software-products/pslf.

[26] General Electric International, Inc, *PSLF User's Manual*, 2016.

[27] W. B. Gish, "Automatic generation control algorithm - general concepts and application to the watertown energy control center," Bureau of Reclamation Engineering and Research Center, 1980.

[28] J. D. Glover, M. S. Sarma, and T. J. Overbye, *Power System Analysis & Design*, 5e. Cengage Learning, 2012.

[29] R. Gonzales. (2019). Pg&e transmission lines caused california's deadliest wildfire, state officials say, NPR, [Online]. Available: https://www.npr.org/2019/05/15/723753237/pg-e-transmission-lines-caused-californias-deadliest-wildfire-state-officials-sa.

[30] M. Goossens, F. Mittelbach, and A. Samarin, *The LaTeX Companion*. Addison-Wesley, 1993.

[31] R. Hallett, "Improving a transient stability control scheme with wide-area synchrophasors and the microwecc, a reduced-order model of the western interconnect," Master's thesis, Montana Tech, 2018.

[32] E. Heredia, D. Kosterev, and M. Donnelly, "Wind hub reactive resource coordination and voltage control study by sequence power flow," IEEE, 2013.

[33] J. JMesserly. (2008). Electricity_grid_simple-_north_america.svg, United States Department of Energy, [Online]. Available: https://commons.wikimedia.org/wiki/File:Electricity_grid_simple-_North_America.svg.

[34] T. Kennedy, S. M. Hoyt, and C. F. Abell, "Variable, non-linear tie-line frequency bias for interconnected systems control," IEEE Transactions on Power Systems, 1988.

[35] Y. G. Kim, H. Song, and B. Lee, "Governor-response power flow (grpf) based long-term voltage stability simulation," IEEE T&D Asia, 2009.

[36] G. Kou, P. Markham, S. Hadley, T. King, and Y. Liu, "Impact of governor deadband on frequency response of u.s. eastern interconnection," IEEE Transactions on Smart Grid, 2016.

[37] D. Kuhlman, *A Python Book: Beginning Python, Advanced Python, and Python Exercises*. 2009.

[38] P. Kundur, *Power System Stability and Control*. McGraw-Hill, 1994.

[39] K. H. LaCommare and J. H. Eto, "Understanding the cost of power interruptions to u.s. electricity consumers," Ernest Orlando Lawrence Berkeley National Laboratory, 2004.

[40] M. Liedtke. (2020). Court approves pg&e's $23b bankruptcy financing package, AP News, [Online]. Available: https://apnews.com/b70582ee8d4bb7f781553215612da993.

[41] P. Maloney. (2018). What is the value of electric reliability for your operation? [Online]. Available: https://microgridknowledge.com/power-outage-costs-electric-reliability/.

[42] Y. Mobarak, "Effects of the droop speed governor and automatic generation control agc on generator load sharing of power system," International Journal of Applied Power Engineering, 2015.

[43] NERC, "Frequency response initiative report," North American Electric Reliability Corporation, 2012.

[44] NERC, "Procedure for ero support of frequency response and frequency bias setting standard," North American Electric Reliability Corporation, 2012.

[45] NERC, "Standard bal-003-1.1 — frequency response and frequency bias setting," North American Electric Reliability Corporation, 2015.

[46] NERC, "Standard bal-001-2 – real power balancing control performance," North American Electric Reliability Corporation, 2016.

[47] NERC. (2017). About nerc, [Online]. Available: https://www.nerc.com/AboutNERC/Pages/default.aspx.

[48] NERC. (2017). Nerc interconnections map, [Online]. Available: https://www.nerc.com/AboutNERC/keyplayers/PublishingImages/NERC%20Interconnections.pdf.

[49] NERC, "Bal-002-3 – disturbance control standard – contingency reserve for recovery from a balancing contingency event," North American Electric Reliability Corporation, 2018.

[50] NERC, "Frequency response annual analysis," North American Electric Reliability Corporation, 2018.

[51] NERC, "Reliability guideline application guide for modeling turbine-governor and active power-frequency controls in interconnection-wide stability studies," 2019.

[52] NERC, "Reliability guideline primary frequency control," 2019.

[53] NERC. (2020). Glossary of terms used in nerc reliability standards, NERC, [Online]. Available: https://www.nerc.com/files/glossary_of_terms.pdf.

[54] NERC Resources Subcommittee, "Balancing and frequency control," North American Electric Reliability Corporation, 2011.

[55] NERC Resources Subcommittee, "Bal-001-tre-1 — primary frequency response in the ercot region," North American Electric Reliability Corporation, 2016.

[56] J. W. Nilsson and S. A. Riedel, *Electric Circuits*, Ninth. Pearson Education, Inc., 2011.

[57] P. W. Parfomak, "Physical security of the u.s. power grid: High-voltage transformer substations," Congressional Research Service, 2014.

[58] PowerWorld Corporation. (2020). Powerworld main page, [Online]. Available: `https://www.powerworld.com/`.

[59] Python Software Foundataion. (2019). About python, [Online]. Available: `https://www.python.org/about/`.

[60] B. Rand. (2018). Agent-based modeling: What is agent-based modeling? Youtube, [Online]. Available: `https://www.youtube.com/watch?v=FVmQbfsOkGc`.

[61] C. W. Ross, "Error adaptive control computer for interconnected power systems," IEEE Transactions on Power Apparatus and Systems, 1966.

[62] G. van Rossum. (2009). A brief timeline of python, [Online]. Available: `https://python-history.blogspot.com/2009/01/brief-timeline-of-python.html`.

[63] RTDS Technologies. (2020). Rscad main page, [Online]. Available: `https://legacy.rtds.com/the-simulator/our-software/about-rscad/`.

[64] J. Sanchez-Gasca, M. Donnelly, R. Concepcion, A. Ellis, and R. Elliott, "Dynamic simulation over long time periods with 100% solar generation," Sandia National Laboratories, SAND2015-11084R, 2015.

[65] P. W. Sauer, M. A. Pai, and J. H. Chow, *Power System Dynamics and Stability With Synchrophasor Measurement and Power System Toolbox*, Second Edition. John Wiley & Sons Ltd, 2018.

[66] SciPy developers. (2019). About scipy, [Online]. Available: `https://www.scipy.org/about.html`.

[67] K. Siegel. (2012). The true cost of power outages, Yale Environment Review, [Online]. Available: `https://environment-review.yale.edu/true-cost-power-outages-0`.

[68] Siemens AG. (2018). Siemens main page, [Online]. Available: `https://pss-store.siemens.com/store/sipti/en_US/home`.

[69] M. Stajcar, "Power system simulation using an adaptive modeling framework," Master's thesis, Montana Tech, 2016.

[70] C. W. Taylor and R. L. Cresap, "Real-time power system simulation for automatic generation control," IEEE Transactions on Power Apparatus and Systems, 1976.

[71] The SciPy community. (2019). Scipy scipy.integrate.solve_ivp page, [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html.

[72] The SciPy community. (2019). Scipy scipy.signal.lsim page, [Online]. Available: https://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.signal.lsim.html.

[73] D. Trudnowski, "Properties of the dominant inter-area modes in the wecc interconnect," Montana Tech, 2012.

[74] T. Van Cutsem and C. Vournas, *Voltage Stability of Electric Power Systems*, 1st ed. Springer US, 1998.

[75] WECC. (2015). About wecc, [Online]. Available: https://www.wecc.org/Pages/AboutWECC.aspx.

A Numerical Methods

PSLTDSim utilizes a variety of numerical methods to perform integration. Some of the methods are coded 'by hand', while others are included in Python packages. This appendix is meant to introduce some numerical integration techniques, provide basic information about two Python functions used to perform numerical integration, compare results of numerical methods to exact solutions via examples, and briefly explain how some dynamic agents utilize the explained techniques.

A.1 Integration Methods

The options included in PSLTDSim to solve the combined swing equation for a new system frequency are Euler, Adams-Bashforth, and Runge-Kutta. Each of these methods are numerical approximations that provide an *approximation* to the solution of an initial value problem. Method equations presented below were adapted from [5].

A.1.1 Euler Method

Of the integration methods available, the Euler method is the simplest. In general terms, the next y value associated with a given differential function $f(t, y)$ is

$$y_{n+1} = y_n + f(t_n, y_n)t_s \tag{A.1}$$

where t_s is desired time step. The y_{n+1} solution is simply a projection along a line tangent to $f(t_n, y_n)$. It should be noted that the accuracy of this approximation method, and others described, is often related to the time step size, or the distance between approximations.

A.1.2 Runge-Kutta Method

Improving on the Euler method, the Runge-Kutta method combines numerous projections through a weighted average to approximate the next y value. The fourth order four-stage Runge-Kutta method is defined as Equation Block A.2.

$$\begin{aligned} k_1 &= f(t_n,\ y_n) \\ k_2 &= f(t_n + t_s/2,\ y_n + t_s k_1/2) \\ k_3 &= f(t_n + t_s/2,\ y_n + t_s k_2/2) \\ k_4 &= f(t_n + t_s,\ y_n + t_s k_3) \\ y_{n+1} &= y_n + t_s(k_1 + 2k_2 + 2k_3 + k_4)/6 \end{aligned} \tag{A.2}$$

It can be seen that k_1 and k_4 are solutions on either side of the interval of approximation defined by the time step t_s, and that k_2 and k_3 represent midpoint estimations.

A.1.3 Adams-Bashforth Method

Unlike previously introduced methods, the Adams-Bashforth method requires data from previous solution steps. Methods of this nature are sometimes referred to as multistep methods. A two-step Adams-Bashforth method is described in Equation A.3, however, larger step methods do exist.

$$y_{n+1} = y_n + t_s\left(1.5f(t_n, y_n) - 0.5f(t_{n-1}, y_{n-1})\right) \tag{A.3}$$

Regardless of the number of steps, Adams-Bashforth methods utilize a weighted combination of values similar to the Runge-Kutta method, but using only previously known values instead of projected future values.

A.1.4 Trapezoidal Integration

To integrate known values generated each time step, PSLTDSim uses a trapezoidal integration method. Given some value $y(t)$, the trapezoidal method states that

$$\int_{t-t_s}^{t} y(t)\ \mathrm{d}t \approx t_s \left(y(t) + y(t-ts)\right)/2, \tag{A.4}$$

where t_s is the time step used between calculated values of y. Visually, this method can be thought of connecting the two y values with a straight line, and then calculating the area of the trapezoid formed between them. As with previously described methods, the accuracy of this method depends on step size.

A.2 Python Functions

To allow for more robust solution methods, two Python functions were incorporated into PSLTDSim. The two functions are from the Scipy package for scientific computing. General information about these two functions is presented in this section.

A.2.1 scipy.integrate.solve_ivp

The Scipy solve_ipv function is capable of numerically integrating ordinary differential equations with initial values using a variety of techniques. A generic call to the function is shown in Figure A.1. Required inputs include a multi-variable function of x and y (i.e. some $f(x, y)$), a tuple describing the range of integration, and an initial value list. The output is an object with various collections of time points, solution points, and other information about the returned solution.

```
soln = scipy.integrate.solve_ivp(fp, (t0, t1), [initVal])
```

Figure A.1: Generic call to solve_ivp.

The default integration method used by solve_ivp is an explicit Runge-Kutta of order 5(4). This method is similar to the previously discussed 4th order Runge-Kutta, but with an

additional estimation factor. The four in parenthesis describes an approximation generated by a 4th order method which is used to calculate an error term between the 5th order solution and adjust the approximation time step accordingly. The exact execution of this process may be studied in the source code of the function itself. Other possible integration methods and function usage suggestions are described in [71].

A.2.2 scipy.signal.lsim

The Scipy function that simulates the output from a continuous-time linear system is called lsim. A general call to lsim is shown in Figure A.2. The inputs include an 'lti' system, an input vector, a time vector, and an initial state vector.

```
tout, y, x = scipy.signal.lsim(system, [U,U], [t0,t1], initialStates)
```

Figure A.2: Generic call to lsim.

Accepted lti systems passed into lsim may be transfer functions or state space systems created by the Scipy signal package. Function output includes a simulated time vector, system output, and state history. The computations performed by lsim utilize a state space solution centered around a matrix exponential that solves a system of first order differential equations. More complete information about the usage of lsim may be found in its source code or in [72].

A.3 Method Comparisons via Python Code Examples

Approximations from each method or function described above were compared to an exact solution by way of a Python script. This section includes full code from each test case, equations required to solve integrals exactly, and simulation results. Due to the lack of an accepted code listing format for this document, code is presented in figures that may span page breaks. Despite the breaks in code presentation, code line numbers are continuous where applicable.

A.3.1 General Approximation Comparisons

The code used to compare the Euler, Adams-Bashforth, and Runge-Kutta method to an exact solution is presented below. As most code does, the created script begins with package imports. Numpy was imported for its math capabilities, such as the exponential function, and Matplotlib was imported for its plotting functions.

```
"""
File meant to show numerical integration methods applied via python
Structured in a way that is related to the simulation method in PSLTDSim

NOTE: lambda is the python equivalent to matlab anonymous functions
"""
# Package Imports
import numpy as np
import matplotlib.pyplot as plt
```

Figure A.3: Approximation comparison package imports.

Each approximation method described in Equation A.1-A.4 was coded as a Python function. It should be noted that trapezoidal integration was intended to be performed after the simulation is run and full data is collected. This choice was made because of the various time steps involved with solution results.

```
# Function Definitions
def euler(fp, x0, y0, ts):
    """
    fp = Some derivative function of x and y
    x0 = Current x value
    y0 = Current y value
    ts = time step
    Returns y1 using Euler or tangent line method
    """
    return y0 + fp(x0,y0)*ts

def adams2(fp, x0, y0, xN, yN, ts):
```

```
    """
    fp = Some derivative function of x and y
    x0 = Current x value
    y0 = Current y value
    xN = Previous x value
    yN = Previous y value
    ts = time step
    Returns y1 using Adams-Bashforth two step method
    """
    return y0 + (1.5*fp(x0,y0) - 0.5*fp(xN,yN))*ts

def rk45(fp, x0, y0, ts):
    """
    fp = Some derivative function of x and y
    x0 = Current x value
    y0 = Current y value
    ts = time step
    Returns y1 using Runge-Kutta method
    """
    k1 = fp(x0, y0)
    k2 = fp(x0 +ts/2, y0+ts/2*k1)
    k3 = fp(x0 +ts/2, y0+ts/2*k2)
    k4 = fp(x0 +ts, y0+ts*k3)
    return y0 + ts/6*(k1+2*k2+2*k3+k4)

def trapezoidalPost(x,y):
    """
    x = list of x values
    y = list of y values
    Returns integral of y over x.
    Assumes full lists / ran post simulation
    """
    integral = 0
    for ndx in range(1,len(x)):
        integral+= (y[ndx]+y[ndx-1])/2 * (x[ndx]-x[ndx-1])
    return integral
```

Figure A.4: Approximation comparison function definitions.

To enable one file to execute all desired tests, a for loop that cycles through a case number variable was created. Each case if statement contains definitions for case name, simulation start and stop times, number of points to plot, the initial value problem, the exact

solution, and the exact integral solution. Equations from each case are further described in future sections. The Python `lambda` command was used to create temporary functions that are passed to other functions.

```
# Case Selection
for caseN in range(0,3):
    blkFlag = False # for holding plots open
    if caseN == 0:
        # Trig example
        caseName = 'Sinusoidal Example'
        tStart =0
        tEnd = 3
        numPoints = 6*2

        ic = [0,0] # initial condition x,y
        fp = lambda x, y: -2*np.pi*np.cos(2*np.pi*x)
        f = lambda x,c: -np.sin(2*np.pi*x)+c
        findC = lambda x,y: y+np.sin(2*np.pi*x)
        c = findC(ic[0],ic[1])
        calcInt = ( 1/(2*np.pi)*np.cos(2*np.pi*tEnd)+c*tEnd -
                    1/(2*np.pi)*np.cos(2*np.pi*ic[0])-c*ic[0] )

    elif caseN == 1:
        # Exp example
        caseName = 'Exponential Example'
        tStart =0
        tEnd = 3
        numPoints = 3

        ic = [0,0] # initial condition x,y
        fp = lambda x, y: np.exp(x)
        f = lambda x,c: np.exp(x)+c
        findC = lambda x, y: y-np.exp(x)
        c= findC(ic[0],ic[1])
        calcInt = np.exp(tEnd)+c*tEnd-np.exp(ic[0])+c*ic[0]

    elif caseN == 2:
        # Log example
        caseName = 'Logarithmic Example'
        tStart =1
        tEnd = 4
        numPoints = 3
        blkFlag = True # for holding plots open
```

```
97
98          ic = [1,1] # initial condition x,y
99          fp = lambda x, y: 1/x
100         f = lambda x,c: np.log(x)+c
101         findC = lambda x, y: y-np.log(x)
102         c= findC(ic[0],ic[1])
103         calcInt = (tEnd*np.log(tEnd)- tEnd +c*tEnd -
104                    ic[0]*np.log(ic[0])+ ic[0] -c*ic[0])
```

Figure A.5: Approximation comparison case definitions.

After case selection, a current value dictionary `cv` was initialized to mimic how PSLTDSim stores current values. Unlike PSLTDSim, the lists used to store history values were not initialized to the full length they were expected to be. This required logged values to be appended to the list after each solution. The reasoning behind this choice was again due to the various time steps involved with solution results.

```
105     # Initialize current value dictionary
106     # Shown to mimic PSLTDSim record keeping
107     cv={
108         't' :ic[0],
109         'yE': ic[1],
110         'yRK': ic[1],
111         'yAB': ic[1],
112         }
113
114     # Initialize running value lists
115     t=[]
116     yE=[]
117     yRK =[]
118     yAB = []
119
120     t.append(cv['t'])
121     yE.append(cv['yE'])
122     yRK.append(cv['yRK'])
123     yAB.append(cv['yAB'])
```

Figure A.6: Approximation comparison variable initialization.

An exact solution was computed using a hand-derived exact function. The code then entered a while loop that solved the selected differential equation for the next y value using the Euler, Runge-Kutta, and Adams-Bashforth methods. It should be noted that Python enables negative indexing of lists. Intuitively, negative indexes step backwards through an iterable object. An if statement was required to handle the first step of the Adams-Bashforth method as a -2 index does not exist in a list of length 1. After each approximation method was executed, and the solution stored in the current value dictionary, all values were logged and simulation time increased.

```
124     # Find C from integrated equation for exact soln
125     c = findC(ic[0], ic[1])
126 
127     # Calculate time step
128     ts = (tEnd-tStart)/numPoints
129 
130     # Calculate exact solution
131     tExact = np.linspace(tStart,tEnd, 10000)
132     yExact = f(tExact, c)
133 
134     # Start Simulation
135     while cv['t']< tEnd:
136 
137         # Calculate Euler result
138         cv['yE'] = euler( fp, cv['t'], cv['yE'],  ts )
139 
140         # Calculate Runge-Kutta result
141         cv['yRK'] = rk45( fp, cv['t'], cv['yRK'],  ts )
142 
143         # Calculate Adams-Bashforth result
144         if len(t)>=2:
145             cv['yAB'] = adams2( fp, cv['t'], cv['yAB'], t[-2], yAB[-2], ts )
146         else:
147             # Required to handle first step when a -2 index doesn't exist
148             cv['yAB'] = adams2( fp, cv['t'], cv['yAB'], t[-1], yAB[-1], ts )
149 
150         # Log calculated results
151         yE.append(cv['yE'])
152         yRK.append(cv['yRK'])
153         yAB.append(cv['yAB'])
```

```
154
155         # Increment and log time
156         cv['t'] += ts
157         t.append(cv['t'])
```

Figure A.7: Approximation comparison solution calculations.

Matplotlib functions were used to generate result plots after simulated time accumulated to a point that the while loop exited. Each line color, legend label, and various other superficial options were defined before global plot output options were configured and the plot displayed.

```
158     # Generate Plot
159     fig, ax = plt.subplots()
160     ax.set_title('Approximation Comparison\n' + caseName)
161
162     #Plot all lines
163     ax.plot(tExact,yExact,
164             c=[0,0,0],
165             linewidth=2,
166             label="Exact")
167     ax.plot(t,yE,
168             marker='o',
169             fillstyle='none',
170             linestyle=':',
171             c=[0.7,0.7,0.7],
172             label="Euler")
173     ax.plot(t,yRK,
174             marker='*',
175             markersize=10,
176             fillstyle='none',
177             linestyle=':',
178             c=[1,0,1],
179             label="RK4")
180     ax.plot(t,yAB,
181             marker='s',
182             fillstyle='none',
183             linestyle=':',
184             c =[0,1,0],
185             label="AB2")
```

```
186
187     # Format Plot
188     fig.set_dpi(150)
189     fig.set_size_inches(9, 2.5)
190     ax.set_xlim(min(t), max(t))
191     ax.grid(True, alpha=0.25)
192     ax.legend(loc='best',  ncol=2)
193     ax.set_ylabel('y Value')
194     ax.set_xlabel('x Value')
195     fig.tight_layout()
196     plt.show(block = blkFlag)
197     plt.pause(0.00001)
```

Figure A.8: Approximation comparison plotting.

After plotting, trapezoidal integration was performed on all results and compared to the calculated integral. It should be noted that the 'exact' result uses trapezoidal integration on 10,000 points while the calculated integral `calcInt` was computed via calculus. After code line 211 executes, the for loop that started on line 59 is restarted until all case numbers in the selected range are applied.

```
198     # Trapezoidal Integration
199     exactI = trapezoidalPost(tExact,yExact)
200     Eint = trapezoidalPost(t,yE)
201     RKint = trapezoidalPost(t,yRK)
202     ABint = trapezoidalPost(t,yAB)
203
204     print("\n%s" % caseName)
205     print("time step:  %.2f" % ts)
206     print("Method: Trapezoidal Int\t Absolute Error from calculated")
207     print("Calc: \t%.9f\t%.9f" % (calcInt ,abs(calcInt-calcInt)))
208     print("Exact: \t%.9f\t%.9f" % (exactI ,abs(calcInt-exactI)))
209     print("RK4: \t%.9f\t%.9f" % (RKint,abs(calcInt-RKint)))
210     print("AB2: \t%.9f\t%.9f" % (ABint,abs(calcInt-ABint)))
211     print("Euler: \t%.9f\t%.9f" % (Eint,abs(calcInt-Eint)))
```

Figure A.9: Approximation comparison trapezoidal integration and display.

A.3.1.1 Sinusoidal Example and Results

The first initial value example is presented as Equation Block A.5.

$$\begin{aligned} \text{Given: } & y(0) = 0 \\ & y'(x) = 2\pi \cos(2\pi x) \end{aligned} \tag{A.5}$$

Two integrations of Equation A.5 were performed to calculate the exact integral and plot the exact solution. This is shown in Equation Block A.6.

$$\begin{aligned} \int y'(x)\ \mathrm{d}x = y(x) &= -\sin(2\pi x) + C_1 \\ C_1 &= y_0 + \sin(2\pi x_0) \\ \int_0^\tau y(x)\ \mathrm{d}x &= \frac{1}{2\pi}\cos(2\pi x) + C_1 x \Big|_0^\tau \end{aligned} \tag{A.6}$$

Figure A.10 shows that when using a 0.5 step size, the approximations of all methods do not accurately reflect the exact function. This example and step size were contrived to show such behavior. The explanation for such a result lies in the derivatives calculated at the points used to generate each approximation.

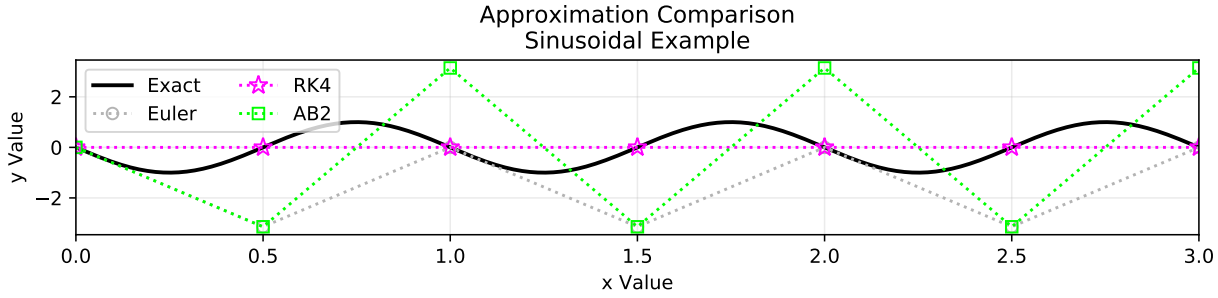

Figure A.10: Approximation comparison of a sinusoidal function using a step of 0.5.

Using a smaller step size of 0.25, as shown in Figure A.11, results with more accurate approximations. For all calculated points, the Runge-Kutta method matches the exact solution while the other two methods do not.

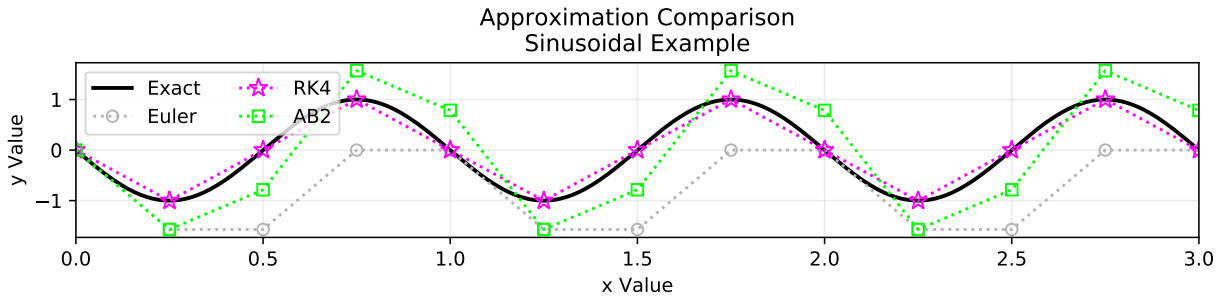

Figure A.11: Approximation comparison of a sinusoidal function using a step of 0.25.

Table A.1 shows the calculated integrals of the 0.25 step size example. It should be noted that because the integral is zero, any function may numerically match the calculated result if it is symmetrical about zero. The Runge-Kutta method meets this criteria despite representing more of a triangle wave instead of a sine wave. The Euler method has the largest error from exact integral as there are no approximated points above zero.

Table A.1: Trapezoidal integration results of a sinusoidal function using an x step of 0.25.

Method	Result	Absolute Error
Calculated	0.000000000	0.000000000
Exact	0.000000000	0.000000000
RK4	-0.000000000	0.000000000
AB2	-0.098174770	0.098174770
Euler	-2.356194490	2.356194490

A.3.1.2 Exponential Example and Results

The second initial value example is presented as Equation Block A.7.

$$\begin{aligned} &\text{Given: } y(0) = 0 \\ &y'(x) = e^x \end{aligned} \tag{A.7}$$

The required integrations of Equation A.7 are shown in Equation Block A.8.

$$\begin{aligned} \int y'(x)\ \mathrm{d}x &= y(x) = e^x + C_1 \\ C_1 &= y_0 - e^x \\ \int_0^\tau y(x)\ \mathrm{d}x &= e^x + C_1 x|_0^\tau \end{aligned} \tag{A.8}$$

Figure A.12 shows the resulting comparison plot using a step size of 1. The Runge-Kutta method matches the exact solution well while the other two approximation methods under-approximate. This is due to the lack of the Euler and Adams-Bashforth methods to accurately represent a constantly changing derivative.

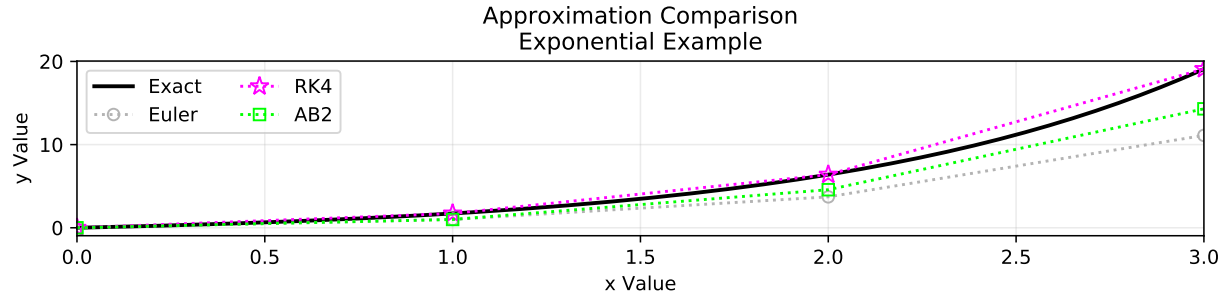

Figure A.12: Approximation comparison of an exponential function.

Table A.2 shows the trapezoidal integration of the exact function does not match the calculated integral. This is due to the exponential function not being well represented by trapezoids. Absolute error continued to increase with the Runge-Kutta, Adams-Bashforth, and Euler methods respectively.

Table A.2: Trapezoidal integration results of an exponential function using an x step of 1.

Method	Result	Absolute Error
Calculated	16.085536923	0.000000000
Exact	16.085537066	0.000000143
RK4	17.656057171	1.570520247
AB2	12.728355731	3.357181192
Euler	10.271950792	5.813586131

A.3.1.3 Logarithmic Example and Results

The third initial value example is presented as Equation Block A.9. Initial values are not zero as this would immediately lead to a divide by zero situation.

$$\begin{aligned} \text{Given: } y(1) &= 1 \\ y'(x) &= \frac{1}{x} \end{aligned} \tag{A.9}$$

The required integrations of Equation A.9 are shown in Equation Block A.10.

$$\begin{aligned} \int y'(x)\ \mathrm{d}x = y(x) &= \ln(x) + C_1 \\ C_1 &= y_0 - \ln(x_0) \\ \int_0^{\tau} y(x)\ \mathrm{d}x &= x\ln(x) - x + C_1 x\Big|_0^{\tau} \end{aligned} \tag{A.10}$$

Figure A.13 shows the resulting comparison plot using a step size of 1. Again the Runge-Kutta method produces the best approximation while the Euler method has the worst. The Adam-Bashforth method appears to be converging to the exact solution. While the exponential function and logarithmic functions both contain constantly changing derivatives, the logarithmic derivative decreases with increasing x values. This produces an over-approximating situation where as the exponential function was generally under-approximated.

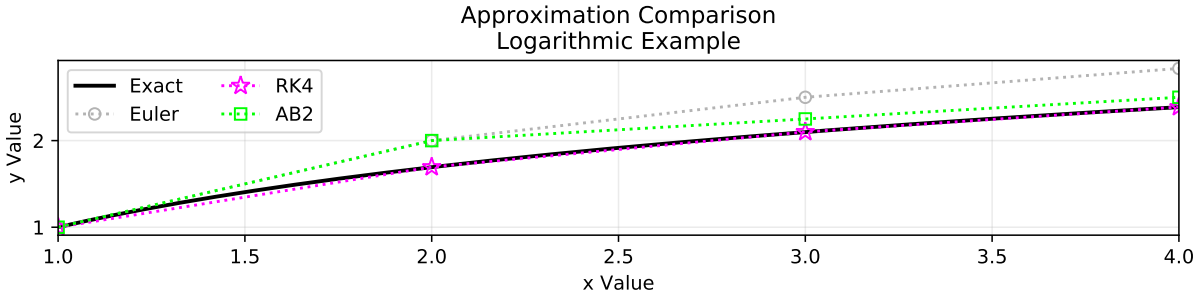

Figure A.13: Approximation comparison of a logarithmic function.

Table A.3 shows the integration results have a similar trend as seen in Table A.2

where the exact trapezoidal method doesn't match the calculated integral, and absolute error gradually increases using the Runge-Kutta, Adams-Bashforth, and Euler methods respectively.

Table A.3: Trapezoidal integration results of a logarithmic function.

Method	Result	Absolute Error
Calulated	5.545177444	0.000000000
Exact	5.545177439	0.000000006
RK4	5.488293651	0.056883794
AB2	6.000000000	0.454822556
Euler	6.416666667	0.871489222

A.3.1.4 General Approximation Result Summary

The chosen examples showed that the Runge-Kutta method typically produces better results than the simpler Euler or Adams-Bashforth methods. Step size is an important factor to consider when using approximation methods as phenomena may be ignored or reported in error elsewise. Depending on step size, trapezoidal integration can produce results that are reasonable approximations of calculated integrals.

A.3.2 Python Function Comparisons

Code used to compare the Python lsim and solve_ivp functions to the exact solution and fourth order Runge-Kutta approximation is presented below. The code is very similar to the previously discussed approximation comparison code and again begins with package imports and function definitions. The solve_ivp function was imported from the integrate methods of Scipy, while the lsim function is part of the signal collection of functions. Only the Runge-Kutta and trapezoidal methods are defined as functions in this code example.

```
# Package Imports
import numpy as np
import matplotlib.pyplot as plt

from scipy.integrate import solve_ivp
from scipy import signal

# Function Definitions
def rk45(fp, x0, y0, ts):
    """
    fp = Some derivative function of x and y
    x0 = Current x value
    y0 = Current y value
    ts = time step
    Returns y1 using Runge-Kutta method
    """
    k1 = fp(x0, y0)
    k2 = fp(x0 +ts/2, y0+ts/2*k1)
    k3 = fp(x0 +ts/2, y0+ts/2*k2)
    k4 = fp(x0 +ts, y0+ts*k3)
    return y0 + ts/6*(k1+2*k2+2*k3+k4)

def trapezoidalPost(x,y):
    """
    x = list of x values
    y = list of y values
    Returns integral of y over x.
    Assumes full lists / ran post simulation
    """
    integral = 0
    for ndx in range(1,len(x)):
```

```
32          integral+= (y[ndx]+y[ndx-1])/2 * (x[ndx]-x[ndx-1])
33      return integral
```

Figure A.14: Python function comparison imports and definitions.

Case definitions were similar to the previous example with the addition of an lti system definition. For simplicity, a transfer function style system was used as input to create each lti system. More specifically, this input consisted of the numerator and denominator of the transfer function as lists of descending powers $ (the Laplace '*s*'). Numerous transforms and calculus based mathematical methods found in [5], [6] and [56] were employed to calculate the exact functions and integrals which are described in more detail after this code discussion.

```
34  # Case Selection
35  for caseN in range(0,3):
36      blkFlag = False # for holding plots open
37
38      if caseN == 0:
39          # step input Integrator example
40          caseName = 'Step Input Integrator Example'
41          tStart =0
42          tEnd = 4
43          numPoints = 4
44
45          U = 1
46          initState = 0
47          ic = [0,initState] # initial condition x,y
48          fp = lambda x, y: 1
49          f = lambda x, c: x+c
50          findC = lambda x, y: y-x
51          system = signal.lti([1],[1,0])
52          calcInt = 0.5*(tEnd**2) # Calculated integral
53
54      elif caseN == 1:
55          # step input Low pass example
56          caseName = 'Step Input Low Pass Example'
57          tStart =0
58          tEnd = 2
59          numPoints = 4
60
```

```
61          A = 0.25
62          U = 1.0
63          initState = 0
64          ic = [0,initState] # initial condition x,y
65          fp = lambda x, y: 1/A*np.exp(-x/A)# via table
66          f = lambda x, c: -np.exp(-x/A) +c
67          findC = lambda x, y : y+np.exp(-x/A)
68          system = signal.lti([1],[A,1])
69          calcInt = tEnd + A*np.exp(-tEnd/A)-A # Calculated integral
70
71      else:
72          # step multi order system
73          caseName = 'Step Input Third Order System Example'
74          tStart =0
75          tEnd = 5
76          numPoints = 5*2
77          blkFlag = True # for holding plots open
78
79          U = 1
80          T0 = 0.4
81          T2 = 4.5
82          T1 = 5
83          T3 = -1
84          T4 = 0.5
85
86          alphaNum = (T1*T3)
87          alphaDen = (T0*T2*T4)
88          alpha = alphaNum/alphaDen
89
90          num = alphaNum*np.array([1, 1/T1+1/T3, 1/(T1*T3)])
91          den  = alphaDen*np.array([1, 1/T4+1/T0+1/T2, 1/(T0*T4)+1/(T2*T4)+1/(T0*T2),
92                          1/(T0*T2*T4)])
93          system = signal.lti(num,den)
94
95          # PFE
96          A = ((1/T1-1/T0)*(1/T3-1/T0))/((1/T2-1/T0)*(1/T4-1/T0))
97          B = ((1/T1-1/T2)*(1/T3-1/T2))/((1/T0-1/T2)*(1/T4-1/T2))
98          C = ((1/T1-1/T4)*(1/T3-1/T4))/((1/T0-1/T4)*(1/T2-1/T4))
99
100         initState = 0 # for steady state start
101         ic = [0,0] # initial condition x,y
102         fp = lambda x, y: alpha*(A*np.exp(-x/T0)+B*np.exp(-x/T2)+C*np.exp(-x/T4))
103         f = lambda x, c:
            ↪  alpha*(-T0*A*np.exp(-x/T0)-T2*B*np.exp(-x/T2)-T4*C*np.exp(-x/T4))+c
```

```
        findC = lambda x, y : alpha*(A*T0+B*T2+C*T4)
        c = findC(ic[0], ic[1])
        calcInt = (
            alpha*A*T0**2*np.exp(-tEnd/T0) +
            alpha*B*T2**2*np.exp(-tEnd/T2) +
            alpha*C*T4**2*np.exp(-tEnd/T4) +
            c*tEnd -
            alpha*(A*T0**2+B*T2**2+C*T4**2)
            )# Calculated integral
```

Figure A.15: Python function comparison case definitions.

Initial conditions and list initializations were performed in a similar manner as the previous example. An additional `xLS` variable was required to track the states associated with the lsim function.

```
    # Initialize current value dictionary
    # Shown to mimic PSLTDSim record keeping
    cv={
        't' :ic[0],
        'yRK': ic[1],
        'ySI': ic[1],
        'yLS': ic[1],
        }

    # Initialize running value lists
    t=[]

    # runge-kutta
    yRK =[]

    # solve ivp
    ySI = []
    tSI = []

    # lsim
    yLS = []
    xLS = [] # required to track state history

    # Log intial values
    t.append(cv['t'])
```

```
138
139     yRK.append(cv['yRK'])
140     yLS.append(cv['yLS'])
141     xLS.append(cv['yLS'])
```

Figure A.16: Python function comparison variable initializations.

The exact solution and Runge-Kutta methods were handled as before, but Python function inputs required slightly different input. The lsim and solve_ivp outputs also required slightly different handling as their output was not just a single value. Again, negative indexing is used to access the last value in an iterable object.

```
142     # Calculate time step
143     ts = (tEnd-tStart)/numPoints
144     # Find C from integrated equation for exact soln
145     c = findC(ic[0], ic[1])
146     # Calculate exact solution
147     tExact = np.linspace(tStart,tEnd, 1000)
148     yExact = f(tExact, c)
149
150     # Start Simulation
151     while cv['t']< tEnd:
152
153         # Calculate Runge-Kutta result
154         cv['yRK'] = rk45( fp, cv['t'], cv['yRK'],  ts )
155
156         # Runge-Kutta 4(5) via solve IVP.
157         soln = solve_ivp(fp, (cv['t'], cv['t']+ts), [cv['ySI']])
158
159         # lsim solution
160         if cv['t'] > 0:
161             tout, ylsim, xlsim = signal.lsim(system, [U,U], [0,ts], xLS[-1])
162         else:
163             tout, ylsim, xlsim = signal.lsim(system, [U,U], [0,ts], initState)
164
165         # Log calculated results
166         yRK.append(cv['yRK'])
167
168         # handle solve_ivp output data
169         ySI += list(soln.y[-1])
```

```
170         tSI += list(soln.t)
171         cv['ySI'] = ySI[-1] # ensure correct cv
172 
173         # handle lsim output data
174         cv['yLS']=ylsim[-1]
175         yLS.append(cv['yLS'])
176         xLS.append(xlsim[-1]) # this is the state
177 
178         # Increment and log time
179         cv['t'] += ts
180         t.append(cv['t'])
```

Figure A.17: Python function comparison solution calculations.

Once the simulation is complete, plotting and trapezoidal integration was carried out in the same manner as previously discussed before the for loop restarts.

```
181     # Generate Plot
182     fig, ax = plt.subplots()
183     ax.set_title('Approximation Comparison\n' + caseName)
184 
185     #Plot all lines
186     ax.plot(tExact,yExact,
187             c=[0,0,0],
188             linewidth=2,
189             label="Exact")
190     ax.plot(t,yRK,
191             marker='*',
192             markersize=10,
193             fillstyle='none',
194             linestyle=':',
195             c=[1,0,1],
196             label="RK45")
197     ax.plot(tSI,ySI,
198             marker='x',
199             markersize=10,
200             fillstyle='none',
201             linestyle=':',
202             c=[1,.647,0],
203             label="solve_ivp")
204     ax.plot(t,yLS,
```

```
            marker='+',
            markersize=10,
            fillstyle='none',
            linestyle=':',
            c ="#17becf",
            label="lsim")

    # Format Plot
    fig.set_dpi(150)
    fig.set_size_inches(9, 2.5)
    ax.set_xlim(min(t), max(t))
    ax.grid(True, alpha=0.25)
    ax.legend(loc='best',  ncol=2)
    ax.set_ylabel('y Value')
    ax.set_xlabel('Time [seconds]')
    fig.tight_layout()
    plt.show(block = blkFlag)
    plt.pause(0.00001)

    # Trapezoidal Integration
    exactI = trapezoidalPost(tExact,yExact)
    SIint = trapezoidalPost(tSI,ySI)
    RKint = trapezoidalPost(t,yRK)
    LSint = trapezoidalPost(t,yLS)

    print("\n%s" % caseName)
    print("time step:  %.2f" % ts)
    print("Method: Trapezoidal Int\t Absolute Error from calculated")
    print("Calc: \t%.9f\t%.9f" % (calcInt ,abs(calcInt-calcInt)))
    print("Exact: \t%.9f\t%.9f" % (exactI ,abs(calcInt-exactI)))
    print("RK4: \t%.9f\t%.9f" % (RKint,abs(calcInt-RKint)))
    print("SI: \t%.9f\t%.9f" % (SIint,abs(calcInt-SIint)))
    print("lsim: \t%.9f\t%.9f" % (LSint,abs(calcInt-LSint)))
```

Figure A.18: Python function comparison plotting and integration code.

A.3.2.1 Integrator Example and Results

The first example is the Laplace domain integrator block shown in Figure A.19.

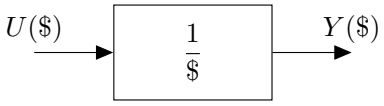

Figure A.19: Integrator block.

Transformation of the block into a time domain derivative function is shown in Equation Block A.11. As step input is a given, this results in a very simple differential equation.

$$\begin{aligned} &\text{Given: Step input, } y(0) = 0 \\ &F(\$) = \frac{Y(\$)}{U(\$)} = \frac{1}{\$} \\ &F(\$) = Y(\$)\$ = U(\$) \\ &\mathscr{L}^{-1}\{F(\$)\} \longrightarrow y'(t) = u(t) = 1 \end{aligned} \tag{A.11}$$

The required integrations are shown in Equation Block A.12.

$$\begin{aligned} \int y'(t)\ \mathrm{d}t &= y(t) = t + C_1 \\ C_1 &= y_0 - t_0 \\ \int_0^\tau y(t)\ \mathrm{d}t &= \frac{1}{2}t^2 + C_1 t \Big|_0^\tau \end{aligned} \tag{A.12}$$

The resulting approximation comparisons are plotted in Figure A.20. While all methods produce the same result, it is worth noting the extra approximations generated by the solve_ivp function near the beginning of each approximation interval.

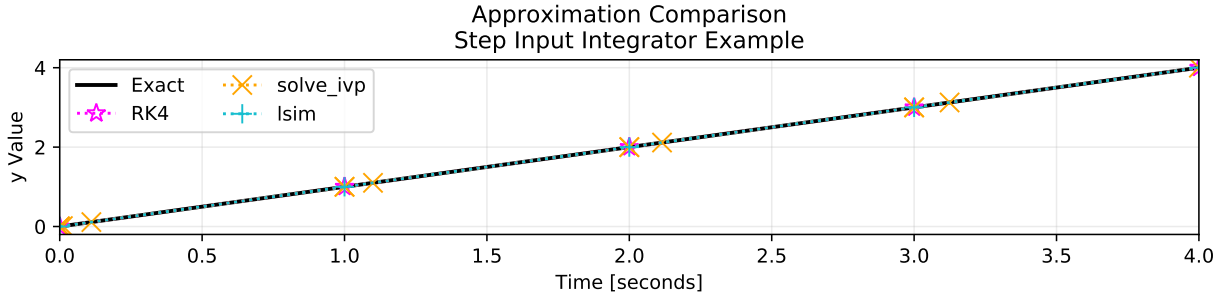

Figure A.20: Approximation comparison of an integrator block.

Table A.4 shows that all methods match the calculated integral. Obviously, this particular linear function can be accurately represented by trapezoids.

Table A.4: Trapezoidal integration results of an integral function.

Method	Result	Absolute Error
Calculated	8.000000000	0.000000000
Exact	8.000000000	0.000000000
RK4	8.000000000	0.000000000
solve_ivp	8.000000000	0.000000000
lsim	8.000000000	0.000000000

A.3.2.2 Low Pass Example and Results

A slightly more interesting example consists of the Laplace low pass filter block shown in Figure A.21.

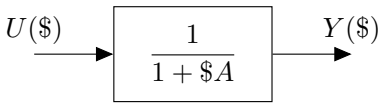

Figure A.21: Low pass filter block.

Equation Block A.13 shows the manipulation of $F(\$)$ to match a common Laplace form so that a conversion table could be used to easily convert the equation from the frequency domain to the time domain.

$$\begin{aligned} &\text{Given: Step input, } A = 0.25, y(0) = 0 \\ &F(\$) = \frac{Y(\$)}{U(\$)} = \left(\frac{1}{A}\right)\left(\frac{1}{\$ + 1/A}\right) \\ &\mathscr{L}^{-1}\{F(\$)\} \longrightarrow y'(t) = \frac{e^{-t/A}}{A} \end{aligned} \tag{A.13}$$

Required integration is shown in Equation Block A.14.

$$\begin{aligned} \int y'(t)\ \mathrm{d}t = y(t) &= -e^{-t/A} + C_1 \\ C_1 &= y_0 + e^{-t_0/A} \\ \int_0^\tau y(t)\ \mathrm{d}t &= Ae^{-t_0/A} + C_1 t\Big|_0^\tau \end{aligned} \tag{A.14}$$

The resulting approximation comparisons are shown in Figure A.22. All methods produce approximations that are very close to the exact solution. The solve_ivp function again produces more approximations between the defined step range.

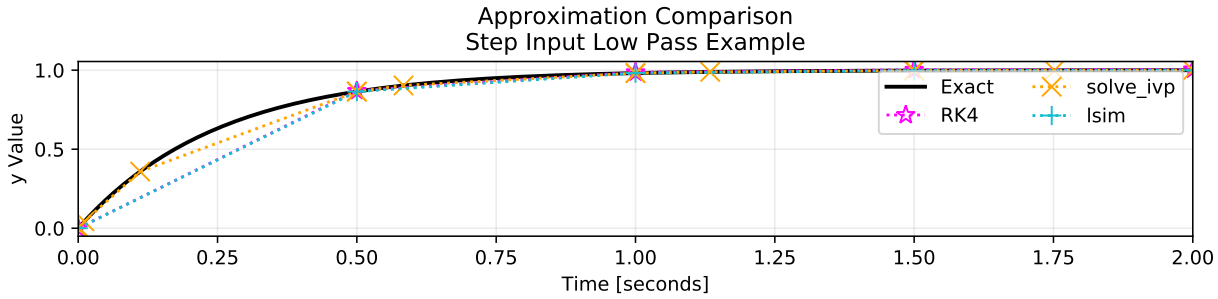

Figure A.22: Approximation comparison of a low pass filter block.

Table A.5 shows the integration results of the low pass example. The exact solution has slight error from the calculated integral as trapezoidal integration provides only an approximate solution. The solve_ivp result has the next smallest error due to the added points between defined approximation steps. Runge-Kutta and lsim results were very similar.

Table A.5: Trapezoidal integration results a of low pass filter using a t step of 0.5.

Method	Result	Absolute Error
Calculated	1.750083866	0.000000000
Exact	1.750082530	0.000001336
RK4	1.680138966	0.069944900
solve_ivp	1.719657220	0.030426646
lsim	1.671851297	0.078232568

A.3.2.3 Third Order System Example and Results

A third order system that resembles the one used in the genericGov is shown in Figure A.23. As the previous example showed, manipulation of Laplace transfer function blocks with poles may be useful when it comes time to convert to the time domain. The resulting modified block diagram is shown in Figure A.24.

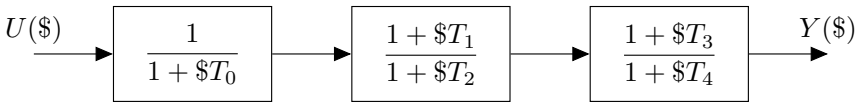

Figure A.23: Third order system block diagram.

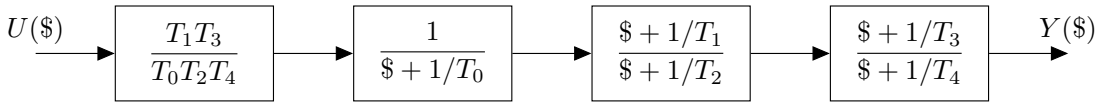

Figure A.24: Modified third order system block diagram.

Example givens and algebraic simplifications are listed at the top of Equation A.15. The time constants chosen were those of the generic hydro governor with the exception of T_2, which was reduced by an order of magnitude so that a steady state was reached within a reasonable amount of time. Partial fraction expansion was used to express the third order equation as sum of first order terms. The rational behind this action was to enable a simpler inverse Laplace transform.

$$
\begin{aligned}
&\text{Given: Step input, } T_0 = 0.4,\ T_1 = 5.0,\ T_2 = 4.5,\\
&T_3 = -1.0,\ T_4 = 0.5,\ y(0) = 0\\
&\text{Let } \alpha = \frac{T_1T_3}{T_0T_2T_4}\\
F(\$) = \alpha\frac{(\$+1/T_1)(\$+1/T_3)}{(\$+1/T_0)(\$+1/T_2)(\$+1/T_4)} &= \alpha\left(\frac{A}{\$+1/T_0} + \frac{B}{\$+1/T_2} + \frac{C}{\$+1/T_4}\right)\\
F(\$)(\$+1/T_0)\big|_{\$=-1/T_0} = A &= \frac{\left(1/T_1 - 1/T_0\right)\left(1/T_3 - 1/T_0\right)}{\left(1/T_2 - 1/T_0\right)\left(1/T_4 - 1/T_0\right)}\\
F(\$)(\$+1/T_2)\big|_{\$=-1/T_2} = B &= \frac{\left(1/T_1 - 1/T_2\right)\left(1/T_3 - 1/T_2\right)}{\left(1/T_0 - 1/T_2\right)\left(1/T_4 - 1/T_2\right)}\\
F(\$)(\$+1/T_4)\big|_{\$=-1/T_4} = C &= \frac{\left(1/T_1 - 1/T_4\right)\left(1/T_3 - 1/T_4\right)}{\left(1/T_0 - 1/T_4\right)\left(1/T_2 - 1/T_4\right)}\\
\mathscr{L}^{-1}\{F(\$)\} \longrightarrow y'(t) &= \alpha\left(Ae^{-t/T_0} + Be^{-t/T_2} + Ce^{-t/T_4}\right)
\end{aligned}
\tag{A.15}
$$

The relatively straight forward integrations required for an exact solution and integral are

shown in Equation Block A.16.

$$\begin{aligned} \int y'(t)\ \mathrm{d}t = y(t) &= -\alpha\left(AT_0e^{-t/T_0} + BT_2e^{-t/T_2} + CT_4e^{-t/T_4}\right) + C_1 \\ C_1 &= y_0 + \alpha\left(AT_0e^{-t_0/T_0} + BT_2e^{-t_0/T_2} + CT_4e^{-t_0/T_4}\right) \\ \int_0^\tau y(t)\ \mathrm{d}t &= \alpha\left(AT_0^2e^{-t/T_0} + BT_2^2e^{-t/T_2} + CT_4^2e^{-t/T_4}\right) + C_1t\Big|_0^\tau \end{aligned} \tag{A.16}$$

Figure A.25 shows the approximation comparison results of the third order system. Using a half second time step produces results that are fairly similar to the exact method. As previously seen, the solve_ivp solution produces more approximations between defined time steps.

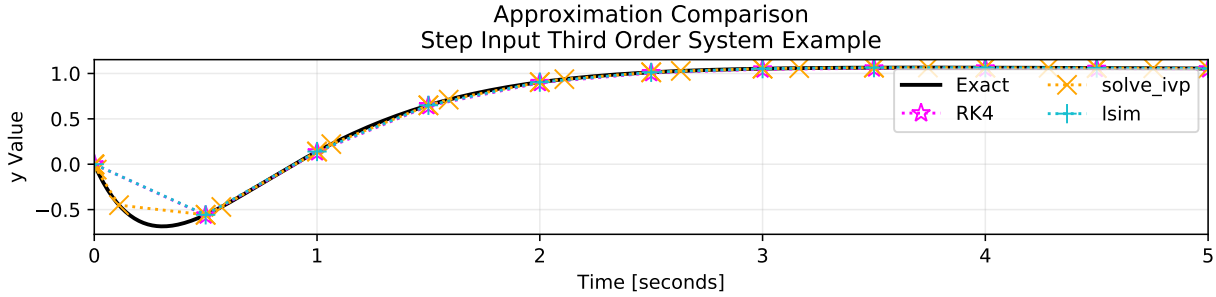

Figure A.25: Third order approximation comparison using half second time step.

Increasing the time step to one second, as shown in Figure A.26, highlights more differences between the methods. The solve_ivp soultion still tracks the exact solution well because of the additional approximations between time steps. Approximations of lsim match the exact solution, however, dynamics between time steps are not represented at all. The Runge-Kutta method also ignores dynamics between approximation results and appears to under-approximate steady state behavior.

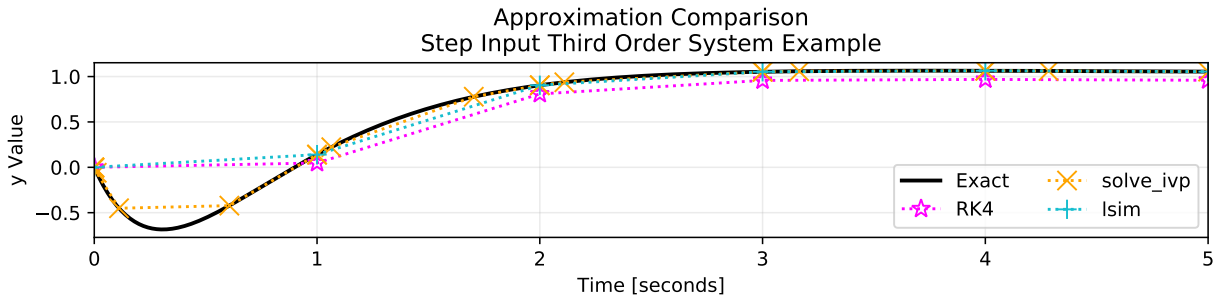

Figure A.26: Third order approximation comparison using one second time step.

Table A.6 lists the integration results from the half second time step test. The exact solution is near the calculated solution, but they do not match completely. The solve_ivp absolute error is the next smallest due to the additional data points generated between set time steps. While the absolute error from the Runge-Kutta solution is calculated as slightly less than the lsim result, this is due to the large negative area both solutions ignore between $t = 0$ and $t = 1$ and the continuous under-approximation by the Runge-Kutta method.

Table A.6: Trapezoidal integration results of a third order function using a t step of 0.5.

Method	Result	Absolute Error
Calculated	3.351959451	0.000000000
Exact	3.351971025	0.000011574
RK4	3.425878989	0.073919538
solve_ivp	3.385138424	0.033178973
lsim	3.458377872	0.106418421

A.3.2.4 Python Approximation Result Summary

As previously stated, distance between approximations dictates much of what one can glean from resulting solutions. As such, the full resolution solve_ivp solution provided the most detail of the tested examples. However, the data at defined time steps were essentially the same between the lsim and solve_ivp results. With a small enough time step, the Runge-Kutta method approximations was also similar to the Python function approximations. When a larger time step was used, the Runge-Kutta method did not match the exact

solution in cases where the other methods did.

Through these experiments and comparisons, it was shown that the lsim and solve_ivp methods are comparable, and in some ways, better than the hand coded Runge-Kutta method. Additionally, trapezoidal integration was shown to produce adequate results depening on the input data.

A.4 Dynamic Agent Numerical Utilizations

This section is meant to better describe the handling of numerical methods by specific agents in PSLTDSim. Specifically, window integration and the combined swing equation function are described in detail before governor and filter agent considerations about integrator wind up and dynamic staging are presented.

A.4.1 Window Integrator

The window integrator agent used by balancing authority agents that integrate ACE applies the trapezoidal integration technique. As this agent is relatively simple, the full Python definition is shown in Figure A.27. While attempts were made to create readable code, window integrator actions are also explained below.

```
1  class WindowIntegratorAgent(object):
2      """A window integrator that initializes a history of window
3      values, then updates the total window area each step."""
4
5      def __init__(self, mirror, length):
6          # Retain Inputs / mirror reference
7          self.mirror = mirror
8          self.length = length # length of window in seconds
9
10         self.windowSize = int(self.length / self.mirror.timeStep)
11
12         self.window = [0.0]*self.windowSize
13         self.windowNDX = -1 # so first step index points to 0
14
15         self.cv = {
16             'windowInt' : 0.0,
17             'totalInt' : 0.0,
```

```
            }

    def step(self, curVal, preVal):
        # calculate current window Area, return value
        self.windowNDX += 1
        self.windowNDX %= self.windowSize

        oldVal = self.window[self.windowNDX]
        newVal = (curVal + preVal)/ 2.0 * self.mirror.timeStep

        self.window[self.windowNDX] = newVal
        self.cv['windowInt'] += newVal - oldVal
        self.cv['totalInt'] += newVal

        return self.cv['windowInt']
```

Figure A.27: Window integrator definition.

The agent is initialized by any agent that is desired to perform window integration. Required input parameters are a reference to the system mirror and window length in seconds. The reference to the system mirror is stored and a list of place holder values is created that is the length of the integration window in seconds, divided by the selected time step. This division result is cast into an integer as lists cannot have float value lengths. This list of history values is not required for integration, but it can be used to verify the correct operation of the integrator. A window index is created with an initial value of negative one so that during the first step, the index correctly points to list item zero. A current value dictionary `cv` is created to keep track of the most recent window integration and total integration values.

The parent agent is responsible for calling the window integrator step function each time step with current and previous values of integration focus. The window index variable is incremented by one, and then the modulo operator is used to ensure the index always points to a location that exists inside the list of history values. The value located at the current index value is stored as `oldVal` and later subtracted from the current window integration

value. The integral between the two passed in values is calculated using the trapezoidal method and stored as `newVal`. This `newVal` is then stored in the window integrator history value list at the current index, and added to both the current value for window and total integration. The agent step ends by returning the current value of the window integrator.

A.4.2 Combined Swing Equation

The full code for the combined swing equation is presented in Figure A.28. The function first checks if frequency effects should be accounted for, and then calculates the PU values required for computation of $\dot{\omega}_{sys}$ (`fdot` in the code). The calculated `fdot` is used by the Adams-Bashforth and Euler solution methods if specified by the user. If the chosen integration method is 'rk45', the Runge-Kutta 4(5) method included in solve_ivp is used instead. While the Euler and Adams-Bashforth methods return only the next y value, the solve_ivp method returns more output variables that must be properly handled. The combined swing equation returns nothing and makes any required changes only to the system mirror.

```
def combinedSwing(mirror, Pacc):
    """Calculates fdot, integrates to find next f, calculates deltaF.
    Pacc in MW, f and fdot are PU
    """

    # Handle frequency effects option
    if mirror.simParams['freqEffects'] == 1:
        f = mirror.cv['f']
    else:
        f = 1.0

    PaccPU = Pacc/mirror.Sbase # for PU value
    HsysPU = mirror.cv['Hsys']/mirror.Sbase # to enable variable inertia
    deltaF = 1.0-mirror.cv['f'] # used for damping

    # Swing equation numerical solution
    fdot = 1/(2*HsysPU)*(PaccPU/f - mirror.Dsys*deltaF)
    mirror.cv['fdot'] = fdot

    # Adams Bashforth
```

```
    if mirror.simParams['integrationMethod'].lower() == 'ab':
        mirror.cv['f'] = f + 1.5*mirror.timeStep*fdot -
        ↪  0.5*mirror.timeStep*mirror.r_fdot[mirror.cv['dp']-1]

    # scipy.integrate.solve_ivp
    elif mirror.simParams['integrationMethod'].lower() == 'rk45':
        tic = time.time() # begin dynamic agent timer

        c = [HsysPU, PaccPU, mirror.Dsys, f] # known variables in swing eqn
        cSwing = lambda t, y: 1/(2*c[0])*(c[1]/y - c[2]*(1-c[3]))
        soln = solve_ivp(cSwing, [0, mirror.timeStep], [f])
        mirror.cv['f'] = float(soln.y[-1][-1]) # set current freq to last value

        mirror.IVPTime += time.time()-tic # accumulate and end timer

    # Euler method - chosen by default
    else:
        mirror.cv['f'] = mirror.cv['f'] + (mirror.timeStep*fdot)

    # Log values
    # NOTE: deltaF changed 6/5/19 to more useful 1-f
    deltaF = 1.0 - mirror.cv['f']
    mirror.cv['deltaF'] = deltaF
```

Figure A.28: Combined swing function definition.

A.4.3 Governor and Filter Agent Considerations

The lsim function was chosen for governor and filter dynamic calculations. This was meant to enable a consistent solution method for these agent types. However, lsim only performs linear simulation and non-linear actions, such as limiting, must be handled manually. Futher, to simplify model creation and allow non-linear action, governor models were created as multiple dynamic stages that pass values to each other. Both of these lsim specific areas are covered in this section.

A.4.3.1 Integrator Wind Up

Non-linear system behavior must be handled outside of, or in between, an lsim solution as lsim only handles linear simulation. A common non-linear action is limiting. An issue

may arise when limiting a pure integrator and not addressing integrator wind up. A simple example demonstrating integrator wind up is shown in Figure A.29. The system used is the same as shown in Figure A.19 but with an output limiter set at ±2, and the input is depicted in Figure A.29.

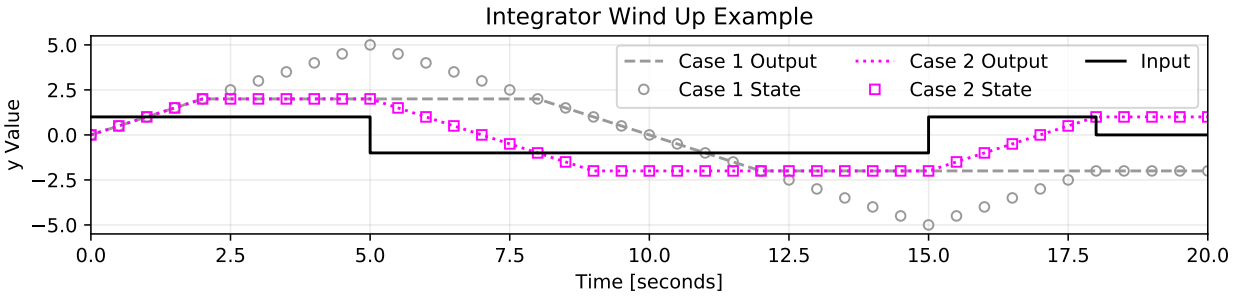

Figure A.29: Effect of integrator wind up.

Results from Case 1 include only an output value limiter, while Case 2 also limits the integrator state. Limiting the state prevents integrator wind up which can be seen in Case 1 between $t = 2.5$ and $t = 7.5$ and again between $t = 12.5$ and $t = 17.5$. The execution of such limiting could be done multiple ways. In this case, a simple if statement was placed after the solution that checks output and state values. The if statement, if executed, adjusts the output and/or state values accordingly.

A.4.3.2 Combined System Comparisons

To allow for a variety of governor models without rewriting code and enable non-linear action, the technique of using a sequence of individual blocks for each part of a specific model was employed in current PSLTDSim governor models. Modeling differences due to interaction of states in multi-order systems represented by a series of single order systems was explored by simulating equivalent systems consisting of various dynamic stages. For example, the block diagram shown in Figure A.23 is mathematically equivalent to the block diagrams shown in Figures A.30 and A.31, however, the computation of each system may not be equivalent.

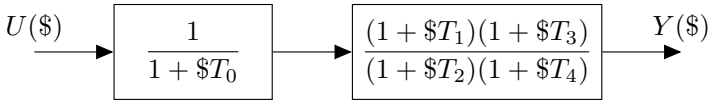

Figure A.30: Third order system as two stages.

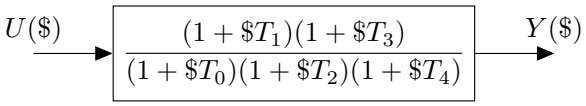

Figure A.31: Third order system as single stage.

Figure A.30 shows the output of a third order system as calculated by various dynamic stage models. The interaction of states affects the resulting output and it can be seen that the three stage system does not capture system dynamics well. A two stage calculation produces output closer to the single stage system, but some dynamics are not represented.

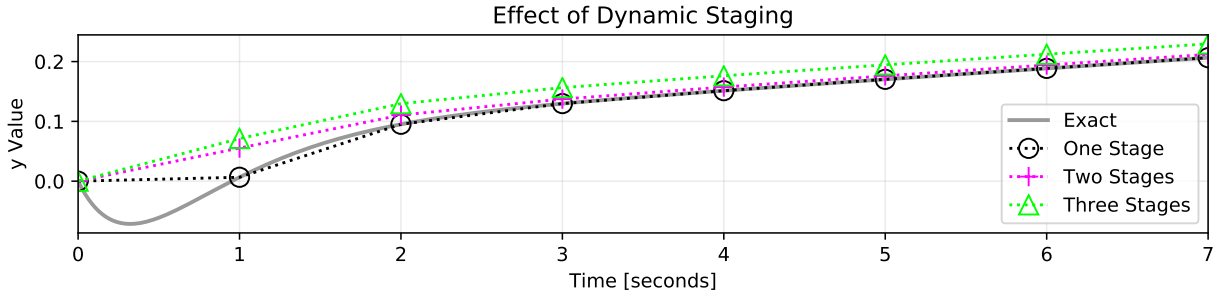

Figure A.32: Effect of dynamic staging using one second time step.

Reducing step size, as shown in Figure A.33, produces similar behavior where the three stage output is most different from the single stage model.

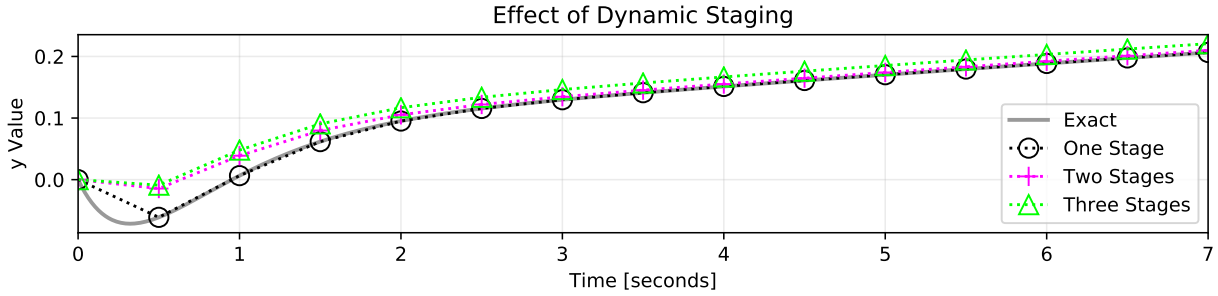

Figure A.33: Effect of dynamic staging using half second time step.

A.4.4 Numerical Utilization Summary

PSLTDSim uses various numerical methods to achieve satisfactory results. However, PSLTDSim was designed to be a customizable simulation environment, and as more use cases arise, previously accepted solution methods may no longer be deemed as such. While there is no currently employed method for integrator wind up prevention, it is certainly possible. Likewise, experiments have shown there is a noticeable reduction in output definition when dynamic models are separated into multiple states. Of course, modifying or creating new, dynamic models to better meet changing user needs is possible. Such modifications require some understanding of the actual code. While documentation such as this can provide some assistance to such an endeavor, actual understanding can be best gained through actual study of the available source code.

B Large Tables

This appendix is used to present large tables too distracting for inclusion in their respective sections.

Table B.1: Balancing authority dictionary input information.

Key	Type	Units	Example	Description
B	String	MW/0.1Hz	"1.0 : permax"	Describes the frequency bias scaling factor B used in the ACE calculation. Various Options exist.
AGCActionTime	Float	Seconds	5	Time between AGC dispatch messages.
AGCType	String	-	"TLB : 2"	Dictates which AGC routine to use and type specific options.
UseAreaDroop	Boolean	-	FALSE	If True, all governed generators under BA control will use the area droop.
AreaDroop	Float	Hz/MW	0.05	Droop value to use if 'UseAreaDroop' is True.
IncludeIACE	Boolean	-	TRUE	If True, include IACE in ACE calculation
IACEconditional	Boolean	-	FALSE	Adds IACE to ACE if signs of deltaw and IACE match.
IACEwindow	Integer	Seconds	60	Defines the length of moving integration window to use in IACE. If set to 0, integration takes place for all time.
IACEscale	Float	-	0.0167	Value used to scale IACE.
IACEdeadband	Float	Hz	0.036	Absolute value of system frequency where IACE will not be calculated below.
ACEFiltering	String	-	PI : 0.03 0.001'	String used to dictate which filter agent is created and filter specific parameters.
AGCDeadband	Float	MW	1.5	Value of ACE to ignore sending in AGC dispatch. Not implemented as of this writing.
GovDeadbandType	String	-	step'	Type of deadband to be applied to area governors.
GovDeadband	Float	Hz	0.036	Absolute value of system frequency that governors will not respond below.
GovAlpha	Float	Hz	0.016	Specific to 'NLDroop' type of deadband. Specifies lower bound of non-linear droop.
GovBeta	Float	Hz	0.036	Specific to 'NLDroop' type of deadband. Specifies upper bound of non-linear droop.
CtrlGens	List of Strings	-	-	List of generators, participation factor, and dispatch signal type.

Table B.2: Simulation parameters dictionary input information.

Key	Type	Units	Example	Description
timeStep	float	Seconds	1	Simulated time between power-flow solutions
endTime	float	Seconds	1800	Number of seconds simulation is to run for.
slackTol	float	MW	0.5	MW Value that slack error must be below for returned solution to be accepted.
PY3msgGroup	integer	-	3	Number of messages to combine into one AMQP message for PY3 to IPY communication.
IPYmsgGroup	integer	-	60	Number of messages to combine into one AMQP message for IPY to PY3 communication.
Hinput	float	MW sec	0	Value to use for total system inertia. Units are MW*sec. If set to 0.0, system inertia will be calculated from the given .sav information.
Dsys	float	PU	0	Value of system damping used in swing equation and governor models. While this option is available, it is untested and typically set to 0.0.
fBase	float	Hz	60	Value of base system frequency.
freqEffects	boolean	-	True	If True, the ω used in the swing equation will be the current system frequency. If this is set to False then ω will be set equal to 1 for the swing equation calculation
integrationMethod	string	-	'rk45'	This option defines how the swing equation is integrated to find current frequency. Valid options are 'rk45', 'ab', and 'euler'. The default is the 'euler' method which is a simple forward Euler integration. The 'ab' option uses a two step Adams-Bashforth method and the 'rk45' options uses the scipy `solve_ivp` function that utilizes an explicit Runge-Kutta 4(5) method.
fileDirectory	string	-	"\\delme\\"	This is a relative path location from the folder where PSLTDSim is exectued in which the output files are saved to.
fileName	string	-	"SimTest"	This is that name used to save files.
exportFinalMirror	int	-	1	If this value is 1 a final system mirror will be exported. If this value is 0 no final mirror will be exported.
exportMat	int	-	1	If this value is 1 a MATLAB .mat file will be exported. If this value is 0 no MATLAB .mat file will be exported.

C Code Examples

This appendix is used to present code examples too large for inclusion in the body of the text. Some examples span multiple pages. To adhere to document format requirements, the description of each code example is presented after the corresponding code figure.

```
# Format of required info for batch runs.
debug = 0
AMQPdebug = 0
debugTimer = 0

simNotes = """
AGC TUNING (no delay)
Delay over response test
Loss of generation in area 1 at t=2
Delayed action by area 2
AGC in both areas
"""

# Simulation Parameters Dictionary
simParams = {
        'timeStep': 1.0, # seconds
        'endTime': 60.0*8, # seconds
        'slackTol': 1, # MW
        'PY3msgGroup' : 3, # number of Agent msgs per AMQP msg
        'IPYmsgGroup' : 60, # number of Agent msgs per AMQP msg
        'Hinput' : 0.0, # MW*sec of entire system, if !> 0.0, will be calculated in code
        'Dsys' : 0.0, # Damping
        'fBase' : 60.0, # System F base in Hertz
        'freqEffects' : True, # w in swing equation will not be assumed 1 if true
        # Mathematical Options
        'integrationMethod' : 'rk45',
        # Data Export Parameters
        'fileDirectory' : "\\delme\\200109-delayScenario1\\", # relative path from cwd
        'fileName' : 'SixMachineDelayStep1',
        'exportFinalMirror': 1, # Export mirror with all data
        'exportMat': 1, # if IPY: requies exportDict == 1 to work
        'exportDict' : 0, # when using python 3 no need to export dicts.
        'deleteInit' : 0, # Delete initialized mirror
        'assumedV' : 'Vsched', # assummed voltage - either Vsched or Vinit
        'logBranch' : True,
}
```

```

savPath = r"C:\LTD\pslf_systems\sixMachine\sixMachineTrips.sav"
dydPath = [r"C:\LTD\pslf_systems\sixMachine\sixMachineDelay.dyd"]
ltdPath = r".\testCases\200109-delayScenario1\sixMachineDelayStep1.ltd.py"
```

Figure C.1: An example of a full .py simulation file.

```
# Format of required info for batch runs.
debug = 0
AMQPdebug = 0
debugTimer = 0

simNotes = """
agc with deadband and nz, area 2 perturbance TLB 4
"""

# Simulation Parameters Dictionary
simParams = {
    'timeStep': 1.0,
    'endTime': 60.0*10,
    'slackTol': 1,
    'PY3msgGroup' : 3,
    'IPYmsgGroup' : 60,
    'Hinput' : 0.0, # MW*sec of entire system, if !> 0.0, will be calculated in code
    'Dsys' : 0.0, # Untested
    'fBase' : 60.0, # System F base in Hertz
    'freqEffects' : True, # w in swing equation will not be assumed 1 if true
    # Mathematical Options
    'integrationMethod' : 'rk45',
    # Data Export Parameters
    'fileDirectory' : "\\delme\\200325-smFinal\\", # relative path from cwd
    'fileName' : 'smAGCt4Ex1',
    'exportFinalMirror': 1, # Export mirror with all data
    'exportMat': 1, # if IPY: requies exportDict == 1 to work
    'exportDict' : 0, # when using python 3 no need to export dicts.
    'deleteInit' : 0, # Delete initialized mirror
    'assumedV' : 'Vsched', # assummed voltage - either Vsched or Vinit
    'logBranch' : True,
    }

savPath = r"C:\LTD\pslf_systems\sixMachine\sixMachineLTD.sav"
```

```
dydPath = [r"C:\LTD\pslf_systems\sixMachine\sixMachineLTD.dyd"]
ltdPath = r".\testCases\200325-smFinals\smAGCt4Ex1.ltd.py"
```

Figure C.2: Required .py file for external AGC event with conditional ACE.

```
# Perturbances
mirror.sysPerturbances = [
    'gen 5 : step Pm 2 -150 rel',
    ]

mirror.NoiseAgent = ltd.perturbance.LoadNoiseAgent(mirror, 0.05, walk=True, delay=0,
↪  damping=0, seed=11)

# Balancing Authorities
mirror.sysBA = {
    'BA1':{
        'Area':1,
        'B': "0.9 : permax", # MW/0.1 Hz
        'AGCActionTime': 30.00, # seconds
        'ACEgain' : 1.0,
        'AGCType':'TLB : 4', # Tie-Line Bias
        'UseAreaDroop' : False,
        'AreaDroop' : 0.05,
        'IncludeIACE' : True,
        'IACEconditional': True,
        'IACEwindow' : 30, # seconds - size of window - 0 for non window
        'IACEscale' : 1/5,
        'IACEdeadband' : 0, # Hz
        'ACEFiltering': 'PI : 0.04 0.0001',
        'AGCDeadband' : None, # MW? -> not implemented
        'GovDeadbandType' : 'nldroop', # step, None, ramp, nldroop
        'GovDeadband' : .036, # Hz
        'GovAlpha' : 0.016, # Hz - for nldroop
        'GovBeta' : 0.036, # Hz - for nldroop
        'CtrlGens': ['gen 1 : 0.5 : rampA',
                     'gen 2 1 : 0.5 : rampA',
                     ]
        },
    'BA2':{
        'Area':2,
        'B': "0.9 : permax", # MW/0.1 Hz
        'AGCActionTime': 45.00, # seconds
```

```
        'ACEgain' : 1.0,
        'AGCType':'TLB : 4', # Tie-Line Bias
        'UseAreaDroop' : False,
        'AreaDroop' : 0.05,
        'IncludeIACE' : True,
        'IACEconditional': True,
        'IACEwindow' : 45, # seconds - size of window - 0 for non window
        'IACEscale' : 1/3,
        'IACEdeadband' : 0, # Hz
        'ACEFiltering': 'PI : 0.04 0.0001',
        'AGCDeadband' : None, # MW? -> not implemented
        'GovDeadbandType' : 'nldroop', # step, None, ramp, nldroop
        'GovDeadband' : .036, # Hz
        'GovAlpha' : 0.016, # Hz - for nldroop
        'GovBeta' : 0.036, # Hz - for nldroop
        'CtrlGens': ['gen 3 : 1.0 : rampA',]
        },
    }
```

Figure C.3: Required .ltd file for external AGC event with conditional ACE.

```
mirror.NoiseAgent = ltd.perturbance.LoadNoiseAgent(mirror, 0.05, walk=True, delay=0,
↪  damping=0, seed=11)

# Balancing Authorities
mirror.sysBA = {
    'BA1':{
        'Area':1,
        'B': "0.9 : permax", # MW/0.1 Hz
        'AGCActionTime': 30.00, # seconds
        'ACEgain' : 1.0,
        'AGCType':'TLB : 4', # Tie-Line Bias
        'UseAreaDroop' : False,
        'AreaDroop' : 0.05,
        'IncludeIACE' : True,
        'IACEconditional': True,
        'IACEwindow' : 30, # seconds - size of window - 0 for non window
        'IACEscale' : 1/5,
        'IACEdeadband' : 0, # Hz
        'ACEFiltering': 'PI : 0.04 0.0001',
        'AGCDeadband' : None, # MW? -> not implemented
        'GovDeadbandType' : 'nldroop', # step, None, ramp, nldroop
```

```
        'GovDeadband' : .036, # Hz
        'GovAlpha' : 0.016, # Hz - for nldroop
        'GovBeta' : 0.036, # Hz - for nldroop
        'CtrlGens': ['gen 1 : 0.5 : rampA',
                     'gen 2 1 : 0.5 : rampA',
                     ]
        },
    'BA2':{
        'Area':2,
        'B': "0.9 : permax", # MW/0.1 Hz
        'AGCActionTime': 45.00, # seconds
        'ACEgain' : 1.0,
        'AGCType':'TLB : 4', # Tie-Line Bias
        'UseAreaDroop' : False,
        'AreaDroop' : 0.05,
        'IncludeIACE' : True,
        'IACEconditional': True,
        'IACEwindow' : 45, # seconds - size of window - 0 for non window
        'IACEscale' : 1/3,
        'IACEdeadband' : 0, # Hz
        'ACEFiltering': 'PI : 0.04 0.0001',
        'AGCDeadband' : None, # MW? -> not implemented
        'GovDeadbandType' : 'nldroop', # step, None, ramp, nldroop
        'GovDeadband' : .036, # Hz
        'GovAlpha' : 0.016, # Hz - for nldroop
        'GovBeta' : 0.036, # Hz - for nldroop
        'CtrlGens': ['gen 3 : 1.0 : rampA',]
        },
    }

# Load and Generation Cycle Agents
mirror.sysGenerationControl = {
    'BPATDispatch' : {
        'Area': 1,
        'startTime' : 2,
        'timeScale' : CTRLtimeScale,
        'rampType' : 'per', # relative percent change
        'CtrlGens': [
            "gen 1 : 0.5",
            "gen 2 1 : 0.5",
            ],
        # Data from: 12/11/2019 PACE
        'forcast' : [
            #(time , Precent change from previous value)
```

```
            (0, 0.0),
            (1, 5.8),
            (2, 8.8),
            (3, 9.9),
            (4, 4.0),
            ],
        }, #end of generation controller def
    'CAISODispatch' : {
        'Area': 2,
        'startTime' : 2,
        'timeScale' : CTRLtimeScale,
        'rampType' : 'per', # relative percent change
        'CtrlGens': [
            "gen 4 : 1.0",
            ],
        # Data from: 12/11/2019 PACE
        'forcast' : [
            #(time , Precent change from previous value)
            (0, 0.0),
            (1, 0.7),
            (2, 7.5),
            (3, 11.2),
            (4, 4.4),
            ],
        }, #end of generation controller def
    }

mirror.sysLoadControl = {
    'BPATDemand' : {
        'Area': 1,
        'startTime' : 2,
        'timeScale' : CTRLtimeScale,
        'rampType' : 'per', # relative percent change
        # Data from: 12/11/2019 BPAT
        'demand' : [
            #(time , Precent change from previous value)
            (0, 0.000),
            (1, 3.2),
            (2, 8.2),
            (3, 9.3),
            (4, 3.8),
        ] ,
    }, # end of demand agent def
    'CAISODemand' : {
```

```
        'Area': 2,
        'startTime' : 2,
        'timeScale' : CTRLtimeScale,
        'rampType' : 'per', # relative percent change
        # Data from: 12/11/2019 CAISO
        'demand' : [
            #(time , Precent change from previous value)
            (0, 0.000),
            (1, 3.0),
            (2, 7.0),
            (3, 10.5),
            (4, 4.4),
        ] ,
    },# end of demand load control definition
}# end of loac control definitions

# Definite Time Controller Definitions
mirror.DTCdict = {
    'bus8caps' : {
        'RefAgents' : {
            'ra1' : 'bus 8 : Vm',
            'ra2' : 'branch 8 9 1 : Qbr', # branches defined from, to, ckID
            },# end Referenc Agents
        'TarAgents' : {
            'tar1' : 'shunt 8 2 : St',
            'tar2' : 'shunt 8 3 : St',
            'tar3' : 'shunt 8 4 : St',
            'tar4' : 'shunt 8 5 : St',
            'tar5' : 'shunt 8 6 : St',
            }, # end Target Agents
        'Timers' : {
            'set' :{ # set shunts
                'logic' : "(ra1 < 1.0)", # or (ra2 < -26)",
                'actTime' : 30, # seconds of true logic before act
                'act' : "anyOFFTar = 1", # set any target off target = 1
            },# end set
            'reset' :{ # reset shunts
                'logic' : "(ra1 > 1.04)",# or (ra2 > 26)",
                'actTime' : 30, # seconds of true logic before act
                'act' : "anyONTar = 0", # set any target On target = 0
            },# end reset
            'hold' : 90, # minimum time between actions
            }, # end timers
        },# end bus8caps
```

```
153     'bus9caps' : {
154         'RefAgents' : {
155             'ra1' : 'bus 9 : Vm',
156             'ra2' : 'branch 8 9 1 : Qbr', # branches defined from, to, ckID
157             },# end Referenc Agents
158         'TarAgents' : {
159             'tar1' : 'shunt 9 2 : St',
160             'tar2' : 'shunt 9 3 : St',
161             'tar3' : 'shunt 9 4 : St',
162             'tar4' : 'shunt 9 5 : St',
163             'tar5' : 'shunt 9 6 : St',
164             }, # end Target Agents
165         'Timers' : {
166             'set' :{ # set shunts
167                 'logic' : "(ra1 < 1.0)",
168                 'actTime' : 45, # seconds of true logic before act
169                 'act' : "anyOFFTar = 1", # set any target off target = 1
170             },# end set
171             'reset' :{ # reset shunts
172                 'logic' : "(ra1 > 1.04)",
173                 'actTime' : 45, # seconds of true logic before act
174                 'act' : "anyONTar = 0", # set any target On target = 0
175             },# end reset
176             'hold' : 120, # minimum time between actions
177             }, # end timers
178         },# end bus8caps
179     }# end DTCdict
```

Figure C.4: Required .ltd file for forecast demand scenario with noise and deadbands.