DQ0 Dynamics—Software Manual

Yoash Levron and Juri Belikov

May 2, 2019

The software website can be found at:
http://a-lab.ee/projects/dq0-dynamics

Contents

1 The basics

New users may wish to start with the tutorial on Section 6

Welcome!

Please take a minute to understand the basics.

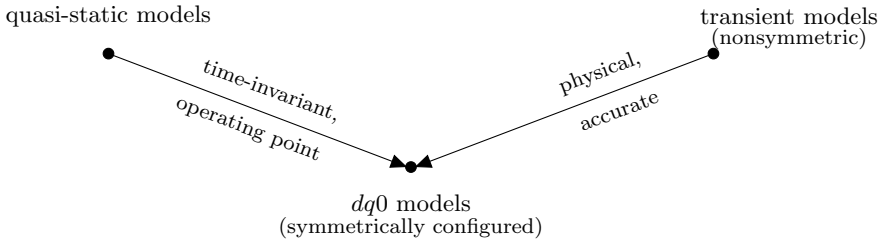
What this tool does

This is a free software tool for analyzing the dynamics of power systems based on $dq0$ signals. It is designed to simulate and analyze power systems that include several generators and loads, and possibly a large transmission network. The software provide tools for constructing dynamic models of the system components, and enables analysis in the frequency domain or the time domain. The manual and software provide simple explanations and examples that can help one get started, even if he has no prior knowledge on power system dynamics, or the $dq0$ transformation.

When to use this tool

Several approaches exist for modeling the dynamic behavior of three-phase power systems. Transient simulations that use the *abc* frame of reference describe the system by means of physical quantities, and thus offer high accuracy and flexibility. This approach is often the most general, since it applies to nonsymmetric systems, and is valid over a wide range of frequencies. Another popular approach is to model the power system using time-varying phasors, often by using the network power flow equations. This approach has many benefits, one of them is that the transmission network is described by means of purely algebraic equations. However, time-varying phasors are only applicable at low frequencies, under the assumption that the system is quasi-static.

A solution that complements these two well-known approaches is to model large power systems on the basis of $dq0$ quantities. This approach is not as general as *abc*-based models, and is advantageous mainly when the network and units are symmetrically configured[1]. However, $dq0$ models combine two properties of interest: similar to transient models, $dq0$-based models are derived from physical models, and are therefore accurate at high frequencies, so the assumption of a quasi-static network is not required. In addition, similarly to time-varying phasors, $dq0$ models are time-invariant. This property allows to define an operating point, and enables small-signal analysis.



This software exploits these properties, and uses $dq0$-based models for:

- **Simulation of complete networks**. The software allows to describe the dynamics of complete power systems that include the transmission network, generators, and loads, based on $dq0$ quantities.
- **Transient simulations** of symmetrically configured power systems. The main advantages in this case is that AC signals are mapped to constant signals at steady-state, so a large simulation step time may be used.

[1] By definition, symmetrically configured networks produce symmetric three-phase currents, if fed by symmetric three-phase voltages. For instance, a transmission network with identical circuit in each of the three-phases is symmetrically configured.

- **Small-signal analysis**. The $dq0$-based models are time-invariant, and therefore allow definition of an operating point. Primary applications are stability analysis and controller design.
- Dynamic analysis of **large-scale networks**. The software uses state-space models that are minimal order and sparse, and are therefore suitable for large systems.

Note that a primary limitation of this software is that the transmission network must be symmetrically configured. For instance, transient analysis of an asymmetric fault is not supported.

Contact

Any questions? Please contact us at
https://yoash-levron.net.technion.ac.il/

2 How this software operates

The software describes power systems by means of signal-flow diagrams, in which each component is modeled by $dq0$ quantities. An example is illustrated in Figs. 1 and 2.

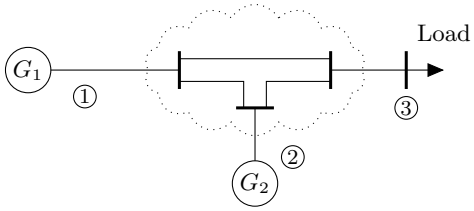


Figure 1: A single-line diagram of a 3-bus system.

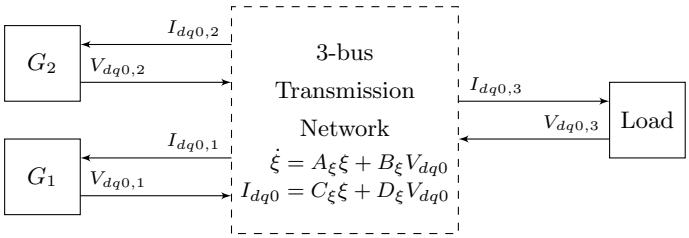


Figure 2: Signal-flow diagram of a 3-bus system.

In Fig. 2 the blocks on the left are dynamic models of the generators, the block in the middle is the dynamic model of the transmission network, and block on the right is a dynamic model of the load. These different components are modeled as follows:

- **The transmission network**. Transmission networks are represented by linear state-space models with voltage inputs and current outputs. As an example, the dynamic model of a 3-bus network is given by

$$\begin{aligned} \dot{\xi} &= A_\xi \xi + B_\xi u \\ y &= C_\xi \xi + D_\xi u, \end{aligned}$$

  where ξ is the state vector, and the input and output vectors are

$$\begin{aligned} u &= [v_{d,1}, v_{d,2}, v_{d,3}, v_{q,1}, v_{q,2}, v_{q,3}, v_{0,1}, v_{0,2}, v_{0,3}]^\mathsf{T}, \\ y &= [i_{d,1}, i_{d,2}, i_{d,3}, i_{q,1}, i_{q,2}, i_{q,3}, i_{0,1}, i_{0,2}, i_{0,3}]^\mathsf{T}, \end{aligned}$$

  in which $v_{d,n}, v_{q,n}, v_{0,n}$ are the $dq0$ voltages of bus n, and $i_{d,n}, i_{q,n}, i_{0,n}$ are the $dq0$ currents injected from the nth unit into bus n.

  The software automatically constructs the matrices $A_\xi, B_\xi, C_\xi, D_\xi$ based on the network data. This can be done using the graphical user interface, or directly from the Matlab command line using the function `ssNetw`, `ssNetwSym`, or `ssNetwMatPower`. More details can be found at Sections 7 and 9.

  The resulting state-space models are of minimal order, and use sparse system matrices, and are therefore suitable for large power systems.

- **Generators (power sources)**. We use the term generators to specify power sources in general. These may include classic synchronous machines, or renewable sources such as photovoltaic inverters or wind turbines. Generators are modeled based on $dq0$ models available in the literature. As an example, the $dq0$ model of a synchronous machine is detailed in Section 9.

  One problem that arises when considering the classic $dq0$ models is that they are typically based on local reference frames, and therefore linking different models to each other is not straightforward. For instance, in synchronous machine models the $dq0$ transformation is typically defined in respect to the rotor angle. However, since different machines may rotate at different frequencies, these models cannot be directly

linked to each other, since the $dq0$ signals describing each machine are defined in respect to the machine's local angle.

We approach this problem by modeling the network and its components using a $dq0$ transformation that is based on a unified reference frame, which rotates with a fixed frequency[2] of $\omega_s = 2\pi f_s$. To this end, the $dq0$ signals associated with each generator are converted from the local reference frame to the unified reference frame. In addition, the state-space model describing the transmission network is also based on this unified reference frame. Since all models are defined in respect to the same unified reference frame, a model of the complete power system may be constructed. Further details may be found in Section 9.

- **Loads**. We use the term loads to specify power consumers in general. Loads are represented by two types of dynamic models. Simple loads are modeled by shunt impedances, which are typically embedded into the transmission network. More complex loads (for instance, synchronous motors) are described by detailed $dq0$-based models, similarly to generators. In this case $dq0$ signals associated with each load are converted from the load's local reference frame to the unified reference frame.

Based on these models, the user may create a complete model of the power system in Simulink, and perform transient simulations. The software also provides several functions dedicated to small-signal and stability analysis. The main functions are listed in Section 7, and several examples are provided in Section 8.

[2]The frequency ω_s is chosen as follows: if there is an infinite bus in the system, ω_s is selected as the frequency of the infinite bus. If no generator is large enough to be modeled as an infinite bus, then ω_s is equal to the steady-state system frequency. In this case $dq0$ signals will be constant at steady-state, and the system will have well-defined equilibrium point. Other choices of ω_s are possible, but the resulting $dq0$ models will be time varying, and therefore will not have an equilibrium point.

3 License

Essentially this is an open-source software. Anyone can use and distribute it (while citing it properly), but the authors are not responsible for any damage.

The exact terms of the license are detailed as follows:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the Technion nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

This software is provided by the copyright holders and contributors "as is", and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the copyright owner or contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business Interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

Please note that the GUI part of the software is licensed differently: The GUI is based on the open-source project TinyCAD, which is licensed under LGPLv2.1 or LGPLv3. Also note that the GUI uses Python, which uses the GPL license, and the Python software packages Decorator and NetworkX, both distributed with the BSD license.

4 Installing the software

Installation and use of this software requires familiarity with the basic operation of Matlab, including setting up your Matlab path.

System requirements

- **Matlab & Simulink** must be installed.
- Several examples require the **MatPower** add-on, which is freely available on-line at:
  http://www.pserc.cornell.edu/matpower/
  This is not needed if you are only getting started.

Basic installation—without a graphical user interface

- Download the software files from Matlab central and copy them to a directory of your choice. The files are available at

  "Toolbox for Modeling and Analysis of Power Networks in the DQ0 Reference Frame"
  http://www.mathworks.com/matlabcentral/fileexchange/58702

- Setup the directory in your MATLAB path. In the MATLAB, go to *File >Set Path...* and click on *Add with Subfolders...* Now, select the directory that contains the DQ0 dynamics folder.

Note: make sure that the library is added to the Matlab path with its subfolders.

Installation with a graphical user interface

Following the basic installation, the user may install the Graphical User Interface (GUI). Most examples in this software package do not require the GUI. However, the GUI may help new users, and is necessary for the tutorial (in Section 6).

The GUI can be downloaded from:

https://github.com/dq0Dynamics/dq0Dynamics

There are two versions to the software, a 32-bit and a 64-bit version. Choose to install whichever suits your hardware. Python 2.7 is required for full functionality, and is therefore integrated as part of DQ0 Dynamics installation. In case the user already has Python 2.7 installed on his computer, he may choose to not reinstall it. However, the recommended and default setting is to install it.

Note for Python 3 users: Python 2.7 is required for full functionally, so please check the Python 2.7 box in the installation wizard.

5 Graphical user interface

5.1 Getting to know the environment

The initial layout can be seen in Fig. 3.

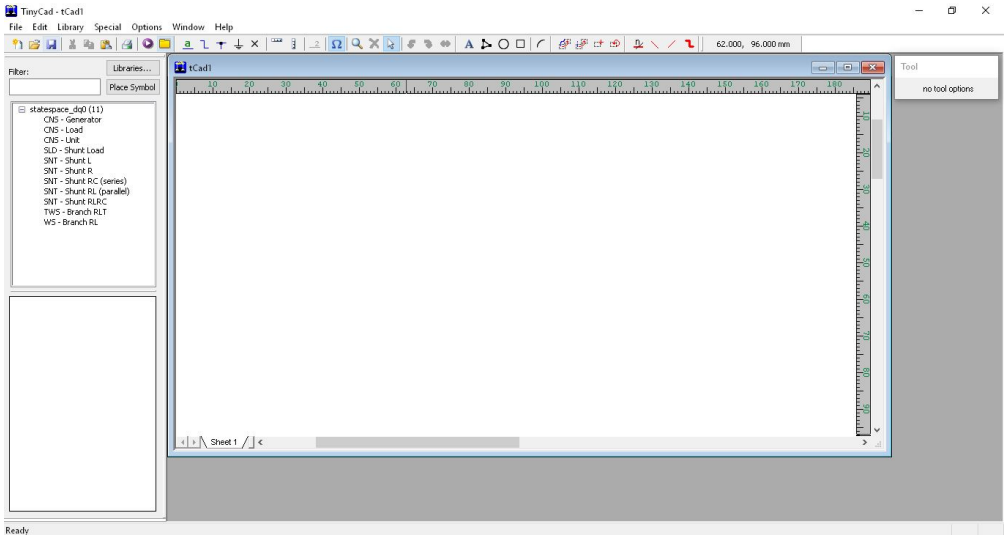


Figure 3: Initial layout.

First let us become more familiar with the top ribbon and explain the tools at our disposal. The top ribbon can be seen in Fig. 4.

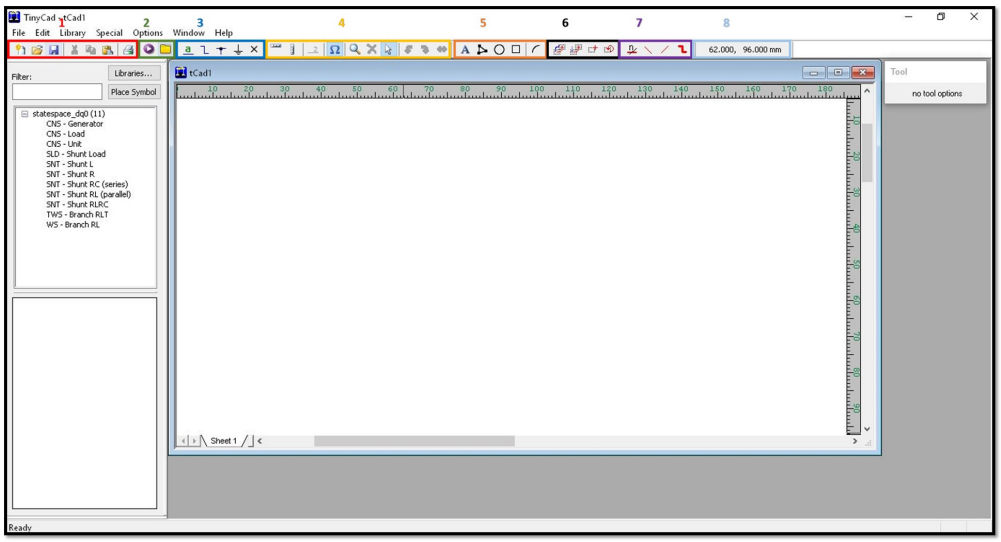


Figure 4: Top ribbon.

The top ribbon is divided into 8 sections:

1. Red frame:
    a. Create new project.
    b. Open existing project.
    c. Save current project.

d. Cut, copy, paste, print functions.

2. Green frame:

   a. Process Network: This is the run command to process the electrical network and run all necessary processes:
      - Create PCB Netlist.
      - Run python engine to create dq0 matrix inputs for Matlab simulation.

   b. Set work-directory for network processing: After processing your network, new files will be created that represent your electrical network. As a default, these files are saved to your DQ0 installation directory. However, it is possible to change this directory by clicking this button. Once pressed, a black command window will prompt you to insert the path of your new working directory. Once a valid path is inserted the working directory will be changed and the outputs of dynamics dq0 will be rerouted to this location.

3. Blue frame:

   a. Add label to a wire.
   b. Add wire to design.
   c. Add junction.
   d. Add power.
   e. Add no connect to wire.

4. Yellow frame:

   a. Horizontal and vertical rulers.
   b. Add symbol pin (not in use).
   c. Symbol picker (toggle cell library panel).
   d. Toggle Zoom in/out of design.
   e. Delete components in design.
   f. Toggle edit design.
   g. Rotate or flip components of design.

5. Orange frame: Add various extras to design:

   a. Text, line polygons, ellipses, rectangles, arcs.

6. Black frame: Block manipulation:

   a. Drag, move, duplicate, rotate.

7. Purple frame: Bus manipulation:

   a. Name, join, place.

8. Gray frame: Mouse pointer coordinates.

The red frame in Fig. 5 contains the symbol library panel. The reduced symbol library can be found here and will be used to drag and drop components into the design.

Further information about the symbols and their properties can be found in Section 5.2. The red frame in Fig. 6 marks the work area. Opening a new project or starting a new project opens a new work area. Various work areas can be opened simultaneously. However, processing the network (green frame in Fig. 4) will run on the active work area that is in the foreground. Additionally, each work area may have various sheets open. This is advantageous when building a design with hierarchical symbols which will be explained shortly.

The tool options window seen in Figure 8 shows the data, parameters and properties for highlighted symbols. Once a symbol is chosen, the tool options window is the most efficient way to see its data and change the values of its parameters.

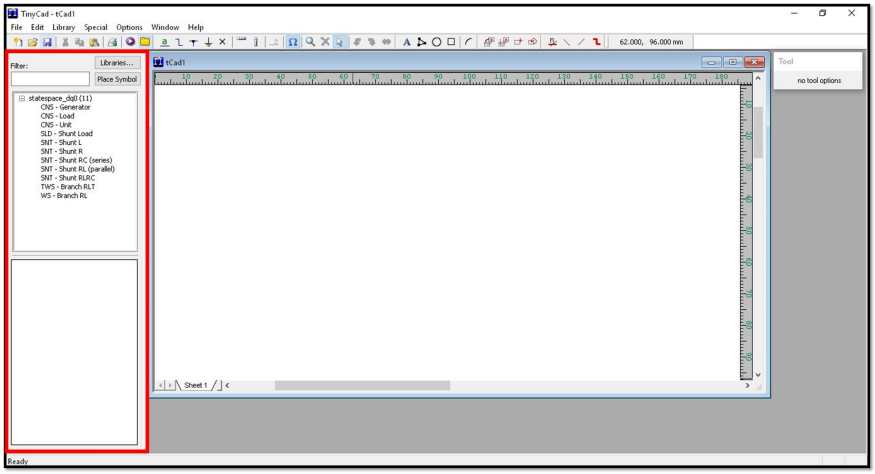


Figure 5: Symbol library panel.

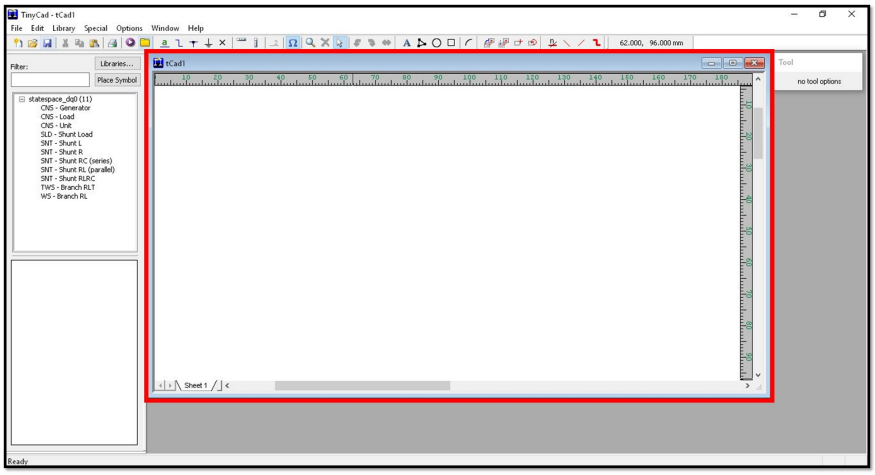


Figure 6: Work area.

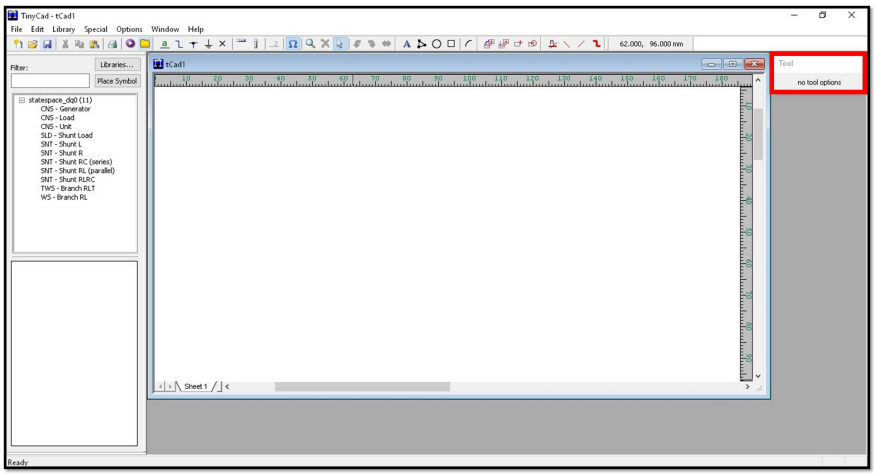


Figure 7: Tool options window.

Two drop down menus that deserve a mention are the Library and Special menus, seen in Figs. 8 and 9, respectively. The library menu allows you to add additional libraries and save newly created libraries in your design. Although all necessary libraries are supplied, if the need should arise for new elements in the design, this is the sure way to add them. The Special menu allows for various operations:

1. The first two, Create PCB Netlist and Check Design Rules are automatically run by the Process Network button (green frame in Fig. 4) and therefore do not require the user to run these individually. However the possibility remains for users to independently check the design rules during the implementation stages of their electrical network if they deem the step necessary.

2. 'Generate Symbol References' is a useful tool that will be explained in detail in the following section.

3. 'Create Parts List' allows users to create a file (.txt or .csv) containing the parts that are used in their design for future reference.

4. The following two features are supported by Dynamics DQ0 but not by the Python parser and therefore will not be successfully translated into dq0 coordinates. We recommend not to use them:

    a. 'Insert Another Design as a Symbol' allows users to insert another design as a black box to be used in their current design. This is especially useful for hierarchical networks with large subsystems.

    b. 'Add Hierarchical Symbol' opens a new special sheet that allows the users to treat it as a black box in their design. If the labels on the pins of the 'Hierarchical Symbol Sheet' match the labels on the pins of the main sheet for the design Dynamics DQ0 will automatically connect the two when processing the network.

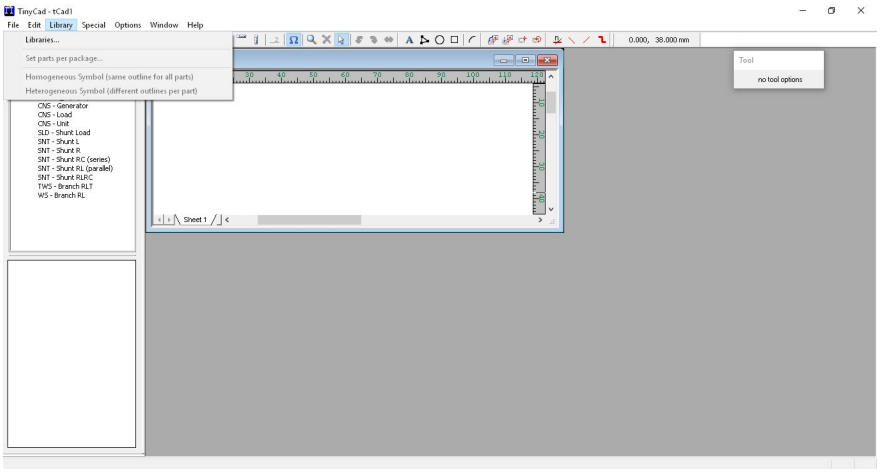


Figure 8: Library drop down menu.

5.2 Library cells

5.2.1 Overview

There are three families of cells provided for the user in the built-in symbol library.

1. Consumers (CNS): these are the end nodes of your electrical grid. The three options are generator, loads and units, see Fig. 10. The user may add properties to this type of component in the tools option window. For example:

    a. Parameter: 'name', Value: 'my generator'.

    b. Parameter: 'Power', Value: '123.456'.

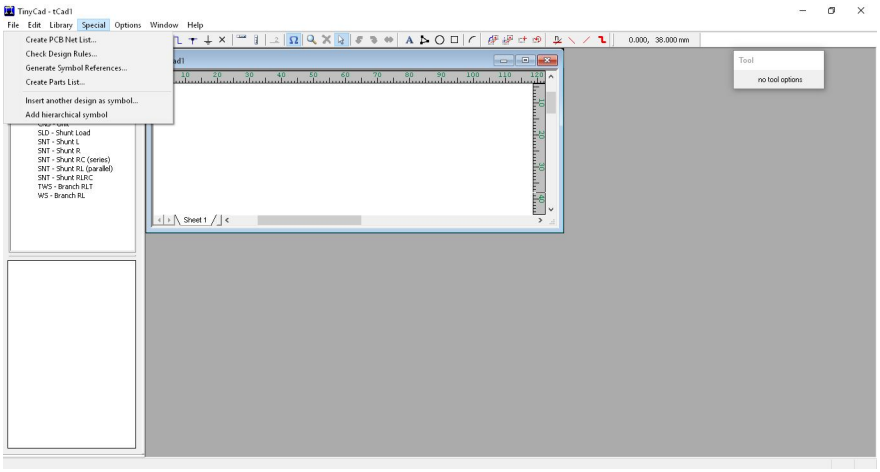


Figure 9: Special drop down menu.

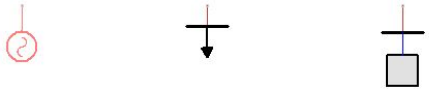


Figure 10: Library consumers.

These parameters are stored in the '.mat' file (dq0 inputs) under the object `unit_params` and are represented in cell arrays. Each index in the array represents the CNS in the bus with the matching number (to the index). For example, `unit_params(3).name` will contain the value of the parameter `name` of the CNS component of bus number 3. Note that numerical numbers are stored as numbers and alphanumerical names as strings. Another point to be noted is that users should not rely on the length of `unit_params` to determine the number of buses. For instance, if there are five buses but no CNS units in the design, the length of `unit_params` will be zero.

2. Branches (TWS/WS): branches connect different consumers and have a built in resistance and inductance in series. There are two different types of branches, one with and one without a transformer, see Fig. 11.

Figure 11: Library branches.

3. Shunts (SLD/SNT): in order to create low resistance paths, the design supports the use for shunts, see Fig. 12.

Once a symbol has been placed in the work area, the tool options window will show all of the pertinent information about the symbol as shown in Fig. 13. In this case we picked a branch symbol with no transformer. In the tool options window the user can view/modify the symbol properties:

1. The symbol orientation: this can be toggled to an up, down, left or right position with mirroring capabilities.

2. The symbol reference name. As a design rule, symbols must have a reference name. For the user's convenience, the 'Generate Symbols Reference' (see Fig. 9), will automatically create legal netlist names

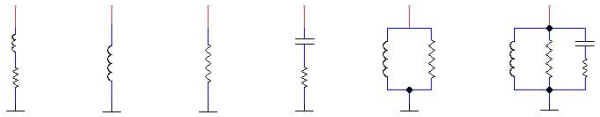


Figure 12: Library shunts.

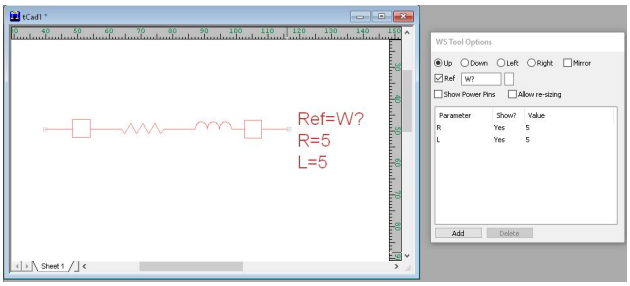


Figure 13: Tool options window.

for the symbols once the design is finished. If symbols do not get assigned a reference name, the built in Design Rule Checker will prompt the user to do so. Each single reference name must be unique. Note: When initially placed, a symbol will have a reference name in the format '<symbol_name>?'. This is an illegal symbol name according to the Dynamics dq0 design rules and should be changed as explained.

3. The symbol's parameters and values. In our example, the symbol has a resistor and an inductor, users may change the values of these parameters for the symbol here.

5.3 Design Rules

Bus Labels
By convention, all buses (added by using the 'add wire to design' button in the blue frame in Fig. 4) must have the same numbered name 'busX' with ascending order, starting from '1'. For example, the first wire should be called 'bus1', the second 'bus2', etc. If this convention is not maintained, the Python Engine will prompt the user to fix the names of the required wires. Note that the name must be 'bus'.

Bus Connectivity
A bus may contain:

1. No more than one CNS symbol.
2. No more than one SNT symbol.
3. No more than one SLD symbol.

In other words, a bus (meaning a blue wire from the blue frame in Fig. 4), may connect at most to one CNS, one SLD, and one SNT.

Branches Connectivity
Between 2 adjacent buses, there is exactly 1 component of Branch type: WS/TWS. In other words, WS/TWS are not allowed to be connected in parallel (to each other). Note, however, that a bus is allowed to be linked to many other buses, each connected with a single Branch (WS/TWS).

Symbol References
As previously mention, when a symbol is first created, it is not assigned a reference name. Reference names can either be individually given to each symbol or given general names by running the 'Generate Symbol References' process under the 'Special' drop down menu. If this process is not executed, the built in Design Rule Checker prompts the user to do so.

Bus Numbers
Every wire should be assigned with a bus number. To do this, use the 'Label' button on the top ribbon (blue frame in Fig. 4). Once we click on the wire button, the tool options window should turn into the 'Label Tool Options' window. Here we can type the names of our buses. Buses must be numbered as 'bus1', 'bus2', etc.

6 Tutorial

In this tutorial we will learn how to design and simulate a 9-bus power system.

Note: If you have not installed the software and the GUI, please do so now (Section 4).

The tutorial is divided to four parts:

- Designing the system using the GUI.
- Processing the network data, and creating an equivalent state-space model.
- Simulating the network dynamics in the time domain using Simulink.
- Creating a small-signal model and analyzing it in the frequency domain.

6.1 Designing the network using the Graphical-User-Interface

This section explains how to construct the 9-bus network, step-by-step. If you wish to skip this section, the final design may be loaded from `Tutorial_1_9bus.dsn`, which may be found in the folder `Tutorial_1`.

Open the GUI software (DQ0 dynamics). The initial screen should show an empty work area. This is where we will build our network.

We will start with some preliminary steps. First, set the page size. To do so, select 'File' from the menu, and then 'Page Setup'. Now choose a page size as you like (this has no effect on the simulation).

Next, set the current working directory (folder). Press the Library symbol (Fig. 14). When asked whether to change the working directory answer 'yes', 'y', or just press enter. Then, specify a library of your choice.

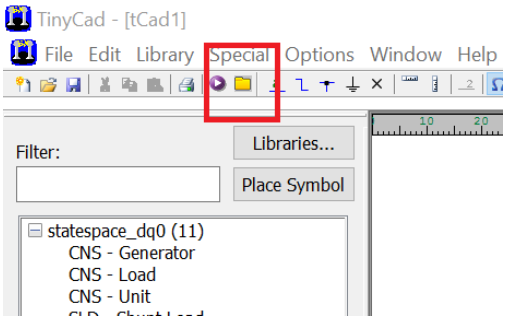


Figure 14: Changing the current working directory

Now, choose a file name for the current design. Select 'File' from the menu, and then 'Save As'. Navigate to the library you chose in the previous step, and save the design. While working, save any time by clicking Ctrl+S.

We will now start building the network. Remember that at any time you may open `Tutorial_1_9bus.dsn`, to see the final design. The first step will be to place the network branches (transmission lines), as seen in Fig. 16. This is done by choosing, from the component library on the left, the component named 'WS - Branch RL'. This is a basic branch that includes a resistor and an inductor in series. Click it, then click 'Place Symbol', and put the component in the main design area. Components may be rotated by clicking Ctrl-R. Notice that three branches contain transformers. These can be placed by selecting the component 'TWS - Branch RLT'.

Next, set the branch parameters. Left-click on any component, and edit its 'R', 'L', and 'Ratio(X)' fields. The specific values are shown in Fig. 16. The units for the resistance and inductance are not specified, but in this tutorial we will assume SI units, meaning that resistances are given in Ohm units, and inductances are given in Henry units. The units will become important later, when the network data is processed in Matlab, but currently these are just numbers. The 'Ratio(X)' field controls the transformer ratio, as shown in Fig. 15. The design should now look as in Fig. 16.

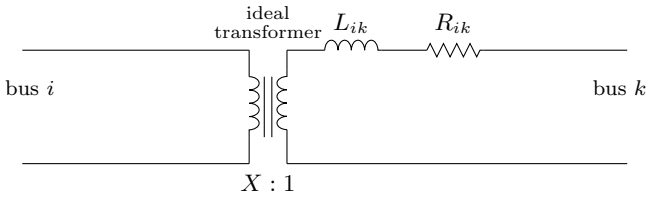


Figure 15: Branch with transformer, resistor and inductor.

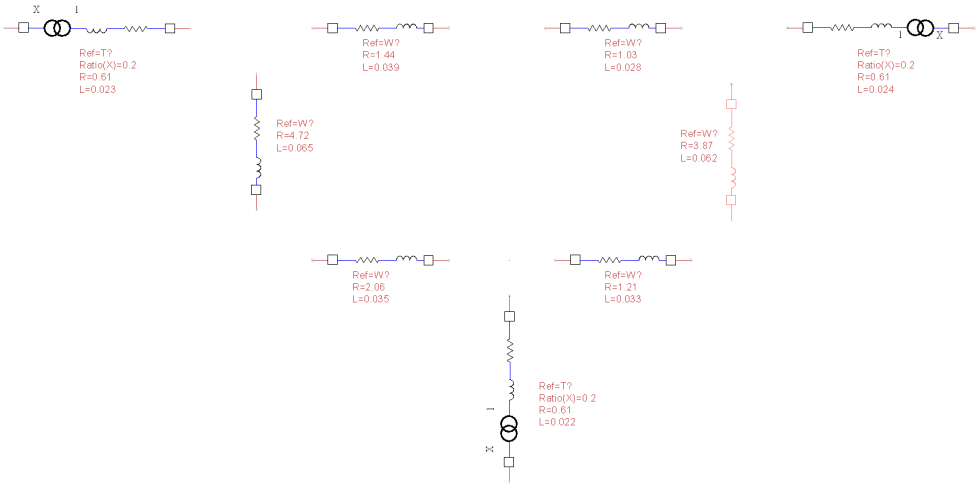


Figure 16: The design after placing the network branches.

The next step is to connect the generators. Generators and other complex units are represented by two types of components, either 'CNS - Generator' or 'CNS - Unit'. These two components are identical, except for a different graphical symbol. Typically 'CNS - Generator' represents physical generators, and 'CNS - Unit' represents other types of complex units. The 9-bus network contains three generators: Generator 1 is an infinite bus, meaning an AC voltage source with a constant frequency and amplitude. In this tutorial this model describes a connection to a larger network, which can provide as much power as needed. Generators 2,3 are synchronous machines.

Table 1: Generator data.

	Type	Component	Parameters
Gen. 1	infinite bus	CNS - Unit	$V = 22$
Gen. 2	synchronous machine	CNS - Generator	$P = 163$ $V = 22$
Gen. 3	synchronous machine	CNS - Generator	$P = 85$ $V = 22$

Place the generators in the design area, as shown in 17. Now click on the generators, to edit their parameters, according to Table 1. For instance, in Generator 2, define a parameter named 'P', and set its value to 163, then define the parameter 'V', and set its value to 22. In this tutorial these parameters mean that the generator's single-phase output power in steady-state is 163 MW, and the no-load voltage is 22 kV (RMS). Generally, parameters can have any name and any value, and their meaning is freely chosen by the user. The GUI does not process these parameters, but only sends them 'as is' to the Matlab environment.

After adding the generators, the design should be similar to Fig. 17.

Next we will add loads to the network. One option to represent loads is to use the component 'CNS - Load'. This component is exactly identical to 'CNS - Generator' or 'CNS - Unit', except for its different graphical

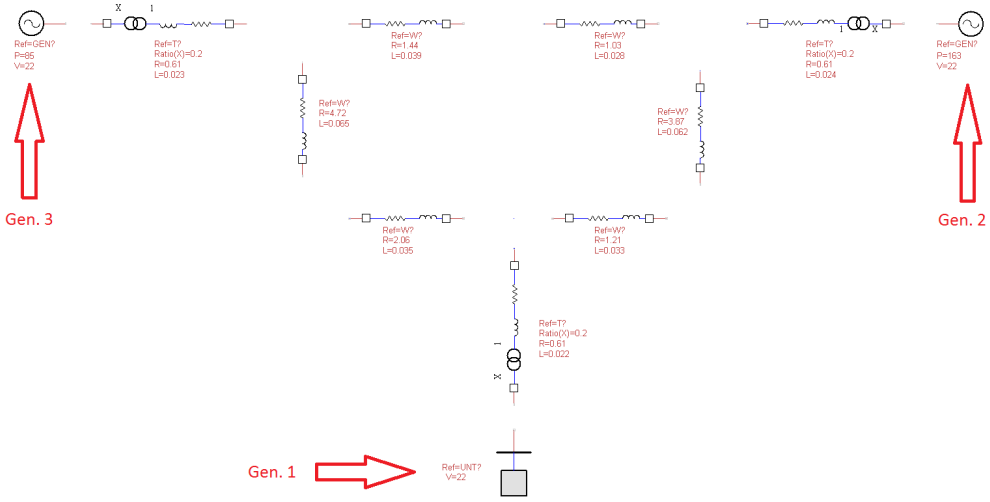


Figure 17: Adding generators to the network.

symbol, and different meaning to the user. When using 'CNS - Load' the GUI does not process the load parameters, but only delivers them 'as is' to the Matlab environment for further processing.

Another option to represent loads is by using shunt elements (elements connected to ground). This type of model is simpler, since the loads are then considered an integral part of the transmission network.

We therefore continue by adding loads using shunt elements. To do so, select the component 'SLD - Shunt Load', and place it in the design area. Edit the load parameters as shown in Fig. 18. In general, the units of resistances and inductances are arbitrary, but in this tutorial we use Ohm and Henry units.

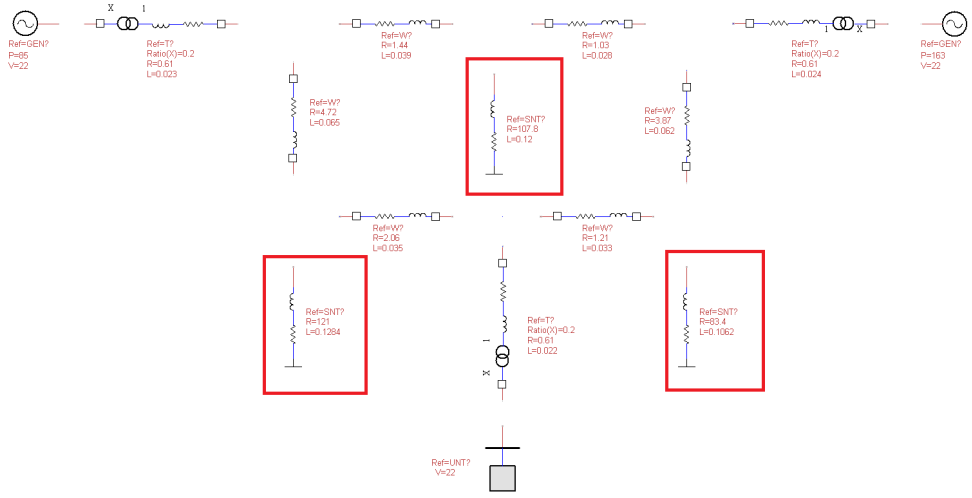


Figure 18: Adding loads to the network.

Now select the Wire tool from the menu (Fig. 19), and connect the components to each other. The final result is shown in Fig. 20

Few final steps are needed to finalize the design:

- **Assign symbol references.** This is necessary to avoid errors when processing the network. To do so, in the menu, click 'Special', and then 'Generate Symbol References', and then 'OK'. Notice that each component now has a unique identifier, shown in its 'Ref' field.
- **Assign bus numbers.** From the menu select the Label tool (Fig. 21), enter a bus number, for instance

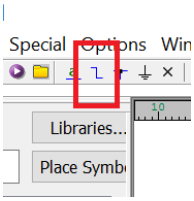


Figure 19: The wire tool.

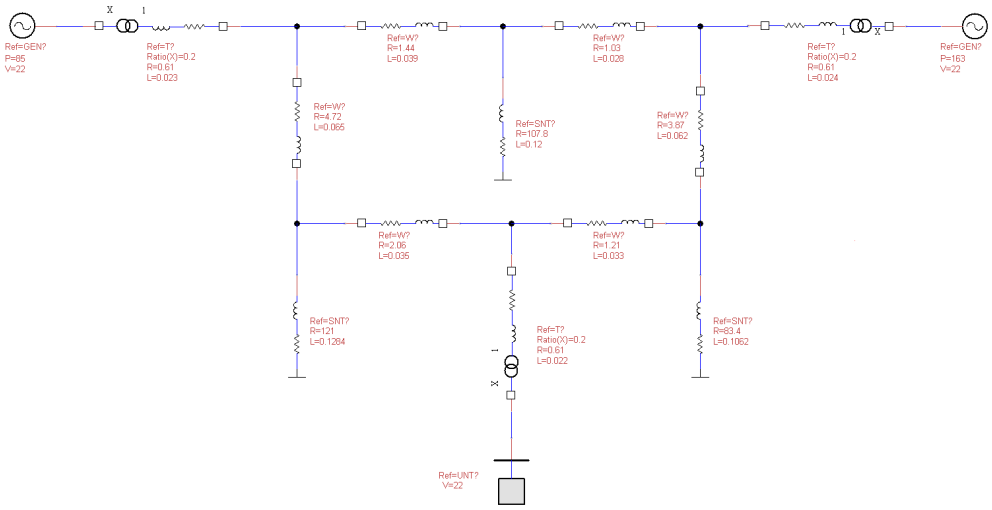


Figure 20: After connecting the components using the wire tool.

'bus1', and then click the desired wire. Assign a bus number to every node. Buses must be numbered 'bus1', 'bus2', etc.

Following these steps, the design should appear as in Fig. 22.

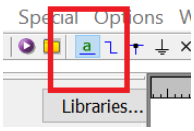


Figure 21: The Label tool.

We are now ready to process the design, and to send the data to the Matlab environment. From the menu, click the 'Process network' tool (Fig. 23). When running this tool for the first time, it will open a temporary Matlab command window, and this may take a few seconds. The software now checks for design rule violations, as explained in Section 5.3. If no errors are found, a file named `sdq0_inputs.mat` will be created in the current working directory. After the process is complete, you may close the Matlab command window, and the GUI.

6.2 Processing the network data in Matlab

This section explains how to process the network data, using Matlab. The script `Tutorial_1.m` contains all the commands used in this section, and may be used as a reference. This script may be found in the folder `Tutorial_1`.

Our objective in this section is twofold: first, we will create a state-space model representing the dynamic

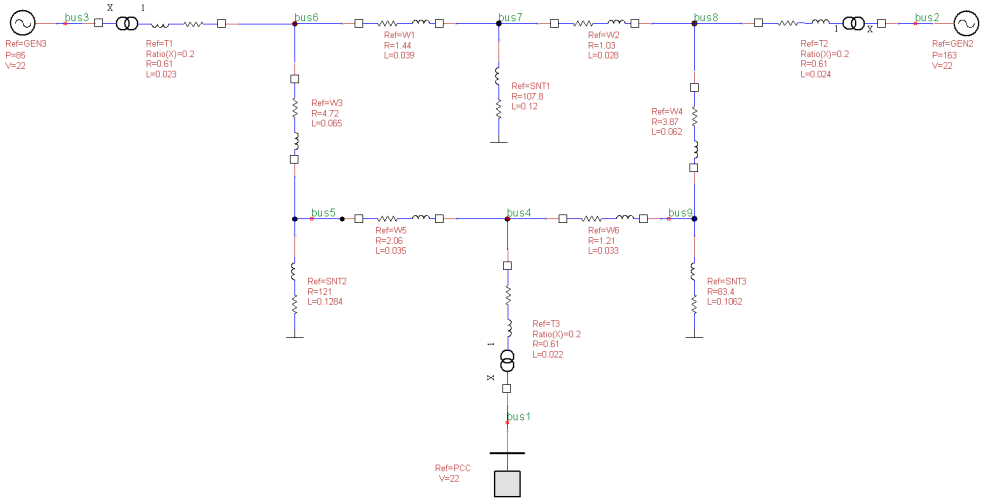


Figure 22: The final design, after assigning symbol references and bus numbers.

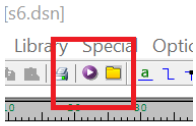


Figure 23: The 'Process network' tool

behavior of the transmission network and loads. Second, we will import the generator parameters, and use them to prepare data for the Simulink model.

Open Matlab. The first step is to set the current folder. This may be done directly through the Matlab environment, or by using the `cd` command. The selected folder should contain the file `sdq0_inputs.mat`, that has been created in the previous section.

Start by loading the network data. To do so, type in the Matlab command window

```
load sdq0_inputs.mat;
```

Now type

```
whos
```

to see the variables. A short explanation about each of the variables that has been loaded is provided in Table 2.

As shown in Table 2, the variables may be divided to four groups. The 1st and 2nd groups describe the transmission network, and the shunt loads. Currently you may skip the details, since these variables will be automatically processed. Two variables of special interest are `subset` and `unit_params`. The vector `subset` contains a list of buses that are connected to either a generator, load, or unit. Note that shunt loads are not included in this list. The structure `unit_params` contain unit parameters. For instance, the generator connected to bus 2 contains two parameters: $P = 163$ and $V = 22$. (These parameters were defined in the GUI). To see these parameters type

```
unit_params{2}
```

Similarly, `unit_params{X}` contains the parameters of the unit connected to Bus X.

Table 2: Variables in `sdq0_inputs.mat`

`R_bus`
`L_bus`
`C_bus`
`Rtil_bus`
`Rb`
`Lb`
`Tau`
	parameters of the transmission network.
	details may be found in the description of the function `ssNetw()` in Section 7

`R_bus_SLD` `L_bus_SLD`	parameters of shunt L-R loads. details may be found in the description of the function `balLoadsRL()` in Section 7
`subset`	vector containing buses that are connected to either a generator, load or unit
`unit_params`	structure containing the unit parameters

Next, type the following code:

```
ws = 2*pi*50;
N = size(Lb,1);
```

The variable `ws` is the nominal system frequency (in rad/s), or the frequency of the infinite bus (in our tutorial, this is bus no. 1). The variable `N` is the number of buses, in our case `N`=9.

Now type the following code, or select it in `Tutorial_1.m` and run it by clicking 'F9':

```
[A1,B1,C1,D1] = ssNetw(R_bus,L_bus,C_bus,Rtil_bus,Rb,Lb,Tau,ws);
[A2,B2,C2,D2] = balLoadsRL(R_bus_SLD,L_bus_SLD,ws);
[A3,B3,C3,D3] = mergeParlNetw(A1,B1,C1,D1,A2,B2,C2,D2);
[A,B,C,D] = elimBus(A3,B3,C3,D3,subset);
```

Note: In case of an error, please check that the software library is added to the Matlab path *with its subfolders*, as explained in Section 4.

This code creates a state-space model (matrices `A,B,C,D`) representing the network and shunt loads. The code is standard, and can be used as is with any design. We currently skip the details. If you are interested, a description of the functions may be found either in Section 7, or in the Matlab code.

The state-space model describes the dynamics of the network and shunt loads in terms of $dq0$ quantities. Its inputs are the d, q, and 0 components of the bus voltages, and its outputs are the d, q, and 0 components of the bus currents. The input and output vectors include **only these buses that are connected to generators, loads, or units**, that is, all the buses in the vector `subset`. In our tutorial, the network has 9 buses, however, only three buses (1,2,3) are connected to generators. The resulting state-space model is

$$\dot{\xi} = A_\xi \xi + B_\xi u$$
$$y = C_\xi \xi + D_\xi u,$$

where ξ is the state vector, and the input and output vectors are

$$u = [v_{d,1}, v_{d,2}, v_{d,3}, v_{q,1}, v_{q,2}, v_{q,3}, v_{0,1}, v_{0,2}, v_{0,3}]^\mathsf{T},$$
$$y = [i_{d,1}, i_{d,2}, i_{d,3}, i_{q,1}, i_{q,2}, i_{q,3}, i_{0,1}, i_{0,2}, i_{0,3}]^\mathsf{T},$$

in which $v_{d,n}, v_{q,n}, v_{0,n}$ are the $dq0$ voltages of bus n, and $i_{d,n}, i_{q,n}, i_{0,n}$ are the $dq0$ currents injected from the nth generator into bus n.

Observe now the structure of the matrices `A,B,C,D`. First, note that these matrices are represented as sparse matrices, since they have many zero elements. Type

```
A
```

to obtain a list of the nonzero elements. Type

```
spy(A)
```

to see the location of these elements in the matrix. Now type

```
size(A)
```

The size of the state matrix is 18×18, meaning that the state-space model has 18 states. Similarly, the size of `B` is 18×9, since the model has 9 inputs, and the size of `C` is 9×18, since the model has 9 outputs.

Next, we will use the parameters in `unit_params` to prepare the necessary data for Simulink. Type the following commands:

```
v1_d = (2^0.5)*1e3*unit_params{1}.V;
v1_q = 0;
v1_0 = 0;
```

These three lines define the $dq0$ voltages of bus 1, in Volts. Remember that `unit_params{1}` contains the parameters of the unit connected to Bus 1. In our case this is the infinite bus, which has a single parameter `unit_params{1}.V`. Notice that `unit_params{1}.V` is given in kV-rms, and is thus multiplied by $\sqrt{2}\cdot 10^3$.

Next, type the following commands:

```
Ert = 1e3*[unit_params{2}.V , unit_params{3}.V];
Pg = 1e6*[unit_params{2}.P , unit_params{3}.P];
```

`Ert` contains the no-load output voltages of the synchronous machines in Vrms, and `Pg` contains the nominal single-phase output power of the machines in W.

Now type

```
poles = 2;
J = 0.0125*Pg/ws;
Kd = 20*J*ws;
```

This code defines several default parameters of the synchronous machines. `poles` is the number of poles of the synchronous machine, `J` is the rotor moment of inertia in kg·m^2, and `Kd` is the machine's damping factor in W·s. Notice that these values are normalized to the machine's nominal power.

Next, type

```
Nu = length(subset);
units_to_net = [1:3:3*Nu , 2:3:3*Nu , 3:3:3*Nu];
[~,net_to_units] = sort(units_to_net);
```

These commands create the routing vectors `units_to_net` and `net_to_units`, which define how models connect to each other in Simulink. `units_to_net` defines the order of voltage signals connected from the generators to the network, and `net_to_units` defines the order of current signals connected from the network back to the generators.

6.3 Time-domain simulations

We are now ready to simulate our 9-bus power system in the time domain, using Simulink.

For this step you will need the file `Tutorial_1_sim.slx`, which may be found in the folder `Tutorial_1`. This file should be copied to the Matlab current folder, if it is not already there.

Open the Simulink model by typing in the Matlab command line:

```
Tutorial_1_sim
```

Let's start by observing the different parts of the model, in order to understand its structure.

On the right side of the model, you should find a relatively small block named 'transmission network, DQ0 state-space model'. This is the state-space model representing the transmission network and shunt loads. The inputs to the model are the $dq0$ voltages of buses 1,2,3, and the outputs of the model are the $dq0$ currents of buses 1,2,3. Click the block twice to view its parameters. The matrices defining the model are `A,B,C,D`, generated in the previous section. The operator `full(x)` is needed since the 'state-space' block does not accept sparse matrices, and requires the inputs to be full matrices.

On the left side, there are three blocks. The top one (smaller than the others) is named 'Bus 1, infinite bus'. This block models the voltage source representing the infinite bus. This voltage source has a constant frequency ω_s, and a constant amplitude, and is therefore modeled by constant voltages in the $dq0$ reference frame. Click on the block to open it. Inside you should find the three constants `v1_d, v1_q, v1_0`, which have been defined in the previous section.

Still on the left side of the model, find the block named 'Bus 2, synchronous generator'. This block models the synchronous machine, connected to bus 2. Observe the block inputs and outputs. The block has two inputs. The bottom one is `idq0`, which is the $dq0$ currents injected from the generator to the transmission network. The upper input is the mechanical power applied to the rotor. Notice that the steady-state value of the mechanical power is three times `Pg(1)`, that is, three times the nominal single-phase power, as defined in the previous section.

The block also has several outputs: `vdq0` is the $dq0$ voltages on bus 2, and `delta` is the rotor angle in respect to a reference frame rotating with $\omega_s t$. Two additional outputs are the rotor electric frequency, and the single-phase (electric) power produced by the generator.

Now click on the block to open it. Inside you will find the relatively simple model of the synchronous machine, as developed in the mathematical background (see Section 9 and Fig. **??**). Several blocks are marked in blue. These contain the constants that have been defined in the previous section.

The last block on the left is named 'Bus 3, synchronous generator'. This block is identical in structure to the block just described.

Now move to the right again, and observe the two 'Selector' blocks, connected to the network state-space model. These block reorder the signals, so that the generators $dq0$ signals match the state-space model inputs and outputs. Click on the 'Selector' blocks to see their parameters. Note that the order of signals is defined by the two vectors `units_to_net` and `net_to_units`, defined in the previous section. `units_to_net` defines the order of voltage signals connected from the generators to the network, and `net_to_units` defines the order of current signals connected from the network back to the generators.

Observe now the 'measurements' block, above the network state-space model. This block accepts the network $dq0$ voltages and currents, and computes several useful quantities: the vectors `P` and `Q` are the single-phase active and reactive powers injected to each of the buses. The powers are positive when injected from the unit into the bus. The vectors `Vm_RMS` and `Vph_deg` are the bus voltage magnitudes and voltage phases (in Vrms and degrees). At steady-state, these four quantities constitute the network power-flow solution.

Lastly, observe the block 'convert signals DQ0 to ABC'. As it name implies, this block converts signals in the $dq0$ reference frame to signals in the abc reference frame. Since the simulation is based on $dq0$ quantities, this block is only used to display the abc signals.

Now let's run the simulation...

Click the 'run' button, and observe the time changing in the display 'sim time'. The simulation time is infinite, so the simulation will run continuously until stopped.

First we shall examine the differences between $dq0$ and abc signals. Go to the block 'convert signals DQ0 to ABC', and open the two scopes connected to its input and output. Now pause the simulation. Zoom-in on the abc signals, and watch their sinusoidal shape. Note that the $dq0$ signals are constant at steady-state. When you finish this test, run the simulation again.

Next, we will change the system operating point (power flow solution). On the left side, while the simulation is running, toggle the manual switch to change the mechanical power input of Generator 2. This will add 90 MW of mechanical power. While toggling the switch, observe the changes in active power (in the network measurements, on the right). When the switch is toggled, 30 MW are added to the active power of Bus 2, and at the same time, 30 MW are subtracted from the active power on Bus 1, which is the infinite bus. The factor of 3 ($90 = 3 \times 30$) is because the mechanical power corresponds to a three-phase output, while the measurements power is single-phase. When you finish this test, toggle the manual switch back to its initial position.

Now we shall observe the system transient behavior.

Go to the block named 'Bus 2 synchronous generator', and open the scope connected to its `delta` output. This output measures the rotor angle, in respect to a reference frame rotating with $\omega_s t$. Notice that in steady-state, the rotor angle is constant. Now, while the simulation is running, toggle the manual switch, and watch the resulting transients. You may have to scale the Y-axis of the scope to observe the full transient.

On the same block, open the scope connected to the 'rotor electric frequency' output. Toggle the manual switch and watch the transients. Remember to scale the Y-axis. Notice that after the transient the frequency always converges back to 50 Hz, This is due to the effect of the infinite bus.

On the same block, open the scope connected to the 'Active Power' output. Toggle the manual switch and watch the transients.

When you finish these last tests, toggle the manual switch back to its initial position.

6.4 Small-signal model and Frequency-domain analysis

A major advantage of $dq0$-based models is that they are time-invariant, so constant inputs produce constant outputs. As a result, $dq0$-based models have a well-defined operating point, and may be linearized to obtain a small-signal model. This linear small-signal model may be then analyzed in the Laplace domain, or in the frequency domain.

In this section we will demonstrate how to linearize our 9-bus power system around an operating point. We will generate Bode plots of the resulting small-signal model, and compute its poles and zeros. Type

```
Tutorial_1_sim
```

to open the Simulink model.

The first step is to define the small-signal model inputs and outputs. In this simulation these are already defined, but nevertheless we will explain how to define them. Go to the block named 'Bus 2 synchronous generator', and left-click on its output 'rotor electric frequency'. Now right-click on this signal, and a floating menu will appear. From this menu choose 'Linear Analysis Points', and observe that the 'Output Measurement' option is selected. This signal is an output of our small-signal model. Notice that the active powers of Generators 2 and 3 are also defined as outputs.

Now left-click on the input 'mechanical power'. Right-click, and choose 'Linear Analysis Points' again. Observe that the 'Input Perturbation' option is selected. This signal is the input signal of our small-signal model.

The next step is to linearize the model. To do so, type the following commands in the Matlab command window:

```
io = getlinio(bdroot);
linsys = linearize(bdroot,io,3);
```

This causes the simulation to run for 3 seconds, to obtain an operating point, and then the model is linearized in the neighborhood of this operating point. You may now type

```
linsys
```

to observe the resulting small-signal model. To obtain the state-space matrices type

```
[Ap,Bp,Cp,Dp]=ssdata(linsys);
```

Check the dimensions of these matrices (for instance, by using `size(Ap)`). According to the dimensions, the small-signal model has 16 states, 1 input, and 3 outputs.

To view the **Bode plot** of the small-signal model type

```
figure; bode(linsys);
```

To view the **poles** of the small-signal model type

```
figure; pzmap(linsys);
```

7 Overview of main functions

Here is a list of several main functions. Detailed explanations can be found in the Matlab code. see Section 9 for more details.

Root functions

bodeSparse()

```
function [Mag, Ph] = bodeSparse(A, B, C, D, u, w)
% BODESPARSE Bode frequency response of a sparse state-space model.
%
% Usage:
%       [Mag, Ph] = bodeSparse(A, B, C, D, u, w)
%
% where
%       A, B, C, D - system matrices
%       u - input's serial number. For single input input=1.
%           For multi-input models select 1 <= input <= size(B,2)
%       w - frequency range [rad/s]
%
% Outputs:
%       Mag - magnitude [dB]
%       Ph - phase [deg]
```

This function computes the frequency response (bode plot) of a state-space model. The system matrices A, B, C, D can be large and sparse. This function performs the same task as Matlab's function `bode`, but can handle large dynamic systems with sparse system matrices. Run-time is primarily determined by the number of nonzero elements in the state matrix A.

- For systems with a single output `Mag(k)` is the magnitude at frequency `W(k)`. For systems with multiple outputs `Mag(:,k)` is the vector of magnitudes at frequency `W(k)`, such that `Mag(j,k)` corresponds to the jth output.
- For systems with a single output `Ph(k)` is the magnitude at frequency `W(k)`. For systems with multiple outputs `Ph(:,k)` is the vector of phases at frequency `W(k)`, such that `Ph(j,k)` corresponds to the jth output.

closedLoop()

```
function [A, B, C, D] = closedLoop(Aksi, Bksi, Cksi, Dksi, unitData)
% CLOSEDLOOP Constructs a linear state-space model of a complete power system:
% transmission network, generators, and loads.

% Usage:
%       [A, B, C, D] = closedLoop(Aksi, Bksi, Cksi, Dksi, unitData)
%
% where
%       Aksi, Bksi, Cksi, Dksi - system matrices of the network model
%       unitData - a cell array (Nx1) that stores the system matrices of the
%           units. Each cell unitData{n} is a structure with the following fields:
%       unitData{n}.A = matrix An
%       unitData{n}.B = matrix Bn
%       unitData{n}.C = matrix Cn
%       unitData{n}.D = matrix Dn
%       unitData{n}.G = matrix Gn
%
% Outputs:
%       A, B, C, D - system matrices of the complete power system
```

The model generated by this function is a feedback connected model, describing the dynamics of the network together with the dynamics of the units connected to it (generators/loads). Since the units are linearized in the neighborhood of the system's operating point (power flow solution), the resulting model is linear, and inputs and outputs are small-signals that represent deviations from the system's operating point. This function enables:

- Low-complexity modeling of the system dynamics. The resulting state-space model has a minimal number of states and uses sparse system matrices, and therefore can be used to efficiently model the dynamics of large power systems in the neighborhood of an operating point.
- Stability analysis. Stability can be tested by examining eigenvalues of the state matrix A. The power system is stable at an operating point if all eigenvalues of the resulting state matrix A have negative real parts.

All the models are given in the state-space form, and use $dq0$ signals. Models of the network and various units can be created automatically by other functions from the package.

createQS()

```
function [Aqs, Bqs, Cqs, Dqs] = createQS(A, B, C, D)
% CREATEQS creates a quasi-static model from a dynamic dq0 model.
%
% Usage:
%       [Aqs, Bqs, Cqs, Dqs] = create_quasi_static(A, B, C, D)
%
% where
%       A, B, C, D - system matrices of dq0 model, computed by functions
%           'ssNetw' or 'ssNetwSym'. These matrices can be full, sparse,
%           numeric, or symbolic
%
% Outputs:
%       Aqs, Bqs, Cqs, Dqs - system matrices of the quasi-static model that
%       are numeric or symbolic, depending on the inputs
```

This function creates a quasi-static model from a $dq0$ model. The quasi-static model is a pure gain model, obtained by the approximation $s \to 0$, i.e., using the so-called low frequency approximation. Under this approximation the system is usually modeled using time-varying phasors. Since the model represents a static gain, all the constructed matrices except `Dqs` are empty.

createYbus()

```
function Ybus = createYbus(A, B, C, D)
% CREATEYBUS Creates admittance matrix Ybus based on system matrices of
% the full dq0 model.
%
% Usage:
%       Ybus = createYbus(A, B, C, D)
%
% where
%       A, B, C, D - system matrices of dq0 model
%
% Outputs:
%       Ybus - the network Ybus matrix
```

This function creates the traditional admittance matrix Y^{bus} based on the full $dq0$ state-space model as input. The admittance matrix is evaluated using the quasi-static approximation ($s \to 0$). Under this approximation Y^{bus} may be extracted from the equality: $[I_d, I_q]^\mathsf{T} = Y^{bus}[V_d, V_q]^\mathsf{T}$, where I_d, I_q, V_d, V_q are vectors from $\mathbb{R}^{N\times 1}$. This function is useful for validating the $dq0$ state-space model, since Y^{bus} can be computed directly from the network topology, and is provided by other software products (such as Matpower), it may be used as a *checksum* to validate the resulting system matrices A, B, C, D. The Y^{bus} is measured in 1/Ohm, and can be converted to the standard per-unit representation as follows: `Ybus_per_unit(i,j)=Ybus(i,j)*Vbase(i)*Vbase(j)/Pbase`, where `Vbase(i)` is the voltage base on bus i (in V), and `Pbase` is the power base (in W, common to all buses).

elimBus()

```
function [Ar, Br, Cr, Dr] = elimBus(A, B, C, D, subset)
% ELIMBUS eliminates disconnected buses. It produces a new equivalent
% dynamic model in which the currents in the buses being eliminated
% are forced to zero.
%
% Usage:
%       [Ar, Br, Cr, Dr] = elimBus(A, B, C, D, subset)
%
% where
%       A, B, C, D - matrices of the original model
%       subset - a vector of bus indices specifying the buses
%           to be included in the new model. Voltages and currents
%           of buses in this list are NOT eliminated, and will be
%           the inputs and outputs of the new model.
%
% Outputs:
%       Ar, Br, Cr, Dr - system matrices of the new dynamic model
```

The need to eliminate disconnected buses arises in several occasions:

- Sometimes certain buses are not connected to either a generator or a load. In this case the current injected into the bus is zero.
- Frequently loads are modeled as shunt elements and are integrated into the network model. In this case load buses appear as disconnected buses with zero current.
- In many scenarios there is a need to analyze the dynamics or stability of a certain subset of units in the network, typically only the generators. In such cases elimination of the disconnected buses results in a simpler dynamic model in which the inputs and outputs relate only to the required subset of buses. This is usually done after integrating the loads in the network model as shunt elements.

Bus elimination is achieved by controlling the inputs of the buses being eliminated, such that corresponding outputs are zeroed. Assume for instance that the ith input and ith output are eliminated. This is done using the dynamic model output equation $y = Cx + Du$. Since the matrix D of the $dq0$ model is diagonal, the output `y(i)` can be zeroed by controlling the ith input such that `u(i)=-(1/D(i,i))*Ci*x`, where `Ci` is the ith row in matrix C. The input `u(i)` and the output `y(i)` are eliminated, and will not appear in the new model.

Handling zero elements on the diagonal of D: `D(i,i)=0`. As described above, elimination is achieved by inverting elements on the main diagonal of matrix D. If several of these elements are zeros, the function transforms the state vector and computes a new dynamic model such that the corresponding rows in matrix C are also zero. In other words, if input/output i are being eliminated, and `D(i,i)=0`, then to zero the output the new state matrix C will have a zero row `Ci=0`, so that `Ci*x+Di*ui=0`, as required. This is done by a LU decomposition of the relevant rows in matrix C. The process is numerically efficient and stable with large sparse state matrices.

elimBusYbus()

```
function YbusR = elimBusYbus(Ybus, subset)
% ELIMBUSYBUS eliminates disconnected buses in the system admittance matrix Ybus.
%
% Usage:
%       YbusR = elimBusYbus(Ybus, subset)
%
% where
%       Ybus - original admittance matrix
%       subset - a vector of bus indices specifying the buses
%           to be included in the new model. Voltages and currents
%           of buses in this list are NOT eliminated, and will be
%           the inputs and outputs of the new model
%
% Outputs:
%       YbusR - the new admittance matrix
```

This function eliminates disconnected buses in the system admittance matrix Y^{bus}. It produces an new

equivalent Y^{bus} matrix for a network in which the currents of the eliminated buses are zeroed.

mergeParlNetw()

```
function [A, B, C, D] = mergeParlNetw(A1, B1, C1, D1, A2, B2, C2, D2)
% MERGEPARLNETW Merges two networks connected in parallel.
%
% Usage:
%       [A, B, C, D] = MERGEPARLNETW(A1, B1, C1, D1, A2, B2, C2, D2)
%
% where
%       A1, B1, C1, D1 - system matrices of network 1
%       A2, B2, C2, D2 - system matrices of network 2
%
% Outputs:
%       A, B, C, D - system matrices of merged network
```

This function merges two networks connected in parallel. The networks are described by state-space models. The model inputs (voltages in $dq0$ coordinates) are the same for both networks. The model outputs (injected currents in $dq0$ coordinates) are summed entry-wise.

ssNetw(), ssNetwSym(), ssNetwMatPower()

```
function [Aksi, Bksi, Cksi, Dksi] = ssNetw(R_bus, L_bus, C_bus, Rtil_bus, Rb, Lb, Tau, ws)
% SSNETW Generates a dynamic model of a three-phase balanced or
% symmetrically configured transmission network.
%
% Usage:
%       [Aksi, Bksi, Cksi, Dksi] = ssNetw(R_bus, L_bus, C_bus, Rtil_bus, Rb, Lb, Tau, ws)

% where
%       R_bus - vector Nx1, shunt resistance on each bus in Ohm
%       L_bus - vector Nx1, shunt inductance on each bus in H
%       C_bus - vector Nx1, shunt capacitance on each bus in F
%       Rtil_bus - vector Nx1, resistance in series with the shunt capacitance in Ohm
%       Rb - matrix NxN, the [i,k] element is the resistance on brach connecting
%         bus i and bus k
%       Lb - matrix NxN, the [i,k] element is the inductance (in H) on brach connecting
%         bus i and bus k
%       Tau - matrix NxN, indicates the branch transformer ratios
%       ws - nominal system frequency [rad/s]
%
% Outputs:
%       Aksi, Bksi, Cksi, Dksi - sparse system matrices
%
% See also: ssNetwSym, ssNetwMatPower
```

The above code represents abridged version taken from the function `ssNetw`, which does not support symbolic inputs. In order to use symbolic math, the function `ssNetwSym` can be used instead. In addition, the software contains a function `ssNetwMatPower`, which is implemented to construct state-space models of a power network represented by a MatPower database (`mpc`).

stepSparse()

```
function [y, t] = stepSparse(A, B, C, D, Tf, T, display)
% STEPSPARSE Computes the step response of a state-space model.
%
% Usage:
%        stepSparse(A, B, C, D, Tf, T)
%           when invoked with no output arguments, this function plots the
%           step response on the screen
%       [y, t] = stepSparse(A, B, C, D, Tf, T)
%           returns outputs, assuming that a unit step is
%           simultaneously applied to each of the inputs
%       [y, t] = stepSparse(A, B(:,m), C, D(:,m), Tf, T)
%           returns outputs, assuming that a unit step is
%           applied to the m'th input
%       [y, t] = stepSparse(A, B(:,m), C(p,:), D(p,m), Tf, T)
%           returns the p'th output, assuming that a unit step is
%           applied to the m'th input
%
% where
%       A, B, C, D - system matrices
%       Tf - simulation final time such that t\in[0,Tf]
%       T - numeric step size (sampling time)
%       display - computation progress is printed on screen (optional)
%
% Outputs:
%       t - time vector
%       y - the step response
%
% See also: ssNetw
```

This function performs the same task as Matlab's function `step`, but is designed for large dynamic systems with sparse system matrices. The function has been tested on multiple systems with up to 10^5 state variables, meaning `size(A)=[100e3,100e3]`. Run-time is primarily determined by the number of nonzero elements in the state matrix A. The step response is computed using the Bilinear transformation (also called Tustin's approximation, or Trapezoidal rule), which is a stiff implicit solution method for ODEs:

$$\begin{aligned} x(k) &= Q_1 x(k-1) + Q_2(u(k) + u(k-1))/2 \\ Q_1 &= (I - (T/2)A)^{-1}(I + (T/2)A) \\ Q_2 &= (I - (T/2)A)^{-1}BT. \end{aligned}$$

This method was selected since it maps stable poles in the s-domain to stable poles in the z-domain, and thus preserves the system stability. The original model is stable if and only if the approximated step response converges to a finite value.

The solver implemented in this file attempts to balance speed and memory requirements depending on the system size. For small systems, the inverse `invAT=inv(speyeNN-(T/2)*A)` is computed explicitly, to obtain fast computation speed. For large systems, `speyeNN-(T/2)*A` is decomposed using the LU decomposition, and the inverse is never computed explicitly. This typically enhances the accuracy and computation speed for very large systems, and also requires less memory.

Models

balLoadsRL()

```
function [A, B, C, D] = balLoadsRL(R_bus, L_bus, ws)
% BALLOADSRL Generates a dynamic model of balanced three-phase series RL loads.
%
% Usage:
%       [A, B, C, D] = BALLOADSRL(R_bus, L_bus, ws)
%
% where
%       R_bus - vector Nx1, resistance on each bus in Ohm
%       L_bus - vector Nx1, inductance on each bus in H
%       ws - the nominal system frequency [rad/s]
%
% Outputs:
%       A, B, C, D - sparse system matrices
```

This function generates simple dynamic model of a linear balanced three-phase load modeled by RL components connected in series.

longLine()

```
function [A, B, C, D] = longLine(Rx, Lx, Cx, len, N, ws)
% LONGLINE Implements an approximate (lumped) dynamic model of a long power line.
%
% Usage:
%       [A, B, C, D] = LONGLINE(Rx, Lx, Cx, len, N, ws)
%
% where
%       Rx - resistance per unit length [Ohm/m]
%       Lx - inductance per unit length [H/m]
%       Cx - capacitance per unit length [F/m]
%       len - transmission line length [m]
%       N - number of T sections
%       ws - nominal system frequency [rad/s]
%
% Outputs:
%       A, B, C, D - system matrices
```

This function constructs an approximate (lumped parameters based) model in the $dq0$ reference frame such that the model can be directly embedded into the main grid.

8 Examples

Here is a list of several examples. For each .m file there is a respective .slx file with assembled simulation scheme. Detailed explanations can be found in the Matlab code.

ex_get_started_here.m

This script demonstrates the use of several basic functions: `ssNetw`, `ssNetwSym`, `createYbus`, `createQS`, `elimBus`, `elimBusesYbus`, `longLine`.

ex2busStability.m

This script illustrates stability analysis of a 2-bus power system. This file demonstrates:

- How to use the function `closedLoop` to construct a state-space model of a complete power system (transmission network + generators/loads).
- How to test stability theoretically by examining eigenvalues of the state matrix A.

ex7busLongLine.m

This script simulates the dynamics of a 7-bus system with long transmission lines, synchronous generators, and photovoltaic generators. The simulation demonstrates the use of dq0 signals to model all the system components, and shows the effects of the long power line (especially delays) on the dynamics and stability of the system.

ex9busSG.m

This script simulates the dynamics of a 9-bus power system in the time domain. The system in this example includes three major blocks: the transmission network, the loads, and the generators. The system converges to a steady-state that matches its power flow solution. The user may trigger a dynamic response by defining an input step in mechanical power.

ex14busSGandPV.m

This script shows dynamic simulation of a 14-bus test case network, with physical synchronous machine models and photovoltaic inverters (renewable sources). The user may choose between transient simulation or small signal analysis.

exDroopA.m, exDroopB.m

These scripts illustrate how to use simple droop control and analyze dynamics and stability of large power systems with physical synchronous machines.

exLargeSysStability.m

This script presents the small-signal modeling and stability analysis of large power systems. This file demonstrates:

- How to model the small-signal dynamics of large power systems using $dq0$ signals.
- How to evaluate the system stability based on eigenvalues.
- How to compute step responses of the complete system.

Power systems ranging in size from 4 to 2383 buses are considered.

exMatPower.m

This script constructs dynamic models of power systems from the MatPower package in the $dq0$ reference frame.

syncMachVsSwing.m

This script provides comparison between the transient response of the physical and approximate (swing equation) synchronous machine models. Both models are tested when connected to an infinite bus.

The physical (accurate) model is based on the book "Electric Machinery" by Fitzgerald. The presented model is based on $dq0$ quantities, and is identical to the model presented in the book, except that the input voltages and output currents are represented in a references frame rotating with angle $\omega_s t$, (instead of θ - the rotor electrical angle), where ω_s is the frequency of the infinite bus. This change of coordinates allows direct connection of the model to the infinite bus, or to any other model (network/load/generator) that is represented in the same reference frame.

The approximated model is based on the (classical) swing equation, and includes the effects of a series synchronous inductance and armature resistance. Similar to the physical model, the input voltages and output currents are represented in a references frame rotating with angle $\omega_s t$.

9 Mathematical background

The mathematical background is explained in a series of lectures freely available here:
https://a-lab.ee/projects/dq0-dynamics#lectures