GridDyn

Sep 27, 2019

Basics

GridDyn is a power system simulator developed at Lawrence Livermore National Laboratory. The name is a concatenation of Grid Dynamics, and as such usually pronounced as "Grid Dine". It was create dto meet a research need for exploring coupling between transmission, distribution, and communications system simulations. While good open source tools existed on the distribution side, the open source tools on the transmission side were limited in usability either in the language or platform or simulation capability, and the commercial tools while quite capable simply did not allow the access to the internals required to conduct the research. Thus the decision was made to design a platform that met the needs of the research project.

Building off of prior efforts in grid simulation, GridDyn was designed to meet the current and future needs of the various grid related research and computational efforts. It is written in C++ making use of recent improvements in the C++ standards. It is intended to be cross platform with regard to operating system and machine scale. The design goals were for the software to be easy to couple with other simulations, and be easy to modify and extend. It is very much still in development and as such, the interfaces and code is likely to change, in some cases significantly as more experience and testing is done. It is our expectation that the performance, reliability, capabilities, and flexibility will continue to improve as projects making use of the code continue and new ones develop. We expect there are still many issues so any bug reports or fixes are welcome. And hopefully even in its current state and as the software improves the broader power systems research community will find it useful.

CHAPTER 1

Getting Started

1.1 Prerequisites

GridDyn is written in C++ and makes use of a few external libraries not included in the released source code. External software packages needing installation prior to compilation of GridDyn include:

1. A modern C/C++ compiler for building
2. The `cmake` command for building
3. The `git` command for getting the code
4. Boost 1.61 or greater
5. Doxygen for building in-source documentation
6. KLU in SuiteSparse - on Linux the SuiteSparse package typically has KLU, on macOS the Homebrew suite-sparse package can be used

Currently supported compilers are:

- Visual Studio 2015
- GCC 4.9.3 or higher
- Clang 3.5 or higher (OpenMP must be off to use 3.4)
- Intel 16.0 (not well tested as of yet)

See *CMake Options* for a list of CMake configuration options to turn on/off different features.

1.2 Installation Notes

1.2.1 Mac

For building on macOS, use Homebrew and make sure git, CMake, suite-sparse, Boost, and OpenMP are installed.

1.2.2 Linux

Depending on the distribution, Boost or an updated version of it may need to be installed (the package in the package manager may be significantly outdated). SuiteSparse/KLU may need to be installed as well. Typically CMake is used to generate Makefiles, but it can also be used to generate Eclipse projects. `BOOST_INSTALL_PATH` and `SuiteSparse_INSTALL_PATH` may need to be user specified if they are not in the system directories. This can be done with the `cmake-gui` or the command line `cmake`. Then running `make` will compile the program. Running `make install` will copy the executables and libraries to the install directory.

1.2.3 Windows

GridDyn has been built with Visual Studio 2015 and MSYS2. The MSYS2 build is like building on Linux, and works fine with GCC, though the current Clang version on MSYS2 has library incompatibilities with some of the Boost libraries due to changes in GCC. I don't fully follow what the exact issue is but Clang won't work on MSYS2 to compile GridDyn unless SUNDIALS, Boost, and KLU are compiled with the same compiler, I suspect the same issue is also present in some other Linux platforms that use GCC 5.0 or greater as the default compiler. The SuiteSparse version available through pacman on MSYS2 seems to work fine.

For compilation with Visual Studio, Boost will need to be built with the same version as is used to compile GridDyn. Otherwise, follow the same instructions.

1.3 Running GridDyn

The main executable for GridDyn is built as *gridDynMain* and is intended to load and run a single simulation. The executables *testSystem*, *testComponents*, *testLibrary*, *testSharedLibrary*, and *extraTests* are test programs for the unit testing of GridDyn. A server mode for interactive sessions is a work in progress, but is not operational at the time of this release.

```
> ./gridDynMain --version
```

will display the version information.

```
> ./gridDynMain --h
```

will display available command line options.

Typical usage is:

```
> ./gridDynMain [options] inputFile [options]
```

The primary input file can be specified with the flag *–input* or a single flagless argument. Additional input files should be specified using *-i* or *–import* flags.

Command line only options:

--help Print the help message

--config-file arg Specify a config file to use

--config-file-output arg File to store current config options

--mpicount Setup for an MPI run, prints out a string listing the number of MPI tasks that are required to execute the specified scenario, then halts execution

--helics Setup for a HELICS run for a coupled co-simulation

--version Print version string

Configuration options:

-o, --powerflow-output filename File output for the powerflow solution. Extension specifies a type (.csv, .xml, .dat, .bin, .txt), unrecognized extensions default to the same format as .txt

-P, --param arg Override simulation file parameters, *-param ParamName=<val>*

-D, --dir directory Add search directory for input files

-i, --import filename Add import files loaded after the main input file

--powerflow-only Set the solver to stop after the power flow solution

--state-output filename File for saving states, corresponds to *–save-state-period*

--save-state-period arg Save state every N ms, -1 for saving only at the end

--log-file filename Log file output

-q, --quiet Set verbosity to zero and printing to none

--jac-output arg Powerflow Jacobian file output

--v, --verbose arg Specify verbosity output, 3=verbose, 2=normal, 1=summary, 0=none

-f, --flags arg Specify flags to feed to the solver, e.g. *–flags=flag1,flag2,flag3* no spaces between flags if multiple flags are specified or enclose in quotes

-w, --warn arg Specify warning level output for input file processing, 2=all, 1=important, 0=none

--auto-capture filename Automatically capture a set of parameters from a dynamic simulation to the specified file format determined by extension. Either .csv or .txt will record the output in csv format, all others will record in the binary format. The filename must be specified with *–auto-capture-period* if used.

--auto-capture-period arg Specifies the automatic capture period in seconds. If specified without a corresponding *–auto-capture* file, a file named *auto_capture.bin* is created.

--auto-capture-field arg Specify the fields ot be captured through the auto capture. Defaults to *auto*. Can be a comma or semicolon separated list, no spaces unless enclosed in quotes.

The configuration routine will look for and load a file named *gridDynConfig.ini* if it is available. It will also load any command line specified config file. The order of precedence is command line, user specified config file, then system config file (if available).

CHAPTER 2

GridDyn Components

Components in GridDyn are divided into three categories: *primary*, *secondary*, and *submodel*. Primary components include *buses*, *links*, *relays*, and *areas* and define the basic building blocks for power grid simulation. Secondary components are those which tie into busses and consume or produce real and reactive power. The two component types in the secondary category are *loads* and *generators*. Submodels are any other component in the system and can form the building blocks of other components. A majority of the differential equations in the dynamic simulations are found in submodels. Submodels include things such as *exciters*, *governors*, *generator models*, *control systems*, *sources*, as well as several others. There are a few other types of objects in GridDyn, but they generally are used for specific purposes and do not take part in the equations unless interfaced through another object. The component types currently available in GridDyn are detailed in another section.

The *Development Notes* section has information on the current development status of various components.

2.1 Buses

Buses form the nodes of a power system. They act as containers for secondary objects and attach to links. The default bus type is an AC bus which in typical operation would have 2 states (voltage and angle). 4 types of bus operation are available, *PQ*, *PV*, *slack*, and *fixed angle*. The practical value of fixed angle buses is unknown but was included for mathematical completeness and describes a bus whose angle and reactive power are known.

The residual equation used in the bus model take one of two forms

$$f_v(X) = \sum_{i=0}^{gens} Qgen_i(V,\theta,f) + \sum_{i=0}^{loads} Qload_i(V,\theta,f) + \sum_{i=0}^{lines} Qline_i(V,\theta,f)$$

for PQ and afix type buses and

$$f_v(X) = V - V_{target}$$

for PV and SLK type buses. The equations for θ are very similar

$$f_\theta(X) = \sum_{i=0}^{gens} Pgen_i(V,\theta,f) + \sum_{i=0}^{loads} Pload_i(V,\theta,f) + \sum_{i=0}^{lines} Pline_i(V,\theta,f)$$

for PQ and PV type buses and

$$f_\theta(X) = \theta - \theta_{target}$$

for fixed angle and SLK type buses.

The frequency can be either extracted from an active generator attached to the bus or computed as a filtered derivative of the angle. If it is computed the bus has an additional state as part of the dynamic calculations.

The bus model implemented in GridDyn also includes some ability to merge buses together to operate in node-breaker type configurations. At present this is not well tested.

2.2 Areas

Areas define regions on the simulated grid. An area can contain other areas, buses, links, and relays. It principally acts as a container for the other objects, though will eventually include controls such as AGC and other wide area controls. The simulation object itself is a specialization of an area.

2.3 Links

In the most general form links connect buses together. As a primary object it can contain other objects, including state information. The basic formulation is that of a standard AC transmission line model connecting two buses together. The code includes a number of possible approximations.

2.4 Relays

Relays are perhaps the most interesting and unusual primary object included in GridDyn. The basic concept is that relays can take in information from one object and act upon another. They add protection and control systems into the simulation environment. They exist as primary objects since they can stand to operate on their own at the same level as buses and areas. They may contain states, other objects, submodels, etc. They also act as gateways into communication simulations, functioning as measurement units and control relays. And through relays a whole host of control and protection schemes can be implemented in simulation alongside normal power flow and dynamic simulations. Examples of relays include fuses, breakers, differential relays, distance relays, and control relays, among others.

CHAPTER 3

XML Input

The following section contains a description of the XML input file format and how to construct and specify an input file in the GridDyn XML format. The XML format is intended to be used solely in GridDyn to enable full access to all the capabilities and models that may or may not be defined in other formats. All the actual interpreters have been designed to use an element tree structure. And as such the same reader code is used for the XML interpreter and for a JSON interpreter, though there is some variance in the definitions of elements and attributes in those two contexts meaning Json objects are somewhat more restricted in format. In the documentation, most of the examples will be in XML, but a few will be in JSON for completeness.

3.1 Initial Example

A simple input case is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<GridDyn name="2bus_test" version="1">

<bus name="bus1">
    <type>SLK</type>
    <angle>0</angle>
    <voltage>1.05</voltage>
    <generator name="gen1">
    </generator>
    <load name="load1">
        <P>1.05</P>
        <Q>0.31</Q>
    </load>
</bus>

<bus name="bus2">
    <load name="load2">
        <P>0.45</P>
        <Q>0.2</Q>
    </load>
```

(continues on next page)

(continued from previous page)

```
</bus>
<link from="bus1" name="bus1_to_bus2" to="bus2">
    <b>0.127</b>
    <r>0.0839</r>
    <x>0.51833</x>
</link>

<flags>powerflow_only</flags>
</GridDyn>
```

This small XML file defines a two bus system. There are 5 sections to this model description. The first line describes the standard XML header information and is not used by GridDyn. The second line defines the simulation element and the name of the simulation. In general properties can be described in either an element or as a property. There are certain aspects of parameters which can only be controlled in the element form, but for simple parameters either works fine. Capitalization of properties also does not matter. All object properties in GridDyn are represented by lower case strings, the XML reader converts all property names to lower case strings before input to GridDyn so capitalization doesn't matter in the XML input. The property values themselves preserve capitalization and it is on a per property basis whether capitalization matters. For naming capitalization is preserved such that "object1" is distinct from "Object1". For this XML file the simulation is given the name *2bus_test*. The version is for record keeping only and has no relevance to the simulation.

The second block defines a bus object with a name of *bus1*. The bus is a slack bus indicated by *<bustype>SLK</bustype>*. Other options for this parameter include *PQ*, *PV*, *SLK*, and *afix*. The angle and voltage are specified. A generator object is included. The element *generator* is recognized as a component and a new generator object is created with a name of *gen1*. Finally a load is created with a name of *load1* and a fixed real power of 1.05 and a reactive power of 0.31.

The second bus is defined in a similar way, except it does not define a bustype which means it defaults to a PQ bus. The *link* is defined by:

```
<link from="bus1" name="bus1_to_bus2" to="bus2">
    <b>0.127</b>
    <r>0.0839</r>
    <x>0.51833</x>
</link>
```

The properties *b*, *r*, and *x* are defined in the XML as elements. The *to* and *from* fields are specified using the names of the buses. These properties must be specified for the lines or the system will spit out a warning.

Finally, the last two lines specify that the simulation should stop after a power flow.

To add in dynamic modeling a few additional pieces of XML can be added. For our example, the powerflow_only flag at the bottom can be removed, and the following lines can be added to the block for *gen1*:

```
<generator name="gen1">
    <dynmodel>typical</dynmodel>
    <pmax>4</pmax>
</generator>
```

This defines the generator to have a typical dynamic model, the meaning of which will be detailed in the section on model parameters for specific models [TODO]Add link to section[/TODO]. It also specifies a *pmax* value of 4 per unit.

Next, an event can be added to the load attached to *bus2* to change a parameter with the code shown below:

```
<bus name="bus2">
    <load name="load2">
```

(continues on next page)

(continued from previous page)

```
        <P>0.45</P>
        <Q>0.2</Q>
        <event>@1|p=1.1</event>
    </load>
</bus>
```

The line *<event>@1|p=1.1</event>* defines an event such that at time 1.0 the *p* field of the load is set to 1.1, from the initial value of 0.45. More details will be explained in the section on event specification [TODO]Add link to section[/TODO].

Finally, a block with a stop time and recorder can be added before the closing GridDyn tag:

```
<stoptime>10</stoptime>
<recorder period=0.5 field="auto">
    <file>twobusdynout.csv</file>
</recorder>
```

This sets the simulation to run until a stoptime of 10 seconds. The recorder xml element defines a recorder to capture a set of automatic fields at a period of 0.05 seconds, and capture it to the file twobusdynout.csv upon completion of the scenario. More dtails on recorder specification are available later in this document [TODO]Add link to section[/TODO].

The final listing after these changes is:

```
<?xml version="1.0" encoding="utf-8"?>
<GridDyn name="2bus_test" version="1">

<bus name="bus1">
    <type>SLK</type>
    <angle>0</angle>
    <voltage>1.05</voltage>
    <generator name="gen1">
        <dynmodel>typical</dynmodel>
        <pmax>4</pmax>
    </generator>
    <load name="load1">
        <P>1.05</P>
        <Q>0.31</Q>
    </load>
</bus>

<bus name="bus2">
    <load name="load2">
        <P>0.45</P>
        <Q>0.2</Q>
        <event>@1|p=1.1</event>
    </load>
</bus>
<link from="bus1" name="bus1_to_bus2" to="bus2">
    <b>0.127</b>
    <r>0.0839</r>
    <x>0.51833</x>
</link>

<stoptime>10</stoptime>
<recorder period=0.5 field="auto">
    <file>twobusdynout.csv</file>
```

(continues on next page)

(continued from previous page)

```
</recorder>

</GridDyn>
```

3.2 Parameter Specification

Simple parameters can be specified via elements or as attributes. Default units are in seconds for all times and time constants unless individual models assume differently. Power and impedance specifications are typically in PU values. Exceptions include *basepower* and *basevoltage* specifications which are in *MW* and *KV* respectively. The default units on any rates are in units per second. However, individual models are free to deviate from this standard as makes sense for them so check with the individual model type specification for details. Parameters in the XML can be specified in a number of different forms that are useful in different contexts. Below is an example showing the various methods.

```
<?xml version="1.0" encoding="utf-8"?>
<!--xml file to test parameter setting methods-->
<GridDyn name="input_tests" version="0.0.1">
<bus name="bus1">
    <load>
        <param name="P" value=0.4></param>
        <param field="q">0.3</param>
        <param field="ip" units="MW">55</param>
        <param>yq=0.11</param>
        <param name="iq(MW)" value=32/>
        <yp>0.5</yp>
    </load>

    <load yq=0.74 >
        <p units="puMW"> 0.31</p>
        <param>q(MW)=14.8</param>
        <param name="yp" unit="MW" value=127/>
    </load>
</bus>
</GridDyn>
```

The main variants involve varying how the units are placed. Units can be placed as an attribute named *unit* or *units* on the parameters either in a *param* element or and element named after the model parameter. They can also be placed in parenthesis at the end of the parameter name when the parameter name is a string contained in the elemental form. Values can be places in a value element, as the content of an element, or following an equal sign when defined as a string like *<param>yq=0.11</param>*. Parameters assuming the default units are allowed to be placed as attributes of the object.

3.3 Functions and Mathematical Operations

GridDyn XML input allows mathematical operators and expressions in any parameter specification, including complex expressions. Supported functions are shown in the tables that follow. In addition, most operators are supported including +, -, *, /, ^, and %. Operator precedence is respected as are parenthesis. String operations are not supported but the definition system has features that support some use cases for string operations.

3.3.1 Zero argument mathematical expressions

function	details
inf()	results in a large number between 0 and 1
nan()	uses nan("0")
pi()	pi
rand()	produces a uniform random number between 0 and 1
randn()	produces a normal random number with mean 0 and standard deviation of 1.0
randexp()	produces a random number from an exponential distribution with a mean of 1.0
randlogn()	produces a random number from a log normal distribution

3.3.2 One argument mathematical expressions

function	details
sin(x)	sine of x
cos(x)	cosine of x
tan(x)	tangent of x
sinh(x)	hyperbolic sine of x
cosh(x)	hyperbolic cosine of x
tanh(x)	hyperbolic tangent of x
abs(x)	absolute value of x
sign(x)	return 1.0 if x>0 and -1.0 if x<0 and 0 if x==0
asin(x)	arcsin of x
acos(x)	arccosine of x
atan(x)	arctangent of x
sqrt(x)	the square root of x
cbrt(x)	the cube root of x
log(x)	the natural logarithm of x *log(exp(x))=x*
exp(x)	the exponential function e^x
log10(x)	the base 10 logarithm of x
log2(x)	the base 2 logarithm of x
exp2(x)	evaluates 2^x
ceil(x)	the smallest integer value such that ceil(x)>=x
floor(x)	the largest integer value such that floor(x)<=x
round(x)	the nearest integer value to x
trunc(x)	the integer portion of x
none(x)	return x
dec(x)	the decimal portion of x *trunc(x)+dec(x)=x*
randexp(x)	an exponential random variable with a mean of x

3.3.3 Two argument mathematical expressions

function	details
atan2(x,y)	the 4 quadrant arctangent function
pow(x,y)	evaluates x^y
plus(x,y)	evaluates *x*+*y*
add(x,y)	evaluates *x*+*y*
minus(x,y)	evaluates *x*-*y*
subtract(x,y)	evaluates *x*-*y*
mult(x,y)	evaluates *x***y*
product(x,y)	evaluates *x***y*
div(x,y)	evaluates *x*/*y*
max(x,y)	returns the greater of x or y
min(x,y)	returns the lesser of x or y
mod(x,y)	return the modulus of x and y e.g. mod(5,3)=2
hypot(x,y)	evaluates $sqrtx^2 + y^2$
rand(x,y)	return a random number between x and y
randn(x,y)	returns a random number from a normal distribution with mean x and variance y
randexp(x,y)	returns a random number from an exponential distribution with mean x and variance y
randlogn(x,y)	returns a random number from a log normal distribution with mean x and variance y
randint(x,y)	returns a uniformly distributed random integer between x and y inclusive of x and y

3.4 Component Description

Components are defined in elements matching the component name. For example

```
<exciter name="ext1">
    <type>type1</type>
    <Aex>0</Aex>
    <Bex>0</Bex>
    <Ka>20</Ka>
    <Ke>1</Ke>
    <Kf>0.040</Kf>
    <Ta>0.200</Ta>
    <Te>0.700</Te>
    <Tf>1</Tf>
    <Urmax>50</Urmax>
    <Urmin>-50</Urmin>
</exciter>
```

describes an exciter component as part of a generator. The name attribute or element is common for all objects. A *description* can also be defined for all objects which is basically a string that can be added to any object. The *type* property is a keyword used to describe the detailed type of the component. In the above example the specific type of the component is *type1*. GridDyn uses polymorphic objects for each of the components. The type defined in the XML file for each component defines the specific object to instantiate. If type is not specified the default type of the component is used.

Predefined components include:

area defines a region of the grid

bus the basic node of the system

link the basic object connecting buses together

relay primary object allowing control and triggers for other objects

sensor a form of a relay specifically targeted at sensing different parameters and allowing some direct signal processing on measurements before output

load the basic consumer of energy

generator the basic producer of electricity

genmodel the dynamic model of an electrical generator

governor a generator governor

exciter an exciter for a generator

pss a power system stabilizer*

controlblock a basic control block

source a signal generator in GridDyn

simulation a simulation object

agc describes an automatic generation control*

reserveddispatcher describes a reserve dispatcher*

scheduler a scheduling controller*

- these objects are in development and do not work consistently

Several components are also defined that map onto the more general components, in some cases these define specific types, in others they are simply maps. Examples include *fuse=>relay*, *breaker=>relay*, *transformer=>line*, *block=>controlblock*, *control=>relay*, and *tie=>line*. Custom definitions can also be defined if desired through a *translate* element.

3.5 Object Identification

There are many instances where it is necessary to identify an object for purposes of creating links or to extract a property from another. Internally there is a hierarchy of objects starting with the root simulation object. This allows a path like specification of the objects. There are 2 different notations for describing an object path, one based on colons, the other more similar to the URI specification for WEB like services, and both allow properties to be specified in a similar fashion. Objects can be referred to by 4 distinct patterns. The first is by the name of the object. The name should be unique within any given parent object. The second uses an object component name and index number, for example *bus#0* would refer to the bus in index location 0; using *bus!0* will also work the same as *bus#0* but can be used in settings where the '#' is not allowed. All indices are 0 based. The fourth makes use of the user id of an object, to use this objects would be identified by *load$2* to locate the load with userID of 2.

When searching for an object the system starts in the current directory for the search. If it is not found it traverses to the root object and starts the search from there.

An example of a path specified with the ':' notation is *area45::bus#6::load#0:p*. The single colon marks that the final string represents a property. The same object in the URI notation would be *area45/bus#6/load#0?p*.

In some cases mixed notation might work but it is not recommended. The property indication can be left off when referencing an entire object. Starting the object identification string with an '@' or a beginning '/' implies searching starting in the root object, otherwise the search starts at whatever the current object of interest is.

3.6 Special Elements

In addition to the components elements, several element names have special purposes.

3.6.1 translate

The translate element is used to create a custom object definition.

```
<translate name="special" component="exciter" type="type1"/>
```

Putting this command in the XML file will allow objects using "special" as the element name instead of specifying "exciter" in the element name and a specific type. Translations, unlike definitions, are global and are only allowed in the root element of an XML file. IF you wish to specify a default type for a component or other defined component translation then the name or component can be left out.

3.6.2 define

The GridDyn XML file allows specification of definition strings that can be used as parameter values or in other definitions.

```
<define name="constant1" value=5/>
<define name="constant2" value=46 locked=1/>
<define name="constant3 value="constant1*constant2" eval=1/>
```

The above snippet of code defines 3 constants. Internally constants are stored as strings. If the *eval* attribute is specified the value string is evaluated before storing it as a string, otherwise it will be stored as a string and evaluated on use. The *locked* attribute defines a global parameter that cannot be overridden by another *define* command. The mechanisms allow programmatic or command line overrides of any internal definitions in the XML file. Inside the XML file they cross scope boundaries like a global variable. Regular definitions are only valid in elements they were defined in and subelements. So if a definition is used, define it in the root scope or it will only be applicable in a subsection of the XML.

When using definitions they can be used as a variable in other languages wherever a string or numerical value would be used. They can also be used in string replacements like the following code.

```
<bus name="bus_$#rowindex$_$#colindex$">
```

When evaluating the expression, the parts of the string between the $ signs gets evaluated first. In this case "#rowindex" and "#colindex" are part of an array structure which is described in the subsection on arrays.[TODO]Add ref to array subheading[/TODO]

After the substring replacement, the entire expression is evaluated again for other definitions. There is a small set of predefined definitions including %date, %datetime, and %time, which contain strings of the expected values at the time of file input.

3.6.3 custom

Translations are useful for readability, library elements allow duplication of objects with only minor modifications. For library elements the object is constructed once from XML and duplicated. Custom objects allow duplication of objects or sets of objects from the XML. A reference to the actual element source is stored and reprocessed at the time the custom object is encountered. This also allows a set of object to be defined in one input form to be imported by a different form and used to create objects described in the first. For instance you could create a library of different object sections and import that into another XML file and only use a few of the custom definitions you are interested

in. Custom objects can use other custom objects but cannot define new custom sections. Custom objects can define a set of required arguments and default values. When calling the custom element arguments can be defined.

A brief example using custom elements is shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<!--xml file to test custom elements-->
<griddyn name="customtest">
<custom name="cbus">
  <bus>
    <voltage>rand(0.95,1.05)</voltage>
  </bus>
</custom>
<array count="10">
  <cbus/>
</array>
</griddyn>
```

Another example using arguments is shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<!--xml file to test custom elements-->
<griddyn name="customtest">
<custom name="busarray" args="1">
<array count="arg1">
  <bus>
    <voltage>rand(0.95,1.05)</voltage>
  </bus>
 </array>
</custom>
<busarray arg1="7"/>
</griddyn>
```

3.6.4 configuration

A *configuration* element can define some parameters and operations for the XML reader itself. There are currently 2 parameters that can be specified, *printlevel* and *match_type*. The *printlevel* controls the verbosity of the output. The *match_type* controls the default match capitalization for searching for specified fields. The following table shows valid values for these fields.

parameter	value	details
printlevel	none(0) summary(1) detailed(2)	print only warnings and errors print only summary info print detailed information
match_type	typical_case_match all exact	match on specified, lower case, and upper case values (this is the default value) match to all cases [lex] only match to the given string. Ths operation can be used to process files where every component and XML description field is in lower case in the xml to speed up processing slightly

3.6.5 event

Events are changes that take place during the course of the GridDyn simulation execution. They can be as simple as the example used previously or contain more complex specifications and multiple times and values. The events are likely to be updated significantly in the near future, and while much of the specification will remain the same some new capabilities will be added.

Parameters for events are specified like those in the components. The available element or attribute names are shown in this table.

parameter	description
target	the object to extract data from
field	the field of the target object to capture
time, t	the time the event is scheduled to occur
units	the desired units of the output
value	the value associated with the event
period	for periodic events sets the period
file	the input file for a player type event
column	the column in a data file to use for the event

The *field* option of event specification is by far the most flexible. Any text directly in the *event* element is captured as the field.

3.6.6 recorder

Recorders are the primary data output system for GridDyn dynamic simulations. Like events, recorders have a set of parameters associated with them. The details are in the table below. Multiple recorder elements can be specified and the recorder for a single file can have multiple elements that get merged even if they are in different objects. They are keyed by recorder name and/or filename. Certain properties like the sampling period are specified on a recorder basis. Others are for the properties and data to record.

parameter	default	description
file	outputfile.csv	the file to save the data to
name	recorder_#	the name of the recorder for easy reference later
description		description that gets put in the header of the output file
column	-1	the column of the recorder in which to place the requested data
target		the object to extract data from
field		the field of the target object to capture
units	defUnit	the desired units of the output
offset	0	an offset index for a particular state
gain	1.0	a multiplier on the measurement
bias	0.0	a measurement bias
precision	7	the number of digits of precision to print for string formats
frequency	1.0	set the frequency of recording
period	1.0	set the measurement period
starttime	-inf	set a start time for the recorder
stoptime	inf	set a stop time for the recorder
autosave	0	set the recorder to save every N samples; 0 for off
reserve	0	reserve space for N samples
period_resolution	0	set the minimum resolution for any time period (not usually user specified)

Recorder fields define which property of an object to capture. This includes all properties and calculations involving

properties. All functions and expressions defined in the func section [TODO]Add ref to func section[/TODO] are valid in recorder expressions.

3.6.7 solver

Solvers can be defined through the XML file. There are some default solvers defined but the *solver* element allows the definition of custom solvers applied to specific problem types. This allows specification of specific approximations or other configuration options for the solvers to use for solving various specific problems. Solver properties are shown in the table below.

parameter	default	description
printlevel	error(1)	may be specified with a string or number, "debug"(2), "error"(1), "none"(0)", "errno" only prints out error messages
approx	"none"	see the table below for details on possible options [TODO]Add ref to the table on solver options[/TODO]
flags		see the table for details on possible options [TODO]Add ref to table on flags[/TODO]
tolerance	1e-8	the residual tolerance to use
name	solver_#	the name of the solver
index	automatic	the specified index of the solver
file		log file for the solver

Solvers have a set of options used to define what types of problems they are intended to solve. And another set of intended approximations. This information gets passed to the models whenever a solve is attempted. A listing of the possible modes is shown in the solver modes table below. In some cases multiple modes can be combined, in other cases they are mutually exclusive and the second will override the earlier specification. A number of approximations are also specified mainly targeting approximations to transmission lines. These approximations are suggestions rather than directives and models are free to ignore them. There are 3 independent approximations that can be used in various combinations and several descriptions which turn on the simplifications in a convenient form. Most approximations target the acline models, but future approximations can be added specifically looking at other models.

mode/approximation	description
local	used for local solutions
dae	solver is intended for solving a set of coupled differential algebraic equations
differential	solver is intended to solve a set of coupled differential equations
algebraic	solver is intended to solve algebraic equations only
dynamic	solver is intended for dynamic simulations
powerflow	solver is intended for powerflow problems (implies !dynamic and algebraic)
extended	instructs the model to use an extended state formulation mainly targetted at state estimation problems
primary	opposite of extend
ac	solve for both voltage and angle on the buses
dc	solve only for the angle on AC buses, assume the voltage is fixed
r, small_r	assume the resistance of transmission lines is small
small_angle	use the small angle approximation (assume $sin(\theta) = \theta$)
coupling	assume there is no coupling between the V and θ states
normal	use full detailed calculations
simple, simplified	use the small_r approximation
decoupled	use the coupling approximation
small_angle_decoupled	use the small angle and decoupled approximations
small_angle_simplified	use the small angle and small r approximations
simplified_decoupled	use the small r and decoupling approximations
fast_decoupled	use all 3 approximations
linear	linear

Solvers can also include a number of options that are common across all solvers (though specific solvers may not implement them). Often specific solvers also include other options specific to that numerical solver.

mode	description
dense, sparse	set the solver to use a dense matrix solve or a sparse matrix (default)
parallel, serial	set the solver to use parallel or serial (default) arrays, this is in the form of openMP array
constant_Jacobian	tell the solver to assume the Jacobian is constant
mask	tell the solver to use a masking element to shield specific variables from the solution. This functionality is used in some cases of initial condition generation and probably shouldn't be used externally

3.6.8 import

Import statements are used to add an external file into the simulation. The file can be of a type capable of being read by GridDyn. Import statements are typically single element statements though they can have subelements if desired. A couple examples are shown in this example.

```
<import>sep_lib.xml</import>
<import prefix="A1">subnetwork.csv</import>
<import final=true ext="xml">last_elements.odx</import>
```

The optional attributes/elements are descripted in the table below.

parameter	valid values	description
prefix	string	a string to prefix all object names from the imported file
final	"true(1)" "false(0)"	if set to true the import is delayed until after all other non-final imports and the local file have been loaded if set to false or not included the import is processed before any locally defined objects and in the order imports are specified
file	string	the file name, can also be interpreted from the element text
filetype	string	the extension to use for interpreting the import file; if not specified the extension is determined from the file name
flags	ignore_step_up_transformers	the flags option is to add in additional options, it will likely be expanded as needed, currently the only option available is to ignore step up transformers in some formats of model input. As the file readers improve and become more integrated and consistent more options will be available

3.6.9 directory

The directory element allows the user to specify additional search paths for GridDyn to locate any files without an absolute path.

```
<directory>/home/usr/user1/GridDyn</directory>
<directory>
```

3.6.10 library

GridDyn file input can include a library of predefined objects. This section is defined through a library element. Any of the components described above can be included as a library element. These library objects get stored in a separate holding area and are copied when any object uses a *ref* fields with a value of the library element name. The *ref* field can be either an element or an attribute. If *type* and *ref* are specified the type definition takes priority and the library object is cloned to the newly created object, if only *ref* is specified a new object is cloned directly from the library object. There can be multiple library sections, they simply get merged. By using import statements libraries can be defined in a separate file. A simple example using libraries and references is shown below. The code describes 4 objects and generator model, an exciter and a governor, and a generator that uses the 3 previously defined submodels to make up the dynamic components of the generator.

```
<library>
    <model name="mod1">
        <type>fourthOrder</type>
        <D>0.040</D>
        <H>5</H>
        <Tdop>8</Tdop>
        <Tqop>1</Tqop>
        <Xd>1.050</Xd>
        <Xdp>0.350</Xdp>
        <Xq>0.850</Xq>
        <Xqp>0.350</Xqp>
    </model>
    <exciter name="ext1">
        <type>type1</type>
        <Aex>0</Aex>
```

(continues on next page)

(continued from previous page)

```
        <Bex>0</Bex>
        <Ka>20</Ka>
        <Ke>1</Ke>
        <Kf>0.040</Kf>
        <Ta>0.200</Ta>
        <Te>0.700</Te>
        <Tf>1</Tf>
        <Urmax>50</Urmax>
        <Urmin>-50</Urmin>
    </exciter>
    <governor name="gov1">
        <type>basic</type>
        <K>16.667</K>
        <T1>0.100</T1>
        <T2>0.150</T2>
        <T3>0.050</T3>
    </governor>
    <generator name="gen1">
        <model ref="mod1"/>
        <exciter ref="ext1"/>
        <governor ref="gov1"/>
    </generator>
</library>
```

Libraries are only allowed to be defined at the root object level, they are not allowed in any element that is a part of the root element so they are directly processed by the interpreter.

3.6.11 array

Arrays and the *if* statement make up the control structures in the XML file. Arrays allow objects and sets of objects to be generated in a loop, they can even contain other loops. An example file used for building some scalability tests is shown below. This file uses many of the concepts discussed previously.

```
<?xml version="1.0" encoding="utf-8"?>
<!--xml file to scalability using arrays-->
<griddyn name="test1" version="0.0.1">
<define name="garraySize" value="20"/>
<define name="gcount" value="ceil(garraySize/3)" eval="1" />
<configuration>
<match_type>exact</match_type>
</configuration>
<library>
<generator name="default">
<P>3.8*(((garraySize^2)/(gcount^2))/9)</P>
<mbase>400</mbase>
 <exciter>
 <type>type1</type>
 <vrmin>-50</vrmin>
 <vrmax>50</vrmax>
 </exciter>
 <model/>
<governor/>
</generator>
<load name="addLoad">
          <Yp>0.5</Yp>
```

(continues on next page)

(continued from previous page)

```
          <Yq>0.2</Yq>
          </load>
          <load name="constLd">
          <P>0.1</P>
          <Q>0.02</Q>
          <Ip>0.1</Ip>
          <Iq>0.02</Iq>
          <Yp>0.1</Yp>
          <Yq>0.02</Yq>
          </load>
</library>
        <array count="garraySize" loopvariable="#rowindex">
        <array count="garraySize" loopvariable="#colindex">
     <bus name="bus_$#rowindex$_$#colindex$">
         <load ref="constLd"/>
         </bus>
         </array>
         </array>
         <!--add in the additional loads -->

          <array start=1 stop="garraySize" loopvariable="#rowindex" interval=2>
         <array start=1 stop="garraySize" loopvariable="#colindex" interval=2>
     <bus name="bus_$#rowindex$_$#colindex$">
         <load ref="addLoad"/>
         </bus>
         </array>
         </array>
          <!--add in the generators -->
          <array start=1 stop="garraySize" loopvariable="#rowindex" interval=3>
         <array start=1 stop="garraySize" loopvariable="#colindex" interval=3>
     <bus name="bus_$#rowindex$_$#colindex$">
         <gen ref="default"/>
         <bustype>PV</bustype>
         <voltage>1.01</voltage>
         </bus>
         </array>
         </array>
         <!--add in the vertical links-->
         <array stop="garraySize" loopvariable="#rowindex" start="2">
        <array count="garraySize" loopvariable="#colindex">
     <link name="link_$#rowindex-1$_$#colindex$_to_$#rowindex$_$#colindex$">
         <r>0.001</r>
         <x>0.07</x>
         <from>bus_$#rowindex-1$_$#colindex$</from>
         <to>bus_$#rowindex$_$#colindex$</to>
         </link>
         </array>
         </array>

    <!--add in the horizonal links-->
          <array count="garraySize" loopvariable="#rowindex">
         <array stop="garraySize" loopvariable="#colindex" start="2">
     <link name="link_$#rowindex$_$#colindex-1$_to_$#rowindex$_$#colindex$">
         <r>0.001</r>
         <x>0.07</x>
         <from>bus_$#rowindex$_$#colindex-1$</from>
         <to>bus_$#rowindex$_$#colindex$</to>
```

(continues on next page)

(continued from previous page)

```
        </link>
        </array>
        </array>
         <!--label the swing bus-->
         <busmodify name="bus_$1+3*floor(garraySize/6)$_$1+3*floor(garraySize/6)$">
         <bustype>SLK</bustype>
         <id>10000000</id>
         <voltage>1.03</voltage>
         </busmodify>

   <basepower>30</basepower>
   <timestart>0</timestart>
   <timestop>60</timestop>
   <timestep>0.02</timestep>
   <solver name="ida">
   <printlevel>1</printlevel>
   </solver>
</griddyn>
```

Arrays can have several attributes which define how the array is handled.

attribute	default	description
start	1	the index to start the array counter
stop	X	the last index to use, either stop or count must be specified
count	X	the number of loops, either stop or count must be specified
loopvariable	#index	the name of the definitions to store the loop variable
interval	1.0	the interval between each iteration of the loop counter

3.6.12 if

If elements create a conditional inclusion. Most often used for conditional inclusion based on fixed parameters to allow a single file to do a few different scenarios. However, they can be tied in with random function generators and arrays to generate random distributions of elements. Any element component along with *import* and *define* statements are allowed in an *if* element.

The *if* element must have an element or attribute named *condition*. The condition is a string specifying a value or two values and a comparison operator. If a single expression is given, the elements in the *if* statement are evaluated as long as the expression does not result in a 0. Otherwise both sides of the expression are evaluated and the comparison is checked. If both sides evaluate to strings, a string comparison is done, otherwise a numerical comparison if both sides result in numerical values. Depending on the file type and reader '>' and '<' may need to be replaced with the XML character codes of *>* and *<*. These codes are interpreted properly. Compound expressions are not yet supported. Eventually the goal will be to support conditions based on object values instead of values that can be evaluated in the element reader itself, but this capability is not yet allowed.

3.6.13 econ

The *econ* element describes data related to the costs and values of an object. It will be used for interaction with optimization solvers and the root object must be an optimization type simulation. While the element works fine, it doesn't do anything with the data.

3.6.14 position

A *position* element describes data related to the geophysical (or relative) position of an object. The element is ignored but will be further developed at a later time.

3.6.15 actions

The gridDynSimulation object can execute a number of types of actions. These can be controlled through the API but also through an action queue. The actions are defined and stored in a queue and executed when the run function is called. If no actions are defined some logic is in place to do something sensible, typically run a power flow then a dynamic simulation if dynamic components were instantiated. Actions allow a much finer grained control over this process. These actions can be loaded through the XML file and eventually in a type of script (not enabled yet). Actions are specified through an *action* element containing the action string. The string is translated into an action and stored in a queue.

```
<action>run 23.7</action>
```

The list of available commands is shown in the table below. For all lines in the table *(s)* implies string parameter, *(d)* implies double parameter, *(i)* integer parameter, *(X)** optional, *(s|d|i)* string or doulbe or int, and for a given line everything following a # at the beginning of a word is considered a comment and ignored.

action string	description
ignore XXXXXX	do nothing
set parameter(s) value(d)	set a particular parameter; the parameter can include an object path
setall objecttype(s) parameter(s) value(d)	set a parameter on all objects of a particular type
setsolver mode(s) solver(s|i)	set the solver to use for a particular mode of operation
settime newtime(d)	set the simulation time
print parameter(s) setstring(s)	print a parameter (can include path)
powerflowstep solutionType(s)*	take a single step of the specified solution type
eventmode stop(d)* step(d)*	run in event driven power flow mode until stop with step
initialize	run the initialization routine
dynamic solutionType(s)* stop(d)* step(d)*	run a dynamic simulation
dynamicdae stop(d)*	run a dynamic simulation using DAE solver
dynamicpart stop(d)* step(d)*	run a dynamic simulation using the partitioned solver
dynamicdecoupled stop(d)* step(d)*	run a dynamic simulation using the decoupled solver
reset level(i)	reset the simulation to the specified level
iterate interval(d)* stop(d)*	run an iterative power flow with the given interval
run time(d)*	run the simulation using the default mode to the given time
save subject(s) file(s)	save a particular type of file
load subject(s) file(s)	load a particular type of file
add addstring(s)	add something to the simulation
rollback point(s|d)	rollback to a saved checkpoint (not implemented yet)
checkpoint name(s)	save a named checkpoint (not implemented yet)
contingency ????	run a contingency analysis (not implemented yet)
continuation ????	run a continuation analysis (not implemented yet)

CHAPTER 4

Design Philosophy

GridDyn was formulated as a tool to aid in research in simulator coupling. Its use has expanded but it is primarily a research tool into grid simulation and power grid related numeric methods, and is designed and constructed to enable that research. It is open source, released under a BSD license. All included code will have a similarly permissive license. Any connections with software of other licenses will require separate download and installation. It is intended to be fully cross platform, enabling use on all major operating systems, all libraries used internally must support the same platforms. However interaction with other simulators, such as for distribution or communication may impose additional platform restrictions. Optional components may not always abide by the same restrictions. Prior to release 1.0 very little effort will be expended in backwards compatibility. GridDyn was written making extensive use of C++11 constructs, and will shortly be making more use of C++14 standard constructs. Specifically allowing any features of the standard supported by GCC 4.9.x versions. It is expected this will be the minimum version supported until newer compilers are much more widely accessible.

4.1 Modularity

GridDyn code makes heavy use of object oriented design and polymorphism and is intended to be modular and replaceable. The design intention is to allow users to define a new object that meets a given component specification and have that be loaded into the simulation as easily as any previously existing object, and require no knowledge of the implementation details of any other object in the simulation. Thus allowing new and more complex models to be added to the system with no disruption to the rest of the system. Models also do not assume the precense of any other object in the system, though they are allowed to check for the existence first. This is exemplified in the interaction of generators with its subcomponents. Any combination of Generator Model, exciter, and governor should form a valid simulation even though some combinations may not make much physical sense or be realistic.

4.2 Mathematics

The GridDyn code itself has only limited facilities for numeric solutions to the differential algebraic equations which define a dynamic power system simulation. Instead it relies on external libraries interacting through a solver interface tailored for each individual solver. The models are intended to be very flexible in support for an assortment of numeric approximations and solution models, and define the equations necessary for model evaluation.

Initial development of dynamic simulation capability is done through a coupled differential algebraic solver with variable time stepping; the primary solver used is IDA from the SUNDIALS package. It can use the dense solver or the KLU sparse solver which is much faster. Recent work incorporates the use of a fixed time step solution mode, with a partitioned set of solvers separating the algebraic from the differential components and solving them in alternating fashion. At present this is much less well tested. Initial formulations use CVODE for the differential equations and KINSOL for the algebraic solution. Kinsol is also used to solve the power flow solution. ARKODE, an ODE solver using Runga-Kutta methods is also available for solving the ODE portion of the partitioned solution.

In order to provide support for current and future models of grid components a decision was made to distribute the grid connectivity information and not use a Y-bus matrix as is typical in power system simulation tools. This allows loads and tranmission lines to be modeled using arbitrary equations. This decision alters the typical equations used to define a power flow solution at buses. Each bus simply sums the real and reactive power produced or consumed by all connected loads, generators, and links. Those components are free to define the power as an arbitrary function of bus voltage, angle, and frequency, provided that function is at least piecewise continuous.

Note: Continuous functions work much better, piecewise continuous functions work but don't really play nicely with the variable timestepping.

Defining the problem in this way comes at a cost of complexity in the complementation and likely a performance hit but allows tremendous flexibility for incorporating novel loads, generators, and other components into power flow and dynamic simulation solutions. The dependency information is extracted through the Jacobian funciton call. Currently the solution always assumes the problem is non-linear even if the approximations used are in fact linear. While GridDyn's interaction with the solvers comes exclusively through interface objects, there may be some inherent biases in the interface definition due to primary testing with the SUNDIALS package. These will likely be exposed when GridDyn is tested with alternative numerical solvers.

4.3 Model Definition

GridDyn is intended to be flexible in its model definition allowing details to be defined through a number of common power system formats. The most flexible definition is through a GridDyn specific XML format. Strictly speaking the most flexible XML cannot be defined by an XML schema due to the fact that the readers allow element names to describe properties of which the complete set of which cannot be described due to support for externally defined models. Alternate formulations exist which could be standardized in a schema but no attempt has been made to do so. The XML formulation includes a variety of programming like concepts to allow construction of complex models quickly, including arrays and conditionals, as well as limited support for equations and variable definitions. The file ingest library also supports importing other files through the XML and defining a library of objects that can be referenced and copied elsewhere. The typical use case is expected to be importing a file of another format that contains a majority of the desired simulation information and only defining the solver information and any GridDyn specific models and adjustments in the XML. The general idea is to be as flexible and easy to use as possible for a text based input format, as GridDyn develops support as many other formats as is practically possible. All the file ingest functionality is contained in a separate library from the model bookkeeping and model evaluation functionality. Other types of input can be added as necessary and some development is taking place towards a GUI which would interact through REST service commands and JSON objects. Included in GridDyn are capabilities of searching through objects by name, index number, or userID.

4.4 Performance

GridDyn was designed for use in an HPC environment. What that means right now is that GridDyn can interoperate with other simulators in that environment and some considerations were put in place in the design, but GridDyn on its own does not really take advantage of parallel processing. As of release 0.5 the transmission power flow and dynamic

solve is not itself parallel in any way. Considerable thought has been put into how that might be accomplished in later versions but it is not presently in place. Initial steps will include adding in optional OpenMP pragmas to take advantage of the inherent independence of the objects in calculation of the mathematical operations such as residual or Jacobian. OpenMP vector operations can be enabled in SUNDIALS, though this is only expected to result in small performance gains and only for models over 5000 buses. Further tests will be done to determine exact performance gains.

Some effort has gone into improving the performance of the power flow solve and only incremental gains are expected at this point using the current solve methodology. No effort has been expended on the dynamic simulation so some performance improvements can be expected in that area when examined.

The system has no inherent size limitations. Limited only by memory on any given system. Scalability studies have been carried out to solving a million bus model, It could probably go higher but the practical value of such a single solve is unclear as of yet.

4.5 Model Libraries

The aim thus far in GridDyn has been the development of the interfaces. The models available are the result of programmatic needs or the need to ensure the simulator is capable of dealing with specific kinds of model interactions. As a result the models presently available represent only a small subset of those defined in power system libraries. More will be available as time goes on, but the idea is not to have a large collection internally but to enable testing of new models, and to incorporate model definition libraries through the use of other tools and interfaces such as FMI, and possibly others as needed.

4.6 Testing

A suite of test cases is available and will continue to grow as more components and systems are thoroughly tested. The nature of the test suite is evolving along with the code and will continue to do so. It makes use of the BOOST test suite of tools and if built creates 5 executable test programs that test the various aspects of the system. While we are still a ways from that target 100% test coverage is a goal though likely not realistic in the near future. The code is regularly compiled on at least 5 different compilers and multiple operating systems and strives for warning free operation.

4.7 Test Programs

If enabled 5 test programs are built. These programs execute the unit test suite for testing GridDyn. They are divided into 5 programs. testLibrary runs tests aimed at testing operation of the various libraries used In GridDyn. The testComponents program executes test cases targeted at the individual model components of GridDyn. The third, testSystem, runs system level tests and some performance and validation tests on GridDyn. The testSharedLibrary tests using GridDyn as a shared library. The last, extraTests includes some longer running tests and performance tests. After installation these test programs are placed in the install directory and can be executed by simply running the executable. Specific tests can be executed with command line parameters.

```
> ./testComponents --run_test=block_tests
> ./testComponents --run_test=block_tests/block_alg_diff_jac_test
> ./testLibrary -h
```

CHAPTER 5

Development Notes

GridDyn is very much a work in progress, development is proceeding on a number of different aspects from a number of directions and many components are in states of partial operation or are awaiting development in other aspects of the code base. The notes in this section attempt to capture the development status of various Griddyn components and note where active and planned development is taking place.

5.1 Interface and Executables

A gridDynServer executable is in development. This program will become the main means of interacting with simulations. The plan will be for it to support multiple running simulations and allow users to interact through a set of interfaces. Planned interfaces include a RESTful service interface for ethernet based interaction, which will eventually be the basis of interaction with a GUI, a command line interface, and a direct application interface through TCP/UDP or MPI.

Also in development is a wrapper around the simulation engine into a Functional Mockup Interface to allow GridDyn to interact with other simulations through the FMI for co-simulation framework.

5.2 Models

The models included in GridDyn are an evolving set. They have been added to address particular research questions or needs or test specific aspects of GridDyn operation. The next several subsections talk about the state of development in the various components available in GridDyn.

5.2.1 Buses

The bus code is well tested but is constantly evolving to simplify the code or areas of responsibility, or to improve operation, even though the equations used in the bus evaluation are quite straightforward. The bus itself is one of the more complex objects in GridDyn in order to handle the management of loads and generators and the associated limits and controls. As well as the associated transition between powerflow and dynamic simulation. Currently available are

an ACbus, a DC bus for association with HVDC transission lines, a trivial bus, and an infinite bus. Some plans are in place for a 3-phase bus but that has been low on the priority list. The DC bus is not thoroughly tested, particularly in dynamic contexts.

5.2.2 Area

At present areas are primary used as a way to group objects. Ongoing development is taking place to add in area wide controls such as AGC. Some of these structures are in place but have yet to be tied in with the Area model itself. There is work ongoing to do this and some form will be functional within the next 3 months. Areas and subareas can be configured through the GridDyn XML format but none of the other available formats such as CDF or PTI currently make use of the area information available in those formats. This will be added alongside the development of area controls.

5.2.3 Links

The basic AC link has been tested thoroughly in powerflow and dynamic simulations by comparison with standard test cases. Other link models such as DC links, and an *adjustableTransformer* model have been tested in power flow simulations, but the dynamics of them are a work in progress. They operate fine in that context but do not include the control dynamics, at least not at a level that is well-tested.

5.2.4 Relays

The generic relay is one of the more complex objects to setup. Most use cases involve using one the specific relay types as they embody the information for setting up a relay. There are no known issues with the relays though given their complexity it is likely there are many circumstances when they do not function appropriately, or cause issues with interaction of the other parts of the system. The basic relay contains tremendous flexibility and it is not recommended that beginning users attempt to directly instantiate it. You are of course welcome to try but the specification of conditions and actions is somewhat more complex than most other system properties through the XML. Other relay types are in development as needed by specific usage requirements.

5.2.5 Loads

A number of types of loads are modeled in GridDyn. The basic model is a ZIP model. Extensions include ramps and a variety of other load shapes and others such as an exponential load and a frequency dependent load. Also included are motor loads, including models of first order, 3rd order and 5th order induction models, and include mechanisms for modeling motor stalling. The 5th order model has some potential issues during certain conditions that have not been fully debugged. All work in powerflow and dynamic simulation. Code for loading a GridLab-D distribution system is included in the release but will not function without corresponding alterations to a GridLab-D instance and operation with Pargrid, neither of which are included in this release, so for all practical purposes it will revert to a debug mode with a simulated distribution simulation intended for debugging operations. The actual functionality necessary for coupling with a distribution system will hopefully be released in the near future, though could be made available for partners. There is a composite load model available. This is a more generic container for containing other load models. This is distinct and more general than the composite load model defined by FERC. Though an instantiation of that model is planned and will make use of the generic composite model in GridDyn.

Generators

These include governors, exciters, generator models, and power system stabilizers. The variable generator also has mechanisms for including sources which are data generators, and filters. The combination of which creates a mechanism for feeding weather data to a solar or wind plant and converting that into power. The generator is specifically formulated to allow any/all/none of the subcomponents to be present and still operate. A default generator model is

put into place if none is specified and a dynamic simulation is required. A third generator which includes a notion of energy storage is in planning stages.

5.2.6 Generator Models

A wide assortment of *genModels* are included. Most have been debugged and tested. The classical generator model includes a notion of a stabilizer due to inherent instabilities under fault conditions when attached to an exciter and/or governor. Not that the classical generator model is an appropriate model to use for such circumstances, but nonetheless a stabilizer was incorporated to make the model stable. The incorporation of saturation into the models is not complete. The models accept the parameters but are not included in the calculation. GENROU and GENSAL models are being developed but are not complete as of release 0.5.

5.2.7 Exciters

Available exciters include simple, IEEE type1, IEEE type2, DC1A, and DC2A. The DC2A model has some undiagnosed issue in particular situations and is not recommended for use at present.

5.2.8 Governors

The basic governor and TGOV1 models are operational, others are not completed and further work is being delayed until a more general control system model is in place which will greatly simplify governor construction as well as other control systems. The deadband is not working in TGOV1.

5.2.9 Power System Stabilizers

The current PSS code is a placeholder for future work. No PSS model is currently available, though some initial design work has taken place. The work has been delayed until the control system code is operational.

5.2.10 Control Blocks

Control blocks are a building block for other models and a number of them are used in other models throughout Griddyn. Development on the generic transfer function block is not finished but the others are working and tested. These will form the building blocks of a set of general control system modules which could be used to build other types of more complex models.

5.3 Others

Other components in Griddyn include *sources* which are operational but not well tested in practice, *schedulers* which are used to control generator scheduling, and other types of controllers for AGC, dispatch, and other sorts of controls. Most of these are in various states of development and not well tested.

5.3.1 Events

Griddyn supports a notion of events which can be scheduled in a simulation and can basically alter any property of the system with the exception of some models prohibiting changing of certain properties after simulation has begun, in this case the event will still be valid, it just won't do anything. Support for more complex events involving multiple devices in a more straightforward fashion is planned.

5.3.2 Recorders

Support for extracting any calculated field or property from an object is supported through *grabber* objects. This can be done directly via the state arrays or from the objects themselves. The files can be saved periodically or at the end of the simulation in a binary format or in CSV. Readers for the binary format are available in C++, Matlab, and Python. If a large amount of data is captured frequently for dynamic simulations there is currently a performance hit. There are ideas for mitigating this that will be addressed when the performance of the dynamic simulation is studied and addressed.

5.3.3 Simulation

Some of the mechanics and interfacing of the planned optimization extension are in place but nothing actually works yet, so don't use it.

5.3.4 FMU Interaction

This works in some cases but is a little more complex to set up than the rest of the code as it is under significant active development, therefore it is not recommended for use at this time.

5.4 File Input

GridDyn is capable of reading XML and Json files defining the GridDyn data directly and these formats can take advantage of all GridDyn capabilities. Json is not as well tested and was targeted mainly for the server interface, but it should work as a file format just fine. A fairly flexible CSV input file reader is also available for inputing larger datasets in a more workable format. CDF files are read though the area and a few other properties not important for powerflow are not loaded into GridDyn yet. Most of the common elements in raw and pti les are also loaded properly. Some of the more exotic elements such as multiterminal DC lines and 3-way transformers are not yet, mainly since we have no examples of such things in example files. EPC files for PSLF are the same though used less extensively than raw files. Matlab files from Matpower and PSAT can also be loaded. Not all dynamic models from PSAT are available, for DYR files models that match those available are loaded and some others are translated to available models. The library of models in GridDyn is much smaller than those available in commercial tools. Support for other formats is added as needed by projects.

CHAPTER 6

CMake Options

The CMake build scripts for GridDyn support a number of configuration options that can be set via either the *cmake-gui* or the command line cmake command using *-D<VAR>=<VALUE>* arguments. The CMake manual available at https://cmake.org/cmake/help/latest/manual/cmake.1.html

describes use of *-D* and other arguments in more detail.

BOOST_INSTALL_PATH Sets the root location of Boost. Can be used if Boost is not found in the system directories or if a different version is desired.

BUILD_SHARED_LIBS Turns on building of the GridDyn C and C++ shared libraries (`BUILD_GRIDDYN_C_SHARED_LIBRARY` and `BUILD_GRIDDYN_CXX_SHARED_LIBRARY` options).

BUILD_TESTING Enable the test executables to be built.

ENABLE_GRIDDYN_LOGGING Enables all normal, debug, and trace logging in GridDyn.

ENABLE_GRIDDYN_DEBUG_LOGGING Unselecting disables all `DEBUG` and `TRACE` log messages from getting compiled.

ENABLE_GRIDDYN_TRACE_LOGGING Unselecting disables all `TRACE` log messages from getting compiled.

DOXYGEN_OUTPUT_DIR Location for the generated doxygen documentation.

ENABLE_64BIT_INDEXING Enables support inside GridDyn for more than 2:sup:*32*-2 states or objects.

ENABLE_FMI Enable support for FMI objects.

ENABLE_FMI_EXPORT Enable construction of a binary FMI shared library for GridDyn.

ENABLE_FSKIT Enable to build additional libraries and support for integration into FSKIT and PARGRID for tool coupling.

ENABLE_HELICS_EXECUTABLE Enable the HELICS executable to be built for tool coupling using HELICS for communication.

ENABLE_KLU Option to disable KLU (not recommended [slow]; prefer to turn on `AUTOBUILD_KLU`)

ENABLE_MULTITHREADING Disable multithreading in GridDyn libraries.

ENABLE_MPI Enable MPI networking library.

ENABLE_OPENMP Enable OpenMP support.

ENABLE_OPENMP_GRIDDYN Enables OpenMP use internal to GridDyn.

ENABLE_OPENMP_SUNDIALS Enables the SUNDIALS NVector OpenMP implementation.

ENABLE_YAML Enables YAML file support in GridDyn.

ENABLE_EXTRA_MODELS Compile and load extraModels.

ENABLE_EXTRA_SOLVERS Compile and load extraSolvers (including braid, paradae).

ENABLE_NETWORKING_LIBRARY Enable network based communication components.

ENABLE_TCP Enable TCP connection library. Depends on Networking.

ENABLE_DIME Enable connection with DIME. Depends on Networking.

ENABLE_ZMQ Enable ZMQ connection library. Depends on Networking.

ENABLE_PLUGINS Build libpluginLibrary

ENABLE_OPTIMIZATION_LIBRARY Enable optimization libraries.

ENABLE_CODE_COVERAGE_TEST Build a target for testing code coverage.

ENABLE_GRIDDYN_DOXYGEN Generate Doxygen doc target.

ENABLE_CLANG_TOOLS If Clang is found, enable some custom targets for Clang formatting and tidy.

ENABLE_PACKAGE_BUILD Add projects for making packages and installers for GridDyn.

ENABLE_EXTRA_COMPILER_WARNINGS Enable more compiler warnings (full list in config/cmake/compiler_flags.cmake)

ENABLE_EXPERIMENTAL_TEST_CASES Enable some experimental test cases in the test suite.

FORCE_DEPENDENCY_REBUILD Rebuild third party dependencies, even if they're already installed.

LOAD_ARKODE Build in support for ARKODE for solving differential equations. Not used at present but will be in the near future.

LOAD_CVODE Build in support for CVODE for solving differential equations. Not used at present but will be in the near future.

SuiteSparse_INSTALL_PATH The location of the KLU installation if it was not found in the system directories.

SUNDIALS_INSTALL_PATH The location of the SUNDIALS installation if it wasn't found (or `AUTOBUILD_SUNDIALS` is disabled).

CHAPTER 7

Settable Object Properties

The tables here describe the parameters for each of the models present in GridDyn as of Version 0.5. The tables are automatically generated via scripts so there are a few bugs and some missing information as of yet. Each table has 4 columns. The first column specifies the string or strings that can be used to set this property, multiple strings that do the same thing are separated by a comma. The second columns defines the type of parameter, number implies a numeric value, string implies a string field, and flag is a flag or boolean variable which can be set to true with "true", or any number greater than 0.1 (typically 1), and set to false for any number less than 0.1 or "false". The third column lists the default value if applicable and the fourth column is a description. In many cases the default units will be described in [] at the beginning of the description, the default units are the units of the default and the unit that is assumed if no units are given to the set command. All the set functions cascade to parent classes which are identified in the table captions.

CHAPTER 8

Style Guide

8.1 Naming Styles

8.1.1 Classes

Camel case names starting with a Capital letter

e.g. `GridDynClass`

8.1.2 Class Methods

Camel case names starting with a lower case letter

e.g. `gridDynMethod`

8.1.3 Class Static Members

Camel case names starting with lower case and preceded by a `s_`

e.g. `s_staticMember`

8.1.4 Class Members

Camel case names starting with lower case and preceded by a `m_`

e.g. `m_classMember`

Model Parameters

A subset of class members specifically referring to settable model parameters.

Engineering reference model parameters are preceded by `mp_`, `K` is used for gains, `T` for time constants, `R` for resistances, `X` for impedances, and others are used as appropriate, typically using a capital letter first followed by a number of other lower case letters.

e.g. `mp_K1`, `mp_T3`, `mp_Rs`

Pointers

Camel case starting with a lower case and preceded by `p_`

e.g. `p_classMemberPointer`

8.1.5 Function Names

Camel case starting with lower case

e.g. `functionName`

8.1.6 Function Arguments

Camel case starting with lower case

e.g. `functionName(type functionArgument1, type2 functionArgument2)`

8.1.7 Enumeration Names

Lower case separated by _ and followed by `_t`

e.g. `enumeration_name_t`

8.1.8 Enumeration Fields

Lower case separated by _

e.g. `enumeration_field`

8.1.9 Global Constants

Capital letters preceded by a lower case `k`

e.g. `kCONSTANT`

8.1.10 Macros

Capital letters with words separated by _

e.g. `MY_MACRO`

CHAPTER 9

Indices and tables

- genindex
- modindex
- search