PyGridSim: A Functional Interface for Distributed System Simulation

by

Angela M. Zhao

S.B. in Computer Science and Engineering, MIT, 2025

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

Authored by:	Angela M. Zhao
	Department of Electrical Engineering and Computer Science
	May 16, 2025
Certified by:	Kalyan Veeramachaneni
	Principal Research Scientist, Thesis Supervisor
Accepted by:	Katrina LaCurts
	Chair, Master of Engineering Thesis Committee

PyGridSim: A Functional Interface for Distributed System Simulation

by

Angela M. Zhao

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

This thesis details the development of PyGridSim, an open source python module that leverages OpenDSS capabilities to provide an efficient and scalable functional interface for building distributed system simulations.

Distributed power systems encompass all components that power an electrical system—from larger power plants to microgrids—and represent the network of electric consumption and production in a system. Simulations of such power systems allow experts to analyze potential faults and risks in a fast, reproducable, and cost-efficient way. Thus, the accessibility of such simulations is critical to supporting the safety and reliability of power systems.

While existing packages built for distributed system simulation provide the necessary computing power and customizability of a distributed system simulator, their interfaces are hard to scale over many nodes and often have difficult-to-learn syntax. PyGridSim aims to build on these existing modules—maintaining customizability while providing a flexible, intuitive, and scalable syntax structure.

Thesis supervisor: Kalyan Veeramachaneni

Title: Principal Research Scientist

Acknowledgments

I would like to acknowledge all of those who made my work on this thesis possible. First, I would like to thank my supervisor, Kalyan Veeramachaneni, for welcoming me to the Data to AI Lab and proposing the initial project. This work would not have been possible without his continued guidance and collaboration throughout the year. Next, I would like to thank Sarah Alnegheimish—from advice in the ideation stage to code reviewing to helping package the final product—her support and collaboration is incredibly appreciated. I would also like to thank Christophe Schmidt—his feedback was essential in creating the final version of the product. I would like to thank the rest of the Data to AI lab for their insightful feedback during group meetings and continued support throughout the year.

Finally, I would like to thank my family and friends for their encouragement throughout my research and creation of this thesis. Their support has always kept me motivated to learn more and continue to explore.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Distributed Power Systems

To begin with, a power system is a way of modeling electrical circuits through its various components—including consumers, producers, and transformers of electricity. These circuits can represent electricity flow over a variety of possible scopes: they can model something as small as a few homes, to representing the power system between many cities.

A distributed power system represents a network with more independent and localized power sources[1]. Instead of solely depending on large centralized power plants, a distributed power system also includes smaller power generation units, such as wind turbines or household solar panels. These distributed power systems are represented as circuits. However, it is important to distinguish these from electrical AC-DC circuits, that specify circuits to a capacitor, inductor level. Distributed power systems describe the state of the circuit from a higher level, with components being an entire load or entire source (instead of each individual electrical component within them)[1].

Distributed power systems are a crucial part of our everyday—they power the electricity of our homes and buildings. As a result, it is critical to analyze when and how these power

[1]We are not experts in power systems. This thesis focuses on creating an open source software system that wraps around a well known power system simulator

systems might go wrong. Distributed system simulations exist to model the power flow in large power systems—analyzing the expected voltages, power losses, and any faults that occur. Thus, work done on distributed system software is essential towards building more efficient, fault-tolerant, and safe distributed power systems.

1.2 PV systems and Electric Vehicles

The last decade has seen a major rise in the use of household electric vehicles and photo voltaic systems. Electric vehicles serve as both consumers and producers, and photovoltaic systems turn a typical power consumer into both a consumer and producer. Electric cars now account for over one in five cars sold, and are making progress towards becoming common household products globally [2]. Similarly, it is projected that U.S. solar power generation will grow 75% from 2023, rising to nearly 300 billion kWh by 2025 [3]. This accounts for the majority of the growth in U.S. electricity generation [3]. With such a drastic rise in the use of such renewable energy resources, it is critical to test the effect of integrating these resources in electrical networks. PV (photovoltaic) systems, the solar panels attached to houses, will thus be an important part of circuit modeling for the introduced package.

1.3 OpenDSS

Many distributed power system simulations are done through OpenDSS[1], a free, open-source, scalable platform designed for simulations. OpenDSS uses its own DSS scripting language as opposed to an existing coding language. OpenDSS is an object-oriented platform, representing each component of an electrical network (each consumption load, power generation unit, power line, etc.) as an object, and defining hierarchies between these objects. The circuit built using OpenDSS can then be "solved", allowing the user to then query a variety of characteristics about the electric circuit at steady-state: what the voltages at each node are, what the total power used is, etc. OpenDSS supports time series analysis, allowing users to enter in load consumption over a period time and assessing the power flow output across a varying load. It is frequently used to perform standard tests like the IEEE test cases, and is designed around easy imports of network data.

Since OpenDSS exists as a separate platform and syntax, there exist packages developed to bridge the gap between OpenDSS and common syntax in pythonic toolkits. This will be discussed in more detail in Chapter 2. The work described by this thesis also aims to transfer OpenDSS' powerful simulation capability to a functional interface.

1.4 Goals of PyGridSim

PyGridSim aims to integrate the logic of OpenDSS and AltDSS to a pythonic interface that easily and efficiently builds circuits. PyGridSim is a distributed system simulator that enables varying levels of customizability, and allows users to generate circuits on a large scale with fewer lines of code. Like OpenDSS, PyGridSim supports the creation of PVSystem objects, to help effectively measure modern circuits with integrated solar panels.

The remainder of this thesis presents the PyGridSim interface and how it stands out from other distributed system simulators. Chapter 2 will discuss relevant works in the space of distributed power system simulations and how they motivate the work of PyGridSim. Chapter 3 will discuss the PyGridSim library, highlighting its design choices in a component-by-component deep dive. Then, Chapter4 will build up a variety of examples, to demonstrate how users would be able to create circuits with the PyGridSim interface. Finally, Chapter 5 will discuss PyGridSim's contributions and shortcomings and Chapter 6 will review and conclude the discussion on PyGridSim.

Chapter 2

Related Works

Many similar works aim to simulate distributed systems and analyze their outcomes with varying parameters in order to improve the safety and consistency of electric systems. However, much of the work in this field is done from electrical engineering experts, and there is limited research done from a computer science perspective. PyGridSim aims to bridge the gap between the power of distributed system simulators like OpenDSS and a usable interface for users with a computer science based background. There are a few other works that also aim to make distributed system simulations more accessible, sharing some goals with PyGridSim.

2.1 AltDSS

AltDSS [4] is an open source python package that improves on the more primitive python translations OpenDSSDirect.py and DSS-Python. Whereas OpenDSSDirect.py and DSS-Python do support circuit creation on a python file, many of the commands are just OpenDSS commands entered as a string form. Thus, they primarily serve just as an engine for python programs to call OpenDSS on the backend. On the other hand, AltDSS takes steps to make it more pythonic, leveraging typical python classes, methods, and class variables.

Like OpenDSS, AltDSS is a powerful and customizable tool to create distributed system simulations and analyze their results. AltDSS provides most of the circuit components that

OpenDSS does, and does so with pythonic function calls. Of the previous discussed papers, I believe AltDSS is the most fitting existing package for building pythonic circuits. As a result, AltDSS is the python package that will be cited the most throughout the paper, and is what PyGridSim ultimately uses to make intermediary calls to OpenDSS.

However, there are still scenarios where users may want to create circuits with AltDSS but find the interface repetitive or tedious. For example, users might not want to explore the full set of customization options for each component or manually construct every element one by one. PyGridSim builds on top of AltDSS to address these gaps, aiming to offer a more streamlined and user-friendly experience for distributed system simulations.

2.2 PyDSS

Another open-source python package in the space of distributed system simulations is PyDSS [5]. PyDSS is a python wrapper focused on extending the analytical and visual components of OpenDSS. It is based off of the package OpenDSSDirect to serve as a medium for backend OpenDSS access. PyDSS allows users to customize simulations, generate snapshot simulations at different points, and generate plots and other post-processing scripts. Users are then able to export many different files based on what analysis they want to save.

The goal still differs somewhat from PyGridSim—with a focus on visualization and analytic extension as opposed to a cleaner interface. However, it still serves as a valuable reference in the space at the intersection of python programming and simulations done with OpenDSS.

2.3 Hybrid Simulation

Hariri, Newaz, and Faruque explore the space of Python–OpenDSS simulations in their work "Open-source Python–OpenDSS Interface for Hybrid Simulation of PV Impact Studies" [6]. They incorporate an EMT model of a PV (photovoltaic) system into the existing OpenDSS environment to study the impacts of PV integration in distributed circuits. The authors justify the use of a hybrid EMT–phasor approach, arguing that it offers advantages over using a single modeling strategy.

PyGridSim will also incorporate modern elements like PV systems into its circuit construction. However, unlike Hariri et al.'s focus on hybrid simulations, PyGridSim aims to support both traditional and modern circuit models, with or without photovoltaic components. Still, their work stands as an important contribution to the broader use of Python-based OpenDSS in simulation studies.

Chapter 3

PyGridSim Interface

3.1 OpenDSS and AltDSS: What's Missing

The primary tool for Distributed Systems Simulators is OpenDSS, a platform with its own scripting langugae that does distributed energy resource grid integration. OpenDSS is a comprehensive interface to create a customizable circuit with various standard and modern circuit components. In order to make this technology and simulation power more accessible, various packages have converted this to a python-callable interface. Among the more comprehensive of these python packages is AltDSS, which builds on the DSS-Python backend infrastructure. While the interfaces of OpenDSS and AltDSS provide the user with a ton of customization and freedom to model various circuits, the syntax is less intuitive and requires research and learning from the user end. Further, the circuits with these interfaces tend to be created with brute force: each circuit component needs to be initialized and specified on its own, making it difficult to build a circuit in a scalable fashion.

These shortcomings of OpenDSS and AltDSS require a new package that upholds their customizability and scope while managing a much simpler and user-focused interface.

3.2 PyGridSim

As a result of the concerns discussed in section 3.1, there is a need for a proper functional interface in the space of distribution system simulations. PyGridSim is a python package that leverages the tools given by OpenDSS and AltDSS to create electrical circuits more efficiently, allowing users to run scalable fault simulations with fewer lines of code than possible before.

To begin with, this table provides a high level overview of the differences between the OpenDSS, AltDSS, and PyGridSim packages. Then, we will delve into how this table reflects the high level goals of PyGridSim, as well as its primary limitations.

Table 3.1: Feature Comparison of DSS Packages

Feature	OpenDSS	AltDSS	PyGridSim
Distribution System Simulator	Yes	Yes	Yes
Customizable Parameters	Yes	Yes	Yes
Python-Interface	No	Yes	Yes
Batch Insertion	No	Yes[1]	Yes
Given Pre-Defined Nodes	No	No	Yes
Complete Wrapper of DSS Capabilities	Yes	No	No

3.2.1 What PyGridSim Is

PyGridSim is a python package (compatible with Python 3.9-3.12) designed to be a functional interface for users to create and simulate various electrical circuits. It uses the circuit building logic from AltDSS and OpenDSS, and provides an alternate user-friendly API to reduce function calls and repetitiveness in object creation. The two primary avenues through which PyGridSim achieves this efficiency is through providing parameter sets as well as efficient batch creation.

Pre-Defined Nodes

PyGridSim's focus is on allowing developers to create various modern fault simulations using a circuit-building interface that focuses on realistically representing neighborhoods and towns. As a result, PyGridSim provides an extensive set of pre-defined nodes that users can lean on, which represent a set of default parameters that can be assigned to a generic node type. This maps vague components that the user may want to create (i.e. a standard residential home) to a set of values that can be assigned to that component. Then, for instance, a user can specify that they want to create many instances of a "house" load node, without having to re-research what typical kV, kW, etc. values of a house may be.

This primary feature enables a user can create circuits without extensive knowledge of what a reasonable value for each parameter to be, and more efficiently build a realistic circuit. Note that it doesn't negate the customizability of the circuit—users who seek to customize their parameters can still choose to do so. They can choose to entirely provide parameters, entirely rely on PyGridSim's parameter set defaults, or any combination of the two (providing the parameters they want to provide and using PyGridSim's defaults for any they don't provide).

Note that OpenDSS (and adjacent packages like AltDSS) have their own default values for when a parameter is underspecified. Note that these are singular values, not ranges, and there is not a set meaning for what this default value should represent, so it does not accomplish what the pre-defined node ranges of PyGridSim does.

Batching

As defined in the overall module goals, PyGridSim wants to simplify the user interface to create the same circuit as in other distributed system simulators. A principal strategy to meet this goal is what we call "batching". A user may seek to create an large n number of load nodes, without wanting to individually build and add each node to the circuit. Instead of requiring users to make several function calls to create each node, PyGridSim encourages the user to add a set of nodes all at the same time.

Note that, as indicated by the chart, a batching functionality does also exist as a recent improvement to the AltDSS interface, added in 2024[7]. However, batching in AltDSS is still not handled in the same way that it is in PyGridSim. For instance, if the user were to create a batch of 15 nodes, they would have to pre-specify the parameters for each of the 15 nodes as a list of 10 tuples, then batch insert them with AltDSS. An example AltDSS' batch functionality is shown below.

```python
# Create loads_data
loads_cols = 'name,Bus1,Phases,Conn,Model,kV,kW,kvar'.split(',')
loads_data = (
    ('671', '671.1.2.3', 3, Conn.delta, LoadModel.ConstantPQ, 4.16, 1155, 660),
    ('634a', '634.1', 1, Conn.wye, LoadModel.ConstantPQ, 0.277, 160, 110),
    ('634b', '634.2', 1, Conn.wye, LoadModel.ConstantPQ, 0.277, 120, 90),
    ('634c', '634.3', 1, Conn.wye, LoadModel.ConstantPQ, 0.277, 120, 90),
    ('645', '645.2', 1, Conn.wye, LoadModel.ConstantPQ, 2.4, 170, 125),
    ('646', '646.2.3', 1, Conn.delta, LoadModel.ConstantZ, 4.16, 230, 132),
    ('692', '692.3.1', 1, Conn.delta, LoadModel.ConstantI, 4.16, 170, 151),
    ('675a', '675.1', 1, Conn.wye, LoadModel.ConstantPQ, 2.4, 485, 190),
    ('675b', '675.2', 1, Conn.wye, LoadModel.ConstantPQ, 2.4, 68, 60),
    ('675c', '675.3', 1, Conn.wye, LoadModel.ConstantPQ, 2.4, 290, 212),
    ('611', '611.3', 1, Conn.wye, LoadModel.ConstantI, 2.4, 170, 80),
    ('652', '652.1', 1, Conn.wye, LoadModel.ConstantZ, 2.4, 128, 86),
    ('670a', '670.1', 1, Conn.wye, LoadModel.ConstantPQ, 2.4, 17, 10),
    ('670b', '670.2', 1, Conn.wye, LoadModel.ConstantPQ, 2.4, 66, 38),
    ('670c', '670.3', 1, Conn.wye, LoadModel.ConstantPQ, 2.4, 117, 68),
)

# Create new loads in circuit
name, bus1, phases, conn, model, kV, kW, kvar = zip(*loads_data)
Load.batch_new(name, bus1=bus1, conn=conn, phases=phases, model=model,
               kV=kV, kW=kW, kvar=kvar)
```

Figure 3.1: Batch Creation with AltDSS[7]

In Figure 3.1, the user is able to call the AltDSS `Load.batch_new` function that allows the loads to be added to the circuit in a group, as opposed to having to call `Load.new` 15 different times. However, in order to use this shortcut function, the user still must write or import the `kV`, `kW`, etc. of each individual node, as seen in `loads_data` to feed into the batching call.

Thus, if a user were to have a pre-formatted list they want to import into a circuit, this AltDSS batch functionality could save many function calls. However, if a user were to model a circuit from scratch, this functionality would still require a lot of data creation to make each individual node, and not save much time. On the other hand, with PyGridSim, a user can create a batch of 15 similar tuples at once. In combination with the pre-defined nodes, a user who seeks to create 15 nodes that are "house-like" can do so in just the following single function call:

```
circuit.add_load_nodes(num=15, load_type="house")
```

Figure 3.2: Simple Batch Creation with PyGridSim

3.2.2 What PyGridSim Isn't

It's important to highlight that PyGridSim is not a comprehensive wrapper of all of OpenDSS functionality. Similarly to AltDSS, PyGridSim is designed to be a partial wrapper of OpenDSS: it leverages OpenDSS tools to create circuits, but does not yet support every component and function call that OpenDSS does. This tradeoff occurs as a result of PyGridSim's focus on creating efficient circuits—every component is designed to be scaled efficiently, so not every component is supported. While PyGridSim implements most major components of an electrical circuit—sources, loads, generators, transformers, and PVSystem—it doesn't support certain specialized circuit components like meters, monitors, fuses, etc. However, PyGridSim is built to be flexible, so these other components could be integrated into the package easily in the future.

Further, at its current state, PyGridSim does not provide complex fault analysis. Whereas OpenDSS includes specialized algorithms for analyzing different types of faults on the circuit, PyGridSim currently focuses on simpler numerical queries after solving a circuit. However, PyGridSim is built with the end goal of fault analysis in mind, and is flexible to the appending of a functional fault analysis interface in the future.

3.3 Library Overview

PyGridSim is designed to be a python package that enables simple creation of distributed system simulations. As with OpenDSS and AltDSS, PyGridSim first prompts the user to create an empty circuit, and builds the circuit by adding components to it. Once the user finishes building their circuit, they can transition it into a solve mode and query the end behavior of the circuit they built. The novelty and focus of this package is to allow users to do fewer instantiations to make a repetitive circuit, allowing efficient creation of large circuits. PyGridSim is designed around the principles of being *efficient* and *intuitive*—it aims for users to build a circuit intuitively without having to make miscellaneous function calls. Note that PyGridSim also aims to be *customizable* in the way that OpenDSS and AltDSS are.

In order for the library to meet these principles, PyGridSim must allow for multiple avenues of object creation. To keep up with the customizability of other DSS interfaces, the user still needs to have the freedom to customize their circuit components as desired. However, to be as intuitive as possible, PyGridSim also wants to provide users a way to do less manual parameter inputs to create their desired circuit. The perpetuation of these two goals is what leads to the following two forms of circuit instantiation:

- **Customized Parameters**: Users can customize the parameters of each circuit component they create, by specifying a dictionary called `params`. 3.4 will discuss the exact customizations allowed by each supported circuit component. The keys of the dictionary are the names of parameters they want to specify, such as kV, kW, and phases, and the values of the dictionary are the (numerical) values that they want to set each parameter to. This customizability exists to maintain the customization comprehensiveness of OpenDSS/AltDSS, allowing the user to define any valid values to make the exact circuit they intended to.

- **Provided Pre-Defined Nodes**: To accommodate a user who does not intend to entirely customize each component, PyGridSim provides a set of "default parameters" for each circuit component type that represent a pre-defined node. The user can choose to specify which default set they want to use as a string, which indicates the high-level intention of that default sets. For example, a user may want to represent a neighborhood at the high-level, and want to build many "house-like" nodes. Instead of researching the expected parameters for a typical house node, they can just create a "house"-type load node, which will randomly create a set of parameters that match that of a standard house. A detailed description of each pre-defined node, their ranges, and the source of their ranges is provided in 3.4.

Note that the interface allows the user to do a combination of these two forms for any circuit component. The user can specify a non-empty parameter list as well as opting to use one of the pre-defined parameter sets. In this case, the explicitly specified parameter list takes priority, and the pre-defined parameters are used for any unspecified parameters. This flexibility of the PyGridSim package aims to allow users to customize their circuit to the degree they see fit, and intuitively settles conflicts between these instantiations.

Further, in line with the *efficiency* principle of PyGridSim, users need to be able to perform the batch creation discussed in 3.1. Each circuit component supported by PyGridSim allows the user to specify a `num` parameter. Regardless of if the user is manually defining parameters or using package-defined parameters, this specification will support the batch creation of `num` such circuit components.

3.4 PyGridSim Circuit Components

As discussed in **??**, the construction of PyGridSim revolves around the goals of providing customizability and default parameter. These principles for circuit creation apply to each circuit component that PyGridSim is designed to create. This consistency across circuit objects helps ensure an easy to learn interface that is most intuitive to the user. AltDSS and OpenDSS have major inconsistencies in how to instantiate different components in the circuit, which makes it harder to adapt to these interfaces. Therefore, PyGridSim tries to minimize the differences between the function calls to initiate each circuit component. This section will detail the design choices behind each circuit component.

On a high level, in order to build a circuit, users must specify load nodes that consume power (3.4.1), source nodes that produce power (3.4.2), and connections between these nodes (3.4.3). We begin by describing how users can use PyGridSim to create the load nodes in the circuit.

3.4.1 Load Nodes

Users can initialize any number of load nodes at the same time, as a part of the batch creation principle discussed above. They are then able to follow the syntax of either customized parameter of provided parameter sets for the initialization of these group of nodes. Note that, in either initiation method, load nodes will be given the names "load0", "load1", etc. based on the number of loads created in the circuit thus far. These names are important to recall as users may want to reference them later when building lines, transformers, and PVsystems with loads.

Customized Load Initialization

With PyGridSim, users can create a batch of *num* load nodes with the same parameters at once. They can customize the parameters to build these consumption nodes—most importantly, they can set the `kV` (voltage level), `kW` (real power consumed), and `kVar` (reactive power) of the nodes they want to create. Any unspecified parameters will use the PyGridSim defaults, which are taken from the OpenDSS/AltDSS defaults. In OpenDSS, if the user were to create a set of many load nodes with the same parameters, they would have to individually add each load node. Similarly, with the batching of AltDSS, users would still have to define a list containing the set of parameters repeated *num* times, before making the batch call. Therefore, using PyGridSim allows them to streamline the batching process when they seek to create several identical nodes at once.

Since one batch creation of load nodes with the customizable parameters can only create exactly identical nodes, the users may want to create multiple batches of nodes to model their circuit. The following figure demonstrates the creation of multiple batches of load nodes, each with different levels of customization.

```python
circuit.add_load_nodes(num=10, params={"kV": 10, "kW": 10, "kVar": 1})
circuit.add_load_nodes(num=3, params={"kV": 5})
load_params = {"kV": 20, "kW": 5, "kVar": 1, "phases": 3, "R0": 0.2, "R1": 0.3}
circuit.add_load_nodes(num=5, params=load_params)
```

Figure 3.3: **Customizable Load Initialization:** The user chooses to create 10 load nodes with one set of parameters, and 3 load nodes with another set. They can also choose to specify parameters in an outside dictionary and feed that into the add_load_nodes function call. Any unspecified parameters will be assigned the default value.

Defaults-Based Load Initialization

A user may also want to create load nodes in their circuit without having to specify an entire list of customized parameters. If the user creates a batch of defaults-based load nodes, they will be creating *num* "similar" nodes for the system. This occurs as a result of the range-based nature of the pre-defined node types. It means that these nodes will have close but not identical parameters, but will be modeling the same type of node. This design choice is to reflect the randomization of the real world—each house on a neighborhood may have a similar consumption level, but it's unlikely we observe every building on a street consuming the exact same level of power.

If the user wants to specify a parameter set for load nodes, they can choose between *house*, *commercial*, and *industrial*. A user that does not specify any parameter set (and might also not be customizing any parameters) will have their load nodes defaulted to the "house" type, as this is one of the most applicable real-world instances of a consumption node.

The following code snippet demonstrates ways to create a batch of load nodes using the pre-defined nodes. Note that since "house" is the parameter set the code defaults to, the first two calls are synonymous:

```
circuit.add_load_nodes(num=20, load_type="house")
circuit.add_load_nodes(num=20)
circuit.add_load_nodes(num=5, load_type="industrial")
```

Figure 3.4: **Defaults-Based Load Initialization:** In the first two lines, a user creates a batch of 20 house-like load nodes. The user then creates 5 industrial-like load nodes.

Load Ranges

The following subsection aims to explore more into the design choice behind the three chosen load categories, and the ranges that each type captures. The load ranges are split into three defined categories to reflect three fundamentally different consumption profiles.

- **House (default)**: The House use case represents the simplest unit for a load consumer, and the most standard use case for a load node. House loads tend to be low voltage and highly variable, with low levels of reactive power.
- **Commercial**: The Commercial use case captures an offices, schools, stores, etc. Their behavior as loads differ from houses because they use more reactive power as a result of requiring more inductive equipment.
- **Industrial**: The industrial use case represents the biggest load consumers, industrial facilities. Industrial facilities have higher power usage than residential and commercial consumption profiles, and particularly use up even more reactive power, making them a necessary distint case. They are the biggest contributors towards power quality issues and are therefore a notably distinct case for fault analysis.

These three consumption profiles together encapsulate a fairly complete range of typical loads in an electrical power system. Most end-user descriptions of a load consumer can be categorized into one of these three. Below is a summary of the key components of the parameter set from each one, with data collected from research on typical load standards[8][9].

Table 3.2: Parameters of Various Load Profiles

Name	kV	kW	kVar
House	[0.12, 0.24]	[1, 1.5]	[0.5, 1]
Commercial	[0.24, 0.48]	[10, 50]	[5, 10]
Industrial	[4.16, 34.5]	[200, 10000]	[150, 480]

Naming Load Nodes

Optionally, users can choose to add names to their load nodes. This is also specified as an optional parameter in load initiation titled `names`. If a user is creating n nodes, they can set `names` to a list of up to n strings. This essentially gives the user "nicknames" to identify these nodes, for when they want to reference the load in a line or PVSystem generation. Consider the example of load name initialization in the following figure:.

```
load_names = ["632", "633", "645", "646"]
circuit.add_load_nodes(num=5, load_type="house", names=load_names)
```

Load Name	**Alternate Name**
"load0"	"632"
"load1"	"633"
"load2"	"645"
"load3"	"646"
"load4"	—

Figure 3.5: **Specifying Load Names with PyGridSim:** All the load nodes have an internal load name as well as an alternate one, except for load4 since only 4 load names were specified.

Note that these loads will still be numbered, and assigned the name *loadk* as described above. Thus, any named nodes can be later referenced through their number or their name, interchangeably. For instance, a user can use "load0" and "632" interchangeably for the remainder of their circuit creation. Also, since the user only specified a list of 4 names for a batch of 5 nodes, the last node does not have a special name and can only be referenced as "load4". Note that even when providing a list of names, the user still has to specify the number of nodes to build—if a user were to specify than *num* names or to not set *num* as a non-default parameter, the program will throw an error on the assumption that the user is misunderstanding how to batch create nodes.

The main motivation for this design choice is that users may seek to model their circuit after an existing circuit, like an IEEE13 standard circuit. Note that the users can set the alternate names to any strings—the examples use numbers (inputted as strings) because IEEE13 often assigns numerical labels o their nodes. These circuits may already have predesignated load names, like 632 and 633. Thus, when users have to map an IEEE13 description of a circuit into a PyGridSim circuit, it will be much easier for them to be able to match the image names with the desired corresponding node. A more thorough example will be unpacked in chapter 4, but this type of circuit creation is what leans on this type of design choice.

3.4.2 Source Nodes: Vsource, Generators, PVSystem

In addition to supporting load nodes, the power consumers of the circuit, the interface supports the creation of source nodes, the power producers of the circuit. There are various forms of sources supported by PyGridSim, each helping produce power to the circuit in various different ways. This section will outline the three primary sources that are accessible through PyGridSim, discuss the design choices behind differentiating these sources, and provide examples on how to perform the two forms of initiation for these sources.

Voltage Source ("Vsource")

As with AltDSS and OpenDSS, PyGridSim has one centralized Voltage Source object for a single circuit. The intent bethind this design is that distributed system analysis relies on a single slack bus to serve as a reference point. This does not mean that the circuit can only be powered by one voltage producer, but rather that other producers must be of different circuit types. Since the user cannot create multiple of these voltage sources, the interface for

`Vsource` differs from the other components, and cannot be batched. Thus, maintaining this voltage source will be through a function titled `update_source`, that creates the Vsource node on the first call and updates its parameters on future calls. When it needs to be referenced later, this Vsource will have the name “source” and cannot be given an alternate name.

In order to specify this source node, the user once again can choose between allocating a customized set of `params` or using one of the provided default sets. In addition to setting voltage and power, users can also set impedance parameters `(R0, R1, X0, X1)` to further fine tune the resistance and reactance properties of the node. PyGridSim supports *turbine*, *power plant*, *LV substation* (low voltage substation), *MV substation* (medium voltage substation), *HV substation* (high voltage substation), and *SHV substation* (super high voltage substation) as the principle voltage source in the system, with *turbine* being the default. The choice to include source types as well as customization is to remain consistent with the rest of the PyGridSim interface, and give the users the choice to create with the level of customization that they want to.

```
1  # Customized VSource Initialization
2  vsource_params = {"kV": 100, "R0": 0.1, "R1": 0.2, "X0": 0.3, "X1": 0.4}
3  circuit.update_source(params=vsource_params)
4  # Defaults-Based VSource Initialization
5  circuit.update_source(source_type="power plant")
```

Figure 3.6: **Vsource Initialization:** The user creates a central source node into their circuit. They can do this in two ways: manually customizing a list of parameters (lines 2-3) or using one of the defaults for sources (line 5).

Consider the shown ways of calling `update_source` as shown in Figure 3.6. Note that the interface intends that the user should pick only one such call if they were to actually create a circuit—duplicate calls to `update_source` are permitted, but the second call completely overrides the parameters specified in the first one.

VSource Ranges

The types of voltage sources described above are distinguished based on varying use cases, as well as widely differing voltage use cases. The voltage source load type differentiates a turbine, power plant, and four substation levels. The four substation levels are differentiated based on standard substation classification, and their parameters are assigned accordingly.

- **Turbine (default)**: Turbines represent a small-scale generation source. For certain grid scenarios, turbines serve as an effective central source in the system, making them important to include in the PyGridSim interface.
- **Power Plant**: Power Plants represent another possible central source for the system, with higher voltage levels reflecting a larger-scale generation facility. These have differing fault behavior and notably larger voltage levels than turbines, making them a necessary distinct category of Vsource. Further, they are a common choice to anchor many large power systems, so are critical to include as a Vsource option.
- **Substations**: Substations also serve as a distinct possible central source that may be effective to model various different grid scenarios. Substations are generally categorized into one of four categories: low voltage, medium voltage, high voltage, and super high voltage, so PyGridSim reflects these categories. Low voltage substations may represent a central source for a neighborhood-scale distribution systems, medium voltage substations may represent a radial distribution network, high voltage substations can serve as a source for larger scale distribution systems, and super high voltage substations could centralize a macro-level distribution systems.

In all, the turbine, power plant, and substations represent critical distinct distribution systems scenario, and provide fairly complete coverage on the possible distribution systems that the user may seek to represent. The user can analyze which scale and geometry their system best fits into, to build the most applicable corresponding distribution system. The following table summarizes the kV ranges assigned for each of the six Vsource values. These ranges were taken from research done on typical voltage source power ranges[10][11].

Table 3.3: Parameters of Various VSource Profiles

Name	kV
Turbine	[1,3]
Power Plant	[10, 20]
LV Substation	[0.2, 0.4]
MV Substation	[6, 35]
HV Substation	[66, 350]
SHV Substation	[500, 1000]

Generators

Generators are another way in which users can include additional voltage source throughout a circuit system. Like Vsources, they provide electrical energy and power to the circuit. Unlike Vsources, they work primarily from converting mechanical energy into electrical energy. Generators can be added to any point in the system, and there is not any limit on the number of generators that can be added to a distributed system. So, the creation of generators through PyGridSim can be batched, much like load nodes.

The interface for generators remains consistent with that of load node creation. Users can opt for their own customized `params` or use one of the default sets for generators. PyGridSim supports the creation of *small*, *large*, and *industrial* generators, with *small* being the default. More on these generator types will be detailed in the next subsection. Note that the naming functionality of load nodes also carries over to generators. Generators are named according to the number of generators already in the system, titled “generator0”, “generator1”, etc. However, users that seek to name their generators can choose to specify a list of `names` up to `num`, which will serve as naming shortcuts to represent those nodes.

```python
# Customized Generator Initialization
circuit.add_generators(num=5,params={"kV": 200, "kW": 1000})
# Defaults-Based Generator Initialization
circuit.add_generators(num=10, gen_type="small")
# Naming: Creates single generator with name "650"
circuit.add_generators(gen_type="large", names=["650"])
```

Figure 3.7: **Generators Initialization:** The user creates a batch of 5 generators with the same customized parameters. The user can alternatively create a batch of 10 generators that all represent a unique small generator. Finally, users can also choose to give an alternate name for their generators upon creation, which is "650" here.

Figure 3.7 displays how to add generator(s) through the PyGridSim interface, showing the features of customization, pre-defined nodes, and naming.

Generator Ranges

Now that users are familiar with how they can create these default-based generators, this subsection will dive more into the division between these generator types and the choices behind dividing them this way.

Generators are a type of source that converts mechanical energy into electrical energy, and they may take different forms. As a result, users may want to build generators in their system based off of varying preset types. In order to represent varying scales of generators the user may want to create, PyGridSim supports three distinct levels of generators: small, large, and industrial. These categories are common distinctions that reflect not only a difference in voltage, but the context on which the generator operates.

- **Small (default)**: Small generators operate at lower voltages, and have lower short-circuit strength, making them perform distinctly in fault analysis. As a result, they are a necessary separate component to represent in the PyGridSim interface.
- **Large**: Large generators represent larger-scale sources, providing more real and reactive power to the circuit than small generators. Such generators are still common but

operate on a different level than small generators, and are thus included as a separate type.

- **Industrial**: Industrial generators often have a distinct use case than the other two. They tend to be used for industrial facilities, either for backup or for continuous use. Their size and use case differ enough from the prior two description of generators that they should be modeled distinctly.

Ultimately, the small, large, and industrial capacities of generators represent three critically distinct categories of roles the generators take up in the grid, as well as the power levels they provide. In conjunction, they offer comprehensive coverage of the generator types most relevant to both distribution systems. The user can assess the scale, location, and other characteristics of their generators to model the most representative behavior for their system. Below is a table representing the key parameters, kV and kW, of the different generator types[12].

Table 3.4: Parameters of Various Generator Profiles

Name	kV	kW
Small	[0.2, 0.6]	[2, 5]
Large	[1, 35]	[5, 10]
Industrial	[35, 100]	[10, 20]

PhotoVoltaic system ("PVSystem")

In modern distributed systems, there may be nodes that serve as both consumers and load nodes. The primary real-life example of this is a house that has solar panels on it. In addition to using up power, the house is also generating power through solar panels. PyGridSim supports this through the same PVSystem component that OpenDSS does. It is essential that a modern distributed system like PyGridSim supports PVSystems, because PVSystems represent the present and future of electrical circuits. Because these photovoltaic systems are designed to represent solar panels attached to a consumer node, the interface attaches PVSystems to a corresponding load object. (For instance, it "adds" solar panels to an existing "house" in the graph).

To add PVSystems with the PyGridSim interface users must specify a list of `load_nodes` to add PVSystems to. Then, the users can choose to either specify the parameters of this PVSystem through `params` or use the PVSystem default parameters. Note that these two choices remain consistent with the rest of the PyGridSim interface, in order to make the interface functional and easy to adapt to. However, there was not enough varied different "types" of PVSystems the way there are with loads and generators, so there is no PVSystem type to specify—users creating the PVSystem will have the single set of PVSystem parameters defaulted to automatically.

Finally, initialization of these PVSystem objects allows the user to specify `num_panels` symbolizing the number of solar panels they want each PVSystem object to correspond to. If users do not specify `num_panels`, it defaults to 1. Then, PyGridSim will create a randomized PVSystem object that represents a solar panel system with the number of panels that the user specifies. Because the power output of solar panels roughly scales linearly[13], the power output of `num_panels` solar panels will be calculated accordingly.

```python
# Customized PVSystem Initialization
pv_params = {"kV": 200, "kW": 1000}
circuit.add_PVSystems(load_nodes=["load0"], params=pv_params, num_panels=100)
# Default PVSystem Initialization (no parameter specification)
circuit.add_PVSystems(load_nodes=["load1", "load2"], num_panels=10)
circuit.add_PVSystems(load_nodes=["load3"])
```

Figure 3.8: **PVSystem Initialization:** The user chooses to create a customized PVSystem on load0 representing 100 solar panels each with the voltage levels given. They also choose to add 10 generic solar panels to load1 and load2, and a single generic solar panel to load3.

The creation of PVSystem objects through PyGridSim is demonstrated above in Figure 3.8. This assumes the previous creation of at least 4 load nodes, otherwise the program would error in trying to add a PVSystem to a load that doesn't exist. Further, note that the lists are provided in terms of the default load names (i.e. `[‘‘load0’’, ‘‘load1’’]`). If the user gave the loads an alternate name upon load creation, they are also able to call those alternate names as the load nodes they want to add PVSystems to. For instance, if `load1` was binded to the name 633 in the load call, line 5 could optionally replace "*load*1" with "633" for the same functionality.

Overall, users can create these source objects and use them in tandem, or choose to only stick to one centralized source object. PyGridSim provides the flexibility of various source types, so that users can select the source objects that represents their desired circuit setup the best.

3.4.3 Lines and Transformers

In order to create a proper circuit on PyGridSim, the user also needs to create lines to connect their different circuit components together. In order to do this, the user needs to be able to specify which pairs of components they want to connect, and optionally add any special parameters for their lines. In order to support potential batch creation of lines, the user is prompted to enter a list of `connections`: a list of tuples representing the paris of nodes to connect.

Customized Line Initialization

To create a customized line, the user can enter whichever parameters they wish to in the `params` field. The support of this line customization is designed to match the creation of the other circuit components, noting that consistency will make this an easy to understand interface. However, note that the main parameter the user can alter here currently is the `length` parameter (in km). Examples of this customized line initialization are shown below, assuming a Vsource and at least 4 load nodes have already been created.

```
circuit.add_lines(connections=[("source", "load0")], params={"length": 100})
connections_list = [("source", "load1"), ("source", "load2"), ("source", "load3"),
                    ("load1", "load2"), ("load1", "load3")]
circuit.add_lines(connections=connections_list, params={"length": 200})
```

Figure 3.9: **Customized Line Initialization:** The user creates a single line that is 100km long between the source and load0 nodes. The user then specifies a batch of other connections they want to make and creates lines of length 200km.

Defaults-Based Line Initialization

As with other components, PyGridSim supports various types of lines that the user can input as a string in the `line_type` field. The options supported are *lv* (low voltage line), *mv* (medium voltage line), and *hv* (high voltage line), with *lv* being the default.

```
circuit.add_lines(connections=[("source", "load0")], load_type="mv")
connections_list = [("source", "load1"), ("source", "load2")]
circuit.add_lines(connections=connections_list, load_type="hv")
```

Figure 3.10: **Defaults-Based Line Initialization:** The user creates a line between source and load0 representing a generic medium-voltage connection. They then create generic high voltage lines between source and load1, and source and load2.

Figure 3.10 demonstrates the initialization of a few medium and high-voltage lines, assuming the corresponding source and load nodes have previously been initialized.

Line Ranges

This subsection will go into more detail on the divisions between the different line types supported by PyGridSim. PyGridSim distinguishes these lines into three categories—low voltage, medium voltage, and high voltage. These types represent fundamental differences in how voltage is carried across the lines, and thus reflect necessary distinct use cases. Importantly, higher voltage allows for longer-distance transfer of voltage with minimized losses. As a result, in the real world, longer lines can handle higher voltage and lines that only need to carry low voltage tend to be shorter.

- **Low Voltage (default)**: Low voltage lines will be the most common to connect individual loads, often representing the transmission lines between neighboring individual houses or buildings.

- **Medium Voltage**: Medium voltage lines can span much longer than low voltae lines can, carrying higher voltage transmissions on a neighborhood-scale.

- **High Voltage**: High voltage lines are the most effective for transmitting high operating voltage across a larger scale with minimal losses. Thus, they operate at a different voltage level and distance than the other two line types, necessitating a distinct category.

Overall, the three of these voltage categories, in combination, should represent the majority of end use cases for a circuit build. The following table details the profiles of these three line types, noting that a line that carries a higher voltage is reflected by a longer length in km.

Table 3.5: Parameters of Various Line Profiles

Name	length (km)
Low Voltage	[30, 60]
Medium Voltage	[60, 160]
High Voltage	[160, 300]

Transformers

Just like its parent packages OpenDSS and AltDSS, PyGridSim prioritizes the ability for users to add transformers to their circuit. Transformers are essential to accurately representing a circuit for a few reasons. To begin with, transformers are able to handle voltage differences across a circuit, such as between a connected source and load node. Without transformers, mistmatched voltage just overloads, and the circuit always has an equal voltage output between all nodes, which is an inaccurate model. Further, on an electrical level, transformers build an essential foundation towards more fault-resistant electrical circuits[14]. The existence of transformers help isolate any node faults, without having them propogate to all of their connected nodes.

As a result, PyGridSim prioritizes the existence of transformers by having them as the default—a transformer will exist unless specified otherwise. Specifically, each line built through the `add_lines` function will have a transformer attached to them. Therefore, there is no separate `add_transformers` function, the customization is baked into the `add_lines` function. The user can customize their transformer however they desire in the same `params` field that they specify any line parameters in—PyGridSim supports the customization of *windings*, *XHL*, and *Conns* of the transformer.

Of course, the user may want to specifically *not* include transformers—perhaps to test faults in a transformer-less system, or see how the voltage distribution differs without a certain transformer. So, PyGridSim still allows a user to circumvent this default (assigning `transformer=False`), but it expects transformers to usually be used. Therefore, the interface opts for using a transformer as the default setting instead of requiring extra calls in order for one to be implemented.

```python
# Customized transformer
transformer_and_line_params = {"length": 50, "XHL": 3, "Windings": 4}
circuit.add_lines(connections=[("source", "load0")], params=transformer_and_line_params)
# Opt out of Transformer
circuit.add_lines(connections=[("source", "load1")], line_type="hv", transformer=False)
```

Figure 3.11: **Transformer Customization:** The user specifies transformers in the `params` field of add_lines, choosing to specify the number of windings and XHL. The user also creates a line at the end between source and load1 that does not contain a transformer.

A demonstration on how to customize transformer parameters, as well as how to explicitly avoid building a transformer is shown in Figure 3.11. Overall, the interface for building lines and transformers is designed around PyGridSim's major principles—keeping *efficiency* by avoiding extraneous extra calls for transformer initiation, and allowing for batch creation of many lines, while keeping *customizability* of the lines and transformers attached to them.

OpenDSS and AltDSS requires users to separately define each line and then a transformer to add onto that line for each pair of nodes.

3.5 Features of PyGridSim Ranges

A key component of PyGridSim's novelty is that it provides various application-based "defaults" that the user can use when specifying the types of circuit components they want to build. The user inputs these types as a string, and the string is translated to its corresponding internal Enum type. The component-based information on the ranges was detailed in 3.4, containing all available types for each circuit component as well as what the ranges are. The following section will discuss some of the design choices. behind these ranges and pre-defined node types.

3.5.1 Flexible Type Naming

First, note that these strings are not case-sensitive nor space-sensitive, so to specify a `source_type` of "powerplant", "power plant", "Power Plant", etc. would also work. Users may have different preferences on how to case their parameters, and this flexibility thus helps streamline the creation of these pre-defined nodes.

3.5.2 Randomization

A novel feature of PyGridSim is that users can leverage a built-in randomization feature when creating their nodes. Particularly, whenever users are creating pre-defined nodes, they are creating randomized nodes that fall under the range of reasonable values for their pre-defined types. The rationale behind this decision is that real world circuits will contain many similar values, but not necessarily many exactly identical nodes. Houses in the same neighborhood could have a similar scale of power consumption, but they will not have the exact same number of kV outputted. To ignore these variations between similar nodes could lead to an incomplete picture of all the potential faults in the system. Thus, for each of the parameters of a given node, PyGridSim chooses a random value (i.e. random kV) in the range to assign to that node. When a user creates a batch of `num` nodes, they will all be assigned a different random set of parameters, making them all retrieve different parameters in the same range.

Currently, PyGridSim chooses a random value assuming a uniform distribution. However, the library is written to be easy to modify and improve on—a future iteration could give the users the option to select a different type of distribution to randomly select from, or alternatively allow the users to specify their own manual distribution/function.

3.5.3 How to Find Ranges

Users are expected to return a string that correctly corresponds to a type of the component they are trying to create. They have a few options to learning which types are available:

- **Documentation:** PyGridSim comes with provided documentation that states all available pre-specified types for each circuit component. Users can reference this document to see which string matches their preferences best, or determine they want to instead customize their node if none of them match.

- **Internal Code:** Being open source, users can also view the PyGridSim source code if they want to see the specific types supported. The file `pygridsim/enums.py` lists all the string values that are valid to enter, and `pygridsim/defaults.py` stores their corresponding ranges.

- **Type Query Function:** Finally, a user is able to query for all available types and ranges. These query functions are also a part of the circuit class, but note that it should return the same sets of values independent of what circuit is created. The user can use the function `get_types()` with the `component` parameter set to any of `load, source, generator, line` to fetch a list of all corresponding types. Further, the user can specify `show_ranges = True` to get the output to include a dictionary-formatted set of ranges for each type in the list. The choice given by this `show_ranges` option reflects different user intentions—one user may find it sufficient to view the string titles to determine which type they want, whereas another may want to verify the type's internal parameters before deciding.

```python
print(circuit.get_types("load"))
# Expected result:
# ["house", "commercial", "industrial"]
print(circuit.get_types("load", show_ranges = True))
# Expected result:
# [("house", {"kV": [0.12, 0.24], "kW": [1, 1.5], "kVar": [0.5, 1]}),
#  ("commercial", {"kV": [0.24, 0.48], "kW": [10, 50], "kVar": [5, 10]}),
#  ("indsutrial", {"kV": [0.24, 0.48], "kW": [30, 100], "kVar": [20, 25]})]
```

Figure 3.12: **Calling Type Query Functions:** To understand what load types they should use in their circuit building, the user queries for all possible load types, and the ranges they correspond to.

The above figure demonstrates a call to `get_types(‘‘load’’)`, with and without the `show_ranges` specification, as well as their corresponding outputs.

Overall, PyGridSim supports these varying ways to query ranges so that the user can learn about the package in the way that might be most intuitive to them. PyGridSim wants to make the available inputs as clear as possible, to avoid any confusion on the user end.

3.6 Solve Mode and Results

When a user is done building their circuit in PyGridSim, they must transition the code into "solve mode" in order to view any results on the code. Once the circuit is in solve mode, changes to the circuit are frozen and the circuit begins to simulate the current steady-state of the circuit. The motivation behind having two separate phases of circuit building (build, followed by solve) is modeled after the other distributed system simulators, OpenDSS and AltDSS, that do the same thing. Instead of frivolously running simulations when the user may still be building the circuit, it is better practice to wait until the user declares that they are done, before moving on with simulations and results.

Once the solve mode is initiated, PyGridSim is able to output various results on the current parameters of the overall circuit. Users specify a list of `queries` that they want to see the results of. Note that the keys of these queries are not case sensitive or space-sensitive (so `Total Power` and `totalpower` are both valid and will return the same result). This nature of being able to assess multiple queries at once is an improvement done by PyGridSim that other distributed system interfaces do not have—users of OpenDSS and AltDSS are expected to individually query each result they want to see on the solved circuit. Once the circuit is simulated and queried, PyGridSim will output results as a dictionary, with each inputted query as a key and their corresponding result. Currently, PyGridSim supports the following result queries:

- **Voltages:** The result for the key `Voltages` is a dictionary mapping each node in the circuit to their voltage at steady-state. This includes each load, source, and generator (does not include PVsystem, as those are attached to their corresponding nodes). The nodes here use their default name (i.e. `load0, load1`, etc.), even if an alternate name is assigned to them.

- **Losses:** The loss of an electrical circuit refers to the energy dissipated by the circuit, which leads to a reduction in the remaining usable power of the circuit. PyGridSim supports the querying of two different losses in the system: Real/Active Loss and Reactive Loss. Real Loss refers to the energy dissipated due to resistance in lines and transformers, whereas Reactive Loss refers to the energy lost through other means, like through magnetic fields.

  If the user were to simply query `Loss` or `Losses`, they would get a dictionary displaying the Active Loss and the Reactive Loss. Note that this differs from the behavior of other distributed system simulators—loss results are typically given as a vector, with real corresponding to real loss and imaginary corresponding to reactive loss. While compact, this can be unintuitive and hard to decode for a user trying to understand the circuit, so PyGridSim gives the string understanding of each type of loss. Additionally, if the user wanted to just query for real/active loss, and of the queries `RealLoss`, `ActiveLoss`, `RealPowerLoss`, and `ActivePowerLoss` would all equivalently output just the real loss component of Losses. Similarly, users can query `ReactiveLoss` or `ReactivePowerLoss` to get the reactive loss only.

- **Total Power:** PyGridSim also supports the querying of two types of power in the system: real/active power and reactive power. Real power refers to the total power supplied by the circuit in order to overcome the real power demands of the loads plus the real power losses. On the other hand, reactive power refers to the power supplied to by the circuit to overcome the reactive power demands of the loads plus the circuit's reactive power losses. Similarly, the user can either query `Power` or `TotalPower` to get the dictionary with both the Real and Reactive Power. However, the user can also just query `ActivePower` or `RealPower` to only fetch the Active Power component of the total power, or alternatively just seek the `ReactivePower` component of the total power.

Note, however, that the PyGridSim codebase is designed to be modified and iterated on. A future version of PyGridSim would plan on supporting fault analysis and further results after solve mode is initiated. The following code demonstrates how the user may enter solve mode and query for results, assuming the circuit was already built:

```python
1 circuit.solve()
2
3 print(circuit.results(["Voltages", "Losses", "TotalPower"]))
4 # Result: A dictionary with "Voltages", "Losses", "TotalPower"
5 # and their corresponding results will be printed to their terminal.
6
7 circuit.results(["Voltages","Losses"], export_path="simulation.json")
8 # Result: A JSON file containing the "voltage" and "losses" keys
9 # with their corresponding results will be created at the path simulation.json.
```

Figure 3.13: **Querying for Results:** Assuming circuit components and lines have already been built prior to line 1, the user solves the circuit and proceeds to query for the voltages and losses of the resulting system. They then choose to save these results under the path "simulation.json".

3.7 Error Handling

OpenDSS and AltDSS provide some preliminary error handling, but PyGridSim aims to build on this error handling. In order to make the package easy to use, PyGridSim aims to flag most typing errors in compile time. However, there are many errors that must be raised during the runtime of the program. Some of the main error handling cases in PyGridSim are as follows:

- **Invalid Default Type:** Users should seek a list of supported default types through the type query function specified above in section 3.5.3. If they misspell or otherwise attempt to create an object with an invalid default type, a `KeyError` will be thrown.

- **Invalid Parameter:** While PyGridSim allows for parameter customization on its nodes, it does not have the complete suite of parameters that OpenDSS does. As a result, the package throws a `KeyError` when the user inputs a parameter that is not supported in the `params` field.

- **Negative Voltages:** PyGridSim throws errors upon creation of any circuit component with a specified negative kV in `params`, as it is not a physically possible model for the circuit. A `ValueError` is raised in this case.

- **Invalid Result Queries:** When querying results, users can input a set of queries. If the query does not match a suported query, that query will return "invalid". Note that unlike the other error cases, this doesn't explicitly run an error–this is because the error doesn't compromise the build of the circuit nor any other valid queries.

- **Miscellany:** PyGridSim also supports other miscellaneous errors, such as creating a line between two nodes with negative length.

The primary goal of this added error handling is to help users distinguish a circuit fault/error (something useful for fault analysis) compared to a user input error (error because the module doesn't support the desired input). Note that through importing AltDSS and OpenDSS, PyGridSim also throws errors when any of the AltDSS/OpenDSS errors are thrown.

3.8 Testing and Continuous Integration

PyGridSim used `unittest` for its unit testing of the class, and integrated these tests into a continuous integration pipeline on GitHub Actions. The test integration on Github runs these tests for each supported version of python (3.9, 3.10, 3.11, and 3.12), to ensure no extraneous errors across python versions.

The test suite of PyGridSim is broken up into three sections `Default Circuit`, `Customized Circuit` and `Type Querying` as follows:

- The `Default Circuit` test suite contains various test cases to exhaustively create each kind of "default parameter set" allowed by the PyGridSim interface. It verifies that each of these circuits build as intended, and that invalid default types will raise an error.

- The `Customized Circuit` test suite verifies the creation of a circuit with user-customized parameters. It exhaustively checks that the circuit can be built with user-customized parameters for each of the supported circuit components and their corresponding parameter keys. Further, this section tests the error handling on many of the possible errors the user could make—that entering a negative voltage, adding an invalid parameter query, etc. will lead to a thrown error.

- The `Type Querying` exhaustively verifies the behavior of the type query function detailed in 3.5.3, for each possible input of `component` and `show_ranges`.

Critically, the test suite of PyGridSim has a test coverage of over 99% over the entire codebase. This high test coverage ensures that almost all functions and methods within the codebase are validated, minimizing the risk of undetected bugs and errors. This provides confidence in the robustness and reliability of the package across all the supported user scenarios.

Chapter 4

Building and Testing Circuits with PyGridSim

After having a good understanding of how individual circuit components are built and designed through PyGridSim, users will want to create distributed system circuits of their choosing. This section will build a walkthrough of circuit creation through iterated examples, building from the simplest circuit to a larger one. It will highlight the creation of these circuits through both the custom and default methods described above, and serve as a guide for users wanting to build their own circuits. Finally, the section will analyze some results from the example circuits—discussing how tweaking parameters impacts the circuit end-state, and comparing different configurations.

Throughout this section, the examples are focused at featuring a breadth of the capabilities of the PyGridSim system. The examples, of course, are not entirely comprehensive, but aim to give users options on where to start based on the circuit they are trying to create.

4.1 One Source, One Load

To start with, we will outline ways a user can code the simplest circuit possible—one source, one load, and a line connecting them—in PyGridSim. This represents the most basic functioning circuit you can generate, and serves as a starting point for how to develop PyGridSim circuits. First, we will customize the parameters of the circuit, and demonstrate how this circuit functions with and without a transformer, comparing the results. Then, we will see how the circuit performs under various trials of random initiation by default parameter.

4.1.1 Customizing Simple Circuit

These examples will build a single source, single load circuit with the same parameters, with and without a transformer. To start, we will build a load with `0.12 kV`, `1 kW` of power, and `1 kvar` of reactive powe, akin to a small house. Then, we will build a source with `0.5 kV`, akin to a very small substation. We will build a line between these two of length `1 km`. Consider the use case where a user wants to query the voltages and the losses of this system.

To begin with, we want to build the following transformer-less single-source single-load circuit using PyGridSim. Note that these drawn examples will demonstrate each voltage producer in blue, and voltage consumer in red.

source
load0

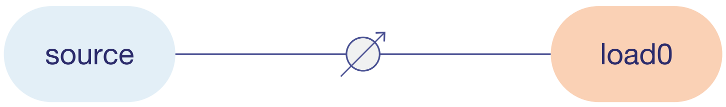
Figure 4.1: Diagram: Single Source, Single Load

To build the above circuit, we first want to initialize a circuit by initializing a PyGridSim object. Then, note that we do not yet need to take advantage of PyGridSim's batching functionality. Rather, we just use the functions `add_load_nodes`, `update_source`, and `add_lines` to add one of each. Note that we also do not need to specify `num` as that defaults to 1. For each function call, we specify the above parameters in the `param` fields. Then, after solving the circuit, we will query for `Voltages` and `Losses` because of the defined use case. Note that the latter should return a dictionary with both real power loss and reactive power loss. Finally, users should clear the circuit when they are done if they plan to continue building new circuits.

```python
# Initialize Circuit
circuit = PyGridSim()

# Add Custom Source and Load
circuit.add_load_nodes(params={"kV": 0.12, "kW": 1, "kvar": 1})
circuit.update_source(params={"kV": 0.5})

# Add Line without Transformer
circuit.add_lines([("source", "load0")], params={"length": 1}, transformer=False)

# Solve and Print Results
circuit.solve()
print(circuit.results(["Voltages", "Losses"]))
circuit.clear()
```

Figure 4.2: Single Source, Single Load, No Transformer

Next, we want to build the same circuit with a transformer. Note that the drawn example circuit contains the typical transformer symbol for distributed systems, a circuit with an arrow.



In order to modify the non-transformer code to transformer code, the only change is in the `add_lines` function call. Specifically, we no longer want to define `transformer = False`, and the circuit will automatically place the transformer. Note that the interface was built for this use case: because transformers are critical to efficient power systems, we default to using them instead of excluding them. The full code, with only line 4 changed, is as follows:

```python
# Initialize Circuit
circuit = PyGridSim()

# Add Customized Source and Load
circuit.add_load_nodes(params={"kV": 0.12, "kW": 1, "kvar": 1})
circuit.update_source(params={"kV": 0.5})

# Add Line With Transformer (default)
circuit.add_lines([("source", "load0")], params={"length": 1, "kVA": 0.1})

# Solve and Print Results
circuit.solve()
print(circuit.results(["Voltages", "Losses"]))
circuit.clear()
```

Figure 4.3: Code: Single Source, Single Load, Transformer

Now, after having built these two types of single-source, single-load circuits, we can compare their results (respective voltages of nodes, and losses). Note that we used identical, non-randomized parameters for both circuits, so all differences can be attributed to the existence of the transformer. Further, in the above code, note that we specified `kVA = 0.1` for the transformer parameter. This parameter represents an estimate: kVA refers to the apparent power of the electrical system, which can only be exactly calculated after the circuit is solved. It is useful as a transformer parameter as an estimate for the initializing size of the transformer—and the kVA is usually on the scale of the kV of nodes[15]. Creating too big of a transformer can lead to excessive losses, whereas a very small transformer minimizes the role of the transformer. The following table compares losses for no transformer and transformers of various sizes (as determined by the kVA parameters set). If the user does not choose to set a kVA, PyGridSim will provide an estimate based on the kV values of the source and destination nodes. In the example above, this estimate is `0.18`.

Note that without the transformer, the source and the load result in nearly identical end kV values. This means that after the circuit is solved there is only a negligible difference in the kV values of the source and load node, despite the fact that they started with wildly different kV values. This model is not quite realistic—we don't necessarily expect or want

Table 4.1: Transformers vs. No Transformer

Result	None	kVA = 0.001	kVA = 0.1	kVA (default, 1.8)	kVA = 10
Source (Volts)	500	500	500	500	500
Load1 (Volts)	494	493.7	478	467	187
Active Loss (kW)	0.13	0.24	10	18	349
Reactive Loss (kW)	0.28	0.76	46	80	816

the load node to consume the kV of the source node. This model certainly leads to less power losses: the voltage is able to be split approximately evenly between the two nodes therefore minimizing chance for loss. However, it may not be realistic or ideal to split the voltage evenly.

We can observe that the transformers observe much more loss, with bigger transformers corresponding to higher power loss yet higher voltage differences. The transformer with the lowest `kVA = 0.001` represents the smallest transformer, and gets results most similar to not placing a transformer at all. On the other hand, the higher `kVA = 10` overestimates the amount of transformer power needed, and thus contributes more losses with a higher difference between source and load values. This is certainly a tradeoff—voltage losses cause inefficiency in the system, but being able to separate the load and source node tends to reduce risks of faults, and may be a more realistic model overall. Typically, connections between source and load nodes will have some transformer—the size depending on the efficiency needs of the distribution system.

Of course, it is important to note that the values observed in this example are in part because of the context presented—we are building a small substation to support only a single house. This explains the unrealistically large losses in the setup—by providing far more voltage production than needed consumption. In a more realistic example, a single substation would be able to power much more than a single house.

4.1.2 Using Various Default Parameters for Simple Circuit

Within a single-source, single-load distributed system, the user can also forego specifying parameters entirely. This will still lead to the losses problem detailed in the previous section (simulating one large voltage source powering a single house will inevitably lead to voltage imbalances and large losses), but is still a valid form of circuit creation using PyGridSim. Note that this will come with some randomness—as described in section 3.5, the default parameters are defined by ranges and not a fixed value. So, different runs with the same code will lead to similar but not identical results.

The following code walks through how a user would generate this single-source, single-load system using default parameters. This example sets the load to be a typical house, the source being a typical wind turbine, and the connection between them to be a low voltage line.

```python
# Initialize Circuit
circuit = PyGridSim()

# Add Source and Load
circuit.add_load_nodes(load_type="house")
circuit.update_source(source_type="turbine")

# Connect Source and Load
circuit.add_lines([("source", "load0")], line_type="lv")

# Solve and Print Results
circuit.solve()
print(circuit.results(["Voltages", "Losses"]))
circuit.clear()
```

Figure 4.4: Code: Single Source, Single Load, Defaults

Note that the creation of the circuit remains approximately the same: the user still initializes, builds components, adds lines, then queries for results. The only difference from section 4.1.1 is that instead of feeding in a dictionary of `params` upon initializing load, source, and lines, the code just uses default parameter sets for each one. A user could even choose to not specify which parameter set to use (while also not passing in a dictionary of parameters)—in this case, a chosen default will be usen. As discussed in section 3.4, PyGridSim is configured to default to a house load, turbine source, and low voltage line. Thus, a user could have written the following code, simpler but underspecified, to achieve the same behavior.

```
# Initialize Circuit
circuit = PyGridSim()

# Add Source and Load
circuit.add_load_nodes()
circuit.update_source()

# Connect Source and Load
circuit.add_lines([("source", "load0")])

# Solve and Print Results
circuit.solve()
print(circuit.results(["Voltages", "Losses"]))
circuit.clear()
```

Figure 4.5: Code: Single Source, Single Load, Unspecified Defaults

Now, a user may want to experiment with the different types provided. For instance, while still modeling a single-source single-load circuit, they may want to alter which default load type to use. This experiment will keep the source type constant so all changes can be attributed to load type changes and some randomness. PyGridSim supports the types *house*, *commercial*, and *industrial* as described in section 3.5, and the following table represents the results of simulating a single-source, single-load system with each possible load type, over 100 trials:

Table 4.2: Results: Single-Source Single-Load with Default Load Types

Result	House	Commercial	Industrial
Source (Volts)	2175	2137	2128
Load1 (Volts)	721	823	1940
Active Loss (kW)	192	163	40
Reactive Loss (kW)	478	379	153

Note that the above simulations were run 100 times—each creation of a load and source node will generate a new set of parameters, uniformly and at random, from the realistic range for their respective load and source types. Thus, an average is taken to show the average end result over several runs.

As we can see, the typical consumption for each load type are bigger than the example we generated in section 4.1.1, as is the typical production of the wind turbine source type. These examples thus operate on a larger scale than the simulation we ran before. Note that the industrial load type is the closest in voltage level to the wind turbine, resulting in the lowest active and reactive losses as power flows between the two components. A user of PyGridSim could similarly run simulations on varying source types, varying line, length, etc.

4.2 One Source, Many Loads

The next sample scenario is a circuit with a singular central voltage source and many loads. This is one step towards modeling a more realistic circuit, where a large source likely has the power production to power many consumer nodes. In this example, we will seek to represent many "house" consumer nodes being powered by a center source node. We assume a transformer will exist in all of these connections. In this case, the user will seek to query the `Losses` as well as the `Power` of the system after running its simulation. As a starting case, assume a turbine as the central voltage source, with 8 house consumer nodes surrounding it. The desired setup for this distributed system is shown below:

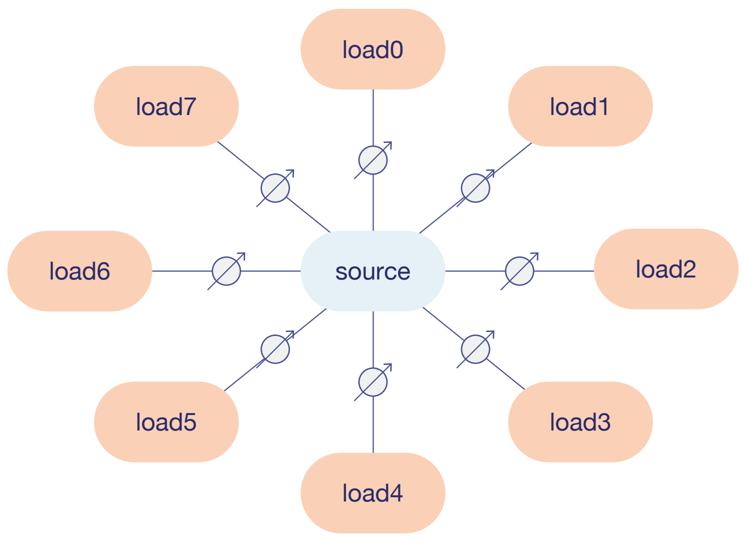


Figure 4.6: Diagram: One Source, Many Load

The PyGridSim code to generate such a circuit with `num = 8` is as follows:

```python
# Initialize Circuit
circuit = PyGridSim()

# Add Source and Loads
circuit.add_load_nodes(num=8, load_type="house")
circuit.update_source(source_type="turbine")

# Add Lines from Source to each Load
conn_list = []
for i in range(8):
    conn_list.append(("source", "load" + str(i)))
circuit.add_lines(conn_list, line_type="lv")

# Solve and Print Results
circuit.solve()
print(circuit.results(["Losses", "Power"]))
circuit.clear()
```

Figure 4.7: Code: Single Source, Many Load, Turbine

If a user were to run this same code for a few different values of `num`, the could observe the following trends in Losses and Power (taken over 100 trials of each choice of `num`).

First, note that the real and reactive power represents the power from the source, and is negative because the power is flowing from the source. Observe that as the circuit expands

Table 4.3: Single-Source With Different Num Loads

Result	num=2	num=4	num=8	num=20	num=100
Active Loss (kW)	21149	42798	85055	212584	1011337
Reactive Loss (kW)	52863	106270	211211	528208	2515161
Active Power (kW)	-22770	-45919	-91375	-228275	-1086621
Reactive Power (kW)	-332	-1153	-1562	-2596	-17868

from a single load to multiple loads, the total power demand increases, since each additional load contributes to the cumulative consumption. To meet this increased demand, the source must supply more power, which is reflected as a more negative real and reactive power value—indicating greater output from the source. However, the addition of new loads and lines leads to more opportunities for overall real and reactive loss in the system. This trend—higher losses and increasingly negative source power—is a natural result of scaling up a distributed circuit from a centralized voltage source. We can also observe that the power output doesn't quite increase linearly–doubling the number of loads does not exactly lead to twice as much power or twice as much loss. Researchers can use these tools and trends to further understand the behavior of such distributed systems. Thus, when picking how many nodes a source can supply, the user should think about whether they value a certain amount of power being generated, care about losses being minimized, etc.

This simulation serves as just one example setup that the user can use to explore power and losses of their circuit. We could also have expanded this experiment for any of the other source types provided, by replacing the `source_type = turbine` to any of the other source types. For each of the other source types described in section 3.5, we could run a similar experiment to determine how they perform with varying number of load nodes. Note that users could also run a similar experiment with this single-source multi-load setup with customization nodes instead of using the prepackaged "house", "turbine", etc. They could similarly discover, based on their input parameters, the appropriate number of houses their single-source can effectively power. Users who seek to run similar experiments could use this code as a starting template, and further enhance and customize their code to fit their research questions.

4.3 Various Sources and Loads

Next, this section will demonstrate the building of a circuit with different types of circuit components, in order to help demonstrate the full scope of PyGridSim's system simulation capabilities. Instead of just having a single voltage source, this system will also have generators that can also help power the system. Further, some of the load nodes will serve both as producers and consumers, modeling houses that generate energy through the addition of solar panels. A simple setup like this would most closely resemble a microgrid in the real world, that can serve independently or as part of a larger distributed system network. Consider the following setup, modeling a system featuring all of those circuit components. As before, electrical consumers are indicated in red whereas electrical producers are indicated in blue. Loads with a PV system are indicated in purple, as they both produce and consume electricity.

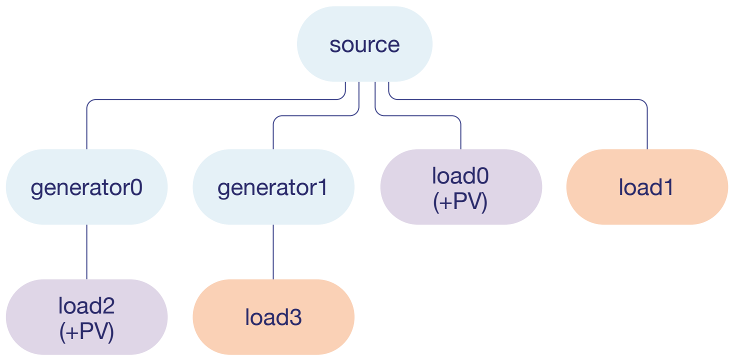


Figure 4.8: Diagram: Many Source, Many Load

Now, assume that a user of PyGridSim seeks to create this circuit setup by customizing some realistic parameters (similar to section 4.1) for each node. For the solar panels they seek to attach to two of the load nodes, assume they are using 10 panels. Assume they want to view the final voltages of each component as well as the real loss accumulated by the system, and export these final results to a file titled `sim_results.json`. They could then build, simulate, and save this circuit with the following code:

```python
# Initialize Circuit
circuit = PyGridSim()

# Add Components
circuit.add_load_nodes(num=4, params={"kV": 0.12, "kW": 1, "kvar": 1})
circuit.update_source(params={"kV": 0.5})
circuit.add_generators(num=2, params={"kV": 1})
circuit.add_PVSystems(["load0", "load2"], num_panels=10)

# Add Lines
src_lines = [("source", "generator0"), ("source", "generator1"),
             ("source", "load0"), ("source", "load1")]
circuit.add_lines(src_lines, params={"length": 0.2})
circuit.add_lines([("generator0", "load2"), ("generator1", "load3")], params={"length": 0.1})

# Solve and Save Results
circuit.solve()
circuit.results(["Voltages", "RealLoss], export_path="sim_results.json")
circuit.clear()
```

Figure 4.9: Code: Many Source, Many Load

Upon running this code, they would be able to see the following content in a created file titled `sim_results.json`. Note that the voltages are expressed in Volts, and the Loss is in kW.

```json
{
    "Voltages": {
        "source": 500,
        "load0": 493,
        "load1": 492,
        "load2": 489,
        "load3": 489,
        "generator0": 493,
        "generator1": 493
    },
    "RealLoss": 85.8
}
```

Figure 4.10: Results: sim_results.json

This simulation indicates relatively good voltage regulation across the system, with low voltage drops between the source nodes and the components they are connected to. As a result, there is relatively low power loss across the system.

As with sections 4.1 and 4.2, this experiment could have been adapted in many ways to query other types of results—the user could have instead chosen to pick generator and load types instead of defining specific parameters, toggled with different number of solar panels for PVSystems, etc. All of these demonstrate ways in which PyGridSim can be used to research the vried behavior of distributed system circuits.

4.4 IEEE13 Radial Circuit

This section presents a circuit based on the standard IEEE 13-node example. In addition to being larger than the circuits previously discussed, it illustrates how users can leverage PyGridSim's naming feature, which was explined in section 3.4.1. For users working with diagrams that already assign specific names or numbers to nodes, this tutorial serves as a practical guide for translating an existing labeled circuit into a PyGridSim circuit.

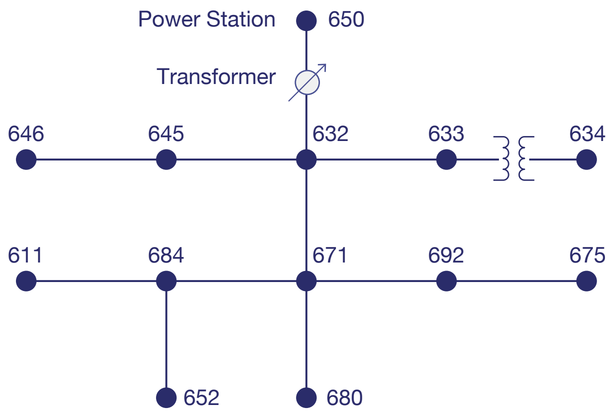


Figure 4.11: IEEE13 Radial Circuit[16]

In order to model the above IEEE13 Radial Circuit on PyGridSim, the user could write the code as follows. Assume the loads are typical houses, the power station is a low voltage substation, and the user wants to query for losses and total power. The user initially assigns the loads the names that were provided, so that line creation is more intuitive and matches up with the drawing. They could have also chosen to just use the default load names, but would have had to make a note of the mapping between on-diagram names and the package's internal names.

```python
# Initialize Circuit
circuit = PyGridSim()

# Add Components and Names
circuit.update_source(source_type="lvsub")
# Add the 12 Load Nodes (houses)
load_names = ["646", "645", "632", "633", "634"]
load_names += ["611", "684", "671", "692", "675", "652", "680"]
circuit.add_load_nodes(num=12, load_type="house", load_names=load_names)

# Add Two Lines with Marked Transformers
circuit.add_lines([("source", "632"), ("633", "634")], line_type="hv")
# Represent Longer lines between (632, 645), (632, 633)
row_lines = [("646", "645"), ("645", "632"), ("632", "633")]
row_lines += [("611", "684"), ("684", "671"), ("671", "692"), ("692", "675")]
col_lines = [("632", "671"), ("671", "680"), ("684", "652")]
circuit.add_lines(row_lines + col_lines, line_type="mv", transformer=False)

# Solve and Query Results
circuit.solve()
circuit.results(["Losses", "TotalPower"])
circuit.clear()
```

Figure 4.12: Code: IEEE13 Radial Circuit

Note that this example replicates the shape of the IEEE13 Radial Circuit, but not the parameters, as those were not provided in the diagram. This example aims to display how to use the randomized default-based initiation to generate a circuit; the user could alternatively use PyGridSim to match IEEE13 standards precisely by porting over every customization parameter. After solving the PyGridSim circuit created above, the user will observe the following results for voltages and losses.

Table 4.4: Results: Single-Source Single-Load with Default Load Types

Active Loss (kW)	1.55
Reactive Loss (kW)	3.20
Active Power (kW)	-4.02
Reactive Power (kW)	-4.45

We observe that under this radial design, we observe power flowing out of the source node, as expected, with relatively low active loss yet substantial reactive loss. This suggests that this setup is inefficient in reactive power handling.

Overall, because of PyGridSim's scalable design, users can expand these experiments to build even larger circuits, with as many nodes as they would like. While they would need to specify more lines manually, scaling component creation only involves updating the `num` parameter. This reflects a core design goal of PyGridSim: enabling the modeling of even complex circuits with fewer lines of code compared to alternatives like AltDSS or OpenDSS.

Chapter 5

Discussion

5.1 Why PyGridSim?

Overall, the space of python-accessible distributed system simulators is growing, and it is worth considering where PyGridSim fits into this space. PyGridSim balances the customizability found in most DSS interfaces with the convenience of sensible defaults, and requires fewer function calls to construct equivalent circuits compared to other simulators. Its interface is consistent and user-friendly, and it offers string-based parameter descriptors to assist users who may be less familiar with reasonable parameter assignments, making circuit creation both accessible and efficient.

As discussed throughout the thesis, the design choices of PyGridSim were based primarily on the principles of *efficiency* and *intuitiveness*. This does come with certain tradeoffs—PyGridSim supports less circuit components than the comprehensive OpenDSS, and has more limited post-processing results in terms of analysis and visualization. Still, PyGridSim still provides a valuable improvement to the space of distributed system simulators, making circuit creation easier and faster for many end-user use cases. Notably, the codebase of PyGridSim is designed to be extensible—supporting continued development and improvements for better distributed system simulation and analysis.

5.2 Future Directions

It is worth outlining some specific changes that could be made to the PyGridSim interface that can further help PyGridSim meet its primary design principles. As stated in section 5.1, PyGridSim is dynamically built to be easily adapted to accommodate new goals and extensions. Some possible future directions of the package are as follows.

- **Improvements on Randomization:** A critical part of the PyGridSim design setup is that the package allows for randomness when users want to create identical nodes. This means that a user can represent having 10 similar houses without making them exactly identical or otherwise manually specifying small differences—making for more realistic modeling of a residential system. However, the PyGridSim interface could also be improved if the user could additionally request randomness even when they customize parameters. For instance, if a user could instead state that they want to make 10 load nodes each with `1 < kV < 2`. Then, the interface would randomly make 10 nodes each following the given constraint. This feature was not included in the initial release of PyGridSim, as it could clutter and complicate the `params` field. However, it could be a useful tool for users in the future.

  Also, randomness in PyGridSim is done through the uniform distribution. While this is a generally reasonable distribution to set for the parameters that it is used for, it could be nice to model different types of randomness distributions. Perhaps some components' distribution falls closer to a normal distribution, and PyGridSim could improve accuracy of simulations by modeling a normal distribution. Further, a user may want to specify their own distribution, by providing a lambda function $\lambda \to [0, 1]$ modeling the likelihood of the parameter falling in each end of the range. An improved version of PyGridSim could support new or user-specified randomness functions for parameter specification.

- **Support for Fault Injection:** There are ways in which PyGridSim can be more catered to conduct certain types of fault studies. For instance, a user may want to simulate a fault at a certain point in their system ("injecting" a fault), and see how that fault spreads throughout the system. This can then help users assess if a point-wise fault will lead to a system-wide fault, or if the system setup is otherwise resilient to individual component faults. This can be incredibly important for analyzing the overall safety of a distributed system setup.

- **Results Visualization:** Next, PyGridSim could improve on the results of the system by providing results visualization. One practical goal would be to generate and save an image of the circuit once the `circuit.solve()` method is called. This visualization could display the circuit components, the electrical lines connecting them, and the final parameter values assigned to each component. Such a feature would help users visually verify that the constructed circuit matches their intended design and that all parameters match their expectation. Implementing this would likely involve integrating PyGridSim with an external Python library specialized in visualization or diagram generation.

  A more ambitious extension would reverse this process—enabling PyGridSim to accept a well-specified circuit diagram as input and automatically generate the corresponding circuit-building commands. The package would then have to parse the data from an image into concrete parameters and circuit components, or perhaps leverage ML tools to learn how to process images of varying styles.

- **Real-Time Simulation:** Another extension on the results and post-analysis portion of the project would be to allow real-time simulation playback. This feature would enable users to observe step-by-step voltage changes throughout the simulation, rather than viewing only the final results. There are certain packages, like PyDSS discussed in section 2.2, with a focus on this type of post-analysis behavior. Adding this simulation playback into the PyGridSim package would further allow users to get a comprehensive understanding of their simulated circuit's behavior over time.

Overall, PyGridSim contributes to the space of distributed system simulators by providing an efficient, easy-to-use interface that streamlines circuit creation. However, there are still notable tradeoffs and ways that the package can be improved in the future to extend the set of user end goals that it meets. Fortunately, PyGridSim is designed with flexibility in mind, allowing for future enhancements and extensions without requiring a complete system overhaul.

Chapter 6

Conclusion

As discussed throughout this thesis, PyGridSim poses many advantages when it comes to certain use cases of PyGridSim. This thesis highlighted some of the major components and extensions that PyGridSim provides over other python-based OpenDSS packages:

- **Functional Interface:** PyGridSim provides a functional Python interface with consistent syntax for node creation.
- **Batch Creation:** Users are able to more easily create scalable circuits with PyGridSim by leveraging the batch creation ability of the interface.
- **Defaults and Customization:** Users can choose to either entirely customize their node, rely on default parameter sets, or settle somwhere in between.

Overall, PyGridSim expands on the space of distributed system simulators, making them more accessible and efficient to generate. Finally, note the work done in the PyGridSim interface is designed to be flexible and improved on, to further improve the accessibility of distributed system simulation and analysis.

References

[1] Electric Power Research Institute (EPRI). *OpenDSS: Open Distribution System Simulator*. Accessed: 2025-04-23. 2024. URL: https://opendss.epri.com/IntroductiontoOpenDSS.html.

[2] IEA. *Global EV Outlook 2024*. 2024. URL: https://www.iea.org/reports/global-ev-outlook-2024/executive-summary.

[3] K. Antonio. *Solar and wind to lead growth of U.S. power generation for the next two years*. Jan. 2024. URL: https://www.eia.gov/todayinenergy/detail.php?id=61242#:~:text=As%20a%20result%20of%20new,476%20billion%20kWh%20in%202025.

[4] *AltDSS-Python*. URL: https://dss-extensions.org/AltDSS-Python/index.html.

[5] *PyDSS*. URL: https://nrel.github.io/PyDSS/index.html.

[6] A. Hariri, A. Newaz, and M. Faruque. "An Open-Source Python-OpenDSS Interface for Hybrid Simulation of PV Impact Studies". In: *IET Generation, Transmission Distribution* 11 (Feb. 2017). DOI: 10.1049/iet-gtd.2016.1572.

[7] P. Meira. *AltDSS-Python*. https://github.com/dss-extensions/AltDSS-Python. Accessed: 2025-04-04. 2025.

[8] IEEE Power Systems Engineering Committee. *IEEE Recommended Practice for Electric Power Distribution for Industrial Plants*. Tech. rep. IEEE Std 141-1993. Revised in 1999. New York, NY: Institute of Electrical and Electronics Engineers (IEEE), 1993.

[9] A. Unknown. *Typical Residential Electrical Load Profiles*. Accessed: 2025-04-23. 2023. URL: https://www.nature.com/articles/s41467-022-31942-9.

[10] S. Topics. *Distribution Voltage Level*. Accessed: 2025-04-23. 2023. URL: https://www.sciencedirect.com/topics/engineering/distribution-voltage-level.

[11] Wind Energy – The Facts. *Electrical Works*. Accessed: 2025-04-23. 2023. URL: https://www.wind-energy-the-facts.org/electrical-works-7.html.

[12] D. Eusebio. *What Size Generator Do I Need?* Accessed: 2025-05-16. 2023. URL: https://www.bigrentz.com/how-to-guides/choose-best-generator-size-job.

[13] Just Energy. *How Big Are Solar Panels: How Panel Size Impacts Your Solar System*. Accessed: 2025-04-23. Oct. 2024. URL: https://justenergy.com/blog/how-big-are-solar-panels-how-panel-size-impacts-your-solar-system/.

[14] Cosmo Ferrites Ltd. *Importance of Power Transformers in the Modern World*. Accessed: 2025-05-15. 2025. URL: https://www.cosmoferrites.com/news-events/importance-of-power-transformers-in-the-modern-world.

[15] Daelim Transformer. *KVA Transformer*. Accessed: 2025-05-16. 2025. URL: https://www.daelimtransformer.com/kva-transformer.html.

[16] B. Hu, J. She, and R. Yokoyama. "Hierarchical Fault Diagnosis for Power Systems Based on Equivalent-Input-Disturbance Approach". In: *ResearchGate* (2013). Figure: IEEE 13-bus radial distribution feeder (IEEE-13 feeder). URL: https://www.researchgate.net/figure/EEE-13-bus-radial-distribution-feeder-IEEE-13-feeder-24_fig10_260541757.

[17] OpenAI. *ChatGPT (GPT-4) [Large language model]*. Used only for grammar correction and proofreading support in this thesis. 2024. URL: https://chat.openai.com.