AN OPEN-SOURCE OBJECT ORIENTED POWER SYSTEMS
ANALYSIS SOFTWARE SOLUTION IMPLEMENTED IN ISO C++

BY

MELVIN R. STEVENS

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Professor Peter Sauer
ABSTRACT

There is a need to have open-source tools for power systems analysis, particularly for academic and other research purposes. This thesis provides the academic and other research communities a beginning set of open-source power analysis tools written in C++. There are many types of power analysis problems, including but not limited to dynamic stability studies, optimal power flow (OPF), security constrained optimal power flow (SCOPF), and economic dispatch (ED). Among all the problems, there is at least one thing often in common. Every power analysis problem requires the power flow of the system, a snapshot of the steady-state of the voltages and angles of each bus of the power system to be studied. A power flow program is a result of this thesis. The power flow module is written in ISO C++14, officially known as ISO International Standard ISO/IEC 14882:2014(E) programming language C++. The power flow module is provided as an open-source project.
To my parents, wife, and children, for their love and support.
ACKNOWLEDGMENTS

There are so many people to thank who have helped and inspired me through this thesis process. It is difficult to acknowledge some and not all; but I must give these explicit acknowledgments. I would like to give a special thanks to my loving wife, Mary. It was her encouragement that brought me to Illinois, and introduced me to the fine institution of the University of Illinois. I would like to thank my parents who stressed that I could achieve any goal I desired if I worked hard at it.

Another special thanks goes to my advisor, Professor Peter Sauer, who has always been available to me whenever I needed his assistance. I am extremely appreciative of his help; words cannot express my full gratitude for his willingness to help me.

I thank Professor Thomas Overbye, the creator of PowerWorld™, for taking the time to answer my questions regarding power flow. I would like to thank Patrick Chapman, who encouraged me to pursue this master’s degree.

Finally, I would like to thank all those in the ECE Department at the University of Illinois at Urbana-Champaign who have given me any assistance. There are too many to name them all but some include: Joyce Mast, Robin Smith, Professor Steve Franke, Jennifer Merry Carlson, Laurie Fisher, Jan Progen and James Hutchinson. I appreciate their patience in working with me to achieve my degree.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER 1</td>
<td>MOTIVATION</td>
</tr>
<tr>
<td>CHAPTER 2</td>
<td>GOALS</td>
</tr>
<tr>
<td>2.1</td>
<td>Project Goal – Pedagogy</td>
</tr>
<tr>
<td>2.2</td>
<td>Project Goal – Open-source</td>
</tr>
<tr>
<td>2.3</td>
<td>Project Goal – ISO C++</td>
</tr>
<tr>
<td>2.4</td>
<td>Design Goal – Platform Independence</td>
</tr>
<tr>
<td>2.5</td>
<td>Design Goal – Clarity</td>
</tr>
<tr>
<td>2.6</td>
<td>Design Goal – Modularity</td>
</tr>
<tr>
<td>2.7</td>
<td>Design Goal – Extensibility</td>
</tr>
<tr>
<td>2.8</td>
<td>Design Goal – Future Computing Technologies</td>
</tr>
<tr>
<td>CHAPTER 3</td>
<td>LITERATURE SEARCH</td>
</tr>
<tr>
<td>CHAPTER 4</td>
<td>SOFTWARE SEARCH</td>
</tr>
<tr>
<td>CHAPTER 5</td>
<td>SYSTEM FORMAT</td>
</tr>
<tr>
<td>CHAPTER 6</td>
<td>TOOLS</td>
</tr>
<tr>
<td>CHAPTER 7</td>
<td>USE OF LIBRARIES</td>
</tr>
<tr>
<td>CHAPTER 8</td>
<td>OBJECTS</td>
</tr>
<tr>
<td>CHAPTER 9</td>
<td>CLASSES</td>
</tr>
<tr>
<td>9.1</td>
<td>Grid Class</td>
</tr>
<tr>
<td>9.2</td>
<td>Branch Struct</td>
</tr>
<tr>
<td>9.3</td>
<td>Node Struct</td>
</tr>
<tr>
<td>9.4</td>
<td>Bus Struct</td>
</tr>
<tr>
<td>9.5</td>
<td>Generator Class</td>
</tr>
<tr>
<td>9.6</td>
<td>PFSolver Class</td>
</tr>
<tr>
<td>CHAPTER 10</td>
<td>NAMING SCHEME</td>
</tr>
<tr>
<td>10.1</td>
<td>Variables</td>
</tr>
<tr>
<td>10.2</td>
<td>Functions</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>11</td>
<td>POWER FLOW ALGORITHM</td>
</tr>
<tr>
<td>12</td>
<td>CHALLENGES</td>
</tr>
<tr>
<td></td>
<td>12.1 Software Design</td>
</tr>
<tr>
<td></td>
<td>12.2 Checking Generator Reactive Power Limits</td>
</tr>
<tr>
<td>13</td>
<td>USE</td>
</tr>
<tr>
<td>14</td>
<td>RESULTS</td>
</tr>
<tr>
<td>15</td>
<td>FUTURE WORK</td>
</tr>
<tr>
<td>16</td>
<td>CONCLUSION</td>
</tr>
<tr>
<td>A</td>
<td>THE POWER FLOW PROBLEM</td>
</tr>
<tr>
<td>B</td>
<td>THE SOLUTION</td>
</tr>
<tr>
<td>C</td>
<td>CDF FORMAT</td>
</tr>
<tr>
<td>D</td>
<td>CONSULT WITH THE FATHER OF C++</td>
</tr>
<tr>
<td>E</td>
<td>HIDDEN GEMS</td>
</tr>
<tr>
<td>F</td>
<td>THE CODE</td>
</tr>
<tr>
<td>References</td>
<td></td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Use of Microsoft DLL for the Power Flow Module</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Modularity of the Code. Red Denotes Steady-State Power Flow</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>C++ Inheritance</td>
<td>8</td>
</tr>
<tr>
<td>5.1</td>
<td>Graphical Format of a 5-Bus Power System [1]</td>
<td>11</td>
</tr>
<tr>
<td>5.2</td>
<td>CDF Format of a 5-Bus Power System [1]</td>
<td>13</td>
</tr>
<tr>
<td>9.1</td>
<td>Grid Class</td>
<td>19</td>
</tr>
<tr>
<td>9.2</td>
<td>Branch Struct</td>
<td>20</td>
</tr>
<tr>
<td>9.3</td>
<td>Node Struct</td>
<td>21</td>
</tr>
<tr>
<td>9.4</td>
<td>Bus Struct</td>
<td>21</td>
</tr>
<tr>
<td>9.5</td>
<td>Generator Struct</td>
<td>22</td>
</tr>
<tr>
<td>11.1</td>
<td>Flow Chart of the Power Flow Algorithm</td>
<td>27</td>
</tr>
<tr>
<td>A.1</td>
<td>3-Bus Simple System</td>
<td>37</td>
</tr>
<tr>
<td>A.2</td>
<td>GSO 5-Bus Case</td>
<td>41</td>
</tr>
<tr>
<td>A.3</td>
<td>Simplification of the GSO 5-Bus Case</td>
<td>42</td>
</tr>
</tbody>
</table>
A search for an open-source power systems simulation package written in C++, with provided source code, proved to be futile. That search found zero packages that were truly open-source, with all source code provided and written in the C++ programming language. Perusing the internet for such a package yielded several sites where there were queries for such a package but no answers were given to those queries. This indicated that there has been interest.

A goal of this thesis was to do some dynamic stability studies using some open-source software written in C++ that could be extended, but no such software could be found. Further, at the Trustworthy Cyber Infrastructure for the Power Grid (TCIPG) Summer School, Summer 2015, it was mentioned that there is a need for more open-source power systems tools.

There are many problems to be solved in power systems. Among these are economic problems, security problems, optimizing problems and many others. Common to most of those problems, however, is the power flow, the steady-state snapshot of the system. The power flow of a system must be solved prior to working on most power-related problems of that system; in order to solve many types of power analysis problems, a perturbation of the power flow is often required. Thus, writing a power flow solver is a good first step in building a real open-source power systems simulation package.

While there are open-source power systems software packages available now, they do not seem to be extensible by the academic student; they do not provide source code, and almost none of them are written in ISO C++. Most open-source power analysis tools are written using Matlab; thus, those tools require the users to have a licensed copy of Matlab installed on their machine. This work is done with the student and academic researcher being the targeted end users, creating an alternative to using Matlab for such studies.
Finally, a discussion with Professor Thomas Overbye revealed that this material is not documented. He explained that although several have coded power flow algorithms, no one has taken the initiative and time to document their efforts. The major goal and motivation of this work is to document the details of building a power flow algorithm so than anyone wishing to do so would have that ability after reading this thesis.
A very important goal of this project was to produce a body of work which anyone, wanting to learn power flow, could read and learn the basics of power flow computation without referring to any document other than this thesis and perhaps one or two of its references. The initial goal of this thesis was ambitious: to provide an open-source power systems analysis software package that was extensible. Writing such software in its entirety is what made this goal so ambitious. Writing the power flow module of that larger package thus became the final overall goal of this thesis; the module is the result of this thesis work and is named **FreePFlow**. Sub-goals of this project were to make the final code open-source, to use the C++ programming language, to make the code platform-independent, to make the code clear and easy to understand, to make the code both modular and extensible, and finally to consider future computing technologies. The project goals are now briefly considered.

### 2.1 Project Goal – Pedagogy

There seems to be very little comprehensive documentation to coding a power flow algorithm. Anyone with the relevant basic knowledge, provided by reading this thesis, should have a thorough understanding of how to write a power flow algorithm and implement it in a programming language. Further, the code contained in this project is written in ISO C++ so it should also allow the reader the opportunity to learn that language as a result of working with and extending the code.
2.2 Project Goal – Open-source

The FreePFlow is given to the world; it is given so that others are free to experiment and learn from it. Open-source is defined as computer software along with its source code that is made available with a license in which the copyright holder gives the rights to study, change, and distribute the software [2], [3]. After a search for open-source power analysis software written in C++, with source code, yielded very few options, the need to fill this void was identified. This C++ code that was written for this thesis is to be released under an open-source license. Others are encouraged to use the code and extend it adhering only to the open-source license under which the code was released.

2.3 Project Goal – ISO C++

Many open-source power systems software packages are written in Matlab. Those packages require their user to have Matlab installed on the system on which they are executed; they are not standalone packages. Writing this power flow module eliminated that requirement. ISO C++ was chosen as the programming language because it is an ISO standard, it is an object-oriented language, it is ubiquitous, its generated executables are extremely small and fast, and code written in it is easily extensible.

Other packages found in a survey of open-source power systems software were written in various other languages including Java [4] and Python [5]. C++ has advantages over both of these languages [6], [7]. Additionally, the search for full source code for packages written in other languages was often unsuccessful.

2.4 Design Goal – Platform Independence

A design goal was to write code that could be compiled and run on a plethora of systems without any change to the code. Three computational programming languages were considered: C, C++, and Java. A comparative analysis of the languages is beyond the scope of this thesis; ISO C++ 14 was chosen
and the other two are only mentioned to acknowledge that careful consideration was given in choosing a language for this project.

Often, computers, on which students must work and write code, are *nix base systems. Today it seems that these are often systems using a Linux-based operating system. The ability to download the source to any computer with an ISO C++ compliant compiler, compile and run it was a major design goal. Deliberate care was taken not to include any platform specific C++ extensions to increase speed, and to decrease the size of the program or otherwise take advantage of any particular operating system features. For example, there was the temptation to compile this **FreePFlow** into a set of Microsoft Dynamic Linked Libraries (DLL) that could be used by or called from other executables. While there are advantages of using DLLs in the Microsoft Windows Operating System environment [8], they were not written and used in this project, which facilitates cross platform compatibility in the code. An illustration of the use of a DLL is displayed in Figure 2.1.

![Figure 2.1: Use of Microsoft DLL for the Power Flow Module](image)

2.5 Design Goal – Clarity

There are enough problems in power systems to solve; debugging hard-to-read or obfuscated code should not be added to that list of problems. The code is concise and clear. Maintainability of code is extremely important given that it is to be open-source. Variables, with meaningful names, are declared and defined where they are used.

2.6 Design Goal – Modularity

Another design goal was to make the overall program modular both from a design and a language perspective. The power flow module is just one
of many modules to be written. Writing the larger program in modules is analogous to the standard engineering black box paradigm. Figure 2.2 illustrates the modularity of the program. Each module of the program can be written, compiled and tested independently.

Figure 2.2: Modularity of the Code. Red Denotes Steady-State Power Flow.

Objects are created from classes and modules operate on those classes. Objects which are just instantiation of a class are passed from module to module. This modularity adds great flexibility to the program. Since the power flow data resides in the system object, it is possible for different individuals or entities to design different front ends or graphical user interfaces (GUI) to display the power flow data. It is even possible to re-write this power flow module, with the only requirement being that the interface remains the same. It is also possible to incorporate all the modules into one monolithic program; the possibilities are endless.

The code itself is also modular. Type and function declarations are in meaningfully named header (.h) files and function definitions are in meaningfully named C++ (.cpp) files. For example, there is a Grid class. The Jacobian class is declared in a grid.h file and is defined in a grid.cpp file. Those files are then included as needed in other files.
2.7 Design Goal – Extensibility

Technologies in electrical engineering change rapidly. Having software that can change as quickly without having to rewrite the entire code base would be a huge benefit. For example, a Node class is created with certain attributes also known in C++ as data members; it also has operations or member functions as they are called in C++.

In power systems, an electrical bus has more information than a basic node. But it also has all the things a basic node has. So the basic node class can be extended to be an electrical bus. Any algorithms that perform operations on a node would then be able to perform those same operations on a bus; e.g., if a function was written to optimally order a set of nodes, that same function would be able to optimally order a set of electrical busses without any change. Extensibility is a powerful feature of the C++ programming language. Figure 2.3 shows this power. Node is the base class for a Bus. A Bus inherits all the attributes and operations of a Node. So, each Bus has a _name, and _number, etc. It also inherits functions from the Node class; functions are often called operations. Each Bus name can be obtained by calling the name() operation without parameters or name(newname) with a string parameter of the newname of the bus.

2.8 Design Goal – Future Computing Technologies

Twenty years ago, many of the technologies in use today were likely not conceivable. It is almost certainly guaranteed that computing technology will continue to evolve. Trending today are Apps, which are used on phones or handheld devices. These devices, such as the Apple iPad Mini, are the most commonly used computing devices by consumers today. Applications are now beginning to move to the cloud. While it is true that in the engineering environment the desktop or workstation is still the most prevalent computing device, it is quite possible this could change. The code produced in this research is written considering that it may someday be ported to a device that fits on a ring on your finger. The code is written to be as generic and as portable as possible; care was taken not to compromise its speed.
Figure 2.3: C++ Inheritance
Searching the standard literary sources yielded the paper by Zhou [9] in which he explores the hypothesis that using object oriented programming (OOP) and the C++ language could result in an acceptable power flow program. “Acceptable” means that the resultant program would be small in size and comparable in speed to other solvers written in other languages. Today C++ has proven to be a language of choice when speed or system-level programming is necessary [10]. The debate over the viability C++ and OOP is over; OOP is now proven as the preferred method of programming for purposes. The literature search only entailed looking for projects in which open-source power systems software was written in C++ and had source code available.

There are many IEEE papers that discuss refinement of algorithms, primarily Newton-Raphson, to solve the power flow problem. Almost none of them describe implementation of the said algorithm in any specific programming language including C++; rather, they give the algorithm in flowchart format. Most papers that give concrete implementations seem to give the implementation as a Matlab program.

During the search, several books were perused, including but not limited to [11], [12], [13], [14]. As programming is just as much style as it is design, the references all had different approaches to the structuring of the software to be written. They had different opinions about what should be classes. While the reading was educational, no specific ideas were taken from any particular source.
CHAPTER 4
SOFTWARE SEARCH

This project is meant to be the beginning of a larger open-source power systems solution solver. The overall vision of the entire package is that it will implement the solvers for the power flow (this work, FreePFlow), optimal power flow (OPF), Security Constrained optimal power flow (SCOPF), dynamic stability and other power systems. A survey of comparable open-source software produced several options, the majority of which were listed on the Task Force on Open-source Software of Power Systems web page [15].

Many of them were written for Matlab; a goal of this work was to provide a standalone package. Complete source code and documentation did not seem to be provided for many of them, as if open-source meant “you can use the software for free.” None were found that were written in C++ and provided full source code for the solution.

While there are commercial software packages that perform the functions of this open-source project, these products are proprietary and do not provide source code; thus, they are not competitors to this project. It is reiterated that this project is provided primarily for academic and research purposes.
To perform power flow analysis of a power system, the system must be described in some concise manner such that it is easily accessible by some modern computational system. In the early 1970s, such a format was proposed and adopted. That data format is known as the IEEE Common Data format (CDF or cdf). The complete specification for this file format can be found in Appendix C.

The program written for this thesis reads system data from a .cdf file [16]. An advantage of using this file format is that there are several IEEE test cases readily available in this format. Disadvantages of this format are its age and that it only stores static data. For this thesis, the CDF format was determined to be sufficient. Figure 5.1 and Figure 5.2 show two formats of a power system. Most humans would prefer the system in the format in Figure 5.1. Computers operate on numbers so Figure 5.2 is preferred when coding; consequently Figure 5.1 must be converted to something like Figure 5.2 in code.

Figure 5.1: Graphical Format of a 5-Bus Power System [1]

There is a new XML format proposed by Milano, Zhou and Hou [17] to
define a power system. This format is more descriptive of a power system, containing such data as synchronous machine and regulator parameters as well as market data just to name a few. This new format is undoubtedly the format of the future, but companies still currently use proprietary formats. This proposed XML format has not been ratified into an IEEE standard. Based on a literature search and a survey of open-source and commercial power systems software, this new XML format has not gained widespread adoption. Should there be no standardization of any particular XML format, future work of this project will use this proposed XML format
```plaintext
01/15/15 MS ARCHIVE           100.0  1961 W GSO 5 Bus Test Case

BUS DATA FOLLOWS              5 ITEMS
1 One  000 1 1 3 1.000   0.0    0.0     0.0     0.0    -16.1   100.0  1.000     0.0     0.0   0.0    0.0        0
2 Two  000 1 1 0 0.0     0.0     800.0    280.0    0.0     0.0   100.0  0.0       0.0     0.0   0.0    0.0        0
3 Three 000 1 1 2 1.050   0.0     80.0     40.0    520.0   337.6   100.0  1.050     0.0     0.0   0.0    0.0        0
4 Four  000 1 1 0 0.0     0.0     0.0      0.0      0.0     0.0   100.0  0.0       0.0     0.0   0.0    0.0        0
5 Five  000 1 1 0 0.0     0.0     0.0      0.0      0.0     0.0   100.0  0.0       0.0     0.0   0.0    0.0        0

BRANCH DATA FOLLOWS           5 ITEMS
-999

LOSS ZONES FOLLOWS             1 ITEMS
1 GSO 30 BUS

INTERCHANGE DATA FOLLOWS       1 ITEMS
-99
1 Claytor  132    0.0  999.99  GSO     GSO 5 Bus Test Case

TIE LINES FOLLOWS              0 ITEMS
-999

END OF DATA
```
The tools used for this thesis research were:

- Intel I7 based computer running Microsoft Windows 8.1 and 10
- Microsoft Office 2016
- Microsoft Visio 2016
- Git and GitHub
- Editor: VIM emulation in Visual Studio
- Editor: Notepad++ using VIM emulation
- UML Modeling tool: Astah [18]
- UML Modeling tool: Enterprise Architect [19]
- VIM
- Notepad++
- Adobe Acrobat
- \LaTeX
- Miktek
- \TeXstudio

This code was compiled on an Intel I7 based machine with 8 GB of RAM. The operating system used was Microsoft Windows 10. All code was written in ISO C++14 using Microsoft Visual Studio Enterprise 2015. Version control is accomplished using git; and the project repository is on GitHub.
The code should be compilable and executable on any x64 machine that has an ISO compliant C++ compiler, with little effort. The final UML models were completed using Enterprise Architect Ultimate by Sparx Systems Pty Ltd. Many of the initial models and concepts were tested using Astah by Astah.net. At the time of the writing of this thesis, Astah provided a free one-year renewable license to the full product. It is easy enough for most coders to code without visualization tools such as UML modelers. UML models make presenting code to others much easier.
There are many libraries that can be used when programming in C++. For matrix manipulations the Matrix Template Library 4 (MTL4) [20] by SimuNova was chosen. The use of this library has many advantages including reduction in coding time. This library has all the support necessary for vector and matrix operations. For example, there is an LU, solver that is used to solve for $x$ given $A$ and $b$.

The MTL4 library is large and fairly robust. MTL4 has several different $Ax = b$ solvers but FreePFlow used only the `lu_solve()` solver to implement the Newton-Raphson method. During this project, difficulties were encountered using this library. Using this library, vectors and matrices are not easily inspected in a debugger such as the one in Microsoft Visual Studio. This requires additional steps in the debugging process, such as writing vector elements to other variables for inspection purposes. For this reason, thought was given to writing a matrix library for this project and the creator of C++ was even consulted (see Appendix D); that idea was abandoned for this initial version of the package.

Reasons for choosing C++ include but are not limited to:

- Object orientation
- Modularity
- Maintainability
- Ability to make libraries accessible by other languages
- Platform portability
- Speed
Definitions of the objects used are given below. Objects are capitalized to differentiate them from the physical entities.

**Grid** – Primarily consists of Branches and Busses. It can contain other devices as necessary.

**Branch** - Is a connection between Busses.

**Bus** - Represents an electrical node or bus to which branches, generators, loads, and other devices such as compensators, can be connected.

**Generator** - Object that models a real-world generator.

**Node** - Represents a conceptual node as in graph theory. It includes the node’s name, number, the total number of other nodes to which it is connected and a list of other connected nodes. It serves as the base class for a bus.

Note that in C++, structs are classes where all the members and member functions are public. Simple data such as bus and branch data are better represented by structs, thereby eliminating the need to have rudimentary member functions such as getmember() and setmember() to access the data [21]. This thesis may use class for struct, since the original code did implement those structs as classes. The released code uses structs for simple data.
In C++, classes are used to encapsulate data along with the functions that operate on that data. Functions in a class are called the member functions. Packaging data along with the functions that operate on the data assists in making code more easily extensible and maintainable. Structs (struct), short for structures, are classes whose members and member functions are all public. This chapter contains a list of the classes and structs used in FreePFlow.

9.1 Grid Class

The grid class is a representation of the power grid to be analyzed. The grid is a class that contains private members that are a vector of branches, and vector of buses. Transformer data is contained in the branch data of the .cdf file and subsequently there is not a transformer class.

A parser was written to parse a .cdf file and initialize the branches and nodes of the grid. This parser is a function of the grid class. A graphical representation of this class can be found in Figure 9.1

9.2 Branch Struct

The branch struct of Figure 9.2 represents a transmission line. Its members hold all the relevant parameters of a transmission line.
9.3 Node Struct

The node struct is a class that encapsulates all of the basic attributes of a node. It is the base struct for the bus struct. This struct can be used to write and experiment with code pertaining to graph theory. The node struct is depicted in Figure 9.3.

9.4 Bus Struct

This Bus struct is a descendant of the Node struct. It extends the Node struct to include all of the attributes specific to an electrical bus in a power grid. It was possible to implement the bus struct fully without creating and implementing the base node struct. Not using inheritance would eliminate using object oriented programming and the Bus struct would have to be implemented completely independent of the Node struct. There would be duplication of some attributes in each struct. The Bus struct is depicted in Figure 9.4.
9.5 Generator Class

The generator class, Figure 9.5, models a generator. This class is a usable base class that can be extended for different generator types. This class was created for future work in dynamic stability studies; this class is not used in the power flow solution.

9.6 PFSolver Class

The PFSolver Class models a power flow solver. This class takes a Grid object and solves the power flow for that grid. This class does all the work to solve the power flow problem. This class contains member functions that will compute the Jacobian and the power mismatches, and perform the Newton-Raphson iterations, among other functions.
class freepower

«struct»
Node
+ connectednodes: vector <int>
+ name: string
+ number: int

Figure 9.3: Node Struct

class freepower

«struct»
Bus
+ angle: precisiontype
+ B: precisiontype
+ baseMW: precisiontype
+ G: precisiontype
+ GenP: precisiontype
+ GenQ: precisiontype
+ loadflowarea: int
+ loadP: precisiontype
+ loadQ: precisiontype
+ losszonenumber: int
+ maxMvarorV: precisiontype
+ minMvarorV: precisiontype
+ pqtopv_count: size_t
+ pvtopq_count: size_t
+ remotebuscontrolled: int
+ type: int
+ vcontrolled: bool
+ voltage: precisiontype
+ Vset: precisiontype

Figure 9.4: Bus Struct

21
class freepower
class freepower
<struct>
Generator
+ angle: precisiontype
+ Edp: complex<precisiontype>
+ Efd: complex<precisiontype>
+ Eqp: complex<precisiontype>
+ H: precisiontype
+ Idq: complex<precisiontype>
+ IGen: complex<precisiontype>
+ KA: precisiontype
+ KE: precisiontype
+ KF: precisiontype
+ Rf: precisiontype
+ Rs: precisiontype
+ SE: complex<precisiontype>
+ TA: precisiontype
+ Tdop: precisiontype
+ TE: precisiontype
+ TF: precisiontype
+ TM: complex<precisiontype>
+ Tqop: precisiontype
+ Vdq: complex<precisiontype>
+ VR: complex<precisiontype>
+ Vref: complex<precisiontype>
+ Xd: complex<precisiontype>
+ Xdp: complex<precisiontype>
+ Xq: complex<precisiontype>
+ Xqp: complex<precisiontype>

Figure 9.5: Generator Struct
CHAPTER 10
NAMING SCHEME

A consistent naming scheme for variables, user-defined types (UDTs), and functions assist in code readability, maintenance, and debugging. There are several style naming conventions, Hungarian, camelCode, and PascalCode being the most common. The C and C++ languages have their naming convention and a derivative of that style is Stroustrup, named after the inventor of the C++ language; Stroustrup is the convention used in this thesis project.

C++ Core Guidelines is a new project hosted on GitHub that gives guidelines on many practices to help C++ programmers to write simpler, more efficient, more maintainable code [21]. Although the code in this project was written prior to the announcement of this project on September 9, 2015, it was edited to adhere as much as possible to those guidelines. Many index variables in loops are suffixed ndx. This adds clarity that the variable is an index and facilitates searching for and modifying it as necessary.

Example

\[
\text{for(size_t k=0; k<iN;++k)}
\]

becomes

\[
\text{for(size_t kndx=0; kndx>buses.sizeof();++kndx)}
\]

Using the above naming scheme trivializes the task of finding and replacing the ndx variable, in any particular routine, file, or even the entire code base. Beginning coders should try searching for and replacing a variable named n to see the difficulties involved with simple names.
10.1 Variables

Names of variables were also chosen to match power flow terms; i.e., $S$ is chosen as the variable for complex power. Effort was made to keep names as concise and descriptive as possible while making them unique enough to be easily located using text searches. Vectors, for instance, were suffixed with \texttt{vec}; e.g., $S_{\text{vec}}$ is the vector of real and reactive powers.

$$S_{\text{vec}} = \begin{bmatrix} P_1 \\ \vdots \\ P_N \\ Q_1 \\ \vdots \\ Q_N \end{bmatrix}$$ (10.1)

where

$$P(\delta, V) = S \begin{bmatrix} 0 \\ \vdots \\ N - 1 \end{bmatrix}$$ (10.2)

and

$$Q[\delta, V] = S \begin{bmatrix} N \\ \vdots \\ 2N - 1 \end{bmatrix}$$ (10.3)

In the code the Jacobian matrix, herein referred to as the Jacobian, is a member of the PFSolver class (see Figure ??). The capital $J$ is unique enough to the code that only it was used for the variable name. The Jacobian is used and defined as follows:

$$J \cdot \begin{bmatrix} \Delta \delta \\ \Delta V \end{bmatrix} = \begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix}$$ (10.4)

where
\[ J = \begin{bmatrix} \frac{\partial P}{\partial \delta} & \frac{\partial P}{\partial V} \\ \frac{\partial Q}{\partial \delta} & \frac{\partial Q}{\partial V} \end{bmatrix} \] (10.5)

10.2 Functions

Function names use all lower case and are descriptive. Words of functions are separated by an underscore. Global functions are discouraged. All functions are members of a class, and as such, a class must be instantiated to use its function; therefore, functions are very unique. To call a particular function its parent class must be instantiated and then it can be accessed via the call

\begin{verbatim}
namespace::classinstance.function_name(args) or
namespace::classinstancefunction_name(args)
\end{verbatim}

if the object was dynamically allocated. This naming scheme for functions is consistent with [21].
The algorithm used is quite simple. It is very common and based on the classical one proposed by Tinney and Hart [22]. The flowchart in Figure 11.1 shows an overview of the algorithm. The program reads the data of the system from a file, sets up all of the necessary objects, does a Newton-Raphson power flow, and then outputs the results to a text file.

This Power Flow module uses the Newton-Raphson method to solve the $Ax = b$ equation of the power system, where $A$ is the Jacobian of the system, $x$ is a vector of unknown voltages and angles, and $b$ is the power mismatch. The implementation of this solver is contained in the PFSolver class.
Figure 11.1: Flow Chart of the Power Flow Algorithm
12.1 Software Design

It is always a challenge to design a complex system. Determining the overall structure and layout of this tool was a daunting task. A wrong decision can lead to unmanageable and non-extensible code. Designing the power flow solver such that it was modular surmounted this challenge. Because the design of the overall power systems analysis tool is a modular one, the FreePFlow module is only one module of many to be included in the larger tool. In engineering terms, this modularity is just “black box” design for software. It is challenging to design a very large black box which is composed of many smaller black boxes, particularly when many of the smaller black boxes may change with respect to their input parameters and their output. Thus, much consideration was made in deciding on what classes were to be used.

12.2 Checking Generator Reactive Power Limits

In the real world, generators cannot produce an unlimited amount of reactive power; Kundur et al. [23] discuss this topic in detail. A search of the internet for the term “generator capability curve” will also give some insight. The most challenging part of this project was getting the simulated code to converge on all test cases while ensuring that the reactive power limits of the generators were met. Succinctly, in the simulation of the power flow for a system, PV busses are those which have generators attached. Ideally PV busses regulate the voltage on that bus to the voltage set point of the generator attached. If the reactive power output of that attached generator
reaches its upper or lower limit, the PV bus is switched to a PQ bus and the voltage on that PQ bus is set equal to the computed voltage. Checking and enforcing generator reactive power limits can be performed at different times using the Newton-Raphson (N-R) method. Generator reactive power limit checking can be done during each N-R iteration, after the N-R method has converged, or a hybrid of the two. The algorithm used to check and adjust the generator reactive power limits is based on the algorithm found in [24].

12.2.1 During Each N-R Iteration

Theoretically, if the reactive power limits of the generators in the system were checked and corrected after each N-R iteration, then once the N-R method has converged, the result would be a solution to the power flow of that system. The code as it was initially written used this paradigm. After many months of testing and tweaking, it has been determined that this method will not work reliably in all cases, all the time. Testing the code on some cases would work, then tweaking the code to work on other cases would cause it to fail on other cases. In most cases, convergence only was achieved if the checking did not begin until after the first N-R iteration; in other words, in almost all instances if checking reactive power limits on the generators began on the first iteration of the N-R algorithm, the systems diverged.

Much effort was given to making this strategy work, mainly because the strategy described in section 12.2.2 required a major rewrite of the software. After the exhaustive effort to get this paradigm working in the code, the conclusion was reached that, succinctly, if the changes to the generated reactive power suggested in [24] are performed too early during the N-R iterations, the guess is pushed out of the region of convergence, commonly referred to in the mathematics literature as the basin of attraction. The discussion of this is beyond the scope of this thesis.

12.2.2 After the N-R Method Has Converged

A more reliable method of checking generator reactive power limits is to allow the N-R algorithm to converge the system (inner loop), be it from a previous solution or a flat start (e.g. $\delta = 0$ and $V = 1$); check the generator
reactive power limit and adjust them as described by [24] (outer loop); if any adjustment were made in the outer loop, then repeat the inner loop; if no adjustments were made in the outer loop, then the system has converged and the result should be a valid power flow solution with all generator reactive power outputs being within each generator’s respective lower and upper limits.

This inner/outer loop paradigm is utilized by PowerWorld. To verify the inner loop of FreePFlow, generator reactive power limit was disabled in PowerWorld. The results from FreePFlow’s inner loop matched PowerWorld’s results within a reasonable tolerance across all the IEEE test cases that were used. Once the outer loop was employed for both, FreePFlow always converged and provided a valid power flow solution and again the results were close to PowerWorld’s solution; although the solutions were similar, the variations in the solutions were more significant than when just an inner loop computation was performed. After a discussion with Professor Thomas Overbye, the author became aware that during the power flow calculations, at different times during the inner and/or outer loops, PowerWorld uses multipliers to perhaps prevent generators from exceeding reactive power limits, or at least not allow those limits to be exceeded by extreme amounts.

With large non-linear systems, there quite possibly are many attractors or solutions to the system, some stable, and some unstable. While it is beyond the scope of this thesis, it is sufficient to mention that it is most probable that PowerWorld’s use of multipliers and other massaging techniques causes PowerWorld to arrive at a valid but slightly different solution from that determined by FreePFlow.

12.2.3 Hybrid

After many hours of writing code and running simulation test cases, it seems conceivable that one could use a hybrid of the two paradigms discussed in sections 12.2.1 and 12.2.2. While it may be possible to devise such a scheme, it is not obvious that there would be any advantages to it over either of the other two ways of checking and correcting generator reactive power limits. Some thought should be put into this idea to be exhaustive; further, Big-O notation should be included for a comparison of the three.
CHAPTER 13
USE

To use the power flow solver, run the executable from a command line with two parameters. The first parameter is the path and name of the input cdf file. The second parameter is the path and name of the output file. It is recommended that the name of the output file be appended with .txt to denote that it is a text file. Using Microsoft Windows to simulate and the power flow of the IEEE 14 bus example that is in c:\temp, write its results to a text file named ieee14bus-results.txt, open a cmd window and type:

    freepflow inputfile outputfile tolerance maxiterations
    freepflow c:\temp\ieee14-bus.cdf c:\temp\ieee14bus-results.txt 0.1 10

If tolerance and maxiterations are omitted, the program will run with defaults of 0.1 and 10 respectively for those parameters.
CHAPTER 14

RESULTS

The code works and gives results that are equivalent to those from the educational version of PowerWorld v17, with generator limits disabled and enabled. The output of the power flow is written to an output text file.

The compiled code was tested using the following test cases:

- Gso5bustest.cdf from [16]
- ieee14-bus.cdf
- ieee30-bus.cdf
- ieee57-bus.cdf
- ieee118-bus.cdf
- ieee300-bus.cdf

After several months, successful implementation of enforcing the reactive power limits of generators was achieved. The simulation results of the IEEE 30 bus test case from PowerWorld and FreePFlow are shown in Table 14.1 and Table 14.2. There are differences in these results. PowerWorld replaces zero value conductances on line with some very small finite amount. FreePFlow does not have a minimum conductance for lines, so that what is specified in the .cdf file is what gets simulated. The MTL4 library also introduces some of this difference.
Table 14.1: PowerWorld IEEE 30 Bus Results

<table>
<thead>
<tr>
<th>Number</th>
<th>PU Volt</th>
<th>Angle (Deg)</th>
<th>Gen(MW)</th>
<th>Gen(Mvar)</th>
<th>Load(MW)</th>
<th>Load(Mvar)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.06</td>
<td>0</td>
<td>260.94828</td>
<td>-16.78749</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.04313</td>
<td>-5.3518</td>
<td>40</td>
<td>50</td>
<td>21.7</td>
<td>12.7</td>
</tr>
<tr>
<td>3</td>
<td>1.02074</td>
<td>-7.53198</td>
<td></td>
<td></td>
<td>2.4</td>
<td>1.2</td>
</tr>
<tr>
<td>4</td>
<td>1.01176</td>
<td>-9.28409</td>
<td></td>
<td></td>
<td>7.6</td>
<td>1.6</td>
</tr>
<tr>
<td>5</td>
<td>1.01</td>
<td>-14.16575</td>
<td>0</td>
<td>36.84996</td>
<td>94.2</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>1.01026</td>
<td>-11.06459</td>
<td></td>
<td></td>
<td>22.8</td>
<td>10.9</td>
</tr>
<tr>
<td>7</td>
<td>1.00238</td>
<td>-12.8651</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.01</td>
<td>-11.81329</td>
<td>0</td>
<td>37.14406</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>9</td>
<td>1.05091</td>
<td>-14.10891</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1.04513</td>
<td>-15.69963</td>
<td>5.8</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1.082</td>
<td>-14.10891</td>
<td>0</td>
<td>16.17158</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1.05712</td>
<td>-14.94329</td>
<td></td>
<td>11.2</td>
<td>7.5</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1.071</td>
<td>-14.94329</td>
<td>0</td>
<td>10.61851</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1.04228</td>
<td>-15.83543</td>
<td>6.2</td>
<td>1.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1.03768</td>
<td>-15.92736</td>
<td>8.2</td>
<td>2.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1.04439</td>
<td>-15.52632</td>
<td>3.5</td>
<td>1.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1.0399</td>
<td>-15.86135</td>
<td>9</td>
<td>5.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>1.02815</td>
<td>-16.54167</td>
<td>3.2</td>
<td>0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1.02565</td>
<td>-16.71541</td>
<td>9.5</td>
<td>3.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>1.02974</td>
<td>-16.51885</td>
<td>2.2</td>
<td>0.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>1.03273</td>
<td>-16.14234</td>
<td>17.5</td>
<td>11.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>1.03326</td>
<td>-16.12811</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>1.02718</td>
<td>-16.31805</td>
<td>3.2</td>
<td>1.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>1.02158</td>
<td>-16.49463</td>
<td>8.7</td>
<td>6.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>1.01734</td>
<td>-16.06676</td>
<td></td>
<td></td>
<td>3.5</td>
<td>2.3</td>
</tr>
<tr>
<td>26</td>
<td>0.99966</td>
<td>-16.48641</td>
<td></td>
<td></td>
<td>3.5</td>
<td>2.3</td>
</tr>
<tr>
<td>27</td>
<td>1.02325</td>
<td>-15.54237</td>
<td></td>
<td></td>
<td>3.5</td>
<td>2.3</td>
</tr>
<tr>
<td>28</td>
<td>1.00682</td>
<td>-11.68838</td>
<td></td>
<td></td>
<td>3.5</td>
<td>2.3</td>
</tr>
<tr>
<td>29</td>
<td>1.00341</td>
<td>-16.77231</td>
<td>2.4</td>
<td>0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>0.99194</td>
<td>-17.65514</td>
<td>10.6</td>
<td>1.9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 14.2: PowerWorld IEEE 30 Bus Results

<table>
<thead>
<tr>
<th>Bus</th>
<th>Voltage(V)</th>
<th>Angle(deg)</th>
<th>GenP(MW)</th>
<th>GenQ(MVar)</th>
<th>loadP(MW)</th>
<th>loadQ(MVar)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.06000</td>
<td>0.00000</td>
<td>260.95189</td>
<td>-16.78737</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>2</td>
<td>1.04313</td>
<td>-5.35188</td>
<td>40.00000</td>
<td>50.00000</td>
<td>21.70000</td>
<td>12.70000</td>
</tr>
<tr>
<td>3</td>
<td>1.02074</td>
<td>-7.53204</td>
<td>0.00000</td>
<td>0.00000</td>
<td>2.40000</td>
<td>1.20000</td>
</tr>
<tr>
<td>4</td>
<td>1.01176</td>
<td>-9.28417</td>
<td>0.00000</td>
<td>0.00000</td>
<td>7.60000</td>
<td>1.60000</td>
</tr>
<tr>
<td>5</td>
<td>1.01000</td>
<td>-14.16587</td>
<td>0.00000</td>
<td>36.85033</td>
<td>94.20000</td>
<td>19.00000</td>
</tr>
<tr>
<td>6</td>
<td>1.01026</td>
<td>-11.06469</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>7</td>
<td>1.00238</td>
<td>-12.86520</td>
<td>0.00000</td>
<td>0.00000</td>
<td>22.80000</td>
<td>10.90000</td>
</tr>
<tr>
<td>8</td>
<td>1.01000</td>
<td>-11.81339</td>
<td>-0.00000</td>
<td>37.14439</td>
<td>30.00000</td>
<td>30.00000</td>
</tr>
<tr>
<td>9</td>
<td>1.05091</td>
<td>-14.10900</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>10</td>
<td>1.04513</td>
<td>-15.69972</td>
<td>0.00000</td>
<td>0.00000</td>
<td>5.80000</td>
<td>2.00000</td>
</tr>
<tr>
<td>11</td>
<td>1.08200</td>
<td>-14.10900</td>
<td>0.00000</td>
<td>16.17160</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>12</td>
<td>1.05712</td>
<td>-14.94338</td>
<td>0.00000</td>
<td>0.00000</td>
<td>11.20000</td>
<td>7.50000</td>
</tr>
<tr>
<td>13</td>
<td>1.07100</td>
<td>-14.94338</td>
<td>0.00000</td>
<td>10.61855</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>14</td>
<td>1.04228</td>
<td>-15.83552</td>
<td>0.00000</td>
<td>0.00000</td>
<td>6.20000</td>
<td>1.60000</td>
</tr>
<tr>
<td>15</td>
<td>1.03768</td>
<td>-15.92745</td>
<td>0.00000</td>
<td>0.00000</td>
<td>8.20000</td>
<td>2.50000</td>
</tr>
<tr>
<td>16</td>
<td>1.04439</td>
<td>-15.52641</td>
<td>0.00000</td>
<td>0.00000</td>
<td>3.50000</td>
<td>1.80000</td>
</tr>
<tr>
<td>17</td>
<td>1.03990</td>
<td>-15.86144</td>
<td>0.00000</td>
<td>0.00000</td>
<td>9.00000</td>
<td>5.80000</td>
</tr>
<tr>
<td>18</td>
<td>1.02815</td>
<td>-16.54176</td>
<td>0.00000</td>
<td>0.00000</td>
<td>3.20000</td>
<td>0.90000</td>
</tr>
<tr>
<td>19</td>
<td>1.02565</td>
<td>-16.71550</td>
<td>0.00000</td>
<td>0.00000</td>
<td>9.50000</td>
<td>3.40000</td>
</tr>
<tr>
<td>20</td>
<td>1.02974</td>
<td>-16.51894</td>
<td>0.00000</td>
<td>0.00000</td>
<td>2.20000</td>
<td>0.70000</td>
</tr>
<tr>
<td>21</td>
<td>1.03273</td>
<td>-16.14244</td>
<td>0.00000</td>
<td>0.00000</td>
<td>17.50000</td>
<td>11.20000</td>
</tr>
<tr>
<td>22</td>
<td>1.03326</td>
<td>-16.12820</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>23</td>
<td>1.02718</td>
<td>-16.31814</td>
<td>0.00000</td>
<td>0.00000</td>
<td>3.20000</td>
<td>1.60000</td>
</tr>
<tr>
<td>24</td>
<td>1.02158</td>
<td>-16.49472</td>
<td>0.00000</td>
<td>0.00000</td>
<td>8.70000</td>
<td>6.70000</td>
</tr>
<tr>
<td>25</td>
<td>1.01734</td>
<td>-16.06685</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>26</td>
<td>0.99966</td>
<td>-16.48651</td>
<td>0.00000</td>
<td>0.00000</td>
<td>3.50000</td>
<td>2.30000</td>
</tr>
<tr>
<td>27</td>
<td>1.02325</td>
<td>-15.54246</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>28</td>
<td>1.00682</td>
<td>-11.68848</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>29</td>
<td>1.00341</td>
<td>-16.77241</td>
<td>0.00000</td>
<td>0.00000</td>
<td>2.40000</td>
<td>0.90000</td>
</tr>
<tr>
<td>30</td>
<td>0.99194</td>
<td>-17.65523</td>
<td>0.00000</td>
<td>0.00000</td>
<td>10.60000</td>
<td>1.90000</td>
</tr>
</tbody>
</table>
Some of the work that can be done to improve the power flow module is:

- Program output should be XML.
- Make this solver 3-phase.
- Add a GUI to draw a power system.
- Possibly add a matrix class - MTL4 is very complex, a simple matrix class may be better.
- Simple matrix class - coding of simple LU factorization (lightweight).
- Introduce CUDA - parallel processing using NVidia graphics processors.
- Create a cloud or web interface for it.
- Optimize the code for particular platforms.

Things that can be done with this code are:

- Code dynamic stability module.
- Code optimal power flow (OPF) model.
- Code security constrained OPF (SCOPF).
- Add features and give back to the GitHub project so that other students can benefit.
The outcome of this project is a working power flow module, FreePFlow, that is written in ISO C++14 and is open-source. Complete source code can be obtained from GitHub. It is hoped that this project will be expanded by the community and additions given back to the community. The results given from this FreePFlow match those given by other products. Speed comparisons were not given as many of them rely on the matrix operations involved, and since those were performed by a third-party matrix library [20], it would be more appropriate to gather that information from the provider or owner of that library.
APPENDIX A

THE POWER FLOW PROBLEM

Figure A.1: 3-Bus Simple System

The power flow solution of an electrical power system, often referred to as a grid, is the solution of the steady-state voltages and angles at each bus in the system. Implicit in that solution is the current and power injected and withdrawn from each bus. Figure A.1 is a simple power system with two generators, one load, three busses and two transmission lines. An overview of solving the power flow of a system is given. To solve the power flow problem:

Compute the $Y_{bus}$ for the system

\[
Y_{bus} = \begin{bmatrix}
Y_{11} & Y_{12} & \cdots & Y_{1j} \\
Y_{21} & \ddots & \ddots \\
\vdots & \ddots & \ddots \\
Y_{i1} & & & Y_{ij}
\end{bmatrix}
\]  

(A.1)

where

\[
Y_{ii} = \sum y_{ii}
\]

(A.2)
\[ Y_{ij} = Y_{ji} = -\sum y_{ij} \forall i \neq j \]  

- The diagonal entries of the \( Y_{bus} \) are the sum of all admittances connected to the bus.

- The \( Y_{bus} \) matrix is computed for the convenience of having the conductances and susceptances required for computing the powers in the power flow equations. The conductances (\( G_s \)) are the real part of the \( Y_{bus} \) entries and the susceptances (\( B_s \)) are the imaginary part of the \( Y_{bus} \) entries, e.g.,

\[ Y_{ij} = G_{ij} + jB_{ij} \]  

- The off-diagonal entries of the \( Y_{bus} \) are the negative of the sum of the admittances connected between buses \( i \) and \( j \).

- The \( Y_{bus} \) in some cases is symmetrical, and when this symmetry occurs only the diagonal and the upper or lower triangle of it need to be stored. If there are tap or phase changing transformers in the system, the \( Y_{bus} \) will not be symmetrical.

Compute the Jacobian for the system

The Jacobian is a gradient of the system. It is a list of sensitivities of real powers to voltages and angles and reactive powers to voltages and angles. The Jacobian of the system is computed by

\[ J = \begin{bmatrix}
\frac{\partial P}{\partial \delta} & \frac{\partial P}{\partial V} \\
\frac{\partial Q}{\partial \delta} & \frac{\partial Q}{\partial V}
\end{bmatrix} \]  

The full Jacobian is a \( 2N \) square matrix, where \( N \) is the number of busses in the system. The system may be simplified, reducing the size of the Jacobian to less than a \( 2N \times 2N \) matrix. Details of simplifying the system are given later in this appendix.

- In a power system, the Jacobian is sparse. Non-zero entries exist only where there are connections between nodes.

Compute the power mismatches for each bus of the system:
# The apparent power of each bus is a function of the voltages and angles of the bus and all the busses to which it is connected, and the admittances of the lines connected the other busses to it. It is:

\[
S_k = P_k + Q_k = V_k \sum_{n}^{N} Y_{kn} V_n e^{j(\delta_k - \delta_n - \delta_{kn})} \quad (A.6)
\]

In rectangular form:

\[
S_k = P_k + jQ_k = V_k \sum_{n=1}^{N} Y_{kn} V_n \left[ \cos(\delta_k - \delta_n - \theta_{kn}) + j \sin(\delta_k - \delta_n - \theta_{kn}) \right] \quad (A.7)
\]

From A.7 the real and reactive power quantities are obtained by taking the real and imaginary parts of the apparent power, respectively:

\[
P_k = \text{Re}\{S_k\} = V_k \sum_{n=1}^{N} Y_{kn} V_n \cos(\delta_k - \delta_n - \theta_{kn}) \quad (A.8)
\]

\[
Q_k = \text{Im}\{S_k\} = V_k \sum_{n=1}^{N} Y_{kn} V_n \sin(\delta_k - \delta_n - \theta_{kn}) \quad (A.9)
\]

- If the norm of the power mismatches is less than a defined tolerance \( \epsilon \), then the Newton-Raphson iterations can stop; the infinity norm is used.

- To solve the power flow, \( A = J \). Initially, the \( x \) vector is set to an initial guess of the angles and voltages of the busses of the system. It is initialized as in equation A.10. Don’t forget that \( J \) is a function of \( x \). \( \nu \) denotes the iteration number of the Newton-Raphson method.
\[
x^\nu = \begin{bmatrix} \delta^\nu \\ V^\nu \end{bmatrix} = \begin{bmatrix} \delta_1^\nu \\ \vdots \\ \delta_i^\nu \\ V_1^\nu \\ \vdots \\ V_i^\nu \end{bmatrix} \quad (A.10)
\]

The \( b \) vector is defined to be the net injected real and reactive powers injected through a path not included in the \( Y_{bus} \), and \( \nu \) denotes the iteration number of the Newton-Raphson method.

\[
b^\nu = S - S^\nu = \begin{bmatrix} P^\nu \\ Q^\nu \end{bmatrix} = \begin{bmatrix} P_1 - P_1^\nu \\ P_2 - P_2^\nu \\ \vdots \\ P_i - P_i^\nu \\ Q_1 - Q_1^\nu \\ Q_2 - Q_2^\nu \\ \vdots \\ Q_i - Q_i^\nu \end{bmatrix} \quad (A.11)
\]

Using A.5, A.10 and A.11 as \( A \), \( x \), and \( b \) respectively, the power flow is solved iteratively using the New-Raphson (NR) method. The NR method solves for the \( x \) vector by iterating through A.12 and A.13, calculating a new Jacobian during each iteration.

\[
J \bigg|_{x^\nu} \Delta x = \Delta b \rightarrow J \begin{bmatrix} \Delta \delta \\ \Delta V \end{bmatrix} = \begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} \quad (A.12)
\]

\[
x^{\nu+1} = x^\nu + \Delta x \quad (A.13)
\]

- We do not invert matrices. The operation is too computationally expensive. Instead we use lower and upper triangular factorization, forward and backward matrix solution schemes to solve A.12.
- The iterative process ceases when a specified tolerance is reached.
• Simplification – This is important! The $A$ matrix, $x$ and $b$ vectors can be simplified; the entries in them can be removed. The entries corresponding to the slack bus can be removed since the voltage and the angle are known for that bus. Likewise, for generator or PV busses, the voltage is known (it is the desired or set voltage of the generator); thus, the reactive power and voltages entries for the PV busses can be removed from the $A$ matrix, $x$ and $b$ vectors. Figure A.2 and Figure A.3 are the GSO 5-bus test [1] case and the described simplifications. If there are $n$ busses in a system that has $s$ slack buses and $p$ generator (PV) buses, then the Jacobian reduces to a $2n - 2s - p$ squared matrix and the $x$ and $b$ vectors become $2n - 2s - p \times 1$ vectors.

![Figure A.2: GSO 5-Bus Case](image-url)
Figure A.3: Simplification of the GSO 5-Bus Case
What constitutes a valid power flow solution of a system? Once a power flow algorithm has solved a system, how does one know if the solution is valid? These questions were posed to others and the answers varied. Based on responses to the above questions, these observations are suggested to determine if a power flow solution may be valid:

1. The algebraic equations used are valid; that is, there are no errors in equations A.1 through A.13.

2. There are no low voltages in the solution.

3. There are no large differences in angles of any bus directly connected to any other bus. The largest difference should be no larger than \( \frac{\pi}{2} \) radians.

4. The solution must not be an unstable equilibrium point (UEP).

Specifics of low and large differ not be given; however, typically, the bus voltages of the system should be close to 1 per unit (pu). And, angle differences between two directly connected busses can only become \( \frac{\pi}{2} \) radians if the line connecting them is purely inductive. As there is always some resistance in a real line, the angle difference between any two directly connected busses should be much less than the limiting \( \frac{\pi}{2} \) radians.

The fourth item in the list above is not trivial to determine. To determine if the solution is not a UEP, the eigenvalues of the system must be found. For the system to be at a stable equilibrium point (SEP), all of the eigenvalues must lie in the left-hand plane of the imaginary coordinate system. For large systems the computation time may be prohibitive.

Finding a solution to the system \( Ax = b \) iteratively is dependent on the initial guess. A wrong guess could cause divergence. Mathematicians use
terms such as attractors, and basins of attraction, in their literature. After working months on the algorithm that ensures that all generators remain within their reactive power limits, algorithm working, there was an epiphany: When checking generator reactive power limits at each Newton-Raphson iteration, what if the changes to the generated reactive power on the generator bus were pushing the current guess outside of the region of convergence for the system? That seems to have been the problem! The topic of power flow, Newton-Raphson and basins of attraction, or region of convergence, will be addressed in a later paper. Some details on this issue are provided in [25], [26]. Perhaps the inner/outer loop method described in chapter 12 of this thesis is the best way of using the Newton-Raphson method to achieve a valid power flow solution. It has proven to work on every case on which it was tested. There may be no need to further investigate basins of attraction, etc., with respect to power systems; however, in the spirit of thoroughness, the investigation has already begun. It is important that one understands what constitutes a valid power flow. Searching for written documentation of such proved to be futile. Hopefully, with this section of this thesis, those seeking to learn power flow will have some idea of what they are looking to achieve.
Partial Description of the IEEE Common Data Format for the Exchange of Solved Load Flow Data


The data file has lines of up to 128 characters. The lines are grouped into sections with section headers. Data items are entered in specific columns. No blank items are allowed, enter zeros instead. Floating point items should have explicit decimal point. No implicit decimal points are used.

Data type codes: A - Alphanumeric (no special characters)
I - Integer
F - Floating point
* - Mandatory item

Title Data
==========

First card in file.

Columns 2-9 Date, in format DD/MM/YY with leading zeros. If no date provided, use 00/00/00 where b is blank.

Columns 11-30 Originator’s name (A)

Columns 32-37 MVA Base (F*)

Columns 39-42 Year (I)

Column 44 Season (S - Summer, W - Winter)

Column 46-73 Case identification (A)
Bus Data *
==============

Section start card *:
---------------------

Columns  1-16 BUS DATA FOLLOWS (not clear that any more than BUS in 1-3 is significant) *

Columns  7- ? NNNNN ITEMS (column not clear, I would not count on this)

Bus data cards *:
--------------------

Columns 1- 4 Bus number (I) *
Columns  7-17 Name (A) (left justify) *
Columns 19-20 Load flow area number (I) Don't use zero! *
Columns 21-23 Loss zone number (I)
Columns 25-26 Type (I) *

0 - Unregulated (load, PQ)
1 - Hold MVAR generation within voltage limits, (PQ)
2 - Hold voltage within VAR limits (gen, PV)
3 - Hold voltage and angle (swing, V-Theta) (must always have one)
Columns 28-33 Final voltage, p.u. (F) *
Columns 34-40 Final angle, degrees (F) *
Columns 41-49 Load MW (F) *
Columns 50-59 Load MVAR (F) *
Columns 60-67 Generation MW (F) *
Columns 68-75 Generation MVAR (F) *
Columns 77-83 Base KV (F)
Columns 85-90 Desired volts (pu) (F) (This is desired remote voltage if this bus is controlling another bus.
Columns 91-98 Maximum MVAR or voltage limit (F)
Columns 99-106 Minimum MVAR or voltage limit (F)
Columns 107-114 Shunt conductance G (per unit) (F) *
Columns 115-122 Shunt susceptance B (per unit) (F) *
Columns 124-127 Remote controlled bus number

Section end card:
-----------------

Columns 1- 4 -999

Branch Data *
==============

Section start card *:
---------------------

Columns  1-16 BRANCH DATA FOLLOWS (not clear that any more than BRANCH is significant) *
Branch data cards *:
------------------------

Columns 1-4 Tap bus number (I) *
For transformers or phase shifters, the side of the model
the non-unity tap is on.

Columns 6-9 Z bus number (I) *
For transformers and phase shifters, the side of the model
the device impedance is on.

Columns 11-12 Load flow area (I)

Columns 13-14 Loss zone (I)

Column 17 Circuit (I) * (Use 1 for single lines)

Column 19 Type (I) *
0 - Transmission line
1 - Fixed tap
2 - Variable tap for voltage control (TCUL, LTC)
3 - Variable tap (turns ratio) for MVAR control
4 - Variable phase angle for MW control (phase shifter)

Columns 20-29 Branch resistance R, per unit (F) *

Columns 30-40 Branch reactance X, per unit (F) * No zero impedance lines

Columns 41-50 Line charging B, per unit (F) * (total line charging, +B)

Columns 51-55 Line MVA rating No 1 (I) Left justify!

Columns 57-61 Line MVA rating No 2 (I) Left justify!

Columns 63-67 Line MVA rating No 3 (I) Left justify!

Columns 69-72 Control bus number

Column 74 Side (I)
0 - Controlled bus is one of the terminals
1 - Controlled bus is near the tap side
2 - Controlled bus is near the impedance side (Z bus)

Columns 77-82 Transformer final turns ratio (F)

Columns 84-90 Transformer (phase shifter) final angle (F)

Columns 91-97 Minimum tap or phase shift (F)

Columns 98-104 Maximum tap or phase shift (F)

Columns 106-111 Step size (F)

Columns 113-119 Minimum voltage, MVAR or MW limit (F)

Columns 120-126 Maximum voltage, MVAR or MW limit (F)

Section end card:
-----------------

Columns 1-4 -999

Loss Zone Data
==============

Section start card
-------------------

Columns 1-16 LOSS ZONES FOLLOWS (not clear that any more than LOSS
is significant)

Columns 40?- ? NNNNN ITEMS (column not clear, I would not count on this)

Loss Zone Cards:
-----------------

Columns 1-3  Loss zone number (I)
Columns 5-16  Loss zone name (A)

Section end card:
-----------------

Columns 1-3  -99

Interchange Data *
==================

Section start card
------------------

Columns 1-16  INTERCHANGE DATA FOLLOWS (not clear that any more than
first word is significant).
Columns 40?- ? NNNNN ITEMS (column not clear, I would not count on this)

Interchange Data Cards *:
--------------------------

Columns 1-2  Area number (I) no zeros! *
Columns 4-7  Interchange slack bus number (I) *
Columns 9-20 Alternate swing bus name (A)
Columns 21-28 Area interchange export, MW (F) (+ = out) *
Columns 30-35 Area interchange tolerance, MW (F) *
Columns 38-43 Area code (abbreviated name) (A) *
Columns 46-75 Area name (A)

Section end card:
-----------------

Columns 1-2  -9

Tie Line Data
=============

Section start card
------------------

Columns 1-16 TIE LINES FOLLOWS (not clear that any more than TIE
is significant)

Columns 40?- ? NNNNN ITEMS (column not clear, I would not count on this)
<table>
<thead>
<tr>
<th>Tie Line Cards:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Columns 1-4</td>
<td>Metered bus number (I)</td>
</tr>
<tr>
<td>Columns 7-8</td>
<td>Metered area number (I)</td>
</tr>
<tr>
<td>Columns 11-14</td>
<td>Non-metered bus number (I)</td>
</tr>
<tr>
<td>Columns 17-18</td>
<td>Non-metered area number (I)</td>
</tr>
<tr>
<td>Column 21</td>
<td>Circuit number</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section end card:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Columns 1-3</td>
<td>-999</td>
</tr>
</tbody>
</table>
Below is a correspondence with Dr. Bjarne Stroustrup, the inventor of C++, in which he explains why there is not a matrix class in the C++ ISO standard.

On 5/12/2015 11:02 AM, Stevens, Melvin R wrote:

Prof./Dr. Stroustrup,

I am coding Power Stability algorithm as an MS/PhD student here at the University of Illinois. I understand that you are a Computer Scientist. From my perspective computer science is a tool used by engineers and scientist to get their work and research done. Linear systems are really at the heart of every scientific, engineering, and mathematic discipline. Matrices are fundamental to all Linear Systems and are ubiquitous.

Why is there not a C++ Standard Matrix STL? I have read some of your books and kind of understand your philosophy of the C++ Language, but I still see no reason that a true Matrix Library is not a part of the C++ Standard. Currently I am using MTL4 and looked at the Boost class but as each has its own idiosyncrasies and shortcomings, I am thinking of just writing my own which I don’t want to do; I have enough other work to do.

The professional wants something very general and complicated and are reluctant to provide simple versions for beginners. Also, several commercial libraries exist soaking up effort. The academics are either not interested or only interested in clever novelty features.

Can you please help work a robust Matrix STL into the C++ standard?
I am probably 3-4 years away from my PhD. I am hoping that I can use a C++ Standard supplied Matrix Library in my code before then. If you have any video where you discuss this matter, can you please share the urls.

*I do what I can (e.g. See the Matrix chapter in TC++PL4) but for anything to work it would have to come from people using such matrix libraries.*

*You might like to look at MTL4: [http://www.simunova.com/de/node/24](http://www.simunova.com/de/node/24)*

Thanks.

I’d also like to thank you for providing Humanity with such a great language!!!

*Thanks.*

Melvin Stevens ECE Graduate Student (Power)
In anticipation of having to write a matrix class for FreePFlow, the author preemptively wrote some of the code that would handle the sparsity of power systems. Functions have been written and are contained in the FreePFlow code that will take as input a grid, which is a form of a graph, and optimally order that grid to minimize matrix operations pertaining to that grid. A description of this process was described by Ogbuobiri et al. [27] and Tinney et al. [28]. Those functions have been thoroughly tested and are known to work.

There are also some routines that pertain to dynamic stability, hence the Generator class in the code. The dynamic stability functions should be moved from this module to a specific module designed for dynamic stability studies. The dynamic stability functions and the Generator class implement an example found in Power System Dynamics and Stability written by Professors Peter Sauer and M.A. Pai [29].
Included below is the complete source code for the project less the copyright and license. Also place GitHub information here.

// FreePFLow.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <iostream>
#include <complex>
#include <vector>
#include <fstream>
#include <algorithm>
#include <iterator>
#include <string>
#include <math.h>
#include <cmath>
#include "fptypes.h"
#include "Grid.h"
#include "PFSolver.h"

using namespace std;
using namespace mtl;
using namespace freepower;

int main(int argc, char* argv[])
{

string inputfile;
if (argc > 1)
    inputfile = argv[1];
else
    //inputfile = "..\\..\\..\Data\GS05BusTest.cdf";
    //inputfile = "..\\..\\..\Data\ieee14-bus.cdf";
    inputfile = "..\\..\\..\Data\ieee30-bus.cdf";
    //inputfile = "..\\..\\..\Data\ieee57-bus.cdf";
    //inputfile = "..\\..\\..\Data\ieee118-bus-3.cf";
    //inputfile = "..\\..\\..\Data\ieee300-bus.cdf";

string outfile;
if (argc > 2)
    outfile = argv[2];
else
    outfile = "PFSolver_out.txt";

string tol;
if (argc > 3)
    tol = argv[3];
else
    tol = "1.0E-1";

Grid* testgrid = new Grid(inputfile);
PFSolver pfs(testgrid, outfile, stod(tol.c_str()), 15);
int gen_Q_violation{ 1 };
auto outer_iterator{ 0 };
while (gen_Q_violation)
{
    #if _FULLDEBUG
    cout << "Outer Loop Iteration " << outer_iterator << endl;
    #endif
    pfs.NRSolve();
gen_Q_violation = pfs.check_generator_limits(pfs.Svec, pfs.
    Svec_old, pfs.xvec);
    #if _FULLDEBUG
    if (gen_Q_violation)
    {
    }
struct Branch
{
  precisiontype Bp;
  int circuit;
  int controlBus;
  int fBus;
  int loadFlowArea;
  int lossZone;
  precisiontype maxLimit;
  precisiontype maxTapOrShift;
  precisiontype minLimit;
  precisiontype minTapOrShift;
  int mva1;
  int mva2;
  int mva3;
  precisiontype Rp;
  int side;
  precisiontype stepSize;
  int tBus;
  int type;
  precisiontype xfmrShifterFinalAngle;
  precisiontype xfmrTurnsRatioFinal;
  precisiontype Xp;
};

struct Node
{
  int number;
  string name;
};
//List of connected busses is used to optimally order the graph
vector<int> connectednodes; //List of connected Bus by their indices

///////////////////////////////////////////////////////////////
// Bus struct
// Implementation of the Class Bus
// Original author: Melvin Stevens
///////////////////////////////////////////////////////////////
struct Bus : public Node
{
    precisiontype angle;
    precisiontype B;
    precisiontype baseMW;
    precisiontype Vset;
    precisiontype G;
    precisiontype GenP;
    precisiontype GenQ;
    int loadflowarea;
    precisiontype loadP;
    precisiontype loadQ;
    int losszonenumber;
    precisiontype maxMvarorV;
    precisiontype minMvarorV;
    size_t pvtopq_count;
    size_t pqtopv_count;
    int remotebuscontrolled;
    int type;
    bool vcontrolled;
    precisiontype voltage;
};

///////////////////////////////////////////////////////////////
// Generator struct
// Implementation of the Class Generator
// Original author: Melvin Stevens

struct Generator
{
    precisiontype angle, Rf;
    complex<precisiontype> IGen, Idq, Vdq, Edp, Eqp, Efd, VR, Vref, TM;

    // Parameters
    precisiontype H, Rs, Tdop, Tqop;
    complex<precisiontype> Xd, Xdp, Xq, Xqp;

    // Exciter Parameters
    precisiontype KA, TA, KE, TE, KF, TF;
    complex<precisiontype> SE;
};


#if !defined(EA_F88E3EA7_D66D_4b6a_BC4C_CDC0C91B8B59__INCLUDED_)
#define EA_F88E3EA7_D66D_4b6a_BC4C_CDC0C91B8B59__INCLUDED_
#include <vector>
#include <set>
#include <string>
#include "fptypes.h"
#include "../..\libs\boost_1_60_0\boost\numeric\mtl\mtl.hpp"

using namespace mtl;
namespace freepower
{
class Grid
{
private:
//std::vector<Branch> branches;
//std::vector<Bus> busses;
//std::string filename;
//precisiontype systembase;

public:

std::vector<Branch> branches;
std::vector<Bus> busses;
set<int> pvs;  //Contains the internal bus number starting
   with 0.
set<int> slacks;
   //vector<int> pvs;  //Contains the internal bus number
   starting with 0.
   //vector<int> slacks;
   //set<int> generator_exceeding_limits;

std::string filename;
precisiontype systembase;
dense2D<complex<precisiontype>> Ybus;

Grid();
Grid(string in_file_name);

virtual ~Grid();

int initpvs();
int initslacks();

int getSystemParameters();
int computeYbus();
std::vector<Bus>& getbusses() { return busses; };
int setup();
};

} #endif // !defined(EA_F88E3EA7_D66D_4b6a_BC4C_CDC0C91B8B59__INCLUDED_)

/////////// End of Grid.h

#include "Grid.h"
#include <iostream>
#include <fstream>
#include <complex>
#include "fptypes.h"
#include "..\..\libs\boost_1_60_0\boost\numeric\mtl\mtl.hpp"

using namespace std;
using namespace mtl;

freepower::Grid::Grid() {

if (filename.empty()) {

    //filename = "C:\Users\Melvin\SkyDrive\3-Phase Research\Data\MSRSingleMachineInfiniteBusTest-Full3Phase.cdf";
    //filename = "C:\Users\Melvin\SkyDrive\3-Phase Research\Data\MSRSingleMachineInfiniteBusTest.cdf";
    //filename = "C:\Users\Melvin\SkyDrive\3-Phase Research\Data\GSO5BusTest.cdf";

}
//filename = "C:\Users\Melvin\SkyDrive\3-Phase Research \Data\ieee14-bus.cdf";
//filename = "C:\Users\Melvin\SkyDrive\3-Phase Research \Data\ieee30-bus.cdf";
//filename = "C:\Users\Melvin\SkyDrive\3-Phase Research \Data\ieee57-bus.cdf";
//filename = "C:\Users\Melvin\SkyDrive\3-Phase Research \Data\ieee118-bus.cdf";
//filename = "C:\Users\Melvin\SkyDrive\3-Phase Research \Data\ieee118-bus-2.cdf";
filename = "C:\Users\Melvin\SkyDrive\3-Phase Research\Data\ieee300-bus.cdf";
//filename = "C:\Users\Melvin\SkyDrive\3-Phase Research \Data\dd17.cdf";
//filename = "C:\Users\Melvin\SkyDrive\3-Phase Research \Data\Anderson9CDF.cdf";
//filename = "C:\Users\Melvin\SkyDrive\3-Phase Research \Data\kundar2area.cdf";
}
setup();
}

freepower::Grid::Grid(string in_file_name)
{
filename = in_file_name;
setup();
}

freepower::Grid::~Grid()
{
}

int freepower::Grid::initpvs()
{
for (int i{ 0 }; i < busses.size(); i++)
{
if (busses[i].type == 2)
pvs.insert(i);
int freepower::Grid::initSlacks()
{
    for (int i{ 0 }; i < busses.size(); i++)
    {
        if (busses[i].type == 3)
            slacks.insert(i);
        //slacks.push_back(i);
    }
    return 0;
}

int freepower::Grid::computeYbus()
{
    for (Branch branch : branches)
    {
        complex<precisiontype> c1(branch.Rp, branch.Xp);
        complex<precisiontype> c2(0.0, branch.Bp / 2.0);

        complex<precisiontype> cfromto = Ybus(branch.fBus, branch.tBus);
        complex<precisiontype> ctofrom = Ybus(branch.tBus, branch.fBus);
        complex<precisiontype> cfromfrom = Ybus(branch.fBus, branch.fBus);
        complex<precisiontype> ctoto = Ybus(branch.tBus, branch.tBus);

        if (branch.xfmrTurnsRatioFinal != 0.0)
        {
            Ybus(branch.fBus, branch.tBus) -= 1.0 / ((complex<precisiontype>(branch.Rp, branch.Xp) * branch.xfmrTurnsRatioFinal));
            Ybus(branch.tBus, branch.fBus) = Ybus(branch.fBus, branch.tBus);
        }
Ybus(branch.tBus, branch.tBus) += 1.0 / complex<
    precisiontype>(branch.Rp, branch.Xp) + complex<
    precisiontype>(0.0, branch.Bp) / 2.0;
Ybus(branch.fBus, branch.fBus) += 1.0 / (complex<
    precisiontype>(branch.Rp, branch.Xp)*branch.xfmrTurnsRatioFinal*branch.xfmrTurnsRatioFinal) + complex
    <precisiontype>(0.0, branch.Bp) / 2.0;
}
else
{
Ybus(branch.fBus, branch.tBus) -= 1.0 / complex<
    precisiontype>(branch.Rp, branch.Xp);
Ybus(branch.tBus, branch.fBus) = Ybus(branch.fBus, branch.tBus);
Ybus(branch.tBus, branch.tBus) += 1.0 / complex<
    precisiontype>(branch.Rp, branch.Xp) + complex<
    precisiontype>(0.0, branch.Bp) / 2.0;
Ybus(branch.fBus, branch.fBus) += 1.0 / complex<
    precisiontype>(branch.Rp, branch.Xp) + complex<
    precisiontype>(0.0, branch.Bp) / 2.0;
}
for (size_t n = 0; n < busses.size(); ++n)
{
    if (busses[n].B > 0.0)
    Ybus(n, n) = Ybus(n, n) + complex<precisiontype>(0.0, busses[n].B);
}
return 0;
}

int freepower::Grid::setup()
{
    getSystemParameters();
    initpvs();
initslacks();
Ybus = dense2D<complex<precisiontype>>(busses.size(), busses
   .size());
computeYbus();
return 0;
}

int freepower::Grid::getSystemParameters()
{

    string line, word;
    ifstream in(filename.c_str());
    if (!in.is_open()) return 1;

    //TODO Need to implement a check to ensure that this line
    //read is the one needed from Title data
    getline(in, line);
    systembase = stod(line.substr(31, 6).c_str());

    while (in >> word)
    {
        //Build the vector of busses
        regex rgex_bus("bus", regex_constants::icase);
        if (regex_match(word, rgex_bus))
        {
            in >> word;
            regex rgex_data("data", regex_constants::icase);
            if (regex_match(word, rgex_data))
            {
                in >> word;
                regex rgex_follows("follows", regex_constants::icase);
                if (regex_match(word, rgex_follows))
                {
                    //in.setf(ios::fixed, ios::floatfield);
                    in.precision(2);
                    in >> word;
                    int Nbusses{ stoi(word) };
                    getline(in, line); //Flush the line
                    for (auto i = 0; i < Nbusses; ++i)
{ 
Bus tn;
getline(in, line);
//int temp = stoi(line.substr(25, 2));

// 0 - Unregulated (load, PQ)
// 1 - Hold MVAR generation within voltage limits, (PQ)
// 2 - Hold voltage within VAR limits(gen, PV)
// 3 - Hold voltage and angle(swing, V - Theta) (must always have one)

if (stoi(line.substr(25, 2)) == 2)
{
    tn.vcontrolled = true;
    //TODO: Removing PVs from this routine ***********************
    //Because the PVs may change
    // pvs.push_back(i); //push the bus index
    // pvs.push_back(stoi(line.substr(0, 4)));
    //*********************************************************
}
else
    tn.vcontrolled = false;
    //TODO: Removing PVs from this routine ***********************
    //Because the PVs may change
    // pvs.push_back(i); //push the bus index
    // pvs.push_back(stoi(line.substr(0, 4)));
    //*********************************************************

if (stoi(line.substr(25, 2)) == 3) // add to include Slack Bus || stoi(line.substr(25, 2)) == 3)
    slacks.push_back(i);
    //*********************************************************

    tn.number = stoi(line.substr(0, 4));
    tn.name = line.substr(4, 13);
    tn.loadflowarea = stoi(line.substr(17, 3));

65
tn.losszonenumber = stoi(line.substr(20, 3));
tn.type = stoi(line.substr(23, 3).c_str());
tn.voltage = stod(line.substr(26, 7).c_str());
tn.angle = stod(line.substr(33, 6).c_str())*pi / 180.0; // Angle is in Degrees in .cdf file
tn.loadP = stod(line.substr(40, 9).c_str());
tn.loadQ = stod(line.substr(49, 10).c_str());
tn.GenP = stod(line.substr(59, 6).c_str());
tn.GenQ = stod(line.substr(67, 8).c_str());
//tn.GenP =0 ;
//tn.GenQ =0;
tn.baseMW = stod(line.substr(75, 8).c_str());
tn.Vset = stod(line.substr(83, 7).c_str());
tn.maxMvarorV = stod(line.substr(90, 8).c_str());
tn.minMvarorV = stod(line.substr(98, 8).c_str());
tn.G = stod(line.substr(106, 8).c_str());
tn.B = stod(line.substr(114, 8).c_str());
tn.remotebuscontrolled = stoi(line.substr(122, 5));
//Added for bus switching during Generator Var limits checking
tn.pqtopv_count = 0;
tn.pvtopq_count = 0;
busses.push_back(tn);
}
}
}

//Build the vector of branches
regex rgex_branch("branch", regex_constants::icase);
if (regex_match(word, rgex_branch))
{
in >> word;
regex rgex_data("data", regex_constants::icase);
if (regex_match(word, rgex_data))
{
in >> word;
regex rgex_follows("follows", regex_constants::icase);
if (regex_match(word, regex_follows))
{
    in >> word;
    int Nbranches{ stoi(word) };
    getline(in, line);  //Flush the line
    Branch tempbranch;
    for (auto n = 0; n < Nbranches; ++n)
    {
        in >> word;
        for (int i{ 0 }; i < busses.size(); ++i)
        {
            if (stoi(word) == busses[i].number)
                tempbranch.fBus = i;
        }
        //tempbranch.fBus = stoi(word);
        in >> word;
        for (int i{ 0 }; i < busses.size(); ++i)
        {
            if (stoi(word) == busses[i].number)
                tempbranch.tBus = i;
        }
        //tempbranch.tBus = stoi(word);

        // Now set the valences and connected busses
        // This sets up the graph of the system
        busses[tempbranch.fBus].connectednodes.push_back(tempbranch.tBus);
        busses[tempbranch.tBus].connectednodes.push_back(tempbranch.fBus);

        in >> word;
        tempbranch.loadFlowArea = stoi(word);
        in >> word;
        tempbranch.lossZone = stoi(word);
        in >> word;
        tempbranch.circuit = stoi(word);
    }
tempbranch.type = stoi(word);
in >> word;
tempbranch.Rp = stod(word.c_str());
//Set minimum line resistance
//if (tempbranch.Rp < 0.0000001)
//    tempbranch.Rp = 0.0000001;
in >> word;
tempbranch.Xp = stod(word.c_str());
in >> word;
tempbranch.Bp = stod(word.c_str());
in >> word;
tempbranch.mva1 = stoi(word);
in >> word;
tempbranch.mva2 = stoi(word);
in >> word;
tempbranch.mva3 = stoi(word);
in >> word;
tempbranch.controlBus = stoi(word);
in >> word;
tempbranch.side = stoi(word);
in >> word;
tempbranch.xfmrTurnsRatioFinal = stod(word.c_str());
in >> word;
tempbranch.xfmrShifterFinalAngle = stod(word.c_str());
in >> word;
tempbranch.minTapOrShift = stod(word.c_str());
in >> word;
tempbranch.maxTapOrShift = stod(word.c_str());
in >> word;
//tempbranch.stepSize= stoi(word);
//in >> word;
//tempbranch.minLimit = stoi(word);
//in >> word;
//tempbranch.maxLimit = stoi(word);
getline(in, line); //Flush the line
branches.push_back(tempbranch);
}
in.close();
return 0;
}

///////////  End of Grid.h


#pragma once

class PFSolver
{
public:
PFSolver();
PFSolver(Grid* g);
PFSolver(Grid* g, precisiontype tol);
PFSolver(Grid* g, string outfile);

namespace freepower
{
    class PFSolver
    {

public:
    PFSolver();
PFSolver(Grid* g);
PFSolver(Grid* g, precisiontype tol);
PFSolver(Grid* g, string outfile);
PFSolver(Grid* g, string outfile, precisiontype tol, size_t max_iters);

virtual ~PFSolver();

int add_loads_back_to_S(dense_vector<precisiontype>& S_vector);
int check_generator_limits(dense_vector<precisiontype>& S_vector, dense_vector<precisiontype>& S_vector_previous, dense_vector<precisiontype>& x_vector);
int compute_Jacobian(dense2D<precisiontype>& Jacobian_matrix, const dense_vector<precisiontype>& x_vector);
int shrink_Jacobian(const dense2D<precisiontype>& Jacobian_matrix, dense2D<precisiontype>& smallJmat);
int compute_Q(dense_vector<precisiontype>& S_vector, dense_vector<precisiontype>& x_vector);
int compute_V(dense_vector<precisiontype>& S_vector, dense_vector<precisiontype>& x_vector);
int compute_S(dense_vector<precisiontype>& S_vector, const dense_vector<precisiontype>& x_vector);
int compute_Sss(dense_vector<precisiontype>& S_steady_state);
int initpvs(set<int>& pv_bus_set);
int initslacks(set<int>& slack_bus_set);
int init_x(dense_vector<precisiontype>& x_vector, const size_t iter);
int init_x_from_small_x(dense_vector<precisiontype>& x_vector, const dense_vector<precisiontype>& smallvec);
int NRSolve();
int NRSolve(string outfile);
int NRSolve(string infile, string outfile);
int outputpu(const dense_vector<precisiontype>& S_vector, const dense_vector<precisiontype>& x_vector, ofstream & outfile);
int output(const dense_vector<precisiontype>& S_vector, const dense_vector<precisiontype>& x_vector, ofstream & outfile);
int save_S(dense_vector<precisiontype>& S_vector);
int save_x(const dense_vector<precisiontype> &x_vector);
int setup();
int shrink_vec(dense_vector<precisiontype> &b, dense_vector<precisiontype> &bsmall);

// private:
Grid* grid = nullptr;
string infilename;
string outfilename;
size_t max_iterations = NULL;
precisiontype tolerance;

//TODO: Remove this line
size_t matrix_dim;
// precisiontype mismatch_error;

// The Steady State Vector should never change. So it's outside the loop.
dense_vector<precisiontype> Sss;// = dense_vector<precisiontype>(matrix_dim, 0.0);
dense_vector<precisiontype> Svec;// { dense_vector<precisiontype>(matrix_dim, 0.0) };
dense_vector<precisiontype> Svec_old;// { dense_vector<precisiontype>(matrix_dim, 0.0) };
dense_vector<precisiontype> Svec_previous;// { dense_vector<precisiontype>(matrix_dim, 0.0) };
dense_vector<precisiontype> dSvec;// { dense_vector<precisiontype>(matrix_dim, 0.0) };
dense_vector<precisiontype> dSvec_old;// { dense_vector<precisiontype>(matrix_dim, 0.0) };
dense2D<precisiontype> Jmat;// { dense2D<precisiontype>(matrix_dim, matrix_dim) };
dense_vector<precisiontype> xvec;// { dense_vector<precisiontype>(matrix_dim, 0.0) };
}
freepower::PFSolver::PFSolver() {

    setup();
}
freepower::PFSolver::PFSolver(Grid* g) {
    grid = g;
    setup();
}

freepower::PFSolver::PFSolver(Grid * g, precisiontype tol) {
    grid = g;
    tolerance = tol;
    setup();
}

freepower::PFSolver::PFSolver(Grid * g, string outfile) {
    grid = g;
    outfilename = outfile;
    setup();
}

freepower::PFSolver::PFSolver(Grid * g, string outfile,
    precisiontype tol, size_t maximum_iterations) {
    grid = g;
    tolerance = tol;
    outfilename = outfile;
    max_iterations = maximum_iterations;
    setup();
}

freepower::PFSolver::~PFSolver() {
}

int freepower::PFSolver::add_loads_back_to_S(dense_vector<
    precisiontype>& S_vector)
for (int indx{ 0 }; indx < grid->busses.size(); indx++)
{
    if (grid->busses[indx].type == 2 || grid->busses[indx].type
        == 3 || grid->busses[indx].vcontrolled)
    {
        S_vector[indx] += grid->busses[indx].loadP / grid->
            systembase;
        S_vector[indx + grid->busses.size()] += grid->busses[indx].
            loadQ / grid->systembase;
    }
}
return 0;

int freepower::PFSolver::check_generator_limits(dense_vector
    <precisiontype>& S_vector, dense_vector<precisiontype>&
    S_vector_previous, dense_vector<precisiontype>& x_vector)
{
    size_t N{ grid->busses.size() };
    int return_status{ 0 };
    for (int kndx{ 0 }; kndx < N; ++kndx)
    {
        //TODO: Adding back to the generator the power consumed by
        // the load
        //S_vector[ kndx] += grid->busses[kndx].loadP / grid->
            systembase;
        //S_vector[ kndx + N] += grid->busses[kndx].loadQ/ grid->
            systembase;
        
        if (grid->busses[kndx].vcontrolled && grid->busses[kndx].
            maxMvarorV != 0 && grid->busses[kndx].minMvarorV != 0)
        {
            // The Apparent Power vector must contain the power without
            // the load for computations.
            // Therefore the reactive power lower limit would be the
            // lower limit minus the load
and the reactive power upper limit would be the upper limit minus the load.

The loads are added back to the Apparent power vector after convergence and before display of the output.

```cpp
precisiontype Qmin{ (grid->busses[kndx].minMvarorV) / grid->systembase };
precisiontype Qmax{ (grid->busses[kndx].maxMvarorV) / grid->systembase };
//precisiontype Qmin{ (grid->busses[kndx].minMvarorV - grid->busses[kndx].loadQ) / grid->systembase };
//precisiontype Qmax{ (grid->busses[kndx].maxMvarorV - grid->busses[kndx].loadQ) / grid->systembase };
#if _DEBUG
//std::cout << endl << "Svec=" << S_vector << endl << "S_old =" << S_vector_previous << endl << endl;
std::cout << endl << "Working on Bus " << kndx << " and it’s Q=" << S_vector[kndx + N] << endl;
std::cout << "Current Sss for the bus is Sss=" << Sss[kndx] << "+j" << Sss[kndx + N] << endl;
#endif

if (grid->busses[kndx].type == 0) // If PQ type
{
#if _DEBUG
std::cout << "Bus " << kndx << " is currently a PQ Bus" << endl;
std::cout << "Currently from Svec_previous, the previous Qgen=" << Svec_previous[kndx + N] << " and the new Q=" << S_vector[kndx + N] << " and Real Power P=" << S_vector[kndx] << endl;
std::cout << "Sss=" << Sss[kndx] << "+j" << Sss[kndx + N] << endl;
std::cout << "Vi=" << x_vector[kndx + N] << " and Vset=" << grid->busses[kndx].Vset << endl;
precisiontype temp123{ Svec_previous[kndx + N] };
cout << "temp123=" << temp123 << endl;

```
cout << "*******ugggg" << "Svec_previous[kndx+N]=" << 
Svec_previous[kndx + N] << "and Qmax is " << Qmax << " 
division=" << Svec_previous[kndx + N] / Qmax << "*****" 
<< endl;
#endif

//Case 1 is a PQ Bus in previous Iteration Q was fixed to 
lower limit
//if (S_vector_previous[kndx + N] == Qmin - grid->busses[ 
kndx].loadQ / grid->systembase) //Was fixed at lower 
limit for previous iteration
if (Svec_previous[kndx + N] <= Qmin) //Was fixed at lower 
limit for previous iteration
  //if (low_limit_set) //Was fixed at lower limit for previous 
iteration
{
  #if _DEBUG
  std::cout << "=== Was MIN for Previouse Iteration ===" 
  << endl;
  #endif
  if (x_vector[kndx + N] >= grid->busses[kndx].Vset)
  {
    #if _DEBUG
    std::cout << "Vi >= Vset so remaining PQ & setting Qi=Qmin" 
    << endl;
    #endif
    S_vector[kndx + N] = Qmin - grid->busses[kndx].loadQ / grid 
    ->systembase;
    //low_limit_set = true;
    return_status = 1;
  }
  else
  {
    #if _DEBUG
    std::cout << "Vi < Vset so checking tempQ against Min and 
    Max" << endl;
    #endif
  }
}

76
if (S_vector[kndx + N] >= Qmax)
{
    S_vector[kndx + N] = Qmax - grid->busses[kndx].loadQ / grid->systembase;
    //high_limit_set = true;
    #if _DEBUG
        std::cout << "And leaving as a PQ Bus" << endl;
        std::cout << "Now S=" << S_vector[kndx + N] << endl;
    #endif
    return_status = 1;
}
else if (S_vector[kndx + N] <= Qmin)
{
    S_vector[kndx + N] = Qmin - grid->busses[kndx].loadQ / grid->systembase;
    //low_limit_set = true;
    #if _DEBUG
        std::cout << "And leaving as a PQ Bus" << endl;
        std::cout << "Now S=" << S_vector[kndx + N] << endl;
    #endif
    return_status = 1;
}
else if ((S_vector[kndx + N] > Qmin) && (S_vector[kndx + N] < Qmax))
{
    #if _DEBUG
        std::cout << "At Bus " << kndx << " between the limits of " << grid->busses[kndx].minMvarorV << " and " << grid->busses[kndx].maxMvarorV << ", so changing back to a type 2 PV bus!" << endl;
    #endif
    grid->busses[kndx].type = 2;
    grid->pvs.insert(kndx);
    grid->busses[kndx].pqtopv_count++; // May want to limit the number of times switch from pq to pv
//compute_V(S_vector, x_vector);//This one should work;
x_vector[kndx + N] = grid->busses[kndx].Vset; //Set to Vi=
   Vset
//high_limit_set = false;
//low_limit_set = false;
return_status = 1;
}
}
}//End of Case 1

//Case 2 - Still in is a PQ Bus in previous Iteration at Q
was fixed to upper limit
//if (S_vector_previous[kndx + N] == Qmax- grid->busses[kndx ]
   .loadQ / grid->systembase) //Was fixed at upper limit
   for previous iteration
if (Svec_previous[kndx + N] >= Qmax) //Was fixed at upper
   limit for previous iteration
   //if (high_limit_set) //Was fixed at upper limit for
   previous iteration
{
#if _DEBUG
std::cout << "=== Was MAX for Previous Iteration ===" <<
   endl;
#endif

if (x_vector[kndx + N] <= grid->busses[kndx].Vset)
{
#if _DEBUG
std::cout << "Vi <= Vset so remaining PQ" << endl;
#endif
S_vector[kndx + N] = Qmax - grid->busses[kndx].loadQ / grid
   ->systembase - tolerance;
std::cout << "Just set Q=Qmax!!! so Q=" << S_vector[kndx +
   N] << endl;
return_status = 1;
}
else if (x_vector[kndx + N] > grid->busses[kndx].Vset)
{

#if _DEBUG
std::cout << "Vi > Vset so checking tempQ against Min and Max" << std::endl;
#endif

if (S_vector[kndx + N] >= Qmax)
{
    S_vector[kndx + N] = Qmax - grid->busses[kndx].loadQ / grid->systembase;
#if _DEBUG
std::cout << "Leaving as a PQ Bus" << std::endl;
std::cout << "Just set Q to Qmax-load which is " << Qmax << "-" << grid->busses[kndx].loadQ / grid->systembase << std::endl;
std::cout << "Now Q[" << kndx << "]=" << S_vector[kndx + N] << std::endl;
#endif
    return_status = 1;
}
else if (S_vector[kndx + N] <= Qmin)
{
    S_vector[kndx + N] = Qmin - grid->busses[kndx].loadQ / grid->systembase;
#if _DEBUG
std::cout << "Leaving as a PQ Bus" << std::endl;
std::cout << "Just set Q to Qmin-load which is " << Qmax << "-" << grid->busses[kndx].loadQ / grid->systembase << std::endl;
std::cout << "Now Q[" << kndx << "]=" << S_vector[kndx + N] << std::endl;
#endif
    return_status = 1;
}
else if ((S_vector[kndx + N] > Qmin) && (S_vector[kndx + N] < Qmax))
{
#if _DEBUG
std::cout << "At Bus " << kndx << " tempQ is " << " between the limits of " << grid->busses[kndx].minMvarorV << " and 

79
" << grid->busses[kndx].maxMvarorV << ", so changing back to a type 2 PV bus!" << endl;
#endif
grid->busses[kndx].type = 2;
grid->pvs.insert(kndx);
grid->busses[kndx].pqtopv_count++; // May want to limit the number of times switch from pq to pv

//compute_V(S_vector, x_vector);//This one should work;
x_vector[kndx + N] = grid->busses[kndx].Vset; //Set to Vi=Vset
return_status = 1;
}
}
} //End of Case 2

//Case 3 if was PV bus in previous iteration
else if (grid->busses[kndx].type == 2)
{
#if _DEBUG
std::cout << "Bus " << kndx << " is currently a PV Bus" << endl;
#endif
if (S_vector[kndx + N] <= Qmin) //Min limit exceeded
{
#if _DEBUG
std::cout << "Bus " << kndx << ", s Q=" << S_vector[kndx + N] << " exceeding its lower limit of " << grid->busses[kndx].minMvarorV << endl;
#endif
grid->busses[kndx].type = 0;
std::set<int>::iterator it;
it = find(grid->pvs.begin(), grid->pvs.end(), kndx);
if (it != grid->pvs.end())
{
#if _DEBUG

std::cout << "Removing the bus from the list of PVs" << endl;
#endif
grid->pvs.erase(it);
}
grid->busses[kndx].pvtopq_count++; //May want to limit the number of times switch from pv to pq
S_vector[kndx + N] = Qmin; //Set the Reactive Power to Qmin
//TODO: Make Sss a parameter of the function
//The Real Power gets set properly in the ComputeSS function
Sss[kndx + N] = Qmin - grid->busses[kndx].loadQ / grid->systembase;
//xvec[kndx] -= .1;
//xvec[kndx + N] -= .1;
return_status = 1;
#if _DEBUG
std::cout << "Setting it to a PQ type also just set S_vector [ " << kndx + N << "] to " << S_vector[kndx + N] * grid->systembase << endl << endl;
//grid->generator_exceeding_limits.insert(kndx);
#endif
}
else if (S_vector[kndx + N] >= Qmax) //Max limit exceeded
{
//TODO: Remove this check
precisiontype temp = S_vector[kndx + N];
#if _DEBUG
std::cout << "Bus " << kndx << " is currently " << S_vector[kndx + N] * grid->systembase << " exceeding its upper limit of " << grid->busses[kndx].maxMvarorV << endl;
#endif
grid->busses[kndx].type = 0;
std::set<int>::iterator it;
it = find(grid->pvs.begin(), grid->pvs.end(), kndx);
if (it != grid->pvs.end())
{

#if _DEBUG
std::cout << "Removing the bus from the list of PVs" << endl
;
#endif
grid->pvs.erase(it);
}
grid->busses[kndx].pvtopq_count++; //May want to limit the
    number of times switch from pv to pq
S_vector[kndx + N] = Qmax; //Set the
    Reactive Power to Qmin
//TODO: Make Sss a parameter of the function
Sss[kndx + N] = Qmax - grid->busses[kndx].loadQ / grid->
    systembase;
//S_vector[kndx] = grid->busses[kndx].GenP - grid->busses[
    kndx].loadQ / grid->systembase;
//xvec[kndx] -= .15;
//xvec[kndx + N] = grid->busses[kndx].Vset;

//xvec[kndx] = 0;
//xvec[kndx + N] = 1;
//no not here compute_V(
    S_vector, x_vector);
return_status = 1;
#endif

#else
{
#if _DEBUG
std::cout << "Setting it to a PQ type; also S_vector[" <<
    kndx + N << "] is now " << S_vector[kndx + N] * grid->
    systembase << endl << endl;
#endif
}
else
{
#if _DEBUG
std::cout << "Bus " << kndx
<< " is within its limits of " << grid->busses[kndx].
    minMvarorV << " and " << grid->busses[kndx].maxMvarorV <<
    endl;
#endif

82
int freepower::PFSolver::compute_Jacobian(dense2D<
precisiontype>& Jacobian_matrix, const dense_vector<
precisiontype>& x_vector) {
Jacobian_matrix = 0.0;
//std::cout << endl << endl << endl;
//std::cout << "To build the Full Jacobian we have:" << endl
<< endl;
//std::cout << "x_vector=" << x_vector << endl << endl;
//std::cout << "xvecsmall=" << xvecsmall << endl << endl;
size_t Nbusses = size(x_vector) / 2;
//Build the Jacobian
for (size_t row{ 0 }; row < Nbusses; ++row)
{
for (size_t col{ 0 }; col < Nbusses; ++col)
{
if (row == col)
{
for (size_t n{ 0 }; n < Nbusses; n++)
{
if (n != row)
{
Jacobian_matrix(row, row) -= x_vector[Nbusses + row] * abs((
    grid->Ybus)(row, n))*x_vector[Nbusses + n] * sin(x_vector
    //Jacobian_matrix1
Jacobian_matrix(row + Nbusses, row) += x_vector[Nbusses +
    row] * abs((grid->Ybus)(row, n))*x_vector[Nbusses + n] *
    cos(x_vector[row] - x_vector[n] - arg((grid->Ybus)(row, n
    ))); //Jacobian_matrix3
}
    //Jacobian_matrix2
    //Jacobian_matrix4

else
{
}


```cpp
int freepower::PFSolver::shrink_Jacobian(const dense2D<precisiontype>& Jacobian_matrix, dense2D<precisiontype>& smallJmat) {
    int Nbusses = grid->busses.size();  
    //Build Jmatsmall  
    auto rowskipped = 0;  
    //Small Jmat1  
    for (int row{ Nbusses - 1 }; row > -1; --row)  
        if (grid->busses[row].type == 3)  
            {  
                rowskipped++;  
            }
        else  
            {  
                auto colskipped = 0;  
                for (int col{ Nbusses - 1 }; col > -1; --col)  
                    if (grid->busses[col].type == 3)  
                        {  
                            colskipped++;  
                        }
                else  
                    {  
                        smallJmat(row - grid->slacks.size() + rowskipped, col - grid
                                ->slacks.size() + colskipped) = Jacobian_matrix(row, col)  
                                ;  
                    }
            }
    } //Small Jmat1 end
```
rowskipped = 0;
//Small Jmat2
for (int row{ Nbusses - 1 }; row > -1; --row)
{
if (grid->busses[row].type == 3)
{
rowskipped++;
}
else
{
int colskipped = 0;
for (int col{ Nbusses - 1 }; col > -1; --col)
{
if (grid->busses[col].type == 2 || grid->busses[col].type == 3)
{
colskipped++;
}
else
{
smallJmat(row - grid->slacks.size() + rowskipped, col - 2 *
grid->slacks.size() - grid->pvs.size() + Nbusses +
colskipped) = Jacobian_matrix(row, col + Nbusses);
}
}
}
} //Small Jmat2 end

rowskipped = 0;
//Small Jmat3
for (int row{ Nbusses - 1 }; row > -1; --row)
{
if (grid->busses[row].type == 2 || grid->busses[row].type == 3)
{
rowskipped++;
}
else

{  
  auto colskipped{ 0 };
  for (int col{ Nbusses - 1 }; col > -1; --col)  
    if (grid->busses[col].type == 3)  
      {  
        colskipped++;
      }  
    else  
      {  
        smallJmat(row - 2 * grid->slacks.size() - grid->pvs.size() + 
          Nbusses + rowskipped, col - grid->slacks.size() + 
          colskipped) = Jacobian_matrix(row + Nbusses, col);
      }
  }
}//end Jmat3

rowskipped = 0;  
//Small Jmat4  
for (int row = { Nbusses - 1 }; row > -1; --row)  
  {  
    if (grid->busses[row].type == 2 || grid->busses[row].type == 3)  
      {  
        rowskipped++;
      }  
    else  
      {  
        int colskipped = 0;
        for (int col = { Nbusses - 1 }; col > -1; --col)  
          {  
            if (grid->busses[col].type == 2 || grid->busses[col].type == 3)  
              {  
                colskipped++;
              }  
            else  
              {  


smallJmat(row - 2 * grid->slacks.size() - grid->pvs.size() +
    Nbusses + rowskipped, col - 2 * grid->slacks.size() -
    grid->pvs.size() + Nbusses + colskipped) =
Jacobian_matrix(row + Nbusses, col + Nbusses);
}
}

//Jmat4 end
//std::cout << Jmatsmall << endl;
return 0;
}

int freepower::PFSolver::compute_Q(dense_vector<
    precisiontype>& S_vector, dense_vector<precisiontype>&
    x_vector) {
    //std::cout << "S before compute_Q_V() is " << S << endl <<
    //endl;
    //precisiontype sumP = 0.0, sumQ = 0.0;
    const auto N = grid->busses.size();
    for (size_t kndx{ 0 }; kndx < N; ++kndx) //Take care of the
        PV busses
    {

        if (grid->busses[kndx].type == 2)
            //if (grid->busses[kndx].vcontrolled)
            //if (grid->busses[kndx].vcontrolled && grid->busses[kndx].
            //type == 2)
            {
                //S_vector[kndx] = grid->busses[kndx].GenP / grid->
                        systembase - grid->busses[kndx].loadP;// grid->systembase
                //The real power output is known
                S_vector[kndx] = 0.0;
                S_vector[kndx + N] = 0.0;
                //x_vector[kndx + N] = grid->busses[kndx].Vset;
                //std::cout << "In compute S Ybus is " << endl << Ybus <<
                        endl << endl;
                //std::cout << "In computeS xvec is " << xvec << endl <<
                        endl;

78
for (size_t nndx{ 0 }; nndx < N; ++nndx)
{
    // S_vector[kndx] += x_vector[N + kndx] * x_vector[N + nndx]
    * abs(grid->Ybus(kndx, nndx)) * cos(x_vector[kndx] -
    x_vector[nndx] - arg(grid->Ybus(kndx, nndx))); // sum for P
    // S_vector[kndx + N] -= x_vector[N + kndx] * x_vector[N +
    nndx] * abs(grid->Ybus(kndx, nndx)) * sin(-x_vector[kndx]
    + x_vector[nndx] + arg(grid->Ybus(kndx, nndx)));
    // sum for Q
    S_vector[kndx + N] += x_vector[N + kndx] * x_vector[N + nndx]
    * abs(grid->Ybus(kndx, nndx)) * sin(x_vector[kndx] -
    x_vector[nndx] - arg(grid->Ybus(kndx, nndx)));
    // sum for Q

    // S_vector[kndx] += x_vector[N + kndx] *
    x_vector[N + nndx] * abs(grid->Ybus(kndx, nndx)) * cos(
    x_vector[kndx] - x_vector[nndx] - arg(grid->Ybus(kndx,
    nndx)));
    // sum for P
    // S_vector[kndx + N] += x_vector[N + kndx] * x_vector[N + nndx]
    * abs(grid->Ybus(kndx, nndx)) * sin(x_vector[kndx] -
    x_vector[nndx] - arg(grid->Ybus(kndx, nndx)));
    // sum for Q
}

// S_vector[kndx] = grid->busses[kndx].GenP / grid->
// systembase - grid->busses[kndx].loadP / grid->
// systembase;
// S_vector[kndx] -= grid->busses[kndx].loadP / grid->
// systembase;
// S_vector[kndx + N] -= grid->busses[kndx].loadQ / grid->
// systembase;
S_vector[kndx] -= grid->busses[kndx].loadP / grid->
// systembase;
}
for (size_t kndx{ 0 }; kndx < N; ++kndx) //Take care of the slack bus
{
if (grid->busses[kndx].type == 3)
{
    ///For Slack bus compute the the real power
#if (grid->busses[kndx].type == 3)
{
    // S_vector[kndx + N] = 0.0;
    //for (size_t nndx{ 0 }; nndx < N; ++nndx)
    //{
    //    S_vector[kndx + N] += x_vector[N + kndx] * 
    //        x_vector[N + nndx] * abs(grid->Ybus(kndx, nndx)) * sin(
    //            x_vector[kndx] - x_vector[nndx] - arg(grid->Ybus(kndx,
    //                nndx))); //sum for Q
    //}
    //}
    
    S_vector[kndx] = 0.0;
    S_vector[kndx + N] = 0.0;
    for (size_t nndx{ 0 }; nndx < N; ++nndx)
    {
        S_vector[kndx] += x_vector[N + kndx] * x_vector[N + nndx] * 
            abs(grid->Ybus(kndx, nndx)) * cos(x_vector[kndx] - 
            x_vector[nndx] - arg(grid->Ybus(kndx, nndx))); //sum for P
        S_vector[kndx + N] += x_vector[N + kndx] * x_vector[N + nndx] * 
            abs(grid->Ybus(kndx, nndx)) * sin(x_vector[kndx] - 
            x_vector[nndx] - arg(grid->Ybus(kndx, nndx))); //sum for Q
    }
}
}
/for (size_t kndx{ 0 }; kndx < N; ++kndx)
{ 
    if (grid->busses[kndx].type == 3)
    { 
        S_vector[kndx] = 0.0;
        S_vector[kndx + N] = 0.0;
        for (size_t nndx{ 0 }; nndx < N; ++nndx)
        { 
            if (grid->busses[nndx].type == 2)
            { 
            }
        }
    }
}

return 0;

int freepower::PFSolver::compute_V(dense_vector<
    precisiontype>& S_vector, dense_vector<
    precisiontype>& x_vector)
{ 
    std::cout << "S before compute_Q_V() is " << S << std::endl;
    precisiontype sumP = 0.0, sumQ = 0.0;
BEFORE V is computed the load must be added back into the reactive power right???

class Grid {
public:
    // constructor...

    void computeS()
    {
        size_t N = busses.size();
        for (size_t kndx = 0; kndx < N; ++kndx)
        {
            precisiontype Qmin = (busses[kndx].minMvarV) / systembase;
            precisiontype Qmax = (busses[kndx].maxMvarV) / systembase;

            if (busses[kndx].type == 2 && (S_vector[kndx + N] == Qmin || S_vector[kndx + N] == Qmax))
                if (busses[kndx].vcontrolled && (S_vector[kndx + N] == Qmin || S_vector[kndx + N] == Qmax))
                {
                    std::cout << "In compute S Ybus is " << endl << Ybus << endl;
                    std::cout << "In compute S xvec is " << xvec << endl;
                }

                precisiontype nodecurrent = 0.0;
                for (size_t nndx = 0; nndx < N; ++nndx)
                {
                    nodecurrent += x_vector[N + nndx] * abs(Ybus(kndx, nndx)) * sin(x_vector[kndx] - x_vector[nndx] - arg(Ybus(kndx, nndx)));
                }

                precisiontype tempV = x_vector[kndx + N], tempS = S_vector[kndx + N] + busses[kndx].loadQ / systembase;
                // Just to see in the debugger
                x_vector[kndx + N] = (S_vector[kndx + N]) / nodecurrent;
                //x_vector[kndx + N] = (S_vector[kndx + N] + grid->busses[kndx].loadQ / grid->systembase) / nodecurrent;

                // The Q is equal to the following.
                // S_vector[kndx + N] += x_vector[N + kndx] * x_vector[N + nndx] * abs(Ybus(kndx, nndx)) * sin(x_vector[kndx]

                //...
- x_vector[nndx] - arg(grid->Ybus(kndx, ndx))); //sum
  for Q

//if (abs(S_vector[kndx]) < (1.0E-6))
//  S_vector[kndx] = 0.0;
//if (abs(S_vector[N + kndx]) < (1.0E-6))
//  S_vector[N + kndx] = 0.0;

//std::cout << "S is now " << S << endl << endl;

}
return 0;
}

int freepower::PFSolver::compute_S(dense_vector<
precisiontype>& S_vector, const dense_vector<
precisiontype>& x_vector) {

//std::cout << "S before computeS() is " << S << endl << endl;
//precisiontype sumP = 0.0, sumQ = 0.0;
const auto N = grid->busses.size();
for (size_t kndx{ 0 }; kndx < N; ++kndx)
  {
    S_vector[kndx] = 0.0;
    S_vector[kndx + N] = 0.0;
    for (size_t ndx{ 0 }; ndx < N; ++ndx)
      {
        S_vector[kndx] += x_vector[N + kndx] * x_vector[N + ndx] * 
                        abs(grid->Ybus(kndx, ndx)) *cos(x_vector[kndx] - 
                        x_vector[ndx] - arg(grid->Ybus(kndx, ndx))); //sum for P
        S_vector[kndx + N] += x_vector[N + kndx] * x_vector[N + ndx ] * abs(grid->Ybus(kndx, ndx)) * sin(x_vector[kndx] - 
                        x_vector[ndx] - arg(grid->Ybus(kndx, ndx))); //sum for Q
int freepower::PFSolver::compute_Sss(dense_vector<decltype(precisiontype)>& S_steady_state) {

for (size_t i{0}; i < grid->busses.size(); ++i) {
    switch (grid->busses[i].type) {
    case(3): // Slack
        S_steady_state[i] = (grid->busses[i].GenP - grid->busses[i].loadP) / grid->systembase;
        //S_steady_state[i] = (-grid->busses[i].loadP) / grid->systembase;
        S_steady_state[i + grid->busses.size()] = (grid->busses[i].GenQ - grid->busses[i].loadQ) / grid->systembase;
        //S_steady_state[i + grid->busses.size()] = (-grid->busses[i].loadQ) / grid->systembase;
        break;
    case(2): // PV
        S_steady_state[i] = (grid->busses[i].GenP - grid->busses[i].loadP) / grid->systembase;
        //S_steady_state[i + grid->busses.size()] = (grid->busses[i].GenQ - grid->busses[i].loadQ) / grid->systembase;
        S_steady_state[i + grid->busses.size()] = 0;
        break;
    case(1): // TODO Build Initial S (Sss) for PQ busses - What is it when Generator not at VAR limit?
        S_steady_state[i] = (grid->busses[i].GenP - grid->busses[i].loadP) / grid->systembase;
    }
// S_steady_state[i + grid->busses.size()] = (grid->busses[i].GenQ - grid->busses[i].loadQ) / grid->systembase;
S_steady_state[i + grid->busses.size()] = 0;
break;
case(0): // Load
    S_steady_state[i] = -grid->busses[i].loadP / grid->systembase;
    S_steady_state[i + grid->busses.size()] = -grid->busses[i].loadQ / grid->systembase;
    break;
}
}
return 0;

int freepower::PFSolver::initpvs(set<int>& pv_bus_set)
{
    for (int i{ 0 }; i < grid->busses.size(); i++)
    {
        if (grid->busses[i].type == 2)
            pv_bus_set.insert(i);
        //pvs.push_back(i);
    }
    return 0;
}

int freepower::PFSolver::initslacks(set<int>& slack_bus_set)
{
    for (int i{ 0 }; i < grid->busses.size(); i++)
    {
        if (grid->busses[i].type == 3)
            slack_bus_set.insert(i);
        //slacks.push_back(i);
    }
    return 0;
}
int freepower::PFSolver::init_x(dense_vector<precisiontype>& x_vector, const size_t iter) {

    // Set the initial guess (x) vector in the Ax=b equation
    if (iter == 0) {
        for (size_t i{ 0 }; i < grid->busses.size(); i++) {
            // There should already be zeros in entire vector. Nbusses need to rewrite them, i.e. all elements in xvec is 0 so no need to init angles..
            // TODO Nbusses need to implement this with a switch
            // xvec[i] = 0;
            #if _FULLDEBUG
                std::cout << "putting theta into index " << i << " so xvec[" << i << "]=" << xvec[i] << endl;
            #endif
            x_vector[i] = 0;
        }
        if (grid->busses[i].type == 2 || grid->busses[i].type == 3) {
            if (grid->busses[i].type == 3) {
                // x_vector[i] = grid->busses[i].angle;
                x_vector[i] = grid->busses[i].angle;
                x_vector[grid->busses.size() + i] = grid->busses[i].Vset; // Voltages begin at index Bus.size()
            }
        } else {
            x_vector[grid->busses.size() + i] = 1.0;
            #if _FULLDEBUG
                std::cout << "putting theta into index " << busses.size() + i << " so xvec[" << busses.size() + i << "]=" << xvec[ busses.size() + i ] << endl;
            #endif
        }
    }
}
#endif
}
#endif

//std::cout << "Initial xvec guess is " << xvec << endl <<
   endl;
#endif
}
//TODO: Can never get here so this should be removed.
else
for (size_t i{ 0 }; i < grid->busses.size(); i++)
{
    x_vector[i] = grid->busses[i].angle;
    x_vector[i + grid->busses.size()] = grid->busses[i].voltage;
}
    // initxFromsmallx(xvec, xvecsmall, busses, pvs, slacks)
;
#if _FULLDEBUG
std::cout << "Done, xvec=" << xvec << endl;
#endif
return 0;
}

int freepower::PFSolver::init_x_from_small_x(dense_vector<
        precisiontype>& x_vector, const dense_vector<
        precisiontype>&smallvec) {

auto N = grid->busses.size();
int skipped = 0;
for (int i = grid->busses.size() - 1; i > -1; --i)
{
    if (grid->busses[i].type == 2 || grid->busses[i].type == 3)
    {
        if (grid->busses[i].type == 3)
        {
            //No need to do anything
            //x_vector[i] = grid->busses[i].angle;
            //x_vector[i + N] = grid->busses[i].Vset;

            // keep this for now
            if (grid->busses[i].type == 3)
            {
                //To check that no need to do anything
                //x_vector[i] = grid->busses[i].angle;
                //x_vector[i + N] = grid->busses[i].Vset;
            }

        }
    }
    else
    {
        // handle other cases
        //...
}  // else
//   xvec[N + i] = busses[i].Vset;
skipped++;
}
else
{
// restore the voltages xvecsmall->xvec
x_vector[i + N] = smallvec[N - 2 * grid->slacks.size() + i -
   grid->pvs.size() + skipped];
}
}
skipped = 0;
for (int i = grid->busses.size() - 1; i > -1; --i)
{
if (grid->busses[i].type == 3)
{
    // No need to do anything
    // x_vector[i] = grid->busses[i].angle;
    skipped++;
}
else
   x_vector[i] = smallvec[i - grid->slacks.size() + skipped];
}
// std::cout << "Done, xvec=" << xvec << endl;
return 0;

}

int freepower::PFSolver::NRSolve()
{
ofstream outfile;
outfile.open(outfilename);
outfile << endl << "Power Flow Simulation of " << grid->
   filename << endl << endl;
outfile.precision(5);
/** cout << endl << endl << "NRSolve(): xvec=" << xvec << endl 

precisiontype mismatch_error;

// The Steady State Vector should never change. So it’s outside the loop.
// dense_vector<precisiontype> Sss = dense_vector<
  precisiontype>(matrix_dim, 0.0);
// compute_Sss(Sss);
// init_x(xvec, 0);
// Svec_previous = Svec;

for (size_t iteration{ 0 }; iteration < max_iterations; ++
   iteration)
{
  if _DEBUG
    outfile << " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " " 

}
//if (check_generator_limits(Svec, Svec_old, xvec))
  // compute_Q(Svec, xvec);

//check_generator_limits(Svec, Svec_old, xvec);
  // compute_V(Svec, xvec);
  // cout << "Just computed voltages" << endl;
  // //NRSolve();
  //
  //}

//if (iteration%2==0)
  /////if (iteration > 0 && iteration % 5 == 0)
  /////if (iteration > 2 )
  //{
    ///// // compute_Q(Svec, xvec);
    //check_generator_limits(Svec, Svec_old, xvec);
    // //compute_V(Svec, xvec);
    //}
  //compute_Q_V(Svec, xvec);
  //add_loads_back_to_S(Svec);
  //cout << "Sss=" << Sss << endl
  //  << "Svec=" << Svec << endl;

//if (iteration)
  // check_generator_limits(Svec, Svec_old, xvec);

//Since generator limits may change the bus types,
//size of samll matrix and vectors know after checking
// generator limits
size_t small_matrix_dim = 2 * (grid->busses.size() - slacks.size()) - grid->pvs.size(); //grid->pvs hold the PV grid ->busses and the Slack bus, Nbusseseed to add the Slack bus back in for the small matrix dim.

//save_S();

#if _FULLDEBUG
  //std::cout <<"**************"<<endl<< "Jmatust after computS" << endl<<endl<<endl;

100
//std::cout << S_vector[1] * grid->systembase << "one " <<
    S_vector[31] * grid->systembase;
#endif
//TODO: Remove the next two comments
//While computing S the PV busses are removed or added as
//necessary so we can determine the size of the small
//vectors
//.... small_matrix_dim = 2 * (grid->busses.size() -
    grid->slacks.size()) - grid->pvs.size(); //added
dense_vector<precisiontype> xvecsmall{ dense_vector<
    precisiontype>(small_matrix_dim, 0.0) };//added
shrink_vec(xvec, xvecsmall);

#if _DEBUG
outfile << "Initial guess:" << endl
    "xvec=" << xvec << endl
    "xvecsmall=" << xvecsmall << endl << endl;
#endif
dSvec_old = dSvec;
dSvec = Sss - Svec;
#if _DEBUG
outfile << "Sss=" << Sss << endl
    "Svec=" << Svec << endl
    "dSvec=Sss-Svec=" << dSvec << endl << endl
    "The two-Norm<15>(dSvec)=" << two_norm(dSvec) << endl;
#endif
dense_vector<precisiontype> dSvecsmall{ dense_vector<
    precisiontype>(small_matrix_dim, 0.0) };//added
shrink_vec(dSvec, dSvecsmall);

#if _FULLDEBUG
out << "After shrinkVec() on dSsmall" << endl;
out << "####### before computeJacobian #### " << endl << "dS=
    Sss-Sx= " << dS << endl;
out << "dSsmall=" << newdSmall << endl << endl;
101
//mismatch_error = abs(infinity_norm(Svec_old - Svec));
mismatch_error = abs(one_norm(Svec_old - Svec));
//mismatch_error = abs(two_norm(Svec_old-Svec));
//cout << "mismatch_error=" << mismatch_error << endl << endl;

#if _DEBUG
outfile << " !!!!!!!!!! <<< Mismatch Error is " << mismatch_error << " and Tolerance=" << tolerance << " >>> !!!!!!!! " << endl << endl;
#endif

//////// End of Step 1

////////// Check for convergence
ios::fmtflags oldFormatSettings = outfile.flags();

if (mismatch_error < tolerance)
{

    //compute_Q(Svec, xvec);
    //compute_V(Svec, xvec);
    //check_generator_limits(Svec, Svec_old, xvec);
    //compute_V(Svec, xvec);

#if _DEBUG
std::cout << "!!!!!!!!!!!!!!!!!!!!!! CONVERGENCE !!!!!!!!!!!!!!!!!!!" << endl << endl;
#endif

//Svec=Svec_old;
streamsize oldPresicion = outfile.precision();
outfile.setf(ios::fixed, ios::floatfield);

#if _DEBUG
outfile << "Final xvec is " << xvec << endl;
#endif
outfile << "After " << iteration + 1 << " iterations the 
    final tolerance is " << setprecision(20) << 
    mismatch_error << setprecision(5) << endl;
//TODO: Add the loads back in!!
add_loads_back_to_S(Svec);
/////per unit output
//outputpu(Svec, xvec, outfile);
Svec *= grid->systembase;
//systembase output
output(Svec, xvec, outfile);
outfile << endl << "Input file name: " << grid->filename << 
    endl << endl;
outfile.close();
//TODO: put back in pu
Svec /= grid->systembase;
Svec_previous = Svec;
return 0;
} 

/////////// End of Check for convergence

dense2D<precisiontype> Jmatsmall{ dense2D<precisiontype>(
    small_matrix_dim, small_matrix_dim) }; //added

/////////// Step 2 - Compute Jacobian
if (iteration % 1 == 0)
{
    compute_Jacobian(Jmat, xvec);
    shrink_Jacobian(Jmat, Jmatsmall);//added
    //compute_Jacobian_reduced();
}

dense_vector<precisiontype> dxvecssmall = dense_vector<
    precisiontype>(small_matrix_dim, 0.0); //added
#if _DEBUG
outfile << "dxvecssmall= " << dxvecssmall << endl;
#endif
dense_vector<precisiontype> mydxvec =
dense_vector<precisiontype>(matrix_dim, 0.0);
dxvecsmall = lu_solve(Jmatsmall, dSvecsmall); //added new

#if _DEBUG
outfile << "After lu_solve" << endl << endl;
outfile << "dxvecsmall " << dxvecsmall << endl;
#endif

///// End of Step 3 /////////////////

///// Step 4 Compute x(i+1) /////////////

xvecsmall += dxvecsmall;//added new
init_x_from_small_x(xvec, xvecsmall);

//initx(x0, testgrid->busses, iteration);
#if _DEBUG
outfile << "So now after Second initx" << endl << "xvec= " <<
vec << endl
<< "xvecsmall= " << xvecsmall << endl << endl;
#endif

///// End of Step 4 Compute x(i+1) /////////////

#if _DEBUG
outfile << "Before saving xvec= " << xvec;
#endif

// check_generator_limits();
//if(iteration) 
//check_generator_limits(Svec, Svec_old, xvec);
///compute_Q(Svec, xvec);
//check_generator_limits(Svec, Svec_old, xvec);
///compute_V(Svec, xvec);
///if (iteration>1)
///{
// compute_Q(Svec, xvec);

///// check_generator_limits(Svec, Svec_old, xvec);
///// //compute_Q(Svec, xvec);
    //compute_V(Svec, xvec);
//}
//compute_V(Svec, xvec);

#if _DEBUG
    outfile << " **************************** END of Run # " <<
         iteration + 1 << "  
  ****************************" << endl << endl
     << endl;
#endif
}
outfile.close();
return 0;
}

int freepower::PFSolver::NRSolve(string outfile)
{
    return 0;
}

int freepower::PFSolver::NRSolve(string infile, string
    outfile)
{
    return 0;
}

int freepower::PFSolver::outputpu(const dense_vector<precisiontype>& S_vector, const dense_vector<precisiontype>& x_vector, ofstream &outfile)
{

outfile << endl <<
"-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-}_
" " << endl <<
"-----------------------------------------------------------
------------------"
"Bus Voltage pu Angle rad GenP pu GenQ pu loadP pu
loadQ pu" << endl <<
"-----------------------------------------------------------
------------------" << endl;

for (size_t n = 0; n < grid->busses.size(); ++n)
{
    precisiontype Pgen{ 0 };
    precisiontype Qgen{ 0 };

    //Add the load back into the Generated Power
    if (grid->busses[n].type == 2 || grid->busses[n].type == 3
        || grid->busses[n].vcontrolled)
    {
        //Pgen = S_vector[ n ];
        //Qgen = S_vector[ n + grid->busses.size() ];
        Pgen = S_vector[n];// +grid->busses[n].loadP / grid->
systembase;
        Qgen = S_vector[n + grid->busses.size() ];// +grid->busses[n
            ].loadQ / grid->systembase;
    }

    //TODO: Fix this limit check for the Per Unit output
    string limitreached = " ";
    if (Qgen <= grid->busses[n].minMvarorV / grid->systembase &&
        grid->busses[n].minMvarorV < 0 || Qgen >= grid->busses[n]
        .maxMvarorV / grid->systembase && grid->busses[n].
        maxMvarorV>0)
        limitreached = "*";

   (outfile << setw(2) << grid->busses[n].number << setw(12)
        << grid->busses[n].voltage << setw(13) << grid->busses[n]
        .angle << setw(12) << Pgen << setw(12) << Qgen << setw
        (12) << grid->busses[n].loadP / grid->systembase << setw
        (12) << grid->busses[n].loadQ / grid->systembase << " ");

106
outfile << endl;
outfile << setw(2) << grid->busses[n].number << setw(12) <<
x_vector[n + grid->busses.size()] << setw(13) << x_vector[n] << setw(12) << Pgen << setw(12) << Qgen << setw(12)
<< grid->busses[n].loadP / grid->systembase << setw(12)
<< grid->busses[n].loadQ / grid->systembase << " " <<
limitreached << endl;
}
return 0;
}

int freepower::PFSolver::output(const dense_vector<
    precisiontype>& S_vector, const dense_vector<
    precisiontype>& x_vector, ofstream &outfile)
{
    outfile << endl <<
    "------------------------------------------------------------------------

            " << endl <<
"Bus Voltage(V) Angle(deg) GenP(MW) GenQ(MVar) loadP(MW)
    loadQ(MVar)" << endl <<
"------------------------------------------------------------------------

            " << endl;

    for (size_t n = 0; n < grid->busses.size(); ++n)
    {

        precisiontype Pgen{ 0 };
        precisiontype Qgen{ 0 };

        //Add the load back into the Generated Power

        if (grid->busses[n].type == 2 || grid->busses[n].type == 3
            || grid->busses[n].vcontrolled)
        {
            //Pgen = S_vector[ n];
            //Qgen = S_vector[ n + grid->busses.size()];
        }
Pgen = S_vector[n]; // +grid->busses[n].loadP;
Qgen = S_vector[n + grid->busses.size()]; // + grid->busses[n]
}

string limitreached = " ";
if (Qgen <= grid->busses[n].minMvarorV && grid->busses[n].
  minMvarorV < 0 || Qgen >= grid->busses[n].maxMvarorV &&
  grid->busses[n].maxMvarorV>0)
limitreached = "*";

outfile << setw(2) << grid->busses[n].number << setw(12) <<
  x_vector[n + grid->busses.size()] << setw(13) << x_vector
  [n] * 180.0 / pi << setw(12) << Pgen << setw(12) << Qgen
  << setw(12) << grid->busses[n].loadP << setw(12) << grid
  ->busses[n].loadQ << " " << limitreached << endl;
}
return 0;
}

int freepower::PFSolver::save_S(dense_vector<precisiontype>&
  S_vector) {

for (size_t i{ 0 }; i < grid->busses.size(); i++)
{
  //grid->busses[i].GenP = S_vector[ i];
  //grid->busses[i].GenQ = S_vector[ grid->busses.size() + i];
  grid->busses[i].GenP = S_vector[i]; // + grid->busses[i].
    loadP / grid->systembase;
  grid->busses[i].GenQ = S_vector[i + grid->busses.size()]; //
    + grid->busses[i].loadQ / grid->systembase;
}
return 0;
}

int freepower::PFSolver::save_x(const dense_vector<
  precisiontype>& x_vector) {

for (size_t i{ 0 }; i < grid->busses.size(); i++)
{
    grid->busses[i].angle = x_vector[i];
    grid->busses[i].voltage = x_vector[grid->busses.size() + i];
}
return 0;

int freepower::PFSolver::setup()
{
    matrix_dim = 2 * grid->busses.size();
xvec = dense_vector<precisiontype>(matrix_dim, 0.0);
dSvec = dense_vector<precisiontype>(matrix_dim, 0.0);
dSvec_old = dense_vector<precisiontype>(matrix_dim, 0.0);
Svec = dense_vector<precisiontype>(matrix_dim, 0.0);
Svec_old = dense_vector<precisiontype>(matrix_dim, 0.0);
Sss = dense_vector<precisiontype>(matrix_dim, 0.0);
Jmat = dense2D<precisiontype>(matrix_dim, matrix_dim);

    //TODO: Moved initx to setup
    init_x(xvec, 0);
    compute_Sss(Sss);
    return 0;
}

int freepower::PFSolver::shrink_vec(dense_vector<precisiontype>&vec, dense_vector<precisiontype>&vecsmall)
{
    //Build xvecsmall
    //std::cout << "%%%%%%%%%%%%%%%%%%%% " << endl << "Before building xvecsmall:" << endl << "xvec=" << xvec << endl
    //    << "%%%%%%%%%%%%%%%%%%%%" << endl << "And xvecsmall is " << xvecsmall << endl << endl;
    int pvCount = 0, smallindex = 0;
auto anglesNumberOf = grid->busses.size() - grid->slacks.size();
auto N = grid->busses.size();
for (size_t i{ 0 }; i < grid->busses.size(); ++i)
{
if (grid->busses[i].type == 0)
{
vecsmall[smallindex] = vec[i];
vecsmall[smallindex + anglesNumberOf - pvCount] = vec[N + i];
smallindex++;
}
else if (grid->busses[i].type == 2)
{
vecsmall[smallindex] = vec[i];
if (grid->pvs.size() == 0)
{
vecsmall[smallindex + anglesNumberOf - pvCount] = vec[N + i];
}
if (grid->pvs.size() > 0)
{
pvCount++;
}
smallindex++;
}
else
{
//This should be bus type 1
}
return 0;

//////////////////////////////// End of PFsolver.cpp
REFERENCES


111


