**GridPACK™ Overview**

**Bruce Palmer, William Perkins, Kevin Glass, Yousu Chen, Shuangshuang Jin, David Callahan, Ruisheng Diao, Mark Rice, Stephen Elbert, Zhenyu Huang**

**Introduction**

The objective of the GridPACK™ toolkit project is to develop a framework to support the rapid development of power grid applications capable of running on high performance computing architectures (HPC) with high levels of performance and scalability. The toolkit will allow power system engineers to focus on developing working applications from their models without getting bogged down in the details of decomposing the computation across multiple processors, managing data transfers between processors, working out index transformations between power grid networks and the matrices generated by different power applications, and managing input and output. GridPACK™ is being designed to encapsulate as much of the book-keeping required to set up HPC applications as possible in high-level programming abstractions that allow developers to concentrate on the physics and mathematics of their problems.

This report will summarize the overall design of the GridPACK™ framework. The initial focus of the GridPACK™ design analysis was to target four power grid applications and to identify common features that span multiple applications as candidates for inclusion in a framework. This analysis included a breakdown of the application into phases and identification within each phase of the functionality required to complete them. The four applications originally targeted within this project were power flow simulations, contingency analysis, state estimation and dynamic simulation. The remainder of this document will describe the effort to obtain design requirements to determine what functionality the GridPACK™ framework would need to incorporate in order to support multiple power grid applications and the initial framework design that resulted from these requirements. The framework will continue to evolve as more real-world experience can be incorporated into the design process but many base classes that have already been identified that are capable of supporting a range of applications.

Four power grid applications were targeted for initial implementation within the GridPack framework. These consisted of

1. Powerflow simulations of the electric grid
2. Contingency analysis of the electric grid
3. State estimation based on electric grid measurements
4. Dynamic simulations of the electric grid

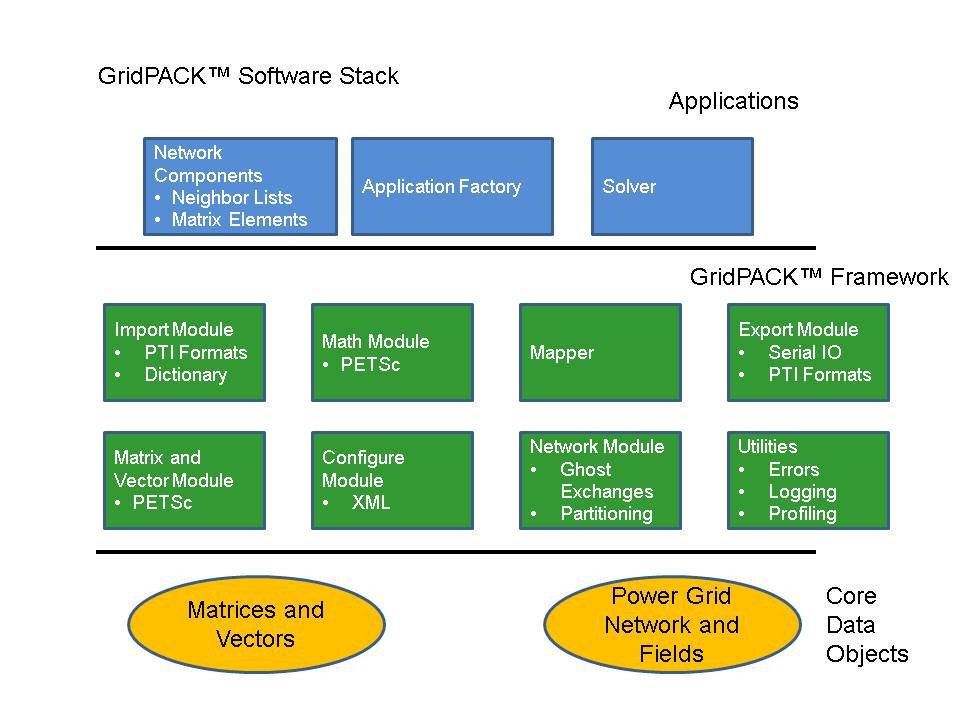
Based on these applications, several cross-cutting functionalities were identified that could be used to support multiple applications. These include modules to support

1. Network topology and behavior. The network topology is the starting point for any power grid analysis. The topology defines the initial network model and is the connection point between the physical problem definition in terms of buses and branches and the solution method, which is usually expressed in terms of matrices and vectors.
2. Network components and their properties (e.g. bus and branch models, measurements, etc.) grid components are the objects associated with the buses and branches of the power grid network. Along with the network topology itself, these define the physical system being modeled and in some cases the analysis that is to be performed. Bus and branch components can be differentiated into things like generators, loads, grounds, lines, transformers, measurements, etc. and depending on the how they are defined and the level of detail incorporated into them, they define different power grid systems and analyses. The behavior of buses and branches can depend on the properties of branches or buses that are directly attached to them, e.g. figuring out the contribution of a particular bus to the solution procedure may require that properties of the branches attached to that bus are made available to the bus. The necessity for exchanging this data is built into the framework. Furthermore, these data exchanges must also be accounted for in a parallel computing context, since the grid component from which data is required may be located on a different processor.
3. Linear algebra and solvers. Basic algebraic objects, such as distributed matrices and vectors, are a core part of the solution algorithms required by power grid analyses. Most solution algorithms are dominated by sparse matrices but a few, such as Kalman filter analyses, require dense matrices. Vectors are typically dense. There exists a rich set of libraries for constructing distributed matrices and vectors and these are coupled to preconditioner and solver libraries. GridPACK™ can leverage this work heavily by creating wrappers within the framework to create matrices and vectors that can be used in solution algorithms. Wrapping these libraries instead of using them directly will have the advantage that creating these algebraic objects can be simplified somewhat for power grid applications but more importantly, it will allow framework developers to investigate new solver and algebraic libraries seamlessly, without disrupting other parts of the code.
4. Mapping between network and algebraic objects. The physical properties of power grid systems are defined by networks and the properties of the network components but the equations describing the networks are algebraic in nature. The mappings between the physical networks and the algebraic equations depend on the indexing scheme used to describe the network and the number of parameters in the network components that appear in the equations. Constructing a map between network parameters and their corresponding locations in a matrix or vector can be complicated and error prone. Fortunately, much of this work can be automated and developers can focus much more on developing code to evaluate individual matrix elements without worrying about where to locate them in the matrix. This can considerably simplify coding.

**GridPACK™ Framework Components**

This section will describe the core components identified so far and the functionality they support. It will start off with two components that directly support the major underlying data objects, the power grid network and its associated network components and matrices and vectors. Additional components are then built on top of these (or at least in conjunction with them). These include partitioners to sort out the grid among the processors, grid components that describe the physics of the different network models or analyses, grid component factories that initialize the grid components, mappers that convert the current state of the grid components into matrices and vectors that are used in the solution algorithms, solvers that supply the preconditioner and solver functionality necessary to implement solution algorithms, input and output modules that allow developers to import and export data, and other utility modules that support standard code develop operations like timing, event logging, and error handling.

Many of these modules are constructed using libraries developed elsewhere so as to minimize framework development time. However, by wrapping these libraries in interfaces geared towards power grid applications these libraries can be made easier to use by power grid engineers. The interfaces also make it possible to exchange libraries in the future for new or improved implementations of specific functionality without requiring application developers to rewrite their codes. This can significantly reduce the cost of introducing new technology into the framework. The software layers in the GridPACK™ framework are shown schematically in Figure 1.



**Figure 1.** A schematic diagram of the GridPack framework software data stack. PETSc is an external library that supports distributed matrices and vectors and supplies extensive support for parallel algebraic operations and linear and non-linear solvers.

Core framework components are described below.

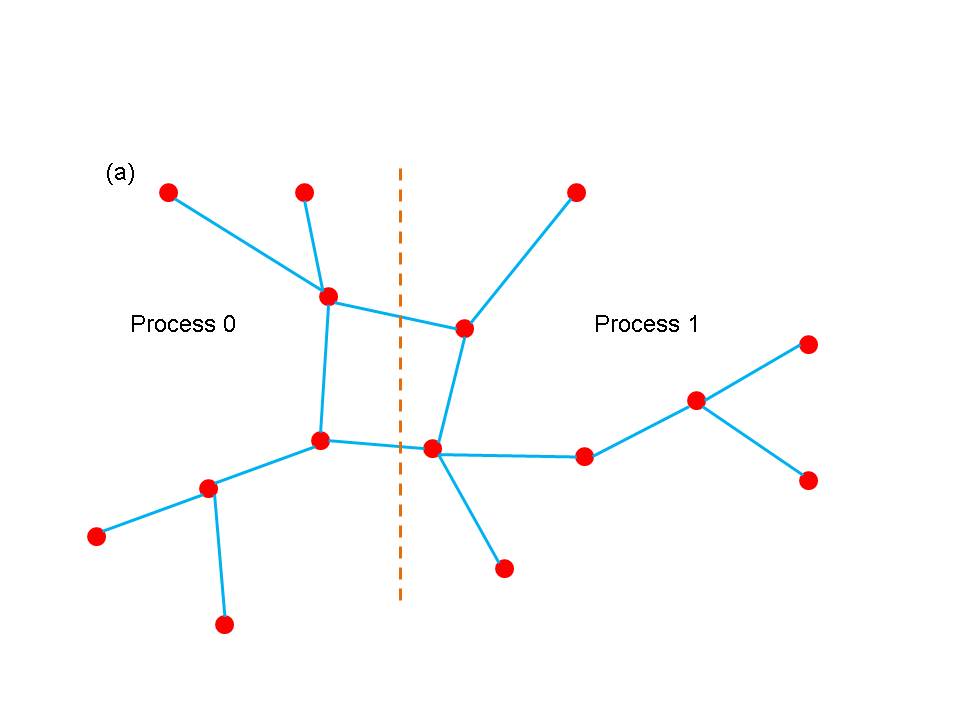
**Network Module:** the network module is designed as a container with four major functions

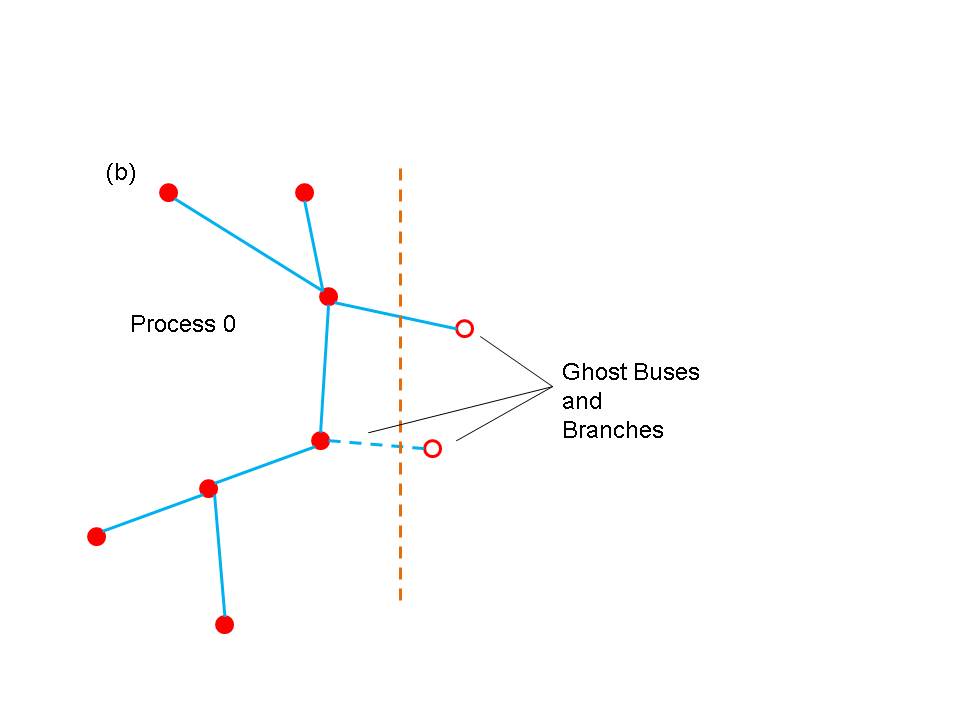
1. The network is a container for the network topology. The connectivity of the network is maintained by the network object and can be made available through requests to the network. The network also maintains the “ghost” status of locally held buses and branches and determines whether a bus or branch is owned by a particular processor or represents a ghost image of a bus or branch owned by a neighboring processor.
2. The network topology can then be decorated with bus and branch objects that reflect the properties of the particular physical system under investigation. These bus and branch objects are written by the application developer and reflect the particular physical system under investigation and the analyses that need to be performed on it. Different applications will use different bus and branch implementations.
3. The network module is responsible for implementing update operations that can be used to fill in the value of ghost cell fields with current data from other processors. The update of ghost buses and ghost branches have been split into separate operations to give users flexibility in optimizing performance by minimizing the amount of data that needs to be communicated in the code.
4. The network contains the partitioner. The partitioner is embedded in the network module but represents a substantial technology in its own right. Partitioning is a key part of parallel application development. It represents the act of dividing up the problem so that each processor is left with approximately equal amounts of work and so that communication between processors (a major source of computational inefficiency in HPC programs) is minimized.

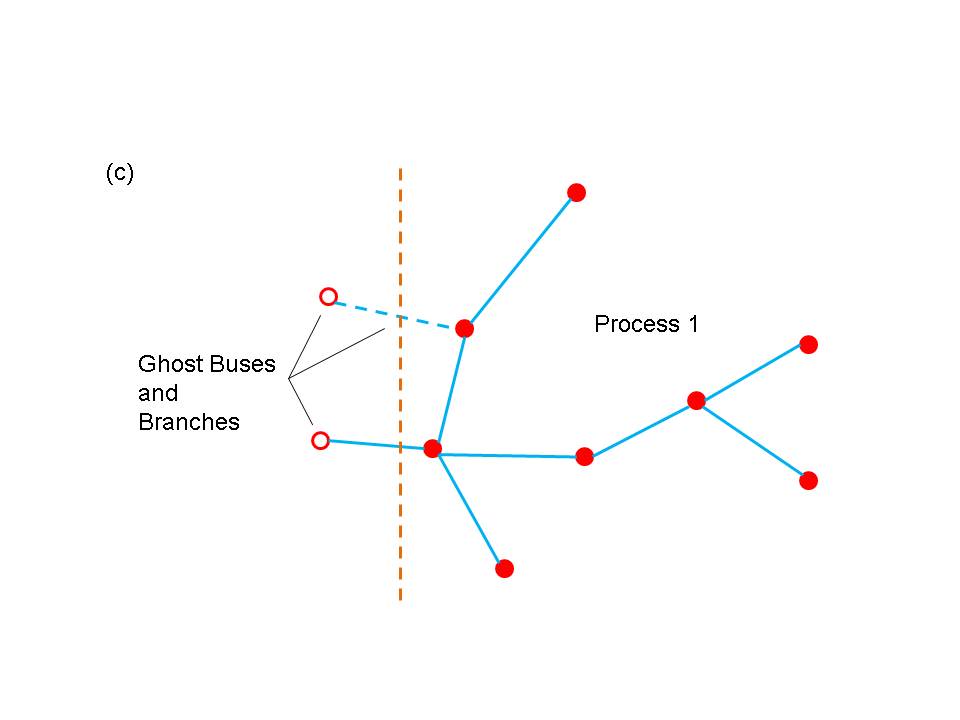
A major use of the partitioner is to rearrange the network in a form that is useful for computation immediately after it is read in from an external file. Typically, the information in the external file is not organized in a way that is necessarily useful for computation, so the partitioner must reorganize data such that large connected blocks are all on the same processor. The partitioner is also responsible for adding the ghost buses and branches to the system.

Ghost buses and branches in a parallel program represent images of buses and branches that are owned by other processes. In order to carry out operations on buses and branches it is frequently necessary to gain access to data associated with attached buses and branches. The most efficient way to do this is to create copies of the buses and branches from other processors on each process so that all locally own objects are attached to these copies (ghosts). The ghost objects are then updated collectively with current information from their home processors at points in the computation. Updating all ghosts at once is almost always more efficient than access data from one bus or branch at a time.

The use of the partitioner to distribute the network between different processors and create ghost nodes and branches is illustrated in Figure 2. Figure 2(a) shows a simple network and Figures 2(b) and 2(c) show the result of distributing the network between two processors.







**Figure 2.** (a) a simple network (b) partition of network on processor 0 (b) partition of network on processor 1. Open circles indicate ghost buses and dotted lines indicate ghost branches.

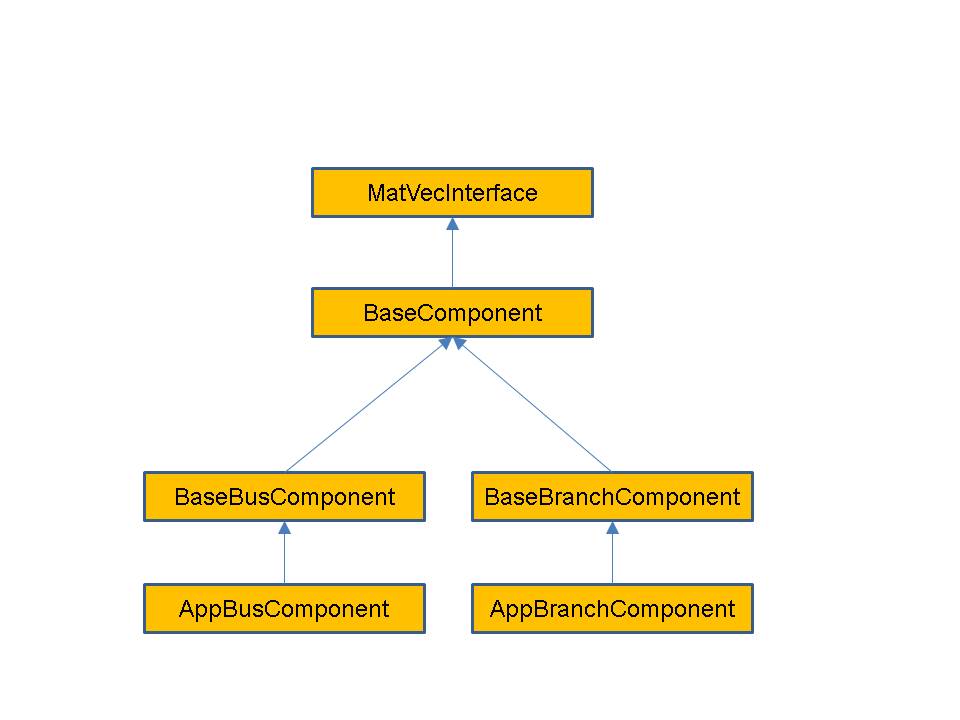
**Math Module:** the math module is used to provide support for distributed matrices and vectors as well as linear solvers, non-linear solvers, and preconditioners. Once created, matrices can be treated as opaque objects and manipulated using a high level syntax that would comparable to writing Matlab code. The distributed matrix and vector data structures themselves are based on existing solver libraries and represent relatively lightweight wrappers on existing code. The current math module is built on the PETSc library but other libraries, such as Hypre and Trilinos could be used instead to implement the math module.

The main functionality associated with the math module is the ability to instantiate new matrices and vectors, add individual matrix and vector elements (and their values) to the matrix/vector objects and invoke and assemble operation on the object. The assemble operation is designed to give the library a chance to set up internal data structures and repartition the matrix elements, etc. in a way that will optimize subsequent calculations. Inclusion of this operation also follows the syntax of most solver libraries when they construct a matrix or vector. This module also includes some basic matrix and vector operations such as matrix-vector multiply and norms.

In addition to basic matrix operations, the math module contains linear and non-linear solvers and preconditioners. The math module provided a simple interface on top of the PETSc libraries that will allow users access to this functionality without having to be familiar with the libraries themselves. This should make it possible to construct solver routines that are comparable in complexity to Matlab scripts. The use of a wrapper instead of having users directly access the libraries will also make it simpler to switch the underlying library in an application. All that will be required will be for developers to link to a different implementation of the math module interface that is built on a different library. There will not be any need to rewrite any application code. This has the advantage that if a different library is used for the math module in one application, it instantly becomes available for other applications.

**Network Components:** Network component is a generic term for objects associated with buses and branches. These objects determine the behavior of the system and the type of analyses being done. Branch components can represent transmission lines and transformers while bus components could model loads, generators, or something else. Both kinds of components could represent measurements (e.g. for a state estimation analysis).

Network components cover a fairly broad range of behaviors and there is little that can be said about them outside the context of a specific problem. Each component inherits from a matrix-vector interface, which enables the framework to generate matrices and vectors from the network in a relatively straightforward way. In addition, buses inherit from a base bus interface and branches inherit from a base branch interface. The relationship between these interfaces is shown if Figure 3.



**Figure 3.** Schematic diagram showing the interface hierarchy for network components.

These base interfaces provide mechanisms for accessing the neighbors of a bus or branch and allow developers to specify what data is transferred in ghost exchanges. They do not define any physical properties of the bus or branch, it is up to application developers to do this.

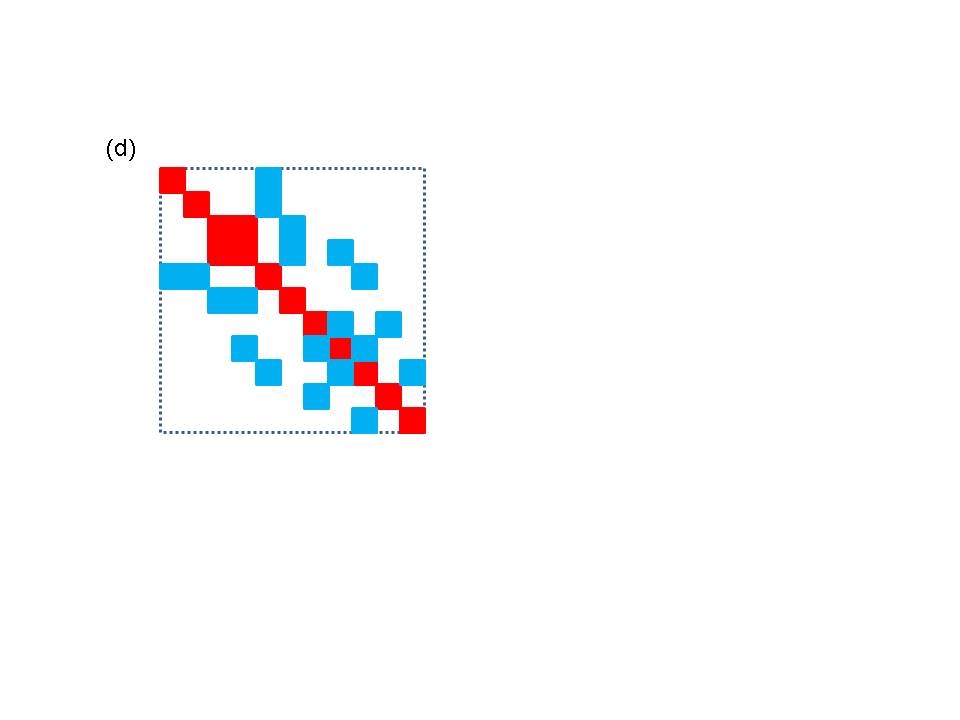
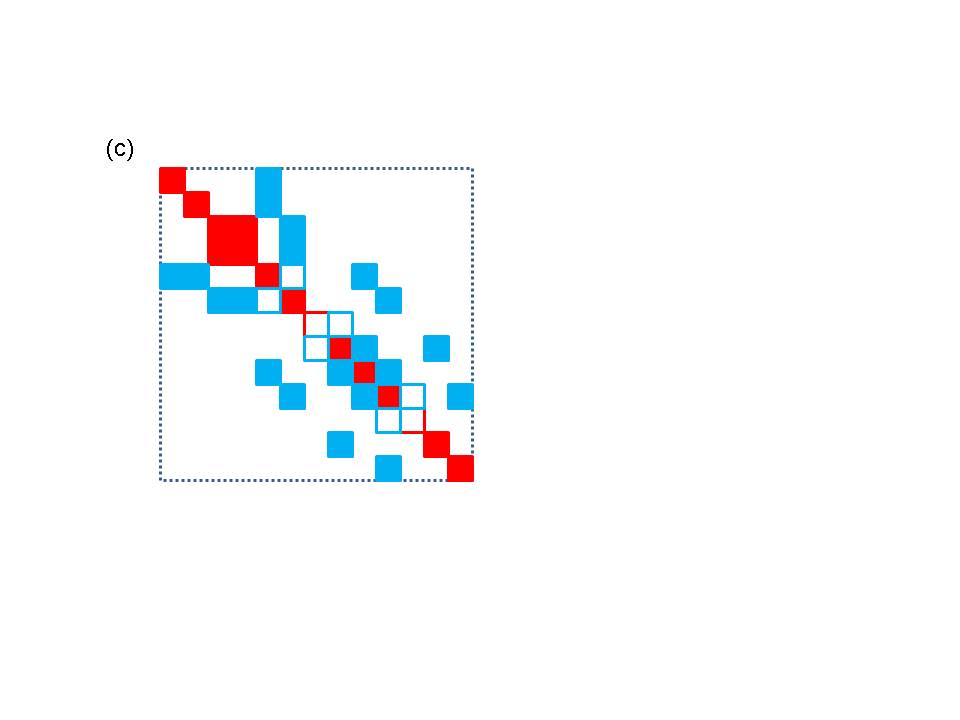
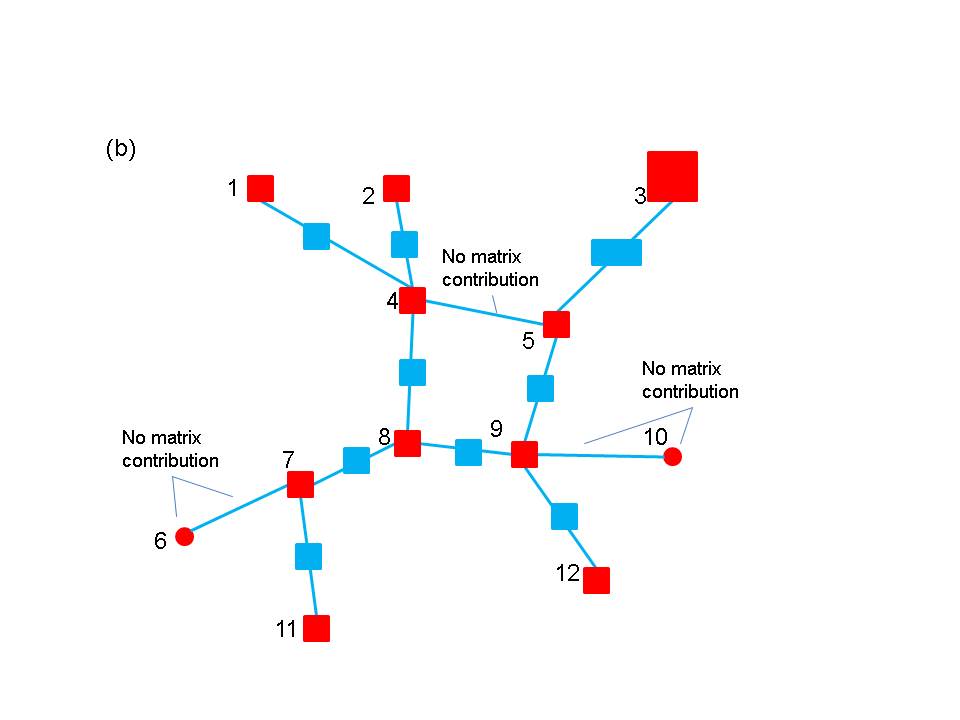
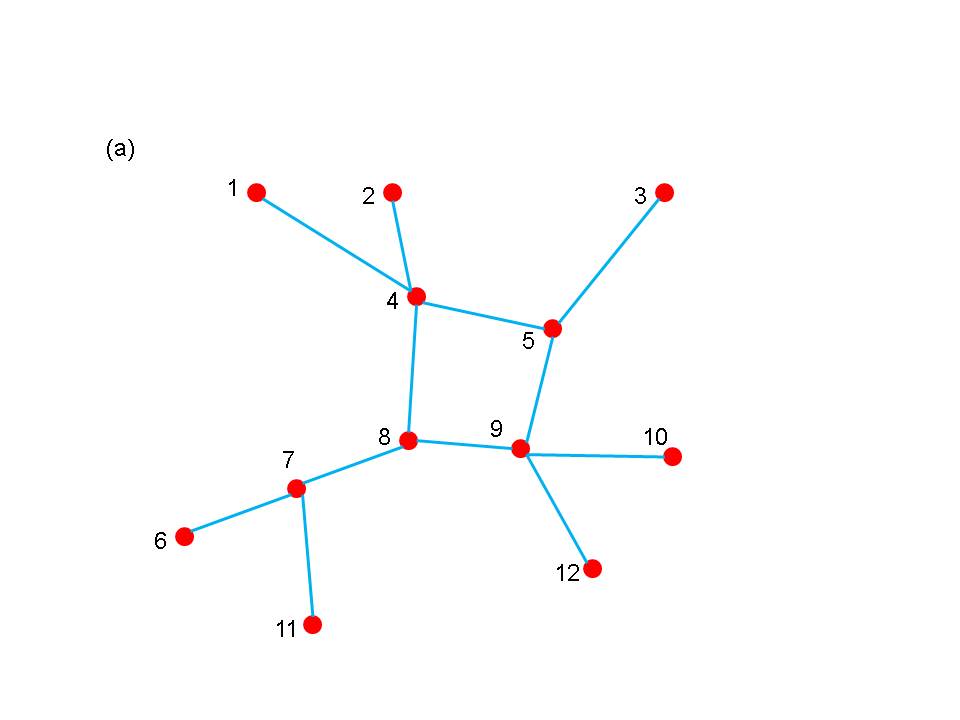
Of these interfaces, the matrix-vector interface is the most important. It answers the question what block of data is contributed by a bus or network and what the dimensions of the block are. For example, if constructing the Y-matrix for a power flow problem using a real-valued formulation, the grid components on buses contribute a 2×2 block to the diagonal of the matrix. Similarly, the grid components on branches contribute a 2×2 block to the off-diagonal elements. (Note that if the Y-matrix is expressed as a complex matrix, then the blocks are of size 1×1.) The location of these blocks in the matrix is determined by the location of the corresponding buses and branches in the network, but the indexing calculations required to determine this location can be made completely transparent to the user via the mapper module.

Because the matrix-vector interface focuses on small blocks, it should be relatively easy for power grid engineers to write the corresponding methods. The full matrices and vectors can then be generated from the network using relatively simple calls to the mapper interface (see the discussion below on the mapper module).

**Network Component Factory:** the network component factory is an application-dependent piece of software that is designed to manage interactions between the network and the network component objects. Most operations in the factory run over all buses and all branches and invoke some operation on each bus and each branch. An example is the “load” operation. After the network is read in from an external file, it consists of a topology and a set of simple data collection objects containing key-value pairs associated with each each bus and branch. The load operation then runs over all buses and branches and instantiates the appropriate objects by invoking a load method in each branch and bus object that takes the values from the data collection object and uses it to instantiate the bus or branch. The application network factory is derived from a base network factory class that contains some additional routines that set up indices, assign neighbors (which initially are only known by the network) to individual buses and branches and assign buffers for The network component factory may also execute other routines that contribute to setting up the network and creating a well-defined state.

**Mapper:** the mapper is a generic capability that can be used to generate a matrix or vector from the network components. This is done by running over all the network components and invoking methods in the matrix-vector interface. The mapper is basically a transformation that converts a set of network components into a matrix or vector based on the behavior of their matrix-vector interfaces. It has no explicit dependencies on either the network components or the type of analyses being performed so this capability is applicable across a wide range of problems.

The matrix-vector interface contains functions that provide two pieces of information about each network component. The first is the size of the matrix block that is contributed by the component and the second is the values in that block. Using this information, the mapper can figure out what the dimensions of the matrix are and where individual elements in the matrix are located. The construction of matrix by the mapper is illustrated in Figure 4 for a small network. Figure 4(a) shows a hypothetical network for which some buses and branches do not contribute to the matrix, as seen in Figure 4(b). This could occur in real systems because the transmission line corresponding to the branch has failed or because a bus represents the reference bus. In addition, it is not necessarily true that all buses and branches contribute the same size elements. The mapping of the individual contributions from the network in Figure 4(b) to initial matrix locations is shown in Figure 4(c). This is followed by elimination of gaps in the matrix in Figure 4(d).



**Figure 4.** A schematic diagram of the matrix map function. The bus numbers in (a) and (b) map to approximate row and column locations in (c). (a) a small network (b) matrix blocks associated with branches and buses. Not that not all blocks are the same size and not all buses and branches contribute (c) initial construction of matrix based on network indices (d) final matrix after eliminating gaps

**Import Module:** the import module is designed to read an external network file, set up the network topology and assign any parameter fields in the file to simple fields. The import module does not partition the network, it is only responsible for reading in the network and distributing the different network elements in a way that guarantees that not too much data ends up on any one processor. The import module is also not responsible for determining if the input is compatible with the analysis being performed. This can be handled by the network factory. The import module is only responsible for determining if it can read the file.

At the moment, it is not clear whether multiple different or import modules will be defined, one for each possible network file format or whether a single import module will be written that takes a file format as an argument. Either way, it will be easy for developers to switch between different file formats for importing the network. The import component will primarily be built on the network module. It will also need to implement its own data transfer strategy for distributing input in a reasonable balanced manner.

**Serial IO Module:** the serial IO module is designed to provide a simple mechanism for writing information from selected buses and/or branches to standard output using a consistent ordering scheme. Individual buses and/or branches implement a write method that will write bus/branch information to a single string. This information usually consists of bus or branch identifiers plus some parameters that are desired in the output. The serial IO module then gathers this information, moves it to the head node, and writes it out in a consistent order. An example of this type of output is shown below.

**Bus Voltages and Phase Angles**

**Bus Number Phase Angle Voltage Magnitude**

**1 0.000000 1.060000**

**2 -4.980000 927.649818**

**3 -12.720000 280.919266**

**4 -10.330000 -1437.822431**

**5 -8.780000 -1320.922177**

**6 -14.220000 548.139123**

**7 -13.370000 -790.995324**

**8 -13.360000 189.293173**

**9 -14.940000 -971.618443**

**10 -15.100000 -589.181023**

**11 -14.790000 -345.309479**

**12 -15.070000 -223.066355**

**13 -15.160000 -426.487761**

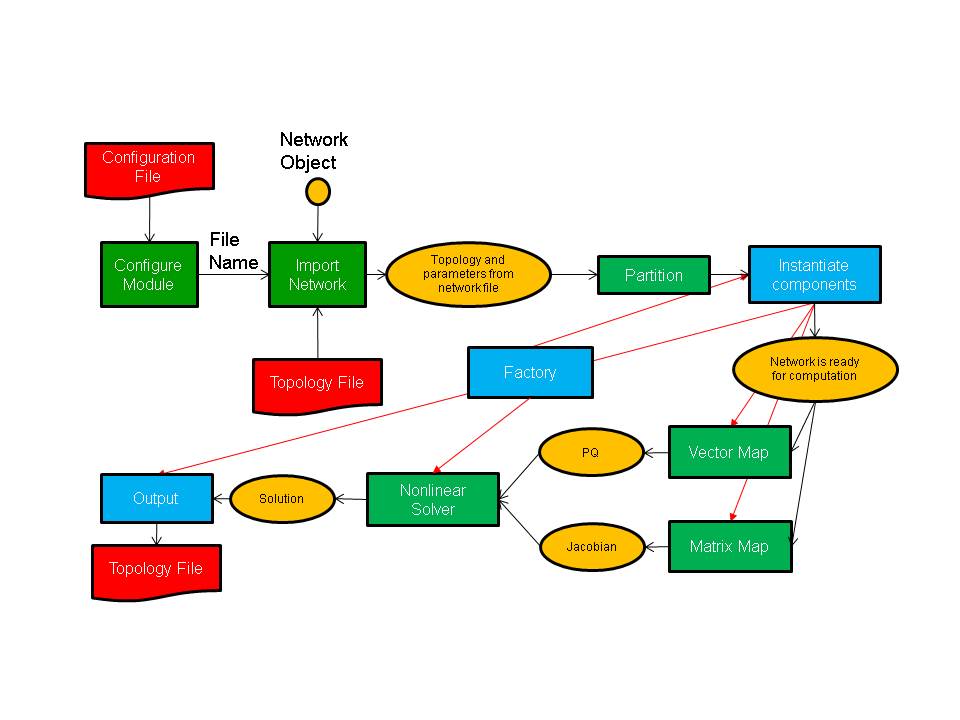
**14 -16.040000 -211.325836**

**Figure 5.** Example output from buses in a 14 bus problem.

**Configuration Module:** the configuration module is designed to provide a central mechanism for directing module specific information to each of the components making up a given application. For example, information about convergence thresholds and maximum numbers of iterations might need to be picked up by the solver module from an external configuration file. The configuration module is designed to read input decks using a simple XML format that supports a hierarchical input. This can be used to control which input gets directed to individual objects in the application, even if the object is a framework component and cannot be modified by the application developer.

**Developing Applications**

The use of these modules in an application such as power flow is outlined in Figure 6. For different power grid problems, the details of the code will be different, but most of these motifs will appear at some point or other. The main differences will probably be in feedback loops as results from one part of the calculation are fed back into other parts of the calculation. For example, an iterative solver will likely need to update the current values of the network components, which can then be used to generate new matrices and vectors that are fed back into the next iteration of the solver. The diagram is not complete, but gives an overall view of code structure and data movement.



**Figure 6.** Schematic of program flow for a power flow simulation. The yellow ovals are distributed data objects, the green blocks are GridPACK™ framework components and the blue blocks are application specific code. External files are red.

As shown in the figure, application developers will need to focus on writing two or three sets of modules. The first is the network components. These are the descriptions of the physics and/or measurements that are associated with buses and branches in the power grid network. The network factory is a module that initializes the grid components on the network after the network is originally created by the import module. The power flow problem is simple enough that it can use a non-linear solver directly from the math module but even a straightforward solution such as this requires the developer to overwrite some functions in the factory that are used in the non-linear solver iterations.