```
#include "gridpack/include/gridpack.hpp"

#include "pf_app.hpp"

#include "pf_factory.hpp"


// Calling program for powerflow application


Basic constructor


gridpack::powerflow::PFApp::PFApp(void)
{
}


Basic destructor


gridpack::powerflow::PFApp::~PFApp(void)
{
}


Enumerated type to distinguish between different configuration file
formats.  PSS/E version 23 and 33 formats are currently supported.

enum Parser{PTI23, PTI33};


Execute application.  The name of XML-formatted input can be used
as an argument to the executable
@param argc number of arguments
@param argv list of character strings
```

```
void gridpack::powerflow::PFApp::execute(int argc, char** argv)
{
  // Define a communicator on the group of all processors (the world group)
  // and create and instance of a power flow network
  gridpack::parallel::Communicator world;
  boost::shared_ptr<PFNetwork> network(new PFNetwork(world));

  // Read configuration file. If file is not specified when invoking the
  // executable, assume the input file is called "input.xml"
  gridpack::utility::Configuration *config
    = gridpack::utility::Configuration::configuration();
  // Echo input file to standard out
  config->enableLogging(&std::cout);
  bool opened;
  if (argc >= 2 && argv[1] != NULL) {
    char inputfile[256];
    sprintf(inputfile,"%s",argv[1]);
    opened = config->open(inputfile,world);
  } else {
    opened = config->open("input.xml",world);
  }
  // If no input file found, return
  if (!opened) return;

  // Find the Configuration.Powerflow block within the input file
  // and set cursor pointer to that block
  gridpack::utility::Configuration::CursorPtr cursor;
  cursor = config->getCursor("Configuration.Powerflow");
  std::string filename;
  int filetype = PTI23;
  // If networkConfiguration field found in input, assume that file
  // is PSS/E version 23 format, otherwise if networkConfiguation_v33
  // field found in input, assume file is version 33 format. If neither
  // field is found, then no configuration file is specified so return
  if (!cursor->get("networkConfiguration",&filename)) {
    if (cursor->get("networkConfiguration_v33",&filename)) {
      filetype = PTI33;
    } else {
```

```
    printf("No network configuration file specified\n");
    return;
  }
}
// Set convergence and iteration parameters from input file. If
// tolerance and maxIteration fields not found, then use defaults
double tolerance = cursor->get("tolerance",1.0e-6);
int max_iteration = cursor->get("maxIteration",50);
ComplexType tol;
// Different files use different conventions for the phase shift sign.
// Allow users to change sign to correspond to the convention used
// in this application.
double phaseShiftSign = cursor->get("phaseShiftSign",1.0);

// Echo network file name to standard out and create appropriate parser.
// Parse the file and change the phase shift sign, if necessary. The
// rank() function on the communicator is used to determine the processor ID
if (world.rank() == 0) printf("Network filename: (%s)\n",filename.c_str());
if (filetype == PTI23) {
  if (world.rank() == 0) printf("Using V23 parser\n");
  gridpack::parser::PTI23_parser<PFNetwork> parser(network);
  parser.parse(filename.c_str());
  if (phaseShiftSign == -1.0) {
    parser.changePhaseShiftSign();
  }
} else if (filetype == PTI33) {
  if (world.rank() == 0) printf("Using V33 parser\n");
  gridpack::parser::PTI33_parser<PFNetwork> parser(network);
  parser.parse(filename.c_str());
  if (phaseShiftSign == -1.0) {
    parser.changePhaseShiftSign();
  }
}

// Partition network between processors
network->partition();

// Echo number of buses and branches to standard out. This message prints from
```

```
// each processor. These numbers may be different for different processors
printf("Process: %d NBUS: %d NBRANCH: %d\n",world.rank(),network->numBuses(),
    network->numBranches());

// Create serial IO object to export data from buses
gridpack::serial_io::SerialBusIO<PFNetwork> busIO(8192,network);
char ioBuf[128];

// Echo convergence parameters to standard out. The use of the header
// method in the SerialBusIO class guarantees that the message is only
// written once to standard out.
sprintf(ioBuf,"\nMaximum number of iterations: %d\n",max_iteration);
busIO.header(ioBuf);
sprintf(ioBuf,"\nConvergence tolerance: %f\n",tolerance);
busIO.header(ioBuf);

// Create factory and call the load method to initialize network components
// from information in configuration file
gridpack::powerflow::PFFactory factory(network);
factory.load();

// Set network components using factory. This includes defining internal
// indices that are used in data exchanges and to manage IO
factory.setComponents();

// Set up bus data exchange buffers. The data to be exchanged is defined
// in the network component classes
factory.setExchange();

// Create bus data exchange. Data exchanges between branches are not needed
// for this calculation
network->initBusUpdate();

// Create components Y-matrix
factory.setYBus();

// Create components of S vector and print out first iteration count
// to standard output
```

```
factory.setSBus();
busIO.header("\nIteration 0\n");

factory.setMode(RHS);
gridpack::mapper::BusVectorMap<PFNetwork> vMap(network);
boost::shared_ptr<gridpack::math::Vector> PQ = vMap.mapToVector();
// Create Jacobian matrix
factory.setMode(Jacobian);
gridpack::mapper::FullMatrixMap<PFNetwork> jMap(network);
boost::shared_ptr<gridpack::math::Matrix> J = jMap.mapToMatrix();

// Create X (solution) vector by cloning PQ
boost::shared_ptr<gridpack::math::Vector> X(PQ->clone());
```

The LinearSolver object solves equations of the form $\overline{\overline{A}} \cdot \overline{X} = \overline{B}$

```
// Create linear solver and configure it with settings from the
// input file (inside the LinearSolver block)
gridpack::math::LinearSolver solver(*J);
solver.configure(cursor);

// Set initial value of the tolerance
tol = 2.0*tolerance;
int iter = 0;

// First iteration of the solver to initialize Newton-Raphson loop
X->zero(); //might not need to do this
busIO.header("\nCalling solver\n");
solver.solve(*PQ, *X);
```

normInfinity evaluates the norm $N = \max_i |X_i|$

```
// The Newton-Raphson algorithm evaluates incremental changes to the solution
// vector. When the algorithm is done, the RHS vector is zero
tol = PQ->normInfinity();


while (real(tol) > tolerance && iter < max_iteration) {
```

```
  // Push current values in X vector back into network components
  // This uses setValues method in PFBus class in order to work
  factory.setMode(RHS);
  vMap.mapToBus(X);

  // Exchange data between ghost buses (We don't need to exchange data
  // between branches)
  network->updateBuses();

  // Update Jacobian and PQ vector with new values
  vMap.mapToVector(PQ);
  factory.setMode(Jacobian);
  jMap.mapToMatrix(J);

  // Resolve equations (the solver can be reused since the number and
  // location of non-zero elements is the same)
  X->zero(); //might not need to do this
  solver.solve(*PQ, *X);

  // Evaluate norm of residual and print out current iteration to standard
  // out
  tol = PQ->normInfinity();
  sprintf(ioBuf,"\nIteration %d Tol: %12.6e\n",iter+1,real(tol));
  busIO.header(ioBuf);
  iter++;
}

// Push final result back onto buses so we get the right values printed out to
// output
factory.setMode(RHS);
vMap.mapToBus(X);

// Make sure that ghost buses have up-to-date values before printing out
// final results (evaluating power flow on branches requires correct values
// of voltages on all buses)
network->updateBuses();

// Write out headers and power flow values for all branches
```

```
  gridpack::serial_io::SerialBranchIO<PFNetwork> branchIO(512,network);
  branchIO.header("\n   Branch Power Flow\n");
  branchIO.header("\n        Bus 1       Bus 2   CKT         P"
                 "                    Q\n");
  // Write branch values
  branchIO.write();


  // Write out headers and voltage values for all buses
  busIO.header("\n   Bus Voltages and Phase Angles\n");
  busIO.header("\n   Bus Number      Phase Angle      Voltage Magnitude\
n");
  // Write bus values values
  busIO.write();
}
```