EECS 598 Homework 3

Jonas Kersulis

October 23, 2013

Part 1

The admittance matrix plays a role in both mismatch and Jacobian calculations, and is based solely on system data, so the first part of my code generates it.
Bus 2 is a PV bus, so it gives rise to an active power mismatch equation. Buses 3 and 4 are PQ buses, so each gives rise to both active and reactive power mismatch equations. The five equations are may be used to find five unknowns: $\theta_2, \theta_3, \theta_4, V_3$, and V_4. I
Next, I wrote functions to calculate active and reactive power mismatches. I also wrote four functions to calculate the four types of partial differentials present in the Jacobian. The two balance and four Jacobian functions may be found in the appendix.
Following is the script I wrote for Part 1. It performs an iterative power flow to find three unknown angles and two unknown voltages.

Step 1: Find Admittance Matrix

```
clc
clear all
% Translate network topology and parameters to matrix form by
% building the admittance matrix Y.

%   Bus types:
%   0   Slack
%   1   PQ
%   2   PV

%        Num  Type  P       Q     theta      V
BusInfo =  [1   0   0       0       0       1.02;
            2   2   0.4     0       0       1.05;
            3   1   -1.0    -0.3    0       0;
            4   1   0       0       0       0;];

n = size(BusInfo,1);
```

```
%          From  To   R       X        B
LineInfo = [1   2   0.10    1.00    0.50;
            1   3   0.05    0.60    0.25;
            2   4   0.05    0.40    0.25;
            3   4   0.00    0.0001  0.00;];

% Mutual admittance is found by taking the inverse of the complex impedance
% between each connected pair of buses:
y_mutual = [LineInfo(:,1), LineInfo(:,2), ...
    (1./complex(LineInfo(:,3), LineInfo(:,4)))];

% Self admittance is the sum of the admittances of all lines connected to
% each bus, plus half of the total charging susceptance associated with
% those lines.
y_self =   [BusInfo(1,1), BusInfo(1,1), ...
    (y_mutual(1,3)+y_mutual(2,3)+ 0.5i.*LineInfo(1,5) + 0.5i.*LineInfo(2,5));
            BusInfo(2,1), BusInfo(2,1), ...
    (y_mutual(1,3)+y_mutual(3,3)+ 0.5i.*LineInfo(1,5) + 0.5i.*LineInfo(3,5));
            BusInfo(3,1), BusInfo(3,1), ...
    (y_mutual(2,3)+y_mutual(4,3)+ 0.5i.*LineInfo(2,5) + 0.5i.*LineInfo(4,5));
            BusInfo(4,1), BusInfo(4,1), ...
    (y_mutual(3,3)+y_mutual(4,3)+ 0.5i.*LineInfo(3,5) + 0.5i.*LineInfo(4,5));];

% The admittance matrix is found by vertically concatenating the mutual and
% self admittance matrices as follows:
Y = sparse([y_mutual(:,1); y_mutual(:,2); y_self(:,1)],...
           [y_mutual(:,2); y_mutual(:,1); y_self(:,2)],...
           [-1.*y_mutual(:,3); -1.*y_mutual(:,3); y_self(:,3)]);
```

Step 2: Initialize with Givens and Guesses

```
% Bus 1: slack bus (no equations)
% theta1    := 0
% V1        := 1.02pu

% Bus 2: PV bus
% Find active power mismatch
% P2_sp     := 0.4pu
% V2        := 1.05pu

% Bus 3: PQ bus
% Find active power mismatch
% Find reactive power mismatch
% P3_sp     := -1.0pu
% Q3_sp     := -0.3pu
% Bus 4: PQ bus
% Find active power mismatch
% Find reactive power mismatch
% P4_sp     := 0
% Q4_sp     := 0

num_unknowns = 5;
% We need to make an initial guess for each of the five unknowns:
theta2_guess = 0.1;%input('Guess for theta2: ');
theta3_guess = 0.1;%input('Guess for theta3: ');
theta4_guess = 0.1;%input('Guess for theta4: ');
V3_guess = 1.0;
V4_guess = 1.0;

% Use guesses and given information to construct a vector for each variable
% type:
P_sp = [0; 0.4; -1.0; 0];
Q_sp = [0; 0; -0.3; 0];
theta = [0; theta2_guess; theta3_guess; theta4_guess];
V = [1.02; 1.05; V3_guess; V4_guess];

% Instantiate nett active and reactive power vectors:
P_nett(1:n) = 0;
Q_nett(1:n) = 0;

Jacobian = zeros(num_unknowns); % Fix the size of the Jacobian
mismatch(1:num_unknowns) = 0;
mismatch = mismatch';
```

Step 3: Newton-Raphson Implementation

```
tolerance = 1e-5;   % When updates are smaller than this value, stop
                    % iterating.
max_iterations = 10;% Do not perform any more iterations than this.

% Set up matrix to store results:
Result(1:num_unknowns+1,1:max_iterations) = 0;

for j = 1:max_iterations
    % Calculate nett active and reactive power at each bus:
    for x = 1:n
        P_nett(x) = activebalance(x,theta,V,Y,n);
        Q_nett(x) = reactivebalance(x,theta,V,Y,n);
    end

    % Calculate mismatches:
```

```
mismatch(1) = P_sp(2) - P_nett(2);   % Bus 2 active
mismatch(2) = P_sp(3) - P_nett(3);   % Bus 3 active
mismatch(3) = P_sp(4) - P_nett(4);   % Bus 4 active

mismatch(4) = Q_sp(3) - Q_nett(3); % Bus 3 reactive
mismatch(5) = Q_sp(4) - Q_nett(4); % Bus 4 reactive

% Use H and N functions to find four parts of the Jacobian matrix:
H = Jacobian(1:3,1:3);
for x = 2:4
    for y = 2:4
        H(x-1,y-1) = Jac_H(x,y,theta,V,Y,Q_nett(y));
    end
end

N = Jacobian(1:3,4:5);
for x = 2:4
    for y = 3:4
        N(x-1,y-2) = Jac_N(x,y,theta,V,Y,P_nett(y));
    end
end

J = Jacobian(4:5,1:3);
for x = 3:4
    for y = 2:4
        J(x-2,y-1) = -Jac_N(x,y,theta,V,Y,P_nett(y));
    end
end

L = Jacobian(4:5,4:5);
for x = 3:4
    for y = 3:4
        L(x-2,y-2) = Jac_H(x,y,theta,V,Y,Q_nett(y));
    end
end

% Piece together Jacobian using four matrices calculated above:
Jacobian = [H N; J L];

% Solve for unknowns:
update = Jacobian\mismatch;

% Use updated values in next iteration:
theta(2) = theta(2) + update(1);
theta(3) = theta(3) + update(2);
theta(4) = theta(4) + update(3);
```

```
    V(3) = V(3)*(1 + update(4));   % Multiply update by V3 because
                                   % algorithm solved for del(V3)/V3
    V(4) = V(4)*(1 + update(5));

    % Store results for later review:
    Result(1,j) = j;
    Result(2:num_unknowns+1,j) = update;
    if(all(abs(update)<tolerance))
        break
    end
end
disp(Result)
P_nett
Q_nett
theta
V
```

Output:

```
  Columns 1 through 7

    1.0000    2.0000    3.0000    4.0000    5.0000    6.0000    7.0000
   -0.2013   -0.0117   -0.0010   -0.0003   -0.0001   -0.0000   -0.0000
   -0.3889   -0.0179   -0.0022   -0.0005   -0.0001   -0.0000   -0.0000
   -0.3888   -0.0179   -0.0022   -0.0005   -0.0001   -0.0000   -0.0000
   -0.0129   -0.0110   -0.0097   -0.0024   -0.0006   -0.0002   -0.0000
   -0.0129   -0.0110   -0.0097   -0.0024   -0.0006   -0.0002   -0.0000

  Columns 8 through 10

    8.0000         0         0
   -0.0000         0         0
   -0.0000         0         0
   -0.0000         0         0
   -0.0000         0         0
   -0.0000         0         0


P_nett =

    0.6289    0.4000   -1.0000   -0.0000


Q_nett =

   -0.2946   -0.1534   -0.3000    0.0000
```

```
theta =

         0
   -0.1144
   -0.3097
   -0.3097


V =

    1.0200
    1.0500
    0.9637
    0.9638
```

As you can see, the algorithm converged after 8 iterations with the given initial guesses.

Part 2

In Part 2, bus 4 is connected with infinite admittance to bus 3, so it can be eliminated. The admittance matrix and power flow are modified accordingly:

Step 1: Find Admittance Matrix

```
clc
clear all
% Translate network topology and parameters to matrix form by
% building the admittance matrix Y.

%   Bus types:
%   0   Slack
%   1   PQ
%   2   PV

%        Num  Type  P       Q     theta      V
BusInfo =  [1   0   0       0       0       1.02;
            2   2   0.4     0       0       1.05;
            3   1   -1.0    -0.3    0       0;];

n = size(BusInfo,1);

%         From  To   R       X        B
```

```
LineInfo = [1   2   0.10    1.00    0.50;
            1   3   0.05    0.60    0.25;
            2   3   0.05    0.40    0.25;];

% Mutual admittance is found by taking the inverse of the complex impedance
% between each connected pair of buses:
y_mutual = [LineInfo(:,1), LineInfo(:,2), ....
    (1./complex(LineInfo(:,3), LineInfo(:,4)))];

% Self admittance is the sum of the admittances of all lines connected to
% each bus, plus half of the total charging susceptance associated with
% those lines.
y_self =   [BusInfo(1,1), BusInfo(1,1), ...
    (y_mutual(1,3)+y_mutual(2,3)+ 0.5i.*LineInfo(1,5) + 0.5i.*LineInfo(2,5));
            BusInfo(2,1), BusInfo(2,1), ...
    (y_mutual(1,3)+y_mutual(3,3)+ 0.5i.*LineInfo(1,5) + 0.5i.*LineInfo(3,5));
            BusInfo(3,1), BusInfo(3,1), ...
    (y_mutual(2,3)+y_mutual(3,3)+ 0.5i.*LineInfo(2,5) + 0.5i.*LineInfo(3,5));];

% The admittance matrix is found by vertically concatenating the mutual and
% self admittance matrices as follows:
Y = sparse([y_mutual(:,1); y_self(:,1); y_mutual(:,2)],...
           [y_mutual(:,2); y_self(:,2); y_mutual(:,1)],...
           [-1.*y_mutual(:,3); y_self(:,3); -1.*y_mutual(:,3)]);
```

Step 2: Initialize with Givens and Guesses

```
% Bus 1: slack bus (no equations)
% theta1    := 0
% V1        := 1.02pu

% Bus 2: PV bus
% Find active power mismatch
% P2_sp     := 0.4pu
% V2        := 1.05pu

% Bus 3: PQ bus
% Find active power mismatch
% Find reactive power mismatch
% P3_sp     := -1.0pu
% Q3_sp     := -0.3pu

num_unknowns = 3;
% We need to make an initial guess for each of the unknowns:
theta2_guess = 0.1;%input('Guess for theta2: ');
theta3_guess = 0.1;%input('Guess for theta3: ');
```

```
V3_guess = 1.0;%input('Guess for V3: ');

% Use guesses and given information to construct a vector for each variable
% type:
P_sp = [0; 0.4; -1.0];
Q_sp = [0; 0; -0.3];
theta = [0; theta2_guess; theta3_guess];
V = [1.02; 1.05; V3_guess];

% Instantiate nett active and reactive power vectors:
P_nett(1:n) = 0;
Q_nett(1:n) = 0;

Jacobian = zeros(num_unknowns); % Fix the size of the Jacobian
mismatch(1:num_unknowns) = 0;
mismatch = mismatch';   % mismatch is a column vector
```

Step 3: Newton-Raphson Implementation

```
tolerance = 1e-5;   % When updates are smaller than this value, stop
                    % iterating.
max_iterations = 10;% Do not perform any more iterations than this.

% Set up matrix to store results:
Result(1:num_unknowns+1,1:max_iterations) = 0;

for j = 1:max_iterations
    % Calculate nett active and reactive power at each bus:
    for x = 1:n
        P_nett(x) = activebalance(x,theta,V,Y,n);
        Q_nett(x) = reactivebalance(x,theta,V,Y,n);
    end

    % Calculate mismatches:
    mismatch(1) = P_sp(2) - P_nett(2);   % Bus 2 active
    mismatch(2) = P_sp(3) - P_nett(3);   % Bus 3 active
    mismatch(3) = Q_sp(3) - Q_nett(3); % Bus 3 reactive

    % Use H and N functions to find four parts of the Jacobian matrix:
    H = Jacobian(1:2,1:2);
    for x = 2:3
        for y = 2:3
            H(x-1,y-1) = Jac_H(x,y,theta,V,Y,Q_nett(y));
        end
    end
    N = Jacobian(1:2,3);
    for x = 2:3
        N(x-1,1) = Jac_N(x,3,theta,V,Y,P_nett(3));
    end

    J = Jacobian(3,1:2);
    for y = 2:3
        J(1,y-1) = Jac_J(3,y,theta,V,Y,P_nett(3));
    end

    L = Jacobian(3,3);
    L(1,1) = Jac_L(3,3,theta,V,Y,Q_nett(3));

    % Piece together Jacobian using four matrices calculated above:
    Jacobian = [H N; J L];

    % Solve for unknowns:
    update = Jacobian\mismatch;

    % Use updated values in next iteration:
    theta(2) = theta(2) + update(1);
    theta(3) = theta(3) + update(2);

    V(3) = V(3)*(1 + update(3));   % Multiply update by V3 because
                                   % algorithm solves for del(V3)/V3

    % Store results for later review:
    Result(1,j) = j;
    Result(2:num_unknowns+1,j) = update;

    % If all update elements are smaller than the tolerance, exit loop:
     if(all(abs(update)<tolerance))
        break
    end
end
disp(Result)
P_nett
Q_nett
theta
V
```

Output:

```
  Columns 1 through 7

    1.0000    2.0000    3.0000    4.0000    5.0000         0         0
   -0.2019   -0.0115   -0.0010   -0.0000   -0.0000         0         0
   -0.3935   -0.0144   -0.0018   -0.0000   -0.0000         0         0
    0.0177   -0.0498   -0.0034   -0.0000   -0.0000         0         0

  Columns 8 through 10

         0         0         0
         0         0         0
         0         0         0
         0         0         0


P_nett =

    0.6289    0.4000   -1.0000


Q_nett =

   -0.2946   -0.1534   -0.3000


theta =

         0
   -0.1144
   -0.3097


V =

    1.0200
    1.0500
    0.9638
```

This time the algorithm converged after 5 iterations instead of 8. V_4 from Part 1 is identical to V_3 here. Also worthy of note is the fact that the angle vectors are identical, but the final element is repeated twice in Part 1. Overall, the resulting data for Parts 1 and 2 are similar, but the Part 2 algorithm converged more quickly.

Part 1 Jacobian

```
   1.0e+03 *
```

```
    0.0035         0   -0.0025         0    0.0002
         0    9.2898   -9.2883   -0.0009   -0.0005
   -0.0024   -9.2883    9.2906    0.0005    0.0003
         0    0.0009    0.0005    9.2898   -9.2883
    0.0008   -0.0005   -0.0003   -9.2883    9.2906
```

Part 1 Jacobian

```
Part 2 Jacobian =

    3.5454   -2.5040    0.1780
   -2.3832    3.8915   -0.5861
    0.7889   -1.4139    3.2915
```

Condition number:

```
>> max(max(abs(Jacobian)))/min(min(abs(Jacobian)))

ans =

   21.8644
```

In Part 1, entering $(V_3, V_4) = (0.95, 1.05)$ resulted in a failure to converge. Similarly, $(1.05, 0.95)$ also caused the algorithm to fail. By contrast, the algorithm in Part 2 was much more robust. It converged to the same operating point even when I let V_3 be 4pu! The only way I was able to make the Part 2 algorithm error was to set V_3 equal to zero. By setting V_3 equal to any small or negative value, I was able to prevent the algorithm from converging, but the performance of the Part 2 algorithm is significantly better than that of Part 1.
The reason that changing V_3 and V_4 has such a dramatic effect on the Part 1 algorithm is due to numerics. The line with the small impedance makes the admittance matrix nearly singular from a numerical standpoint, which causes the algorithm to be unstable and even yield errors. Note that the Part 1 Jacobian contains zeros, so its condition number approaches infinity. When I used the `rank` command on the operating point Jacobian when (V_3, V_4) was $1.05, 0.95)$, the result `MATLAB` returned was 4, which means the next iteration would have been unable to invert the Jacobian, and would have returned divide-by-zero errors. By contrast, the condition number of the Part 2 Jacobian was a reasonable 21.86. The matrices in Part 2 are well-scaled and do not result in near-singularity.

Part 3

In Part 3, we are given that P(1,3) is regulated by generator 2 to be 0.4pu. By the approximate equation for active power, this means theta(3) must be -0.2417 rad. Using this new constraint leaves us with two equations and two unknowns (theta(2) and V(3)).

Step 1: Find Admittance Matrix

```
clc
clear all
% Translate network topology and parameters to matrix form by
% building the admittance matrix Y.

%   Bus types:
%   0   Slack
%   1   PQ
%   2   PV

%        Num  Type  P       Q     theta      V
BusInfo =  [1   0   0       0       0       1.02;
            2   2   0.4     0       0       1.05;
            3   1   -1.0    -0.3    0       0;];

n = size(BusInfo,1);

%         From  To   R       X        B
LineInfo = [1   2   0.10    1.00    0.50;
            1   3   0.05    0.60    0.25;
            2   3   0.05    0.40    0.25;];

% Mutual admittance is found by taking the inverse of the complex impedance
% between each connected pair of buses:
y_mutual = [LineInfo(:,1), LineInfo(:,2), ...
    (1./complex(LineInfo(:,3), LineInfo(:,4)))];

% Self admittance is the sum of the admittances of all lines connected to
% each bus, plus half of the total charging susceptance associated with
% those lines.
y_self =   [BusInfo(1,1), BusInfo(1,1), ...
    (y_mutual(1,3)+y_mutual(2,3)+ 0.5i.*LineInfo(1,5) + 0.5i.*LineInfo(2,5));
            BusInfo(2,1), BusInfo(2,1), ...
    (y_mutual(1,3)+y_mutual(3,3)+ 0.5i.*LineInfo(1,5) + 0.5i.*LineInfo(3,5));
            BusInfo(3,1), BusInfo(3,1), ...
    (y_mutual(2,3)+y_mutual(3,3)+ 0.5i.*LineInfo(2,5) + 0.5i.*LineInfo(3,5));];
```

```
% The admittance matrix is found by vertically concatenating the mutual and
% self admittance matrices as follows:
Y = sparse([y_mutual(:,1); y_self(:,1); y_mutual(:,2)],...
           [y_mutual(:,2); y_self(:,2); y_mutual(:,1)],...
           [-1.*y_mutual(:,3); y_self(:,3); -1.*y_mutual(:,3)]);
```

Step 2: Initialize with Givens and Guesses

```
% Bus 1: slack bus (no equations)
% theta1    := 0
% V1        := 1.02pu

% Bus 2
% V2        := 1.05pu

% Bus 3: PQ bus
% Find active power mismatch
% Find reactive power mismatch
% P3_sp     := -1.0pu
% Q3_sp     := -0.3pu

num_unknowns = 2;
% We need to make an initial guess for each of the unknowns:
theta2_guess = 0.1;%input('Guess for theta2: ');
V3_guess = 1.0;%input('Guess for V3: ');

% Use guesses and given information to construct a vector for each variable
% type:
P_sp = [0; 0; -1.0];    % P2 is no longer specified
Q_sp = [0; 0; -0.3];
theta = [0; theta2_guess; -0.2417];    % New angle constraint
V = [1.02; 1.05; V3_guess];  % New voltage constraint

% Instantiate nett active and reactive power vectors:
P_nett(1:n) = 0;
Q_nett(1:n) = 0;

Jacobian = zeros(num_unknowns); % Fix the size of the Jacobian
mismatch(1:num_unknowns) = 0;
mismatch = mismatch';   % mismatch is a column vector
```

Step 3: Newton-Raphson Implementation

```
tolerance = 1e-5;   % When updates are smaller than this value, stop
                    % iterating.
max_iterations = 25;% Do not perform any more iterations than this.
```

```
% Set up matrix to store results:
Result(1:num_unknowns+1,1:max_iterations) = 0;

for j = 1:max_iterations
    % Calculate nett active and reactive power at each bus:
    for x = 1:n
        P_nett(x) = activebalance(x,theta,V,Y,n);
        Q_nett(x) = reactivebalance(x,theta,V,Y,n);
    end

    % Calculate mismatches:
    mismatch(1) = P_sp(3) - P_nett(3);   % Bus 3 active
    mismatch(2) = Q_sp(3) - Q_nett(3); % Bus 3 reactive

    % Use H and N functions to find four parts of the Jacobian matrix:
    H = Jac_H(2,3,theta,V,Y,Q_nett(3));
    N = Jac_N(2,3,theta,V,Y,P_nett(3));
    J = Jac_J(3,3,theta,V,Y,P_nett(3));
    L = Jac_L(3,3,theta,V,Y,Q_nett(3));

    % Piece together Jacobian using four matrices calculated above:
    Jacobian = [H N; J L];

    % Solve for unknowns:
    update = Jacobian\mismatch;

    % Use updated values in next iteration:
    theta(2) = theta(2) + update(1);
    V(3) = V(3)*(1 + update(2));   % Multiply update by V3 because
                                   % algorithm solves for del(V3)/V3

    % Store results for later review:
    Result(1,j) = j;
    Result(2:num_unknowns+1,j) = update;

    % If all update elements are smaller than the tolerance, exit loop:
     if(all(abs(update)<tolerance))
        break
    end
end
disp(Result)
P_nett
Q_nett
theta
V
```

```
  Columns 1 through 7

    1.0000    2.0000    3.0000    4.0000    5.0000    6.0000    7.0000
   -0.1310    0.0639   -0.0556    0.0314   -0.0191    0.0118   -0.0068
   -0.1206    0.1511   -0.0761    0.0523   -0.0304    0.0182   -0.0112

  Columns 8 through 14

    8.0000    9.0000   10.0000   11.0000   12.0000   13.0000   14.0000
    0.0043   -0.0025    0.0016   -0.0009    0.0006   -0.0003    0.0002
    0.0066   -0.0041    0.0024   -0.0015    0.0009   -0.0005    0.0003

  Columns 15 through 21

   15.0000   16.0000   17.0000   18.0000   19.0000   20.0000   21.0000
   -0.0001    0.0001   -0.0000    0.0000   -0.0000    0.0000   -0.0000
   -0.0002    0.0001   -0.0001    0.0000   -0.0000    0.0000   -0.0000

  Columns 22 through 25

         0         0         0         0
         0         0         0         0
         0         0         0         0


P_nett =

    0.4012    0.6272   -1.0000


Q_nett =

   -0.3120   -0.1636   -0.3000


theta =

         0
   -0.0026
   -0.2417


V =

    1.0200
    1.0500
    0.9644
```

As you can see, the algorithm converged to within $1e^{-5}$ after 21 iterations.

Part 4

In Part 4, we are told that generator 2 active power may vary to maintain an angle difference theta(2) - theta(3) of 0.087266 rad. The active power P_23 that makes this happen is 0.2148pu. But if P_23 is known and P(3) is known, P_13 can be found to be 1.0 - 0.2148 = 0.78519pu. Knowing this allows us to calculate theta_13 as 0.4743 rad. If we pick theta(1) to be 0, then theta(3) is -0.4743 rad.

Step 1: Find Admittance Matrix

```
clc
clear all
% Translate network topology and parameters to matrix form by
% building the admittance matrix Y.

%   Bus types:
%   0   Slack
%   1   PQ
%   2   PV

%        Num  Type  P       Q     theta      V
BusInfo =  [1   0   0       0       0       1.02;
             2   2   0.4     0       0       1.05;
             3   1   -1.0    -0.3    0       0;];

n = size(BusInfo,1);

%         From  To   R       X        B
LineInfo = [1   2   0.10    1.00    0.50;
             1   3   0.05    0.60    0.25;
             2   3   0.05    0.40    0.25;];

% Mutual admittance is found by taking the inverse of the complex impedance
% between each connected pair of buses:
y_mutual = [LineInfo(:,1), LineInfo(:,2), ....
    (1./complex(LineInfo(:,3), LineInfo(:,4)))];

% Self admittance is the sum of the admittances of all lines connected to
% each bus, plus half of the total charging susceptance associated with
% those lines.
```

```
y_self =   [BusInfo(1,1), BusInfo(1,1), ...
    (y_mutual(1,3)+y_mutual(2,3)+ 0.5i.*LineInfo(1,5) + 0.5i.*LineInfo(2,5));
            BusInfo(2,1), BusInfo(2,1), ...
    (y_mutual(1,3)+y_mutual(3,3)+ 0.5i.*LineInfo(1,5) + 0.5i.*LineInfo(3,5));
            BusInfo(3,1), BusInfo(3,1), ...
    (y_mutual(2,3)+y_mutual(3,3)+ 0.5i.*LineInfo(2,5) + 0.5i.*LineInfo(3,5));];

% The admittance matrix is found by vertically concatenating the mutual and
% self admittance matrices as follows:
Y = sparse([y_mutual(:,1); y_self(:,1); y_mutual(:,2)],...
           [y_mutual(:,2); y_self(:,2); y_mutual(:,1)],...
           [-1.*y_mutual(:,3); y_self(:,3); -1.*y_mutual(:,3)]);
```

Step 2: Initialize with Givens and Guesses

```
% Bus 1: slack bus (no equations)
% theta1    := 0
% V1        := 1.02pu

% Bus 2: PV bus
% Find active power mismatch
% P2_sp     := 0.4pu
% V2        := 1.05pu

% Bus 3: PQ bus
% Find active power mismatch
% Find reactive power mismatch
% P3_sp     := -1.0pu
% Q3_sp     := -0.3pu

num_unknowns = 2;
% We need to make an initial guess for each of the unknowns:
theta2_guess = 0.1;
V3_guess = 1.0;%input('Guess for V3: ');

% Use guesses and given information to construct a vector for each variable
% type:
P_sp = [0; 0; -1.0];    % P2 is no longer specified
Q_sp = [0; 0; -0.3];
theta = [0; theta2_guess; -0.47438];    % New angle constraint
V = [1.02; 1.05; V3_guess];  % New voltage constraint

% Instantiate nett active and reactive power vectors:
P_nett(1:n) = 0;
Q_nett(1:n) = 0;
```

```
Jacobian = zeros(num_unknowns); % Fix the size of the Jacobian
mismatch(1:num_unknowns) = 0;
mismatch = mismatch';   % mismatch is a column vector
```

Step 3: Newton-Raphson Implementation

```
tolerance = 1e-5;   % When updates are smaller than this value, stop
                    % iterating.
max_iterations = 25;% Do not perform any more iterations than this.

% Set up matrix to store results:
Result(1:num_unknowns+1,1:max_iterations) = 0;

for j = 1:max_iterations
    % Calculate nett active and reactive power at each bus:
    for x = 1:n
        P_nett(x) = activebalance(x,theta,V,Y,n);
        Q_nett(x) = reactivebalance(x,theta,V,Y,n);
    end

    % Calculate mismatches:
    mismatch(1) = P_sp(3) - P_nett(3);   % Bus 3 active
    mismatch(2) = Q_sp(3) - Q_nett(3); % Bus 3 reactive

    % Use H and N functions to find four parts of the Jacobian matrix:
    H = Jac_H(2,3,theta,V,Y,Q_nett(3));
    N = Jac_N(2,3,theta,V,Y,P_nett(3));
    J = Jac_J(3,3,theta,V,Y,P_nett(3));
    L = Jac_L(3,3,theta,V,Y,Q_nett(3));

    % Piece together Jacobian using four matrices calculated above:
    Jacobian = [H N; J L];

    % Solve for unknowns:
    update = Jacobian\mismatch;

    % Use updated values in next iteration:
    theta(2) = theta(2) + update(1);
    V(3) = V(3)*(1 + update(2));   % Multiply update by V3 because
                                   % algorithm solves for del(V3)/V3

    % Store results for later review:
    Result(1,j) = j;
    Result(2:num_unknowns+1,j) = update;

    % If all update elements are smaller than the tolerance, exit loop:
     if(all(abs(update)<tolerance))
        break
    end
end
disp(Result)
P_nett
Q_nett
theta
V


  Columns 1 through 7

    1.0000    2.0000    3.0000    4.0000    5.0000    6.0000    7.0000
   -0.7912    1.7675   -3.4616   -1.9690   -2.5884   -0.6399    0.1667
   -0.6439   -2.3259    1.7523   -0.8583    6.2124   -0.6238   -0.7995

  Columns 8 through 14

    8.0000    9.0000   10.0000   11.0000   12.0000   13.0000   14.0000
   -3.3014   -0.2549   -3.3706    2.5766   -3.2838   -1.3023    1.8315
   -1.9034   -3.8590    4.2029   -0.3093    0.6134   -0.7104   -1.3958

  Columns 15 through 21

   15.0000   16.0000   17.0000   18.0000   19.0000   20.0000   21.0000
    0.9244    0.6171    0.0327    0.0027    0.0036    0.0002    0.0005
    2.9271    0.4423   -0.0093   -0.0169   -0.0004   -0.0021    0.0001

  Columns 22 through 25

   22.0000   23.0000   24.0000   25.0000
    0.0000    0.0001   -0.0000    0.0000
   -0.0003    0.0000   -0.0000    0.0000


P_nett =

    1.1460   -0.0947   -1.0000


Q_nett =

   -0.1462   -0.0211   -0.3000
```

```
theta =

         0
  -12.9397
   -0.4744


V =

    1.0200
    1.0500
    0.9464
```

Even after 25 iterations, the algorithm was still computing updates greater than $1e^{-5}$. Although the voltage and power results are reasonable, the angle result is clearly undesired, as it violates the initial constraint described in the problem. I attempted to solve for θ_1 and θ_2 by substitution of the initial constraint into the equation $P_{ij} \approx B_{ij}\theta_i j$, but was unable to obtain a reasonable result.

Appendix: MATLAB Functions

activebalance.m

```
function P_i = activebalance(i,theta,V,Y,n)
% This function accepts a row index i and two n-by-1 vectors theta and V.
% It computes active power using the active power balance equation (number
% 1 in Power Flow Equations handout).
G = real(Y);
B = imag(Y);
P_i = 0;

for k = 1:n
    P_i = P_i + V(k)*(G(i,k)*cos(theta(i)-theta(k)) + B(i,k)*sin(theta(i)-theta(k)));
end
P_i = P_i*V(i);
end
```

reactivebalance.m

```
function Q_i = reactivebalance(i,theta,V,Y,n)
% This function accepts a row index i and two n-by-1 vectors theta and V.
% It computes reactive power using the reactive power balance equation (2
% in Power Flow Equations handout).
G = real(Y);
B = imag(Y);
```

```
Q_i = 0;

for k = 1:n
    Q_i = Q_i + V(k)*(G(i,k)*sin(theta(i)-theta(k)) - B(i,k)*cos(theta(i)-theta(k)));
end
Q_i = Q_i*V(i);
end
```

Jac_H.m

```
function H = Jac_H(i,k,theta,V,Y,Q_i)
% This function computes an upper-left Jacobian element "H" given angles,
% voltage, admittance, and reactive power.
G = real(Y);
B = imag(Y);

if i == k
    H = -B(i,i)*(V(i)^2) - Q_i;
else
    H = V(i)*V(k)*(G(i,k)*sin(theta(i)-theta(k)) - B(i,k)*cos(theta(i)-theta(k)));
end
\end{framed}
```

Jac_N.m

```
function N = Jac_N(i,k,theta,V,Y,P_i)
% This function computes an upper-right Jacobian element "H" given angles,
% voltage, admittance, and active power.
G = real(Y);
B = imag(Y);

if i == k
    N = G(i,i)*V(i)^2 + P_i;
else
    N = V(i)*V(k)*(G(i,k)*cos(theta(i)-theta(k)) + B(i,k)*sin(theta(i)-theta(k)));
end
```

Jac_J.m

```
function J = Jac_J(i,k,theta,V,Y,P_i)
% This function computes an upper-right Jacobian element "H" given angles,
% voltage, admittance, and active power.
G = real(Y);
B = imag(Y);

if i == k
    J = -G(i,i)*(V(i)^2) + P_i;
```

```
else
    J = -V(i)*V(k)*(G(i,k)*cos(theta(i)-theta(k)) + B(i,k)*sin(theta(i)-theta(k)));
end
```

Jac_L.m

```
function L = Jac_L(i,k,theta,V,Y,Q_i)
% This function computes an upper-left Jacobian element "H" given angles,
% voltage, admittance, and reactive power.
G = real(Y);
B = imag(Y);

if i == k
    L = -B(i,i)*(V(i)^2) + Q_i;
else
    L = V(i)*V(k)*(G(i,k)*sin(theta(i)-theta(k)) - B(i,k)*cos(theta(i)-theta(k)));
end
```