Computational Methods in Economics Final Project

GitHub Repository:

https://github.com/coquetm/CMiE-2017-Manuel-Coquet (https://github.com/coquetm/CMiE-2017-Manuel-Coquet)

In [1]:
```
using JuMP, Ipopt, Plots, DataFrames, GLPKMathProgInterface, FileIO
gr()
```

```
WARNING: Method definition describe(AbstractArray) in module StatsBa
se at /Users/Manuelcoquet/.julia/v0.5/StatsBase/src/scalarstats.jl:5
73 overwritten in module DataFrames at /Users/Manuelcoquet/.julia/v0
.5/DataFrames/src/abstractdataframe/abstractdataframe.jl:407.
```

Out[1]:
```
Plots.GRBackend()
```

Policy Appications of Power Systems Modelling

Power Flow Algorithms:

- In this project, I start by building Power Flow model for a small network and comparing the computational efficiency of Newton's method and the optimizing package JuMP in Julia.
- Afterwards, I expand the Power Flow model into an Optimal Power Flow to optimize the operation and planning of a power grid based on minimizing generation costs, and to determine Locational Marginal Prices that generators and load consumers face.

Market power regulation:

- I use an Optimal Power Flow model to examine an example of Market Power and the need for regulation due to the incompatibility of incentives between profit-maximizing for a firm and optimizing the welfare of society.
- This is done by determining the social optimal (cost-minimization) grid power flow, and then showing that a generation can withhold generation to manipulate prices to their own benefit.
- At the profit maximization for the generator, I show that there is a substantial deadweight loss to society.

Transmission expansion planning:

- First, I replicate a Mixed Integer Linear (MILP) approach to solving the transmission expansion planning problem from the paper Transmission Expansion Planning: A Mixed-Integer LP Approach (2003) by Natalia Alguacil, Alexis L. Motto and Antonio J. Conejo.
- I apply a DC algorithm without considering losses and another DC LP algorithm linearizing losses to the Garver 6-bus system, and I show that my results are the same as the paper.
- I reach the same conclusions as the author; using a DC model without considering losses to project transmission expansion planning leads to underinvestment - however, computing technology now allows us to approach better solutions using more elaborate algorithms.
- Nonetheless, I also found that considering losses becomes computationally intensive, it takes 25 times more time to solve the model with losses. For larger systems, this may create concerns as computing time rises exponentially.
- Since the DC Algorithm does not guarantee AC Feasibility, I also built a relaxed AC Transmission Expansion Planning model based on the paper Transmission Expansion Planning Using an AC Model: Formulations and Possible Relaxations (2012) by Zhang et al.
- For the AC TEP algorithm, I ran the algorithm in KNITRO in Julia locally - for some reason there is a bug in Jupyter notebook -, but I pasted the code and results (code is also found in github repository).
- AC algorithms do not guarantee global optimum since the problem is non convex. Nonetheless, I show that it is possible to find AC feasible local solutions using a multistart approach at 500 different initial points. The best local solution that I found requires twice as much investment as the DC solution.

1. Power Flow Algorithms

a) AC Power Flow analysis

An alternating current power-flow model is a model used in electrical engineering to analyze power grids. It provides a nonlinear system which describes the energy flow through each transmission line. The goal of a power-flow study is to obtain complete voltages angle and magnitude information for each bus in a power system for specified load and generator real power and voltage conditions.

Once this information is known, real and reactive power flow on each branch as well as generator reactive power output can be analytically determined. Power-flow or load-flow studies are important for planning future expansion of power systems as well as in determining the best operation of existing systems.

Problem description (example)

- Consider a network with 5 buses that forms a cycle (i.e., the lines are (1,2), (2,3), (3,4),(4,5) and (5,1)).
- Assume that Bus 1 is a slack, Bus 2 is PV (generator) and Buses 3-5 are PQ (loads).
- Assume that the resistance and reactance of each line are both equal to 1.

Repeat the following lines 1-3 several times (say 100 times):

- 1: Generate a random vector V such that all voltage magnitudes are somehow close to 1 and all voltage angles are close to 0, and that the phase at the slack bus is zero.
- 2: Based on the random vector of voltages V, compute the loads at the PQ buses, the P and |V| values at the PV buses, and |V| at the slack bus. (constraints/input data needed to run the model)
- 3: Use the measurement data to solve the power flow problem in two ways: (1) Newton's Mehtod, (2) JuMP
- 4: Declare a success if the obtained solution matches the original random state V. Compute how many times each of the above two methods was successful for different values of voltage angles (sensitivity analysis).

NOTE: The problem does not always have a solution, especially at large angles

Mathematical Formulation

The Power flow problem requires us to determine voltages angle and magnitude information for each bus in a power system for specified load and generator real power and voltage conditions:

- For generator buses (also called PV) - we are given P_i & $|V_i|$ -> need to solve for θ_i & Q_i
- For load buses (also called PQ) - we are given P_i & Q_i -> need to solve for θ_i & $|V_i|$
- For the slack bus (used to balance other buses) - we are given $|V_i|$ & θ_i -> need to solve for P_i & Q_i

Therefore our problem can be stated as:

- Known parameters: V_1, θ_1, P_2, V_2, P_3, Q_3, P_4, Q_4, P_5, Q_5
- List of unknowns: P_1, Q_1, Q_2, θ_2, V_3, θ_3, V_4, θ_4, V_5, θ_5
- Need to write equations P_2, P_3, Q_3, P_4, Q_4, P_5, Q_5

Where:

- $P_i = \sum|V_i||V_j|[G_{ij} \cos(\theta_{ij}) + B_{ij} \sin(\theta_{ij})]$
- $Q_i = \sum|V_i||V_j|[G_{ij} \sin(\theta_{ij}) - B_{ij} \cos(\theta_{ij})]$

And:

- $\theta_{ij} = \theta_i - \theta_j$
- $G_{ij} = \text{Real}(Y_{ij})$ # admittance matrix
- $B_{ij} = \text{Imaginary}(Y_{ij})$
- Y_{ij} is the inverse of the impedance matrix which depends on the resistance and reactance of power lines

I will use the equations in terms of vector x to avoid confusion $x = [\theta_2\ \theta_3\ \theta_4\ \theta_5\ V_3\ V_4\ V_5]$

Newton's Method

We will apply a multivariable Newton's method to solve the Non-linear system of power equations according to the following algorithm:

$x = [\theta_2, \theta_3, \theta_4, \theta_5, V_3, V_4, V_5]$

$fx = [P_2(x)-P0_2 , P_3(x)-P0_3 , P_4(x)-P0_4 , P_5(x)-P0_5 , Q_3(x)-Q0_3 , Q_4(x)-Q0_4 , Q_5(x)-Q0_5]$

$J = \delta fx_i / \delta x_j$

while $||fx(x^n)|| < \epsilon$

$$x^{n+1} = x^n - J(x^n)^{-1}f(x^n)$$
$$x^n = x^{n+1}$$

end

Constructing the Admittance Matrix

In [2]:
```
# Since we are not changing resistance or reactance, the admittance ma
trix will always be constant

Zij = zeros(5,5)
Zij = complex(Zij)
Zij[1,2] = 1+1im; Zij[2,1] = 1+1im;
Zij[2,3] = 1+1im; Zij[3,2] = 1+1im;
Zij[3,4] = 1+1im; Zij[4,3] = 1+1im;
Zij[4,5] = 1+1im; Zij[5,4] = 1+1im;
Zij[5,1] = 1+1im; Zij[1,5] = 1+1im;
Y = zeros(5,5);
Y = complex(Y)
for i=1:5, j=1:5
    if i != j
        if Zij[i,j] != 0
            Y[i,j] = -Zij[i,j]^-1;
        else
            Y[i,j] = 0;
        end
    end
end
for i = 1:5
    Y[i,i] = -sum(Y[i,:]);
end
Y
```

Out[2]:
```
5×5 Array{Complex{Float64},2}:
  1.0-1.0im  -0.5+0.5im   0.0+0.0im   0.0+0.0im  -0.5+0.5im
 -0.5+0.5im   1.0-1.0im  -0.5+0.5im   0.0+0.0im   0.0+0.0im
  0.0+0.0im  -0.5+0.5im   1.0-1.0im  -0.5+0.5im   0.0+0.0im
  0.0+0.0im   0.0+0.0im  -0.5+0.5im   1.0-1.0im  -0.5+0.5im
 -0.5+0.5im   0.0+0.0im   0.0+0.0im  -0.5+0.5im   1.0-1.0im
```

Defining a function to evaluate fx (active and reactive power)

In [3]:
```
function feval(Y, V; f0 = zeros(7))

function f(x::Vector)
fx = zeros(7)
θ₁ = 0; #slack reference bus
fx[1] = V[2]*V[1]*(real(Y[2,1])*cos(x[1]-θ₁)+imag(Y[2,1])*sin(x[1]-θ₁)
)+ V[2]*V[2]*real(Y[2,2])+
        V[2]*x[5]*(real(Y[2,3])*cos(x[1]-x[2])+imag(Y[2,3])*sin(x[1]-x
[2])) - f0[1]
fx[2] = x[5]*V[2]*(real(Y[3,2])*cos(x[2]-x[1])+imag(Y[3,2])*sin(x[2]-x
[1]))+ x[5]*x[5]*real(Y[3,3])+
        x[5]*x[6]*(real(Y[3,4])*cos(x[2]-x[3])+imag(Y[3,4])*sin(x[2]-x
[3])) - f0[2]
fx[3] = x[6]*x[5]*(real(Y[4,3])*cos(x[3]-x[2])+imag(Y[4,3])*sin(x[3]-x
[2]))+ x[6]*x[6]*real(Y[4,4])+
        x[6]*x[7]*(real(Y[4,5])*cos(x[3]-x[4])+imag(Y[4,5])*sin(x[3]-x
[4])) - f0[3]
fx[4] = x[7]*x[6]*(real(Y[5,4])*cos(x[4]-x[3])+imag(Y[5,4])*sin(x[4]-x
[3]))+ x[7]*x[7]*real(Y[5,5])+
        x[7]*V[1]*(real(Y[5,1])*cos(x[4]-θ₁)+imag(Y[5,4])*sin(x[4]-θ₁)
) - f0[4]
fx[5] = x[5]*V[2]*(real(Y[3,2])*sin(x[2]-x[1])-imag(Y[3,2])*cos(x[2]-x
[1])) -x[5]*x[5]*imag(Y[3,3])+
        x[5]*x[6]*(real(Y[3,4])*sin(x[2]-x[3])-imag(Y[3,4])*cos(x[2]-x
[3])) - f0[5]
fx[6] = x[6]*x[5]*(real(Y[4,3])*sin(x[3]-x[2])-imag(Y[4,3])*cos(x[3]-x
[2]))-x[6]*x[6]*imag(Y[4,4])+
        x[6]*x[7]*(real(Y[4,5])*sin(x[3]-x[4])-imag(Y[4,5])*cos(x[3]-x
[4])) - f0[6]
fx[7] = x[7]*x[6]*(real(Y[5,4])*sin(x[4]-x[3])-imag(Y[5,4])*cos(x[4]-x
[3]))-x[7]*x[7]*imag(Y[5,5])+
        x[7]*V[1]*(real(Y[5,1])*sin(x[4]-θ₁)-imag(Y[5,1])*cos(x[4]-θ₁)
) - f0[7]
return fx
end

end
```

Defining a function to evaluate Jacobian

I did the analytical Jacobian - tried to use autodiff but i couldn't make it work

In [4]:
```
function Jeval(Y, V)

function J(x::Vector)
θ₁ = 0 #slack reference bus
df1x1 = V[2]*V[1]*(-real(Y[2,1])*sin(x[1]-θ₁)+imag(Y[2,1])*cos(x[1]-θ₁
))+
```

```
            V[2]*x[5]*(-real(Y[2,3])*sin(x[1]-x[2])+imag(Y[2,3])*cos(x
[1]-x[2]))
df1x2 = V[2]*x[5]*(real(Y[2,3])*sin(x[1]-x[2])-imag(Y[2,3])*cos(x[1]-x
[2]))
df1x3 = 0
df1x4 = 0
df1x5 = V[2]*(real(Y[2,3])*cos(x[1]-x[2])+imag(Y[2,3])*sin(x[1]-x[2]))
df1x6 = 0
df1x7 = 0
df2x1 = x[5]*V[2]*(real(Y[3,2])*sin(x[2]-x[1])-imag(Y[3,2])*cos(x[2]-x
[1]))
df2x2 = x[5]*V[2]*(-real(Y[3,2])*sin(x[2]-x[1])+imag(Y[3,2])*cos(x[2]-
x[1]))+
            x[5]*x[6]*(-real(Y[3,4])*sin(x[2]-x[3])+imag(Y[3,4])*cos(x
[2]-x[3]))
df2x3 = x[5]*x[6]*(real(Y[3,4])*sin(x[2]-x[3])-imag(Y[3,4])*cos(x[2]-x
[3]))
df2x4 = 0;
df2x5 = V[2]*(real(Y[3,2])*cos(x[2]-x[1])+imag(Y[3,2])*sin(x[2]-x[1]))
+
            2*x[5]*real(Y[3,3])+x[6]*(real(Y[3,4])*cos(x[2]-x[3])+imag
(Y[3,4])*sin(x[2]-x[3]))
df2x6 = x[5]*(real(Y[3,4])*cos(x[2]-x[3])+imag(Y[3,4])*sin(x[2]-x[3]))
df2x7 = 0
df3x1 = 0
df3x2 = x[6]*x[5]*(real(Y[4,3])*sin(x[3]-x[2])-imag(Y[4,3])*cos(x[3]-x
[2]))
df3x3 = x[6]*x[5]*(-real(Y[4,3])*sin(x[3]-x[2])+imag(Y[4,3])*cos(x[3]-
x[2]))+
            x[6]*x[7]*(-real(Y[4,5])*sin(x[3]-x[4])+imag(Y[4,5])*cos(x
[3]-x[4]))
df3x4 = x[6]*x[7]*(real(Y[4,5])*sin(x[3]-x[4])-imag(Y[4,5])*cos(x[3]-x
[4]))
df3x5 = x[6]*(real(Y[4,3])*cos(x[3]-x[2])+imag(Y[4,3])*sin(x[3]-x[2]))
df3x6 = x[5]*(real(Y[4,3])*cos(x[3]-x[2])+imag(Y[4,3])*sin(x[3]-x[2]))
+
            2*x[6]*real(Y[4,4])+x[7]*(real(Y[4,5])*cos(x[3]-x[4])+imag
(Y[4,5])*sin(x[3]-x[4]))
df3x7 = x[6]*(real(Y[4,5])*cos(x[3]-x[4])+imag(Y[4,5])*sin(x[3]-x[4]))
df4x1 = 0
df4x2 = 0
df4x3 = x[7]*x[6]*(real(Y[5,4])*sin(x[4]-x[3])-imag(Y[5,4])*cos(x[4]-x
[3]))
df4x4 = x[7]*x[6]*(-real(Y[5,4])*sin(x[4]-x[3])+imag(Y[5,4])*cos(x[4]-
x[3]))+
            x[7]*V[1]*(-real(Y[5,1])*sin(x[4]-θ₁)+imag(Y[5,1])*cos(x[4
]-θ₁))
df4x5 = 0
df4x6 = x[7]*(real(Y[5,4])*cos(x[4]-x[3])+imag(Y[5,4])*sin(x[4]-x[3]))
df4x7 = x[6]*(real(Y[5,4])*cos(x[4]-x[3])+imag(Y[5,4])*sin(x[4]-x[3]))
+
            2*x[7]*real(Y[5,5])+V[1]*(real(Y[5,1])*cos(x[4]-θ₁)+imag(Y
[5,1])*sin(x[4]-θ₁))
```

```
df5x1 = x[5]*V[2]*(-real(Y[3,2])*cos(x[2]-x[1])-imag(Y[3,2])*sin(x[2]-
x[1]))
df5x2 = x[5]*V[2]*(real(Y[3,2])*cos(x[2]-x[1])+imag(Y[3,2])*sin(x[2]-x
[1]))+
                        x[5]*x[6]*(real(Y[3,4])*cos(x[2]-x[3])+imag(Y[
3,4])*sin(x[2]-x[3]))
df5x3 = x[5]*x[6]*(-real(Y[3,4])*cos(x[2]-x[3])-imag(Y[3,4])*sin(x[2]-
x[3]))
df5x4 = 0;
df5x5 = V[2]*(real(Y[3,2])*sin(x[2]-x[1])-imag(Y[3,2])*cos(x[2]-x[1]))
-
            2*x[5]*imag(Y[3,3])+x[6]*(real(Y[3,4])*sin(x[2]-x[3])-imag
(Y[3,4])*cos(x[2]-x[3]))
df5x6 = x[5]*(real(Y[3,4])*sin(x[2]-x[3])-imag(Y[3,4])*cos(x[2]-x[3]))
df5x7 = 0
df6x1 = 0
df6x2 = x[6]*x[5]*(-real(Y[4,3])*cos(x[3]-x[2])-imag(Y[4,3])*sin(x[3]-
x[2]))
df6x3 = x[6]*x[5]*(real(Y[4,3])*cos(x[3]-x[2])+imag(Y[4,3])*sin(x[3]-x
[2]))+
            x[6]*x[7]*(real(Y[4,5])*cos(x[3]-x[4])+imag(Y[4,5])*sin(x[
3]-x[4]))
df6x4 = x[6]*x[7]*(-real(Y[4,5])*cos(x[3]-x[4])-imag(Y[4,5])*sin(x[3]-
x[4]))
df6x5 = x[6]*(real(Y[4,3])*sin(x[3]-x[2])-imag(Y[4,3])*cos(x[3]-x[2]))
df6x6 = x[5]*(real(Y[4,3])*sin(x[3]-x[2])-imag(Y[4,3])*cos(x[3]-x[2]))
-
            2*x[6]*imag(Y[4,4])+x[7]*(real(Y[4,5])*sin(x[3]-x[4])-imag
(Y[4,5])*cos(x[3]-x[4]))
df6x7 = x[6]*(real(Y[4,5])*sin(x[3]-x[4])-imag(Y[4,5])*cos(x[3]-x[4]))
df7x1 = 0
df7x2 = 0
df7x3 = x[7]*x[6]*(-real(Y[5,4])*cos(x[4]-x[3])-imag(Y[5,4])*sin(x[4]-
x[3]))
df7x4 = x[7]*x[6]*(real(Y[5,4])*cos(x[4]-x[3])+imag(Y[5,4])*sin(x[4]-x
[3]))+
            x[7]*V[1]*(real(Y[5,1])*cos(x[4]-θ₁)+imag(Y[5,1])*sin(x[4]
-θ₁))
df7x5 = 0
df7x6 = x[7]*(real(Y[5,4])*sin(x[4]-x[3])-imag(Y[5,4])*cos(x[4]-x[3]))
df7x7 = x[6]*(real(Y[5,4])*sin(x[4]-x[3])-imag(Y[5,4])*cos(x[4]-x[3]))
-
            2*x[7]*imag(Y[5,5])+V[1]*(real(Y[5,1])*sin(x[4]-θ₁)-imag(Y
[5,1])*cos(x[4]-θ₁))
Jx = [
    df1x1 df1x2 df1x3 df1x4 df1x5 df1x6 df1x7;
    df2x1 df2x2 df2x3 df2x4 df2x5 df2x6 df2x7;
    df3x1 df3x2 df3x3 df3x4 df3x5 df3x6 df3x7;
    df4x1 df4x2 df4x3 df4x4 df4x5 df4x6 df4x7;
    df5x1 df5x2 df5x3 df5x4 df5x5 df5x6 df5x7;
    df6x1 df6x2 df6x3 df6x4 df6x5 df6x6 df6x7;
    df7x1 df7x2 df7x3 df7x4 df7x5 df7x6 df7x7;
]
```

```
return Jx

end

end
```

Out[4]: Jeval (generic function with 1 method)

Computing the constraints at the PQ & PV buses and the slack bus [P0 & Q0]

In [5]:
```
V = 0.9+0.2*rand(5,1) # random number between 0.9-1.1
θ = (-2+2*2*randn(5,1))*(π/180) # random angle between -2º to 2º
V[1] = 1 # slack bus voltage magnitude (reference)
θ[1] = 0 # slack bus voltage angle (reference)
```

Out[5]: 0

In [6]:
```
f0 = feval(Y,V) # generates a function to evaluate power flows in term
s of x
x_true = [θ[2]; θ[3]; θ[4]; θ[5]; V[3]; V[4]; V[5]]
constraints = f0(x_true) # computes constraints based on initial vecto
r x0
```

Out[6]:
```
7-element Array{Float64,1}:
 -0.102491
  0.00939592
  0.22077
 -0.199723
  0.0704396
 -0.00706236
 -0.0374352
```

Solution with Newton's algorithm

In [7]:
```
function PF_newton(feval, Jeval, V, θ, Y; x0 = [0,0,0,0,1,1,1],tol = 1
e-9, xtol = 1e-4, max_iter = 15)

x_true = [θ[2]; θ[3]; θ[4]; θ[5]; V[3]; V[4]; V[5]] # original x vecto
r (angles in radians)

f0 = feval(Y,V) # generates a function to evaluate power flows in term
s of x
constraints = f0(x_true) # computes constraints based on initial vecto
r x0

# compute new fx function incorporating constraints and Jacobian
fx = feval(Y,V, f0 = constraints)
Jx = Jeval(Y,V)

#initialize algorithm
x = x0

for i = 1:max_iter
    xn = x - Jx(x)\fx(x)
    x = xn

# Was a solution found?
if sum(fx(x).^2) < tol
    break
end

end

# Is it the correct soultion? It is possible to have more than 1 local
solution
if sum((x-x_true).^2) < xtol
    success_newt = true
else
    success_newt = false
end

    return success_newt, x
end
```

Out[7]: PF_newton (generic function with 1 method)

In [8]:
```
(success_newt, x_newt) = PF_newton(feval, Jeval, V, θ, Y)
```

Out[8]: (true,[-0.0668451,-0.0210587,0.0740505,-0.0482533,1.06591,1.0778,0.9
02977])

Solving with JuMP

In [9]:
```
function PF_JuMP(feval, V, θ, Y; xtol = 1e-4)
```

```
x_true = [θ[2]; θ[3]; θ[4]; θ[5]; V[3]; V[4]; V[5]] # original x vecto
r (angles in radians)

# JuMP does not recognize real and imaginary functions so I have to de
fine the Real and Imaginary Parts of Y
YR = real(Y); YI = imag(Y); θ₁ = 0;

f0 = feval(Y,V) # generates a function to evaluate power flows in term
s of x
constraints = f0(x_true) # computes constraints based on initial vecto
r x0

# Initialize Ipopt Solver
m = Model(solver=IpoptSolver(print_level=0))

# Define our x variable
@variable(m, x[1:7])

# Set initial guess
for i in 1:4
    setvalue(x[i], 0.0)
end

for i in 5:7
    setvalue(x[i], 1.0)
end


@NLobjective(m, Max, 10) #doesn't matter we can write constant

# Active Power & Reactive Power non-linear constraints

@NLconstraints(m, begin

    V[2]*V[1]*((YR[2,1])*cos(x[1]-θ₁)+(YI[2,1])*sin(x[1]-θ₁))+ V[2]*V[
2]*(YR[2,2])+
        V[2]*x[5]*((YR[2,3])*cos(x[1]-x[2])+(YI[2,3])*sin(x[1]-x[2]))
== constraints[1]

    x[5]*V[2]*((YR[3,2])*cos(x[2]-x[1])+(YI[3,2])*sin(x[2]-x[1]))+ x[5
]*x[5]*(YR[3,3])+
        x[5]*x[6]*((YR[3,4])*cos(x[2]-x[3])+(YI[3,4])*sin(x[2]-x[3]))
== constraints[2]

    x[6]*x[5]*((YR[4,3])*cos(x[3]-x[2])+(YI[4,3])*sin(x[3]-x[2]))+ x[6
]*x[6]*(YR[4,4])+
        x[6]*x[7]*((YR[4,5])*cos(x[3]-x[4])+(YI[4,5])*sin(x[3]-x[4]))
== constraints[3]

    x[7]*x[6]*((YR[5,4])*cos(x[4]-x[3])+(YI[5,4])*sin(x[4]-x[3]))+ x[7
]*x[7]*(YR[5,5])+
        x[7]*V[1]*((YR[5,1])*cos(x[4]-θ₁)+(YI[5,4])*sin(x[4]-θ₁)) == c
```

```
onstraints[4]

    x[5]*V[2]*((YR[3,2])*sin(x[2]-x[1])-(YI[3,2])*cos(x[2]-x[1]))- x[5
]*x[5]*(YI[3,3])+
        x[5]*x[6]*((YR[3,4])*sin(x[2]-x[3])-(YI[3,4])*cos(x[2]-x[3]))
== constraints[5]

    x[6]*x[5]*((YR[4,3])*sin(x[3]-x[2])-(YI[4,3])*cos(x[3]-x[2]))- x[6
]*x[6]*(YI[4,4])+
        x[6]*x[7]*((YR[4,5])*sin(x[3]-x[4])-(YI[4,5])*cos(x[3]-x[4]))
== constraints[6]

    x[7]*x[6]*((YR[5,4])*sin(x[4]-x[3])-(YI[5,4])*cos(x[4]-x[3]))- x[7
]*x[7]*(YI[5,5])+
        x[7]*V[1]*((YR[5,1])*sin(x[4]-θ₁)-(YI[5,1])*cos(x[4]-θ₁)) == c
onstraints[7]

end)

solve(m; suppress_warnings=true)
x_JuMP = getvalue(x);

# Is it the correct soultion? It is possible to have more than 1 local
solution
if sum((x_JuMP-x_true).^2) < xtol
    success_JuMP = true
else
    success_JuMP = false
end
    return success_JuMP, x_JuMP
end
```

In [10]:
```
(success_JuMP, x_JuMP) = PF_JuMP(feval, V, θ, Y)
```

```
******************************************************************
*********
This program contains Ipopt, a library for large-scale nonlinear opt
imization.
 Ipopt is released as open source code under the Eclipse Public Lice
nse (EPL).
         For more information visit http://projects.coin-or.org/Ipop
t
******************************************************************
*********
```

Results for one iteration

In [11]:
```
Results = DataFrame(x_true = x_true, x_newton = x_newt, x_JuMP = x_JuM
P)
```

Out[11]:

	x_true	x_newton	x_JuMP
1	-0.06684882189500609	-0.06684508743839755	-0.0668488218693944
2	-0.02106193511931002	-0.021058678619364824	-0.02106193509550892
3	0.0740475729233907	0.07405051177019993	0.07404757294163533
4	-0.048255612601959666	-0.04825331623893488	-0.0482556125837295
5	1.065908563663313	1.0659109831139861	1.0659085636796146
6	1.0777992053983179	1.0778038828253786	1.077799205431248
7	0.9029709317281985	0.902976977170994	0.9029709317787932

In [12]:
```
@time PF_JuMP(feval, V, θ, Y)
```

```
  0.003943 seconds (3.43 k allocations: 252.500 KB)
```

Out[12]:
```
(true,[-0.0668488,-0.0210619,0.0740476,-0.0482556,1.06591,1.0778,0.9
02971])
```

In [13]:
```
@time PF_newton(feval,Jeval, V, θ, Y) # although it took me ages to do
the Jacobian manually
```

```
  0.000203 seconds (525 allocations: 80.766 KB)
```

Out[13]:
```
(true,[-0.0668451,-0.0210587,0.0740505,-0.0482533,1.06591,1.0778,0.9
02977])
```

Sensitivity Analysis for voltage angle range

In [14]:

```
for i = 5:10:45

θ_val = i;
s_newt = 0;
s_JuMP = 0;

# perform 100 iterations for each angle range
for k=1:100
    V = 0.9+0.2*rand(5,1) # random number between 0.9-1.1
    θ = (-θ_val+2*θ_val*randn(5,1))*(π/180) # random angle between -5º to 5º
    V[1] = 1 # slack bus voltage magnitude (reference)
    θ[1] = 0 # slack bus voltage angle (reference)

    (success_JuMP, x_JuMP) = PF_JuMP(feval, V, θ, Y)
    (success_newt, x_newt) = PF_newton(feval,Jeval, V, θ, Y)

    if success_JuMP == true
        s_JuMP = s_JuMP +1;
    end
    if success_newt == true
        s_newt = s_newt +1;
    end
end

println("With θ = -$θ_val degrees to $θ_val degrees")
println("Newton success rate is $s_newt")
println("JuMP success rate is $s_JuMP")
println("_________________________________________")
end
```

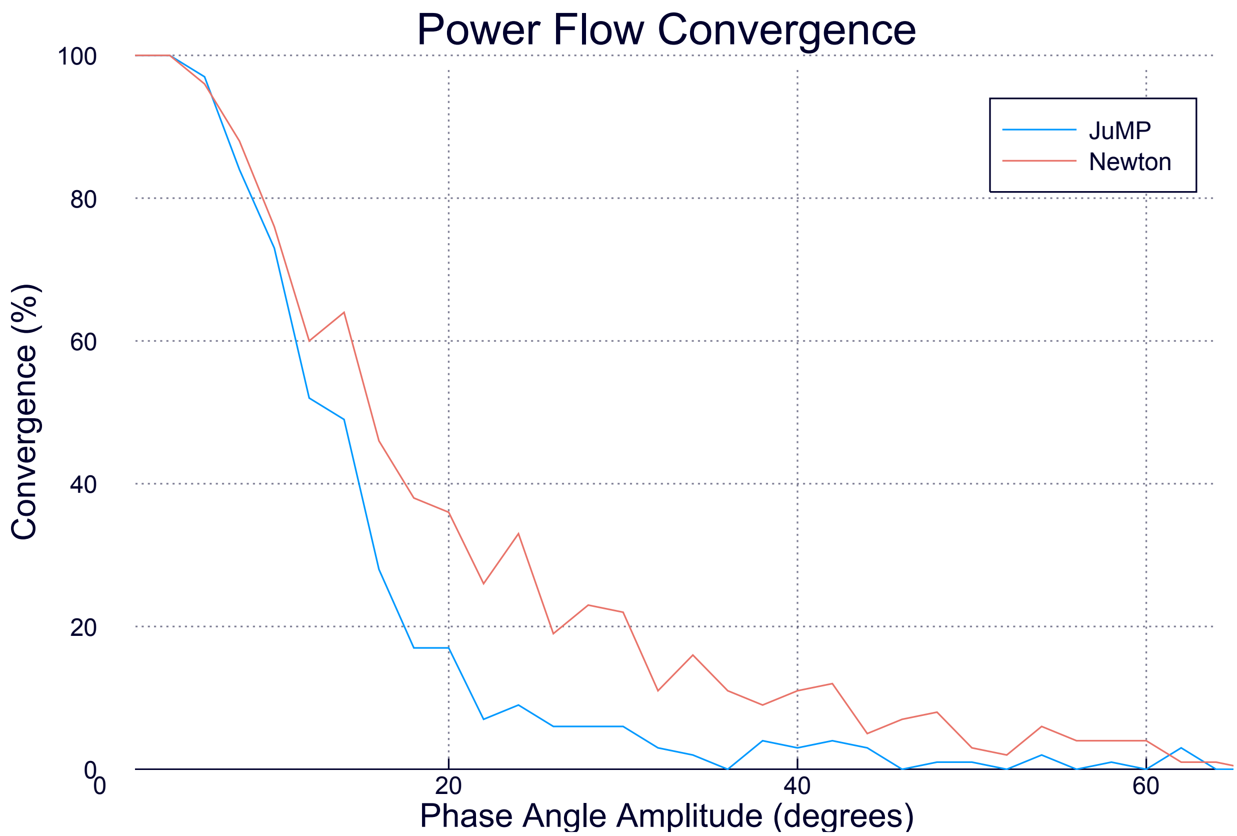
```
With θ = -5 degrees to 5 degrees
Newton success rate is 100
JuMP success rate is 100
_________________________________________
With θ = -15 degrees to 15 degrees
Newton success rate is 44
JuMP success rate is 28
_________________________________________
With θ = -25 degrees to 25 degrees
Newton success rate is 25
JuMP success rate is 12
_________________________________________
With θ = -35 degrees to 35 degrees
Newton success rate is 11
JuMP success rate is 4
_________________________________________
With θ = -45 degrees to 45 degrees
Newton success rate is 4
JuMP success rate is 4
_________________________________________
```

In [15]:

```
# We can also plot the results

angle_amplitude = Int64[]
ss_JuMP = Int64[]
ss_newt = Int64[]

for i = 2:2:90

θ_val = i;
s_newt = 0;
s_JuMP = 0;

# perform 100 iterations for each angle range
for k=1:100
    V = 0.9+0.2*rand(5,1) # random number between 0.9-1.1
    θ = (-θ_val+2*θ_val*randn(5,1))*(π/180) # random angle between -5º
to 5º
    V[1] = 1 # slack bus voltage magnitude (reference)
    θ[1] = 0 # slack bus voltage angle (reference)

    (success_JuMP, x_JuMP) = PF_JuMP(feval, V, θ, Y)
    (success_newt, x_newt) = PF_newton(feval,Jeval, V, θ, Y)

    if success_JuMP == true
        s_JuMP = s_JuMP +1
    end
    if success_newt == true
        s_newt = s_newt +1
    end
end

push!(angle_amplitude, i)
push!(ss_JuMP, s_JuMP)
push!(ss_newt, s_newt)

end
```

In [16]:
```
plot(angle_amplitude,ss_JuMP,label = "JuMP", xlims = (0, 65), title =
"Power Flow Convergence")
plot!(angle_amplitude, ss_newt, label = "Newton", xlabel = "Phase Angl
e Amplitude (degrees)", ylabel = "Convergence (%)")
```

Out[16]:



We can see that the Power Flow algorithm has a higher convergence rate at small voltage angles (θ) and that Newton't method is superior in convergence and time to JuMP using Ipopt

b) AC Optimal Power Flow analysis

Optimal Power Flow (OPF) is an expansion of power flow analysis in which power flow are optimized in order to minimize the cost of generation subject to the power flow constraints and other operational constraints, such as generator minimum output constraints, transmission stability and voltage constraints, and limits on switching mechanical equipment.

Equality constraints

- Power balance at each node - power flow equations

Inequality constraints

- Network operating limits (line flows, voltages)
- Limits on control variables

Solving an OPF is necessary to determine the optimal operation and planning of the grid. In this algorithm, I will simulate an optimal power flow model for a 6-bus system and determine the locational marginal prices (LMPs) and generator power flows.

- Slack: Bus 1
- PV(Generators): Buses 2,3
- PQ(Load): Buses 4,5,6

Locational Marginal Prices (LMPs)

Locational marginal pricing is a way for wholesale electric energy prices to reflect the value of electric energy at different locations, accounting for the patterns of load, generation, and the physical limits of the transmission system.

LMPs are the marginal costs of serving an extra MW-h of electricity at a specific node at a given time. They are composed of energy costs + losses + congestion. LMPs are calculated every five minutes and they are used to settle contracts in energy markets and to deal with transmission congestion.

Problem Set up

Objective: minimize $\sum f_i P_i$ (generators)

Power flow equations:

- $P_i = \sum|V_i||V_j|[G_{ij} \cos(\theta_{ij}) + B_{ij} \sin(\theta_{ij})]$
- $Q_i = \sum|V_i||V_j|[G_{ij} \sin(\theta_{ij}) - B_{ij} \cos(\theta_{ij})]$

Cost functions (generators):

- $f_i = a(P)^2 + b(P) + C$
- a = [0.11, 0.085, 0.1225]
- b = [5, 1.2, 1]
- c = [150, 600, 335]

Constraints:

- Generators & Slack
  - $\theta_1 = 0$ # reference
  - $0.9 <= |V_1| <= 1.1$
  - $0.9 <= |V_2| <= 1.1$
  - $0.9 <= |V_3| <= 1.1$
  - $10 <= P_1 <= 200$
  - $10 <= P_2 <= 100$
  - $10 <= P_3 <= 300$
  - $-300 <= Q_1 <= 300$
  - $-300 <= Q_2 <= 300$
  - $-300 <= Q_3 <= 300$
- Loads
  - $0.9 <= |V_4| <= 1.1$
  - $0.9 <= |V_5| <= 1.1$
  - $0.9 <= |V_6| <= 1.1$
  - $P_4 = -90$
  - $P_5 = -100$
  - $P_6 = -125$
  - $Q_4 = -30$
  - $Q_5 = -35$
  - $Q_6 = -50$
- Lines
  - $S_{14} <= 95$

Constructing the Admittance Matrix

In [17]:
```
res = 0.015*ones(6)
react = 0.01*ones(6)
baseMVA = 100.
Zij = zeros(6,6)
Zij = complex(Zij)
Zij[1,4] = (1/baseMVA)*complex(res[1],react[1]); Zij[4,1] = (1/baseMVA
)*complex(res[1],react[1]);
Zij[4,5] = (1/baseMVA)*complex(res[2],react[2]); Zij[5,4] = (1/baseMVA
)*complex(res[2],react[2]);
Zij[3,5] = (1/baseMVA)*complex(res[3],react[3]); Zij[5,3] = (1/baseMVA
)*complex(res[3],react[3]);
Zij[5,6] = (1/baseMVA)*complex(res[4],react[4]); Zij[6,5] = (1/baseMVA
)*complex(res[4],react[4]);
Zij[6,2] = (1/baseMVA)*complex(res[5],react[5]); Zij[2,6] = (1/baseMVA
)*complex(res[5],react[5]);
Zij[6,4] = (1/baseMVA)*complex(res[6],react[6]); Zij[4,6] = (1/baseMVA
)*complex(res[6],react[6]);

Y = zeros(6,6)
Y = complex(Y)

for i=1:6, j=1:6
    if i != j
        if Zij[i,j] != 0
            Y[i,j] = -Zij[i,j]^-1
        else
            Y[i,j] = 0.0
        end
    end
end
for i = 1:6
    Y[i,i] = -sum(Y[i,:])
end
Y = (Y+conj(Y'))/2;
Y
```

Out[17]:
```
6×6 Array{Complex{Float64},2}:
  4615.38-3076.92im       0.0+0.0im      …       0.0+0.0im
      0.0+0.0im       4615.38-3076.92im     -4615.38+3076.92im
      0.0+0.0im           0.0+0.0im              0.0+0.0im
 -4615.38+3076.92im       0.0+0.0im         -4615.38+3076.92im
      0.0+0.0im           0.0+0.0im         -4615.38+3076.92im
      0.0+0.0im      -4615.38+3076.92im  …   13846.2-9230.77im
```

We set uo new x-vector

$x = [\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, V_1, V_2, V_3, V_4, V_5, V_6]$

Power Flow Model for the new system

In [18]:
```
# generator buses starting at bus 1
vconstraints = [1., 1., 1.]
vlow = 0.9
vhigh = 1.1

# start at bus 2
pconstraints = [150, 75, -90, -100, -125]

# start at bus 4
qconstraints = [-30, -35, -50]

# JuMP does not recognize real and imaginary functions so I have to de
fine the Real and Imaginary Parts of Y
YR = real(Y); YI = imag(Y);

# Initialize Ipopt Solver
m = Model(solver=IpoptSolver(print_level=2))

# Define our x variable
@variable(m, x[1:12])

# Set initial guess
for i in 1:6
    setvalue(x[i], 0.0)
end

for i in 7:12
    setvalue(x[i], 1.0)
end

@NLobjective(m, Max, 10) #doesn't matter we can write as constant

# Voltage magnitude and voltage angle linear constraints

# slackbus reference angle
@constraint(m, x[1] == 0)

# generator voltage constraints
@constraint(m, vconstr[i=1:3], x[i+6] == vconstraints[i])
@constraint(m, vmin[i=1:6], x[i+6] >= vlow)
@constraint(m, vhig[i=1:6], x[i+6] <= vhigh)

# Active Power & Reactive Power non-linear constraints

#active power constraints
@NLconstraint(m, pconstr[i=1:5], sum(x[i+7]*x[j+6]*(YR[i+1,j]*cos(x[i+
1]-x[j])+
            YI[i+1,j]*sin(x[i+1]-x[j])) for j = 1:6) == pconstraints[i
])
```

```
#reactive power constraints
@NLconstraint(m, qconstr[i=1:3], sum(x[i+9]*x[j+6]*(YR[i+3,j]*sin(x[i+
3]-x[j])-
            YI[i+3,j]*cos(x[i+3]-x[j])) for j = 1:6) == qconstraints[i
])

solve(m)
x_PF = getvalue(x)
```

Optimal Power Flow

- We add the corresponding constraints and minimization function

Constraints

In [19]:
```
# All buses starting at bus 1
vlow = 0.9
vhigh = 1.1

#Generators (buses 1-3 - includes slack)

# generators cost coefficients
a = [0.11, 0.085, 0.1225]
b = [5, 1.2, 1.]
c = [150., 600., 335.];

# active power
pgenmin = [10,10,10]
pgenmax = [200,100,300]

# reactive power
qgenmin = [-300,-300,-300]
qgenmax = [300,300,300]

# Loads (buses 4-6)
plconstraints = [-90, -100, -125]
qlconstraints = [-30, -35, -50]

# Lines

l14constraint = 95
```

Solver

In [20]:

```
# Initialize Ipopt Solver
m = Model(solver=IpoptSolver(print_level=0))

# Define our x variable
@variable(m, x[1:12])

# Set initial guess
for i in 1:6
    setvalue(x[i], 0.0)
end

for i in 7:12
    setvalue(x[i], 1.0)
end

# Define power expression P = f(x)
@NLexpression(m, pi[i=1:3], sum(x[i+6]*x[j+6]*(YR[i,j]*cos(x[i]-x[j])+
            YI[i,j]*sin(x[i]-x[j])) for j = 1:6))

# minimize generation costs
@NLobjective(m, Min, sum(a[i]*pi[i]^2+b[i]*pi[i]+c[i] for i = 1:3))

# slackbus reference angle
@constraint(m, x[1] == 0)

# All buses starting at bus 1

@constraint(m, vmin[i=1:6], x[i+6] >= vlow)
@constraint(m, vhig[i=1:6], x[i+6] <= vhigh)

#Generators (buses 1-3 - includes slack)

#active power
@NLconstraint(m, pgenminJP[i=1:3], sum(x[i+6]*x[j+6]*(YR[i,j]*cos(x[i]
-x[j])+
            YI[i,j]*sin(x[i]-x[j])) for j = 1:6) >= pgenmin[i])
@NLconstraint(m, pgenmaxJP[i=1:3], sum(x[i+6]*x[j+6]*(YR[i,j]*cos(x[i]
-x[j])+
            YI[i,j]*sin(x[i]-x[j])) for j = 1:6) <= pgenmax[i])

#reactive power
@NLconstraint(m, qgenminJP[i=1:3], sum(x[i+6]*x[j+6]*(YR[i,j]*sin(x[i]
-x[j])-
            YI[i,j]*cos(x[i]-x[j])) for j = 1:6) >= qgenmin[i])
@NLconstraint(m, qgenmaxJP[i=1:3], sum(x[i+6]*x[j+6]*(YR[i,j]*sin(x[i]
-x[j])-
            YI[i,j]*cos(x[i]-x[j])) for j = 1:6) <= qgenmax[i])



#Loads (buses 4-6)

#active power constraints
@NLconstraint(m, plconstr[i=1:3], sum(x[i+9]*x[j+6]*(YR[i+3,j]*cos(x[i
+3]-x[j])+
```

```
             YI[i+3,j]*sin(x[i+3]-x[j])) for j = 1:6) == plconstraints[
i])

#reactive power constraints
@NLconstraint(m, qlconstr[i=1:3], sum(x[i+9]*x[j+6]*(YR[i+3,j]*sin(x[i
+3]-x[j])-
             YI[i+3,j]*cos(x[i+3]-x[j])) for j = 1:6) == qlconstraints[
i])

#line constraint

@NLexpression(m, p14, x[1+6]*x[1+6]*YR[1,1] + x[1+6]*x[4+6]*(YR[1,4]*c
os(x[1]-x[4])+YI[1,4]*sin(x[1]-x[4])))
@NLexpression(m, q14, -x[1+6]*x[1+6]*YI[1,1] + x[1+6]*x[4+6]*(YR[1,4]*
sin(x[1]-x[4])-YI[1,4]*cos(x[1]-x[4])))

# line constraint must be in terms of complex power
@NLexpression(m, s14sq, p14^2+q14^2)

@NLconstraint(m, l14constrJP, s14sq <= l14constraint^2)

solve(m)
x_OPF = getvalue(x)
```

Out[20]:
```
12-element Array{Float64,1}:
 -5.60637e-33
 -0.00823517
 -0.00424296
 -0.00709477
 -0.00820755
 -0.00927922
  1.08973
  1.0946
  1.1
  1.07602
  1.07814
  1.07556
```

In [21]:
```
getobjectivevalue(m)
```

Out[21]:
```
5570.047199958679
```

Extracting relevant results

In Optimal Power Flow, even though the output you get from the model is the phase angles and voltage magnitude, those are usually not the results that we are interested in. The results that we are more interested in are the active and reactive power flows of generators and the Locational Marginal Prices that loads and generators face. However we have to calculate such values.

Active and Reactive Power Flows

In [22]:
```
v_θ_jump = x_OPF[1:6]; v_jump = x_OPF[7:12]

P_jp = zeros(6,6); Q_jp = zeros(6,6);
P_jump = zeros(6); Q_jump = zeros(6);

for i = 1:6, j = 1:6
    P_jp[i,j] = v_jump[i]*v_jump[j]*(YR[i,j]*cos(v_θ_jump[i]-v_θ_jump[
j]) +
        YI[i,j]*sin(v_θ_jump[i]-v_θ_jump[j]))
    Q_jp[i,j] = v_jump[i]*v_jump[j]*(YR[i,j]*sin(v_θ_jump[i]-v_θ_jump[
j]) -
        YI[i,j]*cos(v_θ_jump[i]-v_θ_jump[j]))
end

for i = 1:6
    P_jump[i] = sum(P_jp[i,:])
    Q_jump[i] = sum(Q_jp[i,:])
end

[P_jump Q_jump]
```

Out[22]:
```
6×2 Array{Float64,2}:
   94.6902    7.66648
  100.0      58.4722
  125.513    52.33
  -90.0     -30.0
 -100.0     -35.0
 -125.0     -50.0
```

Locational Marginal Prices

Locational marginal prices are harder to calculate because they are related to the lagrangian and KKT multipliers from the constraints. However, JuMP is really smart and solves the dual constraint problem simultaneously.

- For loads, LMPs are the langrange multipliers of the active power constraint
  - LMP = λ_p
- For generators, LMPs are the langrange multipliers of the active power constraint + KKT multipliers of Pmin and Pmax constraints
  - LMP = λ_p - μ_l + μ_h
  - where λ_p = Marginal cost = MC(P^{opt})
  - MC(P) = 2*a*P + b

In [23]:
```
LMP_jump = zeros(6)

# For loads
LMP_jump[4:6] = abs(getdual(plconstr))

# For generators

μ_pmin = getdual(pgenminJP)
μ_pmax = abs(getdual(pgenmaxJP))

for i = 1:3
    LMP_jump[i] = 2*a[i]*P_jump[i]+b[i]-μ_pmin[i]+μ_pmax[i]
end

LMP_jump
```

Out[23]:
```
6-element Array{Float64,1}:
 25.8318
 32.1407
 31.7506
 32.872
 32.8045
 32.9602
```

Results JuMP

In [24]:
```
JuMP_r = DataFrame(Bus = 1:6, Voltage_pu = v_jump, θ_deg = v_θ_jump*18
0/π, P_MW = P_jump, Q_MW = Q_jump,
    LMP_JuMP = LMP_jump)
```

Out[24]:

	Bus	Voltage_pu	θ_deg	P_MW	Q_MW
1	1	1.089729748544749	-3.212213280053105e-31	94.690153166669	7.66647739
2	2	1.0946022290235418	-0.4718404170299756	99.99999996753832	58.4721767
3	3	1.1000000098582219	-0.24310393216302847	125.51278928858392	52.3299741
4	4	1.0760193219271599	-0.40650023875230284	-89.99999999972351	-30.000000(
5	5	1.0781358323379941	-0.47025781502015285	-99.99999999949068	-35.000000(
6	6	1.0755573417968547	-0.5316603601786397	-124.99999999947795	-50.000000(

Results Matlab (attached in Github)

- I performed the same simulation in matlab using commercial software Matpower and obtained the following results:

In [25]:
```
v_mat = [1.090,1.095,1.100,1.076, 1.078,1.076]; v_θ = [0.000,-0.472,-0
.243,-0.407, -0.470,-0.532];
p_mat = [94.69,100.00,125.51,-90.00,-100.00,-125.00]; q_mat = [7.67,58
.47,52.33,-30.00, -35.00,-50.00];
LMP_mat = [25.832,32.141,31.751,32.872,32.805,32.960]
Matlab = DataFrame(Bus = 1:6, Voltage_pu = v_mat, θ_deg = v_θ, P_MW =
p_mat, Q_MW = q_mat, LMP_Matlab = LMP_mat)
```

Out[25]:

	Bus	Voltage_pu	θ_deg	P_MW	Q_MW	LMP_Matlab
1	1	1.09	0.0	94.69	7.67	25.832
2	2	1.095	-0.472	100.0	58.47	32.141
3	3	1.1	-0.243	125.51	52.33	31.751
4	4	1.076	-0.407	-90.0	-30.0	32.872
5	5	1.078	-0.47	-100.0	-35.0	32.805
6	6	1.076	-0.532	-125.0	-50.0	32.96

Comparing Matpower from Matlab, we can see that the results are the same and the model works

2. Market Power

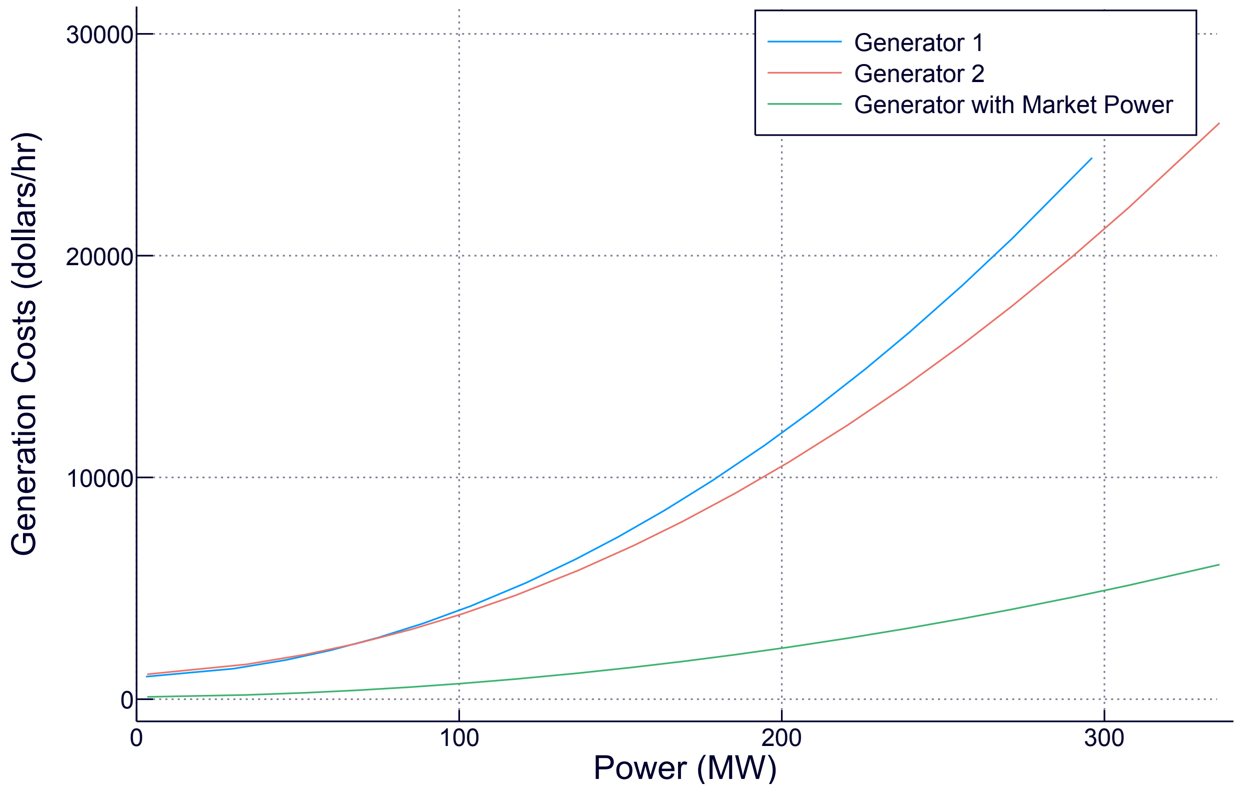
- In this exercise, I will show that when a generator can exert market power (manipulate prices)-large generator with cheaper generation costs-, they will seek to maximize profit and produce at levels suboptimal for society.
- When left unregulated, generators with market power will withhold capacity to maximize profit, causing them to produce away from society's optimal point. This will create Deadweight Loss (DWL) that will be paid by ratepayers, justifying regulating generators with market power in markets.

In order for this problem to work I will adjust the generator costs as shown below:

In [26]:

```
# generator cost functions
f1(x) = 0.25*x^2 + 5*x + 1000
f2(x) = 0.20*x^2 + 7*x + 1100
f3(x) = 0.05*x^2 + 1*x + 100

plot(f1,0,300, label = "Generator 1", xlabel = "Power (MW)", ylabel =
"Generation Costs (dollars/hr)",
 xlims = (0, 340),ylims = (-1000,33000))
plot!(f2, label = "Generator 2")
plot!(f3, label = "Generator with Market Power")
```

Out[26]:



The steps to show the impact of market power will be the following:

- Build an AC OPF algorithm
- Determine the optimal social point by optimizing the power system without restrictions - optimal power flows, generator profit and systemwide costs.
- Run power flow simulations withholding generation capacity from the generator with market power
- Calculate the new optimal power flow, generator profit, systemwide costs and DWL
- Determine the optimal operation point for the generator based on profit maximizing
- Determine the associated DWL to society based on the generator profit-maximizing

Market Power function

- Define a function that takes as input the Power of the generator with market power and calculates system total costs, locational marginal prices and active power flows

In [27]:
```
function mrkt_power(P_cheap)

# All buses starting at bus 1
vlow = 0.9
vhigh = 1.1

#Generators (buses 1-3 - includes slack)

# generators cost coefficients
a = [0.25, 0.20, 0.05]
b = [5., 7., 1.]
c = [1000., 1100., 100.]

# active power
pgenmin = [10,10,10]
pgenmax = [600,600,P_cheap]

# reactive power
qgenmin = [-300,-300,-300]
qgenmax = [300,300,300]

# Loads (buses 4-6)
plconstraints = [-150, -150, -150]
qlconstraints = [-30, -35, -50]

l14constraint = 95

# Initialize Ipopt Solver
m = Model(solver=IpoptSolver(print_level=0))

# Define our x variable
@variable(m, x[1:12])

# Set initial guess
for i in 1:6
    setvalue(x[i], 0.0)
end

for i in 7:12
    setvalue(x[i], 1.0)
end

# Define power expression P = f(x)
@NLexpression(m, pi[i=1:3], sum(x[i+6]*x[j+6]*(YR[i,j]*cos(x[i]-x[j])+
            YI[i,j]*sin(x[i]-x[j])) for j = 1:6))

# minimize generation costs
@NLobjective(m, Min, sum(a[i]*pi[i]^2+b[i]*pi[i]+c[i] for i = 1:3))
```

```
# slackbus reference angle
@constraint(m, x[1] == 0)

# All buses starting at bus 1

@constraint(m, vmin[i=1:6], x[i+6] >= vlow)
@constraint(m, vhig[i=1:6], x[i+6] <= vhigh)

#Generators (buses 1-3 - includes slack)

#active power
@NLconstraint(m, pgenminJP[i=1:3], sum(x[i+6]*x[j+6]*(YR[i,j]*cos(x[i]
-x[j])+
            YI[i,j]*sin(x[i]-x[j])) for j = 1:6) >= pgenmin[i])
@NLconstraint(m, pgenmaxJP[i=1:3], sum(x[i+6]*x[j+6]*(YR[i,j]*cos(x[i]
-x[j])+
            YI[i,j]*sin(x[i]-x[j])) for j = 1:6) <= pgenmax[i])

#reactive power
@NLconstraint(m, qgenminJP[i=1:3], sum(x[i+6]*x[j+6]*(YR[i,j]*sin(x[i]
-x[j])-
            YI[i,j]*cos(x[i]-x[j])) for j = 1:6) >= qgenmin[i])
@NLconstraint(m, qgenmaxJP[i=1:3], sum(x[i+6]*x[j+6]*(YR[i,j]*sin(x[i]
-x[j])-
            YI[i,j]*cos(x[i]-x[j])) for j = 1:6) <= qgenmax[i])


#Loads (buses 4-6)

#active power constraints
@NLconstraint(m, plconstr[i=1:3], sum(x[i+9]*x[j+6]*(YR[i+3,j]*cos(x[i
+3]-x[j])+
            YI[i+3,j]*sin(x[i+3]-x[j])) for j = 1:6) == plconstraints[
i])

#reactive power constraints
@NLconstraint(m, qlconstr[i=1:3], sum(x[i+9]*x[j+6]*(YR[i+3,j]*sin(x[i
+3]-x[j])-
            YI[i+3,j]*cos(x[i+3]-x[j])) for j = 1:6) == qlconstraints[
i])

#line constraint

@NLexpression(m, p14, x[1+6]*x[1+6]*YR[1,1] + x[1+6]*x[4+6]*(YR[1,4]*c
os(x[1]-x[4])+YI[1,4]*sin(x[1]-x[4])))
@NLexpression(m, q14, -x[1+6]*x[1+6]*YI[1,1] + x[1+6]*x[4+6]*(YR[1,4]*
sin(x[1]-x[4])-YI[1,4]*cos(x[1]-x[4])))

# line constraint must be in terms of complex power
@NLconstraint(m, s14sq, p14^2+q14^2 <= l14constraint^2)

solve(m)
```

```
x_OPF = getvalue(x)

v_θ_jump = x_OPF[1:6]; v_jump = x_OPF[7:12]

P_jp = zeros(6,6); Q_jp = zeros(6,6);
P_jump = zeros(6); Q_jump = zeros(6);

for i = 1:6, j = 1:6
    P_jp[i,j] = v_jump[i]*v_jump[j]*(YR[i,j]*cos(v_θ_jump[i]-v_θ_jump[
j]) +
        YI[i,j]*sin(v_θ_jump[i]-v_θ_jump[j]))
    Q_jp[i,j] = v_jump[i]*v_jump[j]*(YR[i,j]*sin(v_θ_jump[i]-v_θ_jump[
j]) -
        YI[i,j]*cos(v_θ_jump[i]-v_θ_jump[j]))
end

for i = 1:6
    P_jump[i] = sum(P_jp[i,:])
    Q_jump[i] = sum(Q_jp[i,:])
end

LMP_jump = zeros(6)

# For loads
LMP_jump[4:6] = abs(getdual(plconstr))

# For generators

μ_pmin = getdual(pgenminJP)
μ_pmax = abs(getdual(pgenmaxJP))

for i = 1:3
    LMP_jump[i] = 2*a[i]*P_jump[i]+b[i]-μ_pmin[i]+μ_pmax[i]
end

obj = getobjectivevalue(m)

return P_jump, Q_jump, LMP_jump, obj, x_OPF

end
```

```
Out[27]: mrkt_power (generic function with 1 method)
```

Determine the Social Optimal Scenario

In [28]:
```
(P_opt, Q_opt, LMP_opt, optval)  = mrkt_power(600)

gen_p = round(P_opt[3],2)
optval = round(optval,2)

println("The optimal production of the generator with market power is $gen_p MW")
println("The total cost of generation the $optval dollars/hr")
println("The Deadweight loss is 0 dollars/hr")
```

```
The optimal production of the generator with market power is 328.14
MW
```

Run the simulation with different levels of production for generator with market power

In [29]:
```
P_iter = 10:2:328

length = size(P_iter)[1]

P_run = zeros(length,6)
Q_run = zeros(length, 6)
LMP_run = zeros(length, 6)
obj_run = zeros(length)

c = 0

for i in P_iter
    c = c + 1
    (P, Q, LMP, obj)  = mrkt_power(i)
    P_run[c,:] = P
    Q_run[c,:] = Q
    LMP_run[c,:] = LMP
    obj_run[c] = obj
end
```

- Determine the costs, revenue & profit of Generator with market power at different levels of production along with the associated DWL

In [30]:

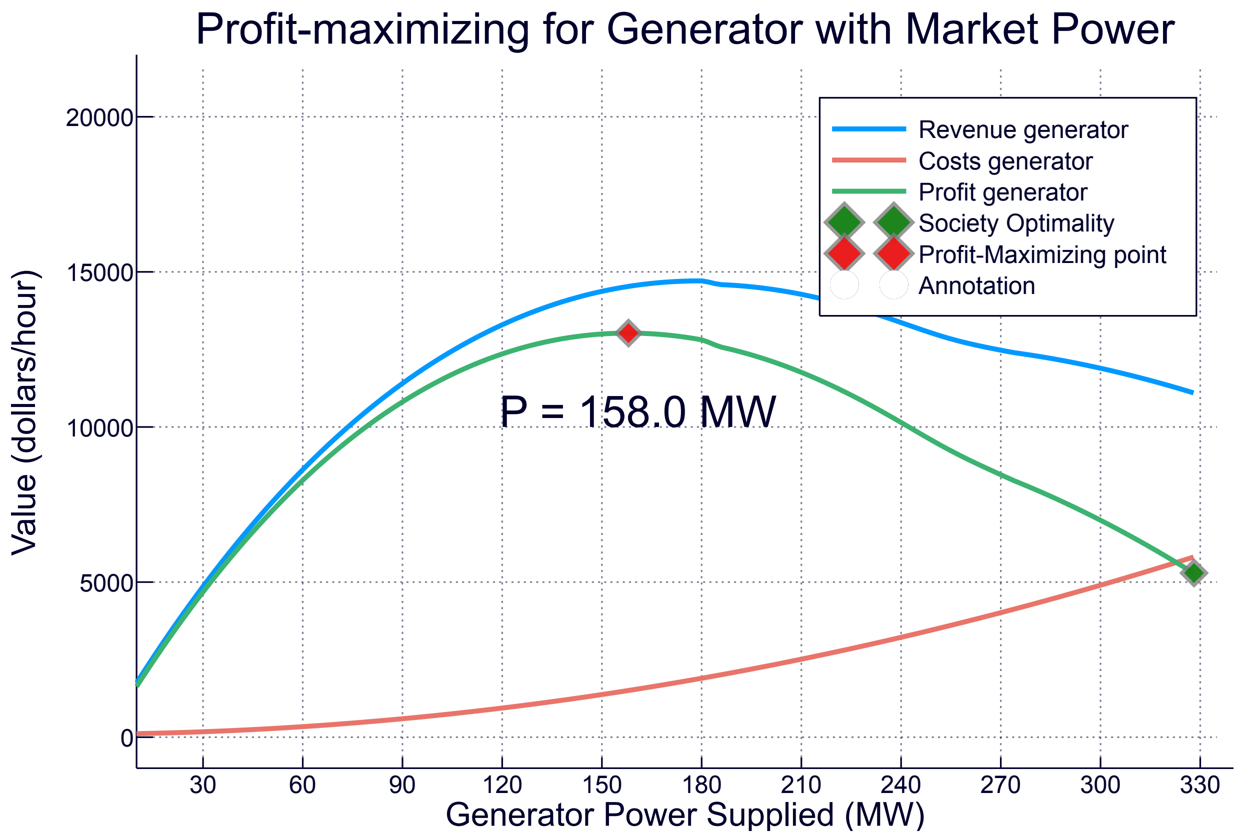
```
# Revenue = PQ = locational marginal prices x production level of gene
rator
rev_gen = LMP_run[:,3].*P_run[:,3]

# Cost = generation cost function evaluated at production level
costs_gen = f3.(P_run[:,3]);

# Profit = Revenue - Costs
profit_gen = rev_gen - costs_gen

# Optimal generation point = profit maximizing generation level
gen_opt_ind = indmax(profit_gen)
gen_opt_P = round(P_run[gen_opt_ind,3], 2)

# DWL = associated system-wide costs - system-wide costs at society's
optimal point
gen_opt_DWL = round(obj_run[gen_opt_ind]-optval,2);
```

Profit Maximization of generator with market power

In [31]:

```
plot(P_run[:,3],rev_gen, label = "Revenue generator",xlims = (10, 340)
,ylims = (-1000,22000),
    xticks = 30:30:330, yticks = 0:5000:25000, xlabel = "Generator Pow
er Supplied (MW)", ylabel = "Value (dollars/hour)",
    w = 3, title = "Profit-maximizing for Generator with Market Power"
)
plot!(P_run[:,3],costs_gen, label = "Costs generator", w = 3)
plot!(P_run[:,3],profit_gen, label = "Profit generator", w = 3)
scatter!([P_opt[3]],[profit_gen[end]],color=[:green],marker=([:d],6,0.
8,stroke(3,:gray)), label = "Society Optimality")
scatter!([gen_opt_P],[profit_gen[gen_opt_ind]],color=[:red],marker=([:
d],6,0.8,stroke(3,:gray)),
    label = "Profit-Maximizing point")
scatter!([gen_opt_P+3], [10_500], series_annotations = ["P = $gen_opt_
P MW"], markersize = 0, color = [:white]
    , label = "Annotation")
```

Out[31]:

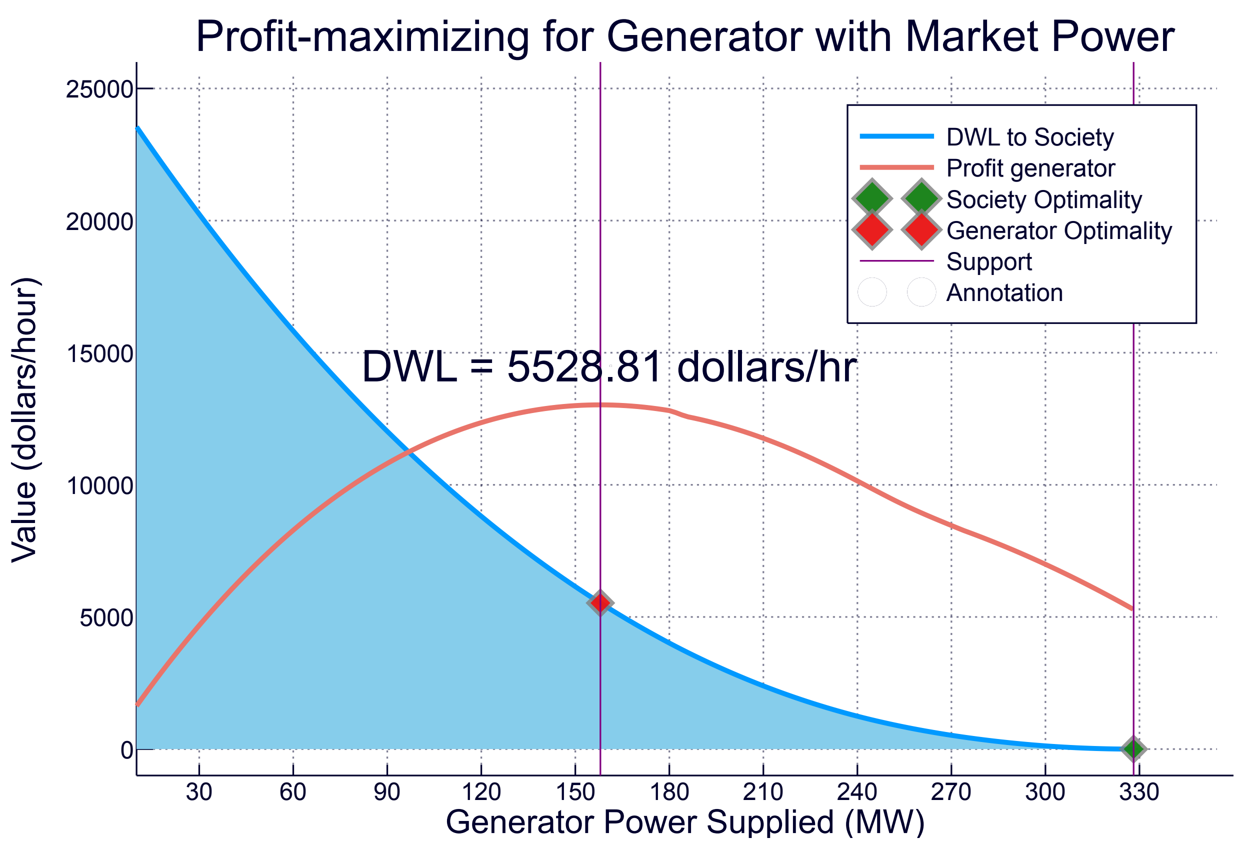


We can see that the profit maximizing function dictates that the generator only produce 158 MW of electricity

DWL associated with generator's optimal production

In [32]:
```
default(legend=true)
plot(P_run[:,3],obj_run-optval, label = "DWL to Society",xlims = (10,
360),ylims = (-1000,26000),
    xticks = 30:30:330, yticks = 0:5000:25000, xlabel = "Generator Pow
er Supplied (MW)", ylabel = "Value (dollars/hour)",
    w = 3, title = "Profit-maximizing for Generator with Market Power"
,fill=(0,:skyblue))
plot!(P_run[:,3],rev_gen-costs_gen, label = "Profit generator", w = 3)
scatter!([P_opt[3]],[0],color=[:green],marker=([:d],6,0.8,stroke(3,:gr
ay)), label = "Society Optimality")
scatter!([gen_opt_P],[gen_opt_DWL],color=[:red],marker=([:d],6,0.8,str
oke(3,:gray)), label = "Generator Optimality")
vline!([gen_opt_P, P_opt[3]],color = [:purple], label = "Support")
scatter!([gen_opt_P+3], [14_500], series_annotations = ["DWL = $gen_op
t_DWL dollars/hr"], markersize = 0, color = [:white]
, label = "Annotation")
```

Out[32]:



- We can see a market flaw because at the profit maximizing point of the generator with market power, society incurs in a deadweight loss of 5528.81 dollars per hour compared to social optimality

3. Transmission Expansion Planning (Mixed Integer Programming)

The transmission planning process must identify and support development of transmission infrastructure that is sufficiently robust and can enable competition among wholesale capacity and energy suppliers in energy markets. However, it is a very complex mathematical problem. In this project, I replicate algorithms that try to tackle this problem through linearizations and relaxations.

The TEP algorithms I will replicate come from the Paper:

- Transmission Expansion Planning: A Mixed-Integer LP Approach (2003) by Natalia Alguacil, Alexis L. Motto and Antonio J. Conejo

- This paper presents a mixed-integer LP approach to the solution of the long-term transmission expansion planning problem
- In general, this problem is large-scale, mixed-integer, nonlinear, and nonconvex. The authors derive a mixed-integer linear formulation that considers losses and guarantees convergence to optimality using existing optimization software
- The proposed model is applied to Garver's 6-bus system, the IEEE Reliability Test System, and a realistic Brazilian system. However, I only apply the algorithm to Garver's 6-bus system.

TEP Algorithm (Original Model)

Objective: minimize { $\sigma \sum \lambda_j^G P_j^G + K_{stk} w_{stk}$ }

σ - weighting factor to make investment and operational costs comparable

λ_j^G - locational marginal prices

K_{stk} - Investment cost of constructing line in corridor (s,t)

w_{stk} - Binary variable that equals 1 if line k from (s,t) corridor is built and equals 0 otherwise

Constraints:

- $\sum P_j^G - P_s = \sum P_s^D$ for every bus s
- $P_s = \sum P_{stk} = \sum [f_{stk} + 1/2\, q_{stk}]$ for every line connected to bus s and for every bus s
- $f_{stk} = -b_{stk} w_{stk} \sin(\delta_s - \delta_t)$ for every line
- $q_{stk} = 2 g_{stk} w_{stk} [1 - \cos(\delta_s - \delta_t)]$ for every line
- $\max(P_{stk}, P_{tsk}) <= P_{stk}^{max}$ for every line
- $0 <= P_j^G <= P_j^{Gmax}$ for every every unit j
- $w_{stk} = 1$ for every existing line that is not prospective
- $w_{stk} \in \{0,1\}$ for every line

b_{stk} - Susceptance of line k in corridor (s,t)

g_{stk} - conductance of line k in corridor (s,t)

q_{stk} - Power losses in line k of corridor (s,t)

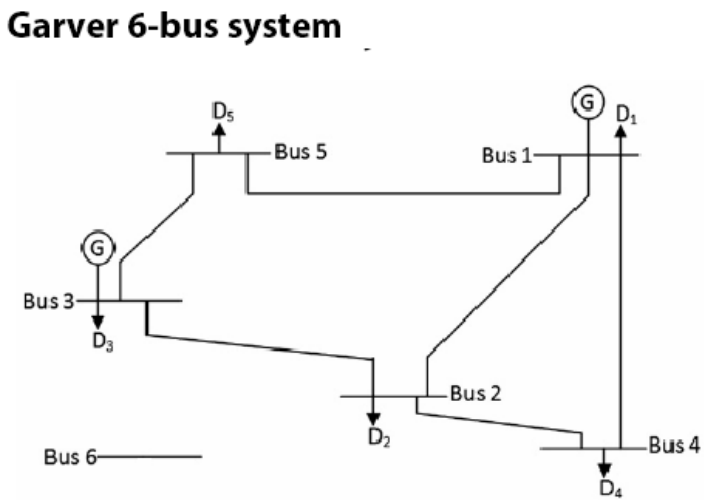
f_{stk} - Lossless power flow in line k of corridor (s,t)

δ_s - Voltage angle at bus s

Gaver 6-bus system

In [33]:
```
load("Garver.png")
```

Out[33]:



In [34]:
```
# Line data for Garver 6 bus example
Line_ID = 1:15;
Corridor = ["1-2","1-3","1-4","1-5","1-6","2-3","2-4","2-5","2-6","3-4
","3-5","3-6","4-5","4-6","5-6"]
Resistance = [0.10,0.09,0.15,0.05,0.17,0.05,0.10,0.08,0.08,0.15,0.05,0
.12,0.16,0.08,0.15]
Reactance = [0.40,0.38,0.60,0.20,0.68,0.20,0.40,0.31,0.30,0.59,0.20,0.
48,0.63,0.30,0.61]
Line_cost = [40.,38,60,20,68,20,40,31,30,59,20,48,63,30,61]
Capacity_MW = [100,100,80,100,70,100,100,100,100,82,100,100,75,100,78]

Garver_line = DataFrame(Line_ID = Line_ID, Corridor = Corridor, Resist
ance = Resistance,
    Reactance = Reactance, Investment_Cost = round(Line_cost), Capacit
y_MW = Capacity_MW)
```

Out[34]:

	Line_ID	Corridor	Resistance	Reactance	Investment_Cost	Capacity_MW
1	1	1-2	0.1	0.4	40.0	100
2	2	1-3	0.09	0.38	38.0	100
3	3	1-4	0.15	0.6	60.0	80
4	4	1-5	0.05	0.2	20.0	100
5	5	1-6	0.17	0.68	68.0	70
6	6	2-3	0.05	0.2	20.0	100
7	7	2-4	0.1	0.4	40.0	100
8	8	2-5	0.08	0.31	31.0	100
9	9	2-6	0.08	0.3	30.0	100
10	10	3-4	0.15	0.59	59.0	82
11	11	3-5	0.05	0.2	20.0	100
12	12	3-6	0.12	0.48	48.0	100
13	13	4-5	0.16	0.63	63.0	75
14	14	4-6	0.08	0.3	30.0	100
15	15	5-6	0.15	0.61	61.0	78

In [35]:

```
# Existing Lines Garver
Ex_ID = [1,3,4,6,7,11]
Ex_lines = DataFrame(Line_ID = Line_ID[Ex_ID], Corridor = Corridor[Ex_
ID],
    Resistance = Resistance[Ex_ID], Reactance = Reactance[Ex_ID], Capa
city_MW = Capacity_MW[Ex_ID])
```

Out[35]:

	Line_ID	Corridor	Resistance	Reactance	Capacity_MW
1	1	1-2	0.1	0.4	100
2	3	1-4	0.15	0.6	80
3	4	1-5	0.05	0.2	100
4	6	2-3	0.05	0.2	100
5	7	2-4	0.1	0.4	100
6	11	3-5	0.05	0.2	100

In [36]:
```
# Bus data for Garver 6 bus example
Bus_ID = 1:6;
PG_max = [150.,0,360,0,0,600]
LMP_G = [10.,0,20,0,0,30]
PD = [80,240,40,160,240,0]

Garver_bus = DataFrame(Bus_ID = Bus_ID, PG_max_MW = PG_max, LMP_G = LM
P_G, PD_MW = PD)
```

Out[36]:

	Bus_ID	PG_max_MW	LMP_G	PD_MW
1	1	150.0	10.0	80
2	2	0.0	0.0	240
3	3	360.0	20.0	40
4	4	0.0	0.0	160
5	5	0.0	0.0	240
6	6	600.0	30.0	0

DC Model Without Losses

- I will start by solving the following DC Model without losses for Garver's 6-bus system:

Algorithm (DC Model without Losses)

Objective: minimize { $\sigma \sum \lambda_j^G P_j^G + K_{stk} w_{stk}$ }

σ - weighting factor to make investment and operational costs comparable

λ_j^G - locational marginal prices

K_{stk} - Investment cost of constructing line in corridor (s,t)

w_{stk} - Binary variable that equals 1 if line k from (s,t) corridor is built and equals 0 otherwise

Constraints

- $P_s^G - \sum P_{stk} = P_s^D$ for every bus s
- $-w_{stk} P_{stk}^{max} <= P_{stk} <= w_{stk} P_{stk}^{max}$: for every line
- $0 <= P_j^G <= P_j^{Gmax}$: for every every unit j
- $w_{stk} = 1$: for every existing line that is not prospective
- $w_{stk} \in \{0,1\}$: for every line

In [37]:

```
# line data
C1 = [1,1,1,1,1,2,2,2,2,3,3,3,4,4,5]
C2 = [2,3,4,5,6,3,4,5,6,4,5,6,5,6,6]

# resized line data vectors (45-long vectors)
Line_ID_jump = 1:45
Corridor_jump = [Corridor;Corridor;Corridor]
lcost_jump = [Line_cost;Line_cost;Line_cost]
Capacity_MW_jump = [Capacity_MW;Capacity_MW;Capacity_MW]
C1_jump = [C1;C1;C1]
C2_jump = [C2;C2;C2]

# NPV factor for 20 years with 10% discount rate to make fixed and ope
rating costs compareable
df = 0.0
for i = 1:20
    df += 1/(1.1)^i
end
fy = 1/df

# bus data
generators = [1,3,6]
loads = [2,4,5];
```

In [38]:
```
# sets of bus and neighboring line IDs
bus_1_c1 = Int64[]; bus_2_c1 = Int64[]; bus_3_c1 = Int64[]; bus_4_c1 =
Int64[]; bus_5_c1 = Int64[]; bus_6_c1 = Int64[]
bus_1_c2 = Int64[]; bus_2_c2 = Int64[]; bus_3_c2 = Int64[]; bus_4_c2 =
Int64[]; bus_5_c2 = Int64[]; bus_6_c2 = Int64[]

for i = 1:45
    if C1_jump[i] == 1
        bus_1_c1 = push!(bus_1_c1,i)
    elseif C1_jump[i] == 2
        bus_2_c1 = push!(bus_2_c1,i)
    elseif C1_jump[i] == 3
        bus_3_c1 = push!(bus_3_c1,i)
    elseif C1_jump[i] == 4
        bus_4_c1 = push!(bus_4_c1,i)
    elseif C1_jump[i] == 5
        bus_5_c1 = push!(bus_5_c1,i)
     elseif C1_jump[i] == 6
        bus_6_c1 = push!(bus_6_c1,i)
    end
end

for i = 1:45
    if C2_jump[i] == 1
        bus_1_c2 = push!(bus_1_c2,i)
    elseif C2_jump[i] == 2
        bus_2_c2 = push!(bus_2_c2,i)
    elseif C2_jump[i] == 3
        bus_3_c2 = push!(bus_3_c2,i)
    elseif C2_jump[i] == 4
        bus_4_c2 = push!(bus_4_c2,i)
    elseif C2_jump[i] == 5
        bus_5_c2 = push!(bus_5_c2,i)
     elseif C2_jump[i] == 6
        bus_6_c2 = push!(bus_6_c2,i)
    end
end
```

In [39]:
```
# Set of lines going from bus s
bus_mat_c1 = [bus_1_c1,bus_2_c1,bus_3_c1,bus_4_c1,bus_5_c1,bus_6_c1]
```

Out[39]:
```
6-element Array{Array{Int64,1},1}:
 [1,2,3,4,5,16,17,18,19,20,31,32,33,34,35]
 [6,7,8,9,21,22,23,24,36,37,38,39]
 [10,11,12,25,26,27,40,41,42]
 [13,14,28,29,43,44]
 [15,30,45]
 Int64[]
```

In [40]:
```
# Set of lines going into bus s
bus_mat_c2 = [bus_1_c2,bus_2_c2,bus_3_c2,bus_4_c2,bus_5_c2,bus_6_c2]
```

Out[40]:
```
6-element Array{Array{Int64,1},1}:
 Int64[]
 [1,16,31]
 [2,6,17,21,32,36]
 [3,7,10,18,22,25,33,37,40]
 [4,8,11,13,19,23,26,28,34,38,41,43]
 [5,9,12,14,15,20,24,27,29,30,35,39,42,44,45]
```

In [41]:
```
# activate a mixed integer-linear solver

function DC_no_loss()

m = Model(solver = GLPKSolverMIP())

@variable(m, w[1:45], Bin) # 45 variables because each line can be bui
lt up to 3 times
@variable(m, pst[1:45]) # lineflow as variables

# Set a constraint for exisiting lines
@constraint(m, exconstr[i=1:6], w[Ex_ID[i]] == 1)

# Calculate an expression for the power flows through each bus
@expression(m, pf[i=1:6], sum(pst[j] for j in bus_mat_c1[i])-sum(pst[j
] for j in bus_mat_c2[i]))

# Calculate an expression for the power generated
@expression(m, pg[i=1:6], PD[i]+pf[i])

# Define the minimization objective
@expression(m, gen_costs, sum(pg[i]*LMP_G[i] for i = 1:6))
@expression(m, line_costs, sum(w[i]*lcost_jump[i] for i = 1:45))

@objective(m, Min, line_costs + gen_costs*(fy*8760/10^6))

# Generator flow constraints
@constraint(m, gen[i=1:6], 0 <= pg[i] <= PG_max[i])

# Line flow constraints
@expression(m, min_flow[i=1:45], -w[i]*Capacity_MW_jump[i])
@expression(m, max_flow[i=1:45], w[i]*Capacity_MW_jump[i])

@constraint(m, min_flow_cstr[i=1:45], pst[i] >= min_flow[i])
@constraint(m, max_flow_cstr[i=1:45], pst[i] <= max_flow[i])

# Load constraints
@constraint(m, load_balance[i = 1:6] , pg[i] - pf[i] == PD[i])

solve(m)

W_jump_nl = getvalue(w)
pst_jump_nl = getvalue(pst)
pg_jump_nl = getvalue(pg)
line_costs_jump_nl = getvalue(line_costs)

obj_nl = getobjectivevalue(m)

return W_jump_nl, pst_jump_nl, pg_jump_nl, line_costs_jump_nl, obj_nl

end
```

Out[41]:
```
DC_no_loss (generic function with 1 method)
```

In [42]:
```
(W_jump_nl, pst_jump_nl, pg_jump_nl, line_costs_jump_nl, obj_nl) = DC_
no_loss();
```

Results

In [43]:
```
nl_lines = find(W_jump_nl[1:end])
nl_new_lines = setdiff(nl_lines,Ex_ID)
n_lines_DC_wo_loss = size(nl_new_lines)[1]
println("the number of new lines required is $n_lines_DC_wo_loss")
```

```
the number of new lines required is 4
```

In [44]:
```
time_nl = @elapsed DC_no_loss();
@time DC_no_loss();
```

```
  0.043428 seconds (5.64 k allocations: 404.297 KB)
```

Additional lines built

In [45]:
```
Ex_lines = DataFrame(Line_ID = Line_ID_jump[nl_new_lines], Corridor =
Corridor_jump[nl_new_lines],
    Capacity_MW = Capacity_MW_jump[nl_new_lines], Investment_Cost = lc
ost_jump[nl_new_lines])
```

Out[45]:

	Line_ID	Corridor	Capacity_MW	Investment_Cost
1	14	4-6	100	30.0
2	29	4-6	100	30.0
3	41	3-5	100	20.0
4	44	4-6	100	30.0

Line Flows

In [46]:
```
DataFrame(Line_ID = Line_ID_jump[nl_lines], Coridor = Corridor_jump[nl
_lines], Line_flow_MW = pst_jump_nl[nl_lines])
```

Out[46]:

	Line_ID	Coridor	Line_flow_MW
1	1	1-2	40.0
2	3	1-4	-10.0
3	4	1-5	40.0
4	6	2-3	-100.0
5	7	2-4	-100.0
6	11	3-5	100.0
7	14	4-6	-70.0
8	29	4-6	-100.0
9	41	3-5	100.0
10	44	4-6	-100.0

Power summary for each bus

In [47]:
```
DataFrame(Bus_ID = Bus_ID, PG_MW = pg_jump_nl, LMP_G = LMP_G, PD_MW =
PD)
```

Out[47]:

	Bus_ID	PG_MW	LMP_G	PD_MW
1	1	150.0	10.0	80
2	2	0.0	0.0	240
3	3	340.0	20.0	40
4	4	0.0	0.0	160
5	5	0.0	0.0	240
6	6	270.0	30.0	0

Investment Costs

In [48]:
```
inv_nl = line_costs_jump_nl - sum(lcost_jump[Ex_ID])
gen_nl = obj_nl-inv_nl
println("The total investment cost of new lines is $inv_nl M dollars")
```

```
The total investment cost of new lines is 110.0 M dollars
```

Algorithm (Linearized LP Model with Losses)

Objective: minimize { $\sigma \sum \lambda_j^G P_j^G + K_{stk} w_{stk}$ }

σ - weighting factor to make investment and operational costs comparable

λ_j^G - locational marginal prices

K_{stk} - Investment cost of constructing line in corridor (s,t)

w_{stk} - Binary variable that equals 1 if line k from (s,t) corridor is built and equals 0 otherwise

Constraints:

- $\sum P_j^G - \sum [\, f_{stk} + 1/2\ q_{stk} \,] = P_s^D$: for every bus s
- $-w_{stk}P_{stk}^{max} <= f_{stk} <= w_{stk}P_{stk}^{max}$: for every line
- $-(1 - w_{stk})\, M_{st} <= f_{stk}/b_{stk} + (\delta_{st}^{+} - \delta_{st}^{-}) <= (1 - w_{stk})\, M_{st}$: for every line
- $0 <= q_{stk} <= w_{stk}P_{stk}^{max}$: for every line
- $0 <= -q_{stk}/g_{stk} + \sum \alpha_{st}(x)\, \delta_{st}(x) <= (1 - w_{stk})\, M_{st}^2$: for every line
- $\delta_{st}^{+} + \delta_{st}^{-} = \sum \delta_{st}(x)$: for every line
- $\delta_s - \delta_t = \delta_{st}^{+} + \delta_{st}^{-}$: for every line
- $f_{stk} + 1/2\ q_{stk} <= P_{stk}^{max}$: for every line
- $-f_{stk} + 1/2\ q_{stk} <= P_{stk}^{max}$: for every line
- $0 <= P_j^G <= P_j^{Gmax}$: for every every unit j
- $w_{stk} = 1$: for every existing line that is not prospective
- $w_{stk} \in \{0,1\}$: for every line
- $\delta_s = 0$: reference bus
- $\delta_{st}^{+} >= 0$; $\delta_{st}^{-} >= 0$: for every line
- $\delta_{st}(x) >= 0$: for every line and for every piecewise segment
- $\delta_{st}(x) <= \Delta\delta_{st} + (1-w_{stk})\, M_{st}$: for every line and for every piecewise segment

b_{stk} - Susceptance of line k in corridor (s,t)

g_{stk} - conductance of line k in corridor (s,t)

q_{stk} - Power losses in line k of corridor (s,t) in scenario c

f_{stk} - Lossless power flow in line k of corridor (s,t) in scenario c

δ_s - Voltage angle at bus s in scenario c

$\delta_{st}(x)$ - Variable used in the linearization of the power losses in corridor (s,t); it represents the the xth angle block relative to this corridor

$\alpha_{st}(x)$ - Slope of the the xth block of the voltage angle for the corridor (s,t)

$\Delta\delta_{st}$ - Upper bound of the angle blocks of corridor (s,t)

M_{st} - Large enough positive constant

δ_{st}^+ - Variable used in the linearization of the power losses in corridor (s,t)

δ_{st}^- - Variable used in the linearization of the power losses in corridor (s,t)

In [49]:
```
# line data susceptance and conductance
b = -Reactance./(Reactance.^2+Resistance.^2)
g = Resistance./(Reactance.^2+Resistance.^2)

# resized line data vectors (45-long vectors)
b_jump = 100*[b;b;b]
g_jump = 100*[g;g;g]

# buses
delta_lim = 20*π/180

# other
Mst = 10^3 # Positive constant needed for relaxation
L = 4 # Number of blocks of the piecewise linearization of power losse
s

αst = zeros(L); #Slope of the lth block of the voltage angle for the co
rridor (s,t)
for i = 1:L
    αst[i] = ((i*delta_lim)^2-(i*delta_lim-delta_lim)^2)/delta_lim
end
```

In [50]:
```
# activate mixed integer-linear solver
function MILP()

n = Model(solver = GLPKSolverMIP(msg_lev=GLPK.MSG_OFF))

@variable(n, δs[1:6])
@variable(n, δst⁺[1:45] >= 0) # support variable
@variable(n, δst⁻[1:45] >= 0) # support variable
@variable(n, δl[1:45,1:L] >= 0) # support variable
@variable(n, w[1:45], Bin) # 45 variables because each line can be bui
lt up to 3 times
@variable(n, fst[1:45])
@variable(n, qst[1:45])
```

```
@variable(n, pg[1:6])

# Set a constraint for exisiting lines
@constraint(n, exconstr[i=1:6], w[Ex_ID[i]] == 1)

# Set constraint for reference bus
@constraint(n, refbus, δs[1] == 1)

# Calculate an expression for the power flows through each bus
@expression(n, f[i=1:6],sum(fst[j]+qst[j] for j in bus_mat_c1[i])-sum(
fst[j] for j in bus_mat_c2[i]))

# Load constraints
@constraint(n, load_balance_loss[i = 1:6] , pg[i] - f[i] == PD[i])

# Line flow constraints
@expression(n, min_flow_loss[i=1:45], -w[i]*Capacity_MW_jump[i])
@expression(n, max_flow_loss[i=1:45], w[i]*Capacity_MW_jump[i])

@constraint(n, min_flow_cstr_loss[i=1:45], fst[i] >= min_flow_loss[i])
@constraint(n, max_flow_cstr_loss[i=1:45], fst[i] <= max_flow_loss[i])

# Elimination of non-linearity constraints
@constraint(n, min_non_l[i=1:45], fst[i]/b_jump[i]+(δst⁺[i]-δst⁻[i]) >=
-(1-w[i])*Mst)
@constraint(n, max_non_l[i=1:45], fst[i]/b_jump[i]+(δst⁺[i]-δst⁻[i]) <=
(1-w[i])*Mst)

# Line loss constraints
@constraint(n, min_loss[i=1:45], qst[i] >= 0)
@constraint(n, max_loss[i=1:45], qst[i] <= w[i]*Capacity_MW_jump[i])

# Linear loss constraints
@expression(n, linear_loss[i=1:45], sum(αst[j]*δl[i,j] for j = 1:L))
@constraint(n, min_loss_lin[i=1:45], -qst[i]/g_jump[i] + linear_loss[i
] >=0)
@constraint(n, max_loss_lin[i=1:45], -qst[i]/g_jump[i] + linear_loss[i
] <= (1-w[i])*Mst^2)

# Angle constraints
@constraint(n, sum_angle[i=1:45], δst⁺[i]+δst⁻[i] == sum(δl[i,j] for j =
1:L))
@constraint(n, diff_angle[i=1:45], δs[C1_jump[i]]-δs[C2_jump[i]] == δst
⁺[i]-δst⁻[i])

# Power balance constraints
@constraint(n, pos_bal[i=1:45], fst[i] + 0.5*qst[i] <= w[i]*Capacity_M
W_jump[i])
@constraint(n, neg_bal[i=1:45],-fst[i] + 0.5*qst[i] <= w[i]*Capacity_M
W_jump[i])

# Generator flow constraints
@constraint(n, gen[i=1:6], 0 <= pg[i] <= PG_max[i])
```

```
    # angle linearization constraint
    @constraint(n, angle_lin[i=1:45,j=1:L], δl[i,j] <= delta_lim + (1-w[i])*Mst)

    # Define the minimization objective
    @expression(n, gen_costs_loss, sum(pg[i]*LMP_G[i] for i = 1:6))
    @expression(n, line_costs_loss, sum(w[i]*lcost_jump[i] for i = 1:45))

    @objective(n, Min, line_costs_loss + gen_costs_loss*(fy*8760/10^6))

    solve(n)

    W_jump_lp = getvalue(w)
    pg_jump_lp = getvalue(pg)
    qst_jump_lp = getvalue(qst)
    fst_jump_lp = getvalue(fst)
    line_costs_jump_lp = getvalue(line_costs_loss)

    obj_lp = getobjectivevalue(n)

    return W_jump_lp, pg_jump_lp, qst_jump_lp, qst_jump_lp, fst_jump_lp, line_costs_jump_lp, obj_lp

end
```

Out[50]: MILP (generic function with 1 method)

In [51]:
```
(W_jump_lp, pg_jump_lp, qst_jump_lp, qst_jump_lp, fst_jump_lp, line_costs_jump_lp, obj_lp) = MILP();
```

Results

In [52]:
```
lp_lines = find(W_jump_lp[1:end])
lp_new_lines = setdiff(lp_lines,Ex_ID)
n_lines_DC_w_loss = size(lp_new_lines)[1]
println("the number of new lines required is $n_lines_DC_w_loss")
```

```
the number of new lines required is 5
```

In [53]:
```
time_lp = @elapsed MILP();
@time MILP();
```

```
  0.691217 seconds (18.89 k allocations: 1.390 MB)
```

Additional lines built

In [54]:
```
Ex_lines = DataFrame(Line_ID = Line_ID_jump[lp_new_lines], Corridor =
Corridor_jump[lp_new_lines],
    Capacity_MW = Capacity_MW_jump[lp_new_lines], Investment_Cost = lc
ost_jump[lp_new_lines])
```

Out[54]:

	Line_ID	Corridor	Capacity_MW	Investment_Cost
1	9	2-6	100	30.0
2	24	2-6	100	30.0
3	29	4-6	100	30.0
4	41	3-5	100	20.0
5	44	4-6	100	30.0

Line Flows

In [55]:
```
DataFrame(Line_ID = Line_ID_jump[lp_lines], Coridor = Corridor_jump[lp
_lines],
    Line_flow_MW = round(fst_jump_lp[lp_lines],3), Losses_MW = round(q
st_jump_lp[lp_lines],3))
```

Out[55]:

	Line_ID	Coridor	Line_flow_MW	Losses_MW
1	1	1-2	10.703	0.934
2	3	1-4	5.317	0.464
3	4	1-5	48.362	4.22
4	6	2-3	-68.863	6.009
5	7	2-4	-2.728	0.238
6	9	2-6	-90.391	8.414
7	11	3-5	95.819	8.362
8	24	2-6	-90.391	8.414
9	29	4-6	-86.783	8.078
10	41	3-5	95.819	8.362
11	44	4-6	-86.783	8.078

Power summary for each bus

In [56]:
```
DataFrame(Bus_ID = Bus_ID, PG_MW = round(pg_jump_lp,1), LMP_G = LMP_G,
PD_MW = PD)
```

Out[56]:

	Bus_ID	PG_MW	LMP_G	PD_MW
1	1	150.0	10.0	80
2	2	-0.0	0.0	240
3	3	317.2	20.0	40
4	4	-0.0	0.0	160
5	5	-0.0	0.0	240
6	6	354.3	30.0	0

Investment Costs

In [57]:
```
inv_lp = line_costs_jump_lp - sum(lcost_jump[Ex_ID])
gen_lp = obj_lp-inv_lp
println("The total investment cost of new lines is $inv_lp M dollars")
```

```
The total investment cost of new lines is 140.0 M dollars
```

Summary of the Results (both models)

In [58]:
```
Results = DataFrame(Model = ["DC w/o Loss","MILP w/ Loss"],Lines_built
= [Corridor_jump[nl_new_lines],
        Corridor_jump[lp_new_lines]],Investment_Musd = [inv_lp,inv_nl]
,Gen_costs_Musd = round([gen_lp,gen_nl],2),
            Total_costs_Musd = round([obj_nl, obj_lp],2), Losses_MW =
round([0, sum(qst_jump_lp)],2),
                Time_seconds = [time_nl,time_lp])
```

Out[58]:

	Model	Lines_built	Investment_Musd	Gen_costs_Musd	Total_costs_Musd	Losses_
1	DC w/o Loss	String["4-6","4-6","3-5","4-6"]	140.0	219.01	326.87	0.0
2	MILP w/ Loss	String["2-6","2-6","4-6","3-5","4-6"]	110.0	216.87	359.01	61.57

In [59]:
```
# Results from the paper
load("Results_paper.png")
```

Out[59]:

TABLE III
SOLUTIONS FOR GARVER'S 6-BUS EXAMPLE

Corridor	Number of lines built	
	No losses	Losses
2-6	0	2
3-5	1	1
4-6	3	2
Investment Cost ($)	110	140

Findings

- We can see that I come up with the same as the author.
- We can also conclude, that without considering losses, we underestimate line investment and the problem becomes infeasible.
- We can also see that considering losses becomes computationally intensive, it takes 25 times more time to solve the model with losses. For larger systems, this may create concerns as computing time rises exponentially.
- One final thing to note is that the solution is very sensitive to:
  - large constant used for the relaxation
  - degree of the polynomial used to interpolate the losses

AC Model with NLP Relaxation

- This approach was replicated from the paper Transmission Expansion Planning Using an AC Model: Formulations and Possible Relaxations (2012) by Zhang et al.
- The relaxation reduces the MINLP to a NLP

Algorithm (Non-Linear AC Relaxation)

Objective: minimize { $\sigma \sum C^G(P_j^G) + K_{stk} w_{stk}$ }

σ - weighting factor to make investment and operational costs comparable

$C^G(P_j^G)$ - cost as a function of power generated

K_{stk} - Investment cost of constructing line in corridor (s,t)

w_{stk} - Binary variable that equals 1 if line k from (s,t) corridor is built and equals 0 otherwise

Constraints:

- $\sum P_j^G - P_s = \sum P_s^D$ for every bus s
- $\sum Q_j^G - Q_s = \sum Q_s^D$ for every bus s
- $P_s = \sum P_{stk}$ for every line connected to bus s and for every bus s
- $p_{stk} = w_{stk} [V_s^2 g_{stk} - V_s V_t (g_{stk} \cos(\delta_s - \delta_t) + b_{stk} \sin(\delta_s - \delta_t)]$ for every line
- $q_{stk} = w_{stk} [-V_s^2 b_{stk} + V_s V_t (b_{stk} \cos(\delta_s - \delta_t) + g_{stk} \sin(\delta_s - \delta_t)]$ for every line
- $P_j^{Gmin} <= P_j^G <= P_j^{Gmax}$ for every unit j
- $Q_j^{Gmin} <= Q_j^G <= Q_j^{Gmax}$ for every unit j
- $V_s^{min} <= V_s <= V_s^{max}$ for every bus s
- $\delta_{st}^{min} <= \delta_{st} <= \delta_{st}^{max}$ for every line
- $0 <= P_{stk}$^2 + Q_{stk}^2 <= S_{stk}^2^{max} for every line
- $w_{stk} = 1$ for every existing line
- $w_{stk}(1-w_{stk}) <= \epsilon$ for every line (relaxation)
- $0 <= w_{stk} <= 1$ for every line

V_s - voltage magnitude at bus s

δ_s - Voltage angle at bus s

δ_{st} - Voltage angle difference in corridor (s,t)

b_{stk} - Susceptance of line k in corridor (s,t)

g_{stk} - conductance of line k in corridor (s,t)

q_{stk} - reactive power flow in line k of corridor (s,t)

p_{stk} - power flow in line k of corridor (s,t)

S_{stk}^{max} - maximum allowed power flow through line k of corridor (s,t)

Bus Data for AC Model including Reactive Power

In [60]:
```
Bus_ID = 1:6
PG_min = [0.,0,0,0,0,0]
PG_max = [150.,0,360,0,0,600]
QG_min = [-10.,0,-10,0,0,-10]
QG_max = [65.,0,150,0,0,200]
LMP_G = [10.,0,20,0,0,30]
PD = [80,240,40,160,240,0]
QD = [16, 48, 8, 32, 48,0]

Garver_bus_ac = DataFrame(Bus_ID = Bus_ID, PD_MW = PD, QD_MVAr = QD, P
G_min = PG_min, PG_max = PG_max,
    QG_min = QG_min, QG_max = QG_max, LMP_G = LMP_G)
```

Out[60]:

	Bus_ID	PD_MW	QD_MVAr	PG_min	PG_max	QG_min	QG_max	LMP_G
1	1	80	16	0.0	150.0	-10.0	65.0	10.0
2	2	240	48	0.0	0.0	0.0	0.0	0.0
3	3	40	8	0.0	360.0	-10.0	150.0	20.0
4	4	160	32	0.0	0.0	0.0	0.0	0.0
5	5	240	48	0.0	0.0	0.0	0.0	0.0
6	6	0	0	0.0	600.0	-10.0	200.0	30.0

There is a bug with KNITRO in Jupyter, I ran the following code locally in Julia

In [61]:
```
using KNITRO

n = Model(solver=KnitroSolver(ms_enable = 1,ms_maxsolves = 500))

# All buses starting at bus 1
```

```
vlow = 0.95
vhigh = 1.05

@variable(n, 0 <= w[1:45] <= 1) # 45 variables because each line can b
e built up to 3 times
@variable(n, -delta_lim <= δ[1:6] <= delta_lim)
@variable(n, vlow <= v[1:6] <= vhigh)

@NLexpression(n, pst[i=1:45], w[i]*(v[C1_jump[i]]^2*g_jump[i]-v[C1_jum
p[i]]*v[C2_jump[i]]*(g_jump[i]*
            cos(δ[C1_jump[i]]-δ[C2_jump[i]])+b_jump[i]*sin(δ[C1_jump[i
]]-δ[C2_jump[i]]))))

@NLexpression(n, qst[i=1:45], w[i]*(-v[C1_jump[i]]^2*b_jump[i]+v[C1_ju
mp[i]]*v[C2_jump[i]]*(b_jump[i]*
            cos(δ[C1_jump[i]]-δ[C2_jump[i]])-g_jump[i]*sin(δ[C1_jump[i
]]-δ[C2_jump[i]]))))

# Set a constraint for exisiting lines
@constraint(n, exconstr[i=1:6], w[Ex_ID[i]] == 1)

# Set constraint for reference bus
@constraint(n, refbus, δ[1] == 0)

# Calculate an expression for the power flows through each bus
@NLexpression(n, pk[i=1:6],sum(pst[j] for j in bus_mat_c1[i])-sum(pst[
j] for j in bus_mat_c2[i]))
@NLexpression(n, qk[i=1:6],sum(qst[j] for j in bus_mat_c1[i])-sum(qst[
j] for j in bus_mat_c2[i]))

@NLconstraint(n, load_balance_ac[i in loads] , pk[i]  == -PD[i])
@NLconstraint(n, load_balance_re[i in loads] , qk[i]  == -QD[i])

@NLconstraint(n, gen_min_ac[i in generators] , pk[i] + PD[i] >= PG_min
[i])
@NLconstraint(n, gen_min_re[i in generators] , qk[i] + QD[i] >= QG_min
[i])

@NLconstraint(n, gen_max_ac[i in generators] , pk[i] + PD[i] <= PG_max
[i])
@NLconstraint(n, gen_max_re[i in generators] , qk[i] + QD[i] <= QG_max
[i])

@NLconstraint(n, max_flow_cstr_ac[i=1:45], pst[i]^2 + qst[i]^2 <= w[i]
*Capacity_MW_jump[i]^2)

@NLconstraint(n, relax[i=1:45],w[i]*(1-w[i]) <= 10^-9)

# Define the minimization objective
@NLexpression(n, gen_costs_ac, sum(pk[i]*LMP_G[i] for i in generators)
)
@NLexpression(n, line_costs_ac, sum(w[i]*lcost_jump[i] for i = 1:45))
@NLexpression(n, obj_ac, line_costs_ac + gen_costs_ac*(fy*8760/10^6))
```

```
@NLobjective(n, Min, obj_ac)

@NLexpression(n,pg[i=1:6],pk[i]+PD[i])

solve(n)
```

```
error compiling loadproblem!: error compiling Type: could not load l
ibrary "libknitro"
dlopen(libknitro.dylib, 1): image not found

 in _buildInternalModel_nlp(::JuMP.Model, ::JuMP.ProblemTraits) at /
Users/Manuelcoquet/.julia/v0.5/JuMP/src/nlp.jl:1248
 in #build#114(::Bool, ::Bool, ::JuMP.ProblemTraits, ::Function, ::J
uMP.Model) at /Users/Manuelcoquet/.julia/v0.5/JuMP/src/solvers.jl:34
8
 in (::JuMP.#kw##build)(::Array{Any,1}, ::JuMP.#build, ::JuMP.Model)
at ./<missing>:0
 in #solve#109(::Bool, ::Bool, ::Bool, ::Array{Any,1}, ::Function, :
:JuMP.Model) at /Users/Manuelcoquet/.julia/v0.5/JuMP/src/solvers.jl:
166
 in solve(::JuMP.Model) at /Users/Manuelcoquet/.julia/v0.5/JuMP/src/
solvers.jl:148
```

Characteristics of the problem:

- #### Non-convex -> can only guarantee local solutions
- #### Ran the problem at 500 different initial points using KNITRO Multistart
- #### It took roughly 70 minutes to run and achieved convergence in about 35% of the scenarios

Sample Output

In [62]:
```
load("sample_output.png")
```

Out[62]:

```
 Solve #  ThreadID  Status     Objective      FeasError    OptError   Real Time
 -------  --------  ------  -------------  ----------  ----------  ---------
       1         0    -200   3.650226e+02   1.055e-01   5.928e+01      3.709
       2         0    -202   2.253067e+03   2.552e+02   6.800e+01      0.635
       3         0       0   7.821621e+02   2.822e-09   1.332e-08     10.060
       4         0    -201   6.355571e+02   6.023e-01   6.800e+01      1.035
       5         0       0   1.881072e+03   2.887e-09   2.758e-06      1.200
       6         0    -200   5.959422e+02   7.465e-01   6.077e+01      1.194
       7         0    -202   7.168067e+03   9.620e+04   6.800e+01      1.162
       8         0    -202   2.896252e+03   1.185e+02   6.800e+01      1.121
       9         0       0   1.733711e+03   2.279e-08   2.338e-05      1.792
      10         0    -202   1.612133e+03   6.885e+03   6.800e+01      0.972
      11         0       0   8.154804e+02   8.576e-09   1.038e-05      0.962
      12         0       0   6.771338e+02   6.613e-11   7.873e-06      0.269
      13         0       0   1.901169e+03   3.511e-07   2.807e-05      0.590
      14         0    -200   7.038157e+02   4.459e-03   4.459e-03      0.824
      15         0    -202   3.533784e+03   5.041e+04   6.800e+01      4.042
      16         0    -202   1.330515e+03   2.253e+02   6.800e+01      0.800
      17         0    -202   6.726573e+02   6.697e+01   6.800e+01      1.339
      18         0    -202   5.065486e+02   9.471e+04   6.800e+01      0.551
      19         0    -202   3.367572e+03   3.674e+04   6.800e+01      0.519
      20         0    -201   1.420199e+04   2.997e+02   6.800e+01      1.164
```

Note: Objective includes existing line costs of $200 M dollars

KNITRO Results

In [63]:
```
load("results_knitro.png")
```

Out[63]:

```
Final Statistics
----------------
Final objective value               =   5.09090347268357e+02
Final feasibility error (abs / rel) =   1.72e-08 / 2.33e-12
Final optimality error  (abs / rel) =   1.29e-07 / 1.90e-09
# of iterations                     =     351310
# of CG iterations                  =     690463
# of function evaluations           =    1684072
# of gradient evaluations           =     351506
# of Hessian evaluations            =     351426
Total program time (secs)           =    4185.39551 (  4173.889 CPU time)

================================================================================
```

Note: Objective includes existing line costs of $200 M dollars

Additional Lines Built

```
In [64]: load("lines_knitro.png")
```

Out[64]:

```
julia> Ad_lines = DataFrame(Line_ID = Line_ID_jump[ac_new_lines], Corridor = Corridor_jump[ac_new_lines],
         Capacity_MW = Capacity_MW_jump[ac_new_lines], Investment_Cost = lcost_jump[ac_new_lines])
8×4 DataFrames.DataFrame
│ Row │ Line_ID │ Corridor │ Capacity_MW │ Investment_Cost │
├─────┼─────────┼──────────┼─────────────┼─────────────────┤
│ 1   │ 21      │ "2-3"    │ 100         │ 20.0            │
│ 2   │ 22      │ "2-4"    │ 100         │ 40.0            │
│ 3   │ 24      │ "2-6"    │ 100         │ 30.0            │
│ 4   │ 28      │ "4-5"    │ 75          │ 63.0            │
│ 5   │ 29      │ "4-6"    │ 100         │ 30.0            │
│ 6   │ 30      │ "5-6"    │ 78          │ 61.0            │
│ 7   │ 41      │ "3-5"    │ 100         │ 20.0            │
│ 8   │ 44      │ "4-6"    │ 100         │ 30.0            │
```

Line Flows

```
In [65]: load("line_flows_knitro.png")
```

Out[65]:

```
julia> line_flows = DataFrame(Line_ID = Line_ID_jump[ac_lines], Corridor = Corridor_jump[ac_lines],
         Active_Power_MW = pst_jump_ac[ac_lines], Reactive_Power_MVAr = qst_jump_ac[ac_lines])
14×4 DataFrames.DataFrame
│ Row │ Line_ID │ Corridor │ Active_Power_MW │ Reactive_Power_MVAr │
├─────┼─────────┼──────────┼─────────────────┼─────────────────────┤
│ 1   │ 1       │ "1-2"    │ 18.0796         │ 16.1978             │
│ 2   │ 3       │ "1-4"    │ 12.5343         │ 12.7448             │
│ 3   │ 4       │ "1-5"    │ 38.4951         │ 19.6164             │
│ 4   │ 6       │ "2-3"    │ -75.4532        │ -18.1297            │
│ 5   │ 7       │ "2-4"    │ 0.817895        │ 2.65839             │
│ 6   │ 11      │ "3-5"    │ 82.1465         │ 18.4954             │
│ 7   │ 21      │ "2-3"    │ -75.4532        │ -18.1297            │
│ 8   │ 22      │ "2-4"    │ 0.817895        │ 2.65839             │
│ 9   │ 24      │ "2-6"    │ -72.6498        │ -0.859567           │
│ 10  │ 28      │ "4-5"    │ -0.0681235      │ -5.39953            │
│ 11  │ 29      │ "4-6"    │ -72.8809        │ -4.26943            │
│ 12  │ 30      │ "5-6"    │ -37.2799        │ 3.20765             │
│ 13  │ 41      │ "3-5"    │ 82.1465         │ 18.4954             │
│ 14  │ 44      │ "4-6"    │ -72.8809        │ -4.26943            │
```

Summary

- ### The optimal line investment costs were $294 M USD
- ### The optimal generation costs were $15.09 M USD

```
In [66]: line_ID_ac = [21,22,24,28,29,30,41,44]; gen_costs_ac = 15.09; line_costs_ac = sum(lcost_jump[line_ID_ac]);
total_costs_ac = gen_costs_ac + line_costs_ac;

Summary = DataFrame(Model = "AC Power Flow NLP", Investment_Musd = line_costs_ac, Gen_costs_Musd = gen_costs_ac,
    Total_costs_Musd = total_costs_ac)
```

Out[66]:

	Model	Investment_Musd	Gen_costs_Musd	Total_costs_Musd
1	AC Power Flow NLP	294.0	15.09	309.09

In [ ]: