PyVIN, Pyomo based optimization model

```
from __future__ import division # This line is used to ensure that int or long
division arguments are converted to floating point values before division is
performed
from pyomo.environ import * # This command makes the symbols used by Pyomo known
to Python
import itertools
# Command to run the model: pyomo --solver=glpk --solver-suffix=dual ###.py
###.dat

model = AbstractModel()

# Nodes in the network
model.N = Set()

# Network arcs
model.k = Set()
model.A = Set(within=model.N*model.N*model.k)

# Source node
model.source = Param(within=model.N)
# Sink node
model.sink = Param(within=model.N)
# Flow capacity limits
model.u = Param(model.A)
# Flow lower bound
model.l = Param(model.A)
# Link amplitude (gain/loss)
model.a = Param(model.A)
# Link cost
model.c = Param(model.A)

# The flow over each arc
model.X = Var(model.A, within=NonNegativeReals)

# Minimize total cost
def total_rule(model):
return sum(model.c[i,j,k]*model.X[i,j,k] for (i,j,k) in model.A)
model.total = Objective(rule=total_rule, sense=minimize)

# Enforce an upper bound limit on the flow across each arc
def limit_rule_upper(model, i, j, k):
return model.X[i,j,k] <= model.u[i,j,k]
model.limit_upper = Constraint(model.A, rule=limit_rule_upper)

# Enforce a lower bound limit on the flow across each arc
def limit_rule_lower(model, i, j, k):
return model.X[i,j,k] >= model.l[i,j,k]
model.limit_lower = Constraint(model.A, rule=limit_rule_lower)
```

```
across each arc
def limit_rule_upper(model, i, j, k):
    return model.X[i,j,k] <= model.u[i,j,k]
model.limit_upper = Constraint(model.A, rule=limit_rule_upper)

# Enforce a lower bound limit on the flow across each arc
def limit_rule_lower(model, i, j, k):
    return model.X[i,j,k] >= model.l[i,j,k]
model.limit_lower = Constraint(model.A, rule=limit_rule_lower)


# To speed up creating the mass balance constraints, first
# create dictionaries of arcs_in and arcs_out of every node
# These are NOT Pyomo data, and Pyomo does not use "model._" at all
arcs_in = {}
arcs_out = {}

def arc_list_hack(model, i,j,k):
  if j not in arcs_in:
    arcs_in[j] = []
  arcs_in[j].append((i,j,k))

  if i not in arcs_out:
    arcs_out[i] = []
  arcs out[i].append((i,j,k))
  return [0]

model._ = Set(model.A, initialize=arc_list_hack)

# Enforce flow through each node (mass balance)
def flow_rule(model, node):
  if node in [value(model.source), value(model.sink)]:
      return Constraint.Skip
  outflow  = sum(model.X[i,j,k]/model.a[i,j,k] for i,j,k in arcs_out[node])
  inflow = sum(model.X[i,j,k] for i,j,k in arcs_in[node])
  return inflow == outflow
model.flow = Constraint(model.N, rule=flow_rule)
```