Process System Engineering #6

#03150796 Amane Suzuki

November 10, 2015

1 Abstract

I use the Simplex Method to find the best conditions of this plant. Simplex Method has enough performance and it can calculate the best conditions.

2 Algorithm

Figure 1 shows a flow of Simplex Method.

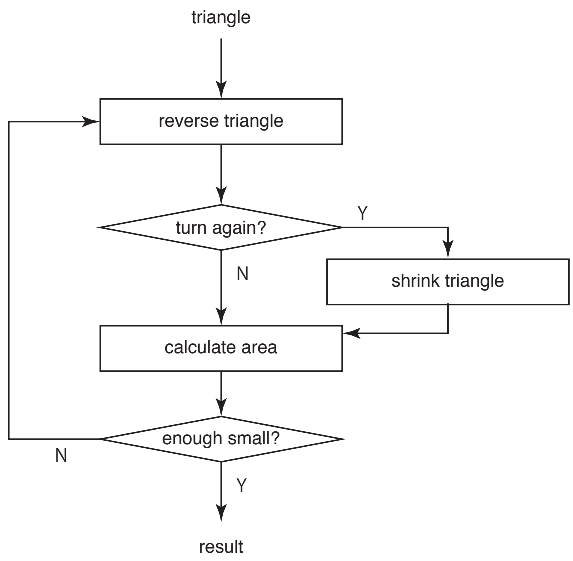

Figure 1: The flow chart of Simplex Method.

Simplex Method is faster than Steppest Decent Method (I used last week). Processing time is about 0.5s.

3 Result and Discussion

Table 1 shows the best conditions in $N = 1, 2, 3, 4, 5$.

	$N = 1$	$N = 2$	$N = 3$	$N = 4$	$N = 5$
γ_0 [wt%]	0.0727	0.0969	0.1	0.1	0.1
Coolant Temperature [K]	253.0	253.0	261.0	266.5	271.1
Total Cost [¥/s]	42.89	30.45	28.05	27.45	27.27

Table 1: The best conditions from N=1 to N=5

Except $N = 1, 2$, the best γ_0 is 0.1. It is because steam cost is dominant in total cost. In $N = 1, 2$, however, too large γ_0 makes power consumption infinite. So best γ_0 are 0.0727 and 0.0969.

The result suggest us to make plant in $N = 5, \gamma_0 = 0.1, T_2 = 271.1$. Thinking realistically, however, we should consider the area of land, plant stability, and ease of control.

4 Source Program

Listing 1: main.rb

```ruby
require "benchmark"
require "./plant"
require "./optimize"

f = lambda{ |conditions|
  gamma_0 =conditions[0]
  t2 =conditions[1]
  plant = Plant.new(5, gamma_0, t2)
  plant.costs[:total]
}

result = nil
time = Benchmark.realtime do
  simplex = Simplex.new(f, Vector[0.03, 260], 1e-12)
  result = simplex.optimize()
end

puts "min cost: #{f.call(result).round(2)}"
puts "gamma_0: #{result[0].round(4)}"
puts "T2: #{result[1].round(1)}"
puts "time: #{time}"
```

Listing 2: plant.rb

```ruby
include Math
require "./const"
require "./reactor"

class Plant
  attr_accessor :costs

  def initialize(n, gamma_0, coolant_temp)
    @n = n
    @gamma_0 = gamma_0
    @coolant_temp = coolant_temp
    @reactors = []
    @costs = {}

    flow_rate = PRODUCT_FLOW_RATE/CONVERSION/DENSITY/gamma_0 # feed speed [m3 h-1]
    k = (1.0/(1-CONVERSION)-1)/RESIDENCE_TIME # reaction constant [h-1]
    tau = (1.0/k)*((1-CONVERSION)**(-1.0/n)-1) # residence time
    volume = flow_rate*tau # [m3]

    prop = (1-CONVERSION)**(1.0/n)
    for i in 0...n
      gamma_in = gamma_0*(prop**(i))
      gamma_out = gamma_0*(prop**(i+1))
      @reactors << Reactor.new(gamma_0, gamma_in, gamma_out, volume, flow_rate, coolant_temp)
    end
    calc_cost()
  end

  def show()
    @reactors.each do |reactor|
      reactor.show()
    end
  end

  def calc_cost()
    if @gamma_0 > 0.1 or @gamma_0 < 0.01
      @costs[:total] = 1000.0
      return
    end

    total_power = 0
    total_heat = 0
    @reactors.each do |reactor|
      total_power += reactor.power
      total_heat += reactor.heat
    end

    @costs[:electricity] = ELECTRICITY_PRICE*total_power/(1000*3600) # [yen s-1]

```

```
    toluene_wt = (PRODUCT_FLOW_RATE/CONVERSION)*((1-@gamma_0)/@gamma_0)/3600 # [kg s-1]
    toluene_heat = toluene_wt*TOLUENE_LATENT_HEAT # [J s-1]
    steam_heat = toluene_heat/THERMAL_EFFICIENCY # [J s-1]
    steam_wt = steam_heat/WATER_LATENT_HEAT # [kg s-1]
    @costs[:steam] = (steam_wt/1000)*STEAM_PRICE # [yen s-1]

    reactor_price = (40000000.0+5.1e6*@reactors[0].volume)*@n
    @costs[:reactor] = reactor_price/(5*330*24*3600) # [yen s-1]

    coolant_wt = total_heat/(WATER_SPECIFIC_HEAT*WATER_TEMP_RISE)/1000 # [ton s-1]
    @costs[:coolant] = coolant_price(@coolant_temp)*coolant_wt # [yen s-1]

    @costs[:total] = @costs[:electricity] + @costs[:steam] + @costs[:reactor] + @costs[:coolant]
  rescue
    @costs[:total] = 1000.0
  end

  def coolant_price(t)
    #TODO: hard coding. it should be rewrittend
    t = t-273.0
    case
    when t>30
      return nil
    when t>10
      return 15+(45-15)*((30-t)*1.0/(30-10))
    when t>0
      return 45+(65-45)*((10-t)*1.0/(10-0))
    when t>-10
      return 65+(90-65)*((0-t)*1.0/(0+10))
    when t>=-20
      return 90+(140-90)*((-10-t)*1.0/(-10+20))
    else
      return nil
    end
  end
end
```

Listing 3: reactor.rb

```ruby
include Math
require "./const"

class Reactor
  attr_accessor :power, :heat, :volume
  def initialize(gamma_0, gamma_in, gamma_out, volume, flow_rate,
      coolant_temp)
    @gamma_0 = gamma_0
    @gamma_in = gamma_in
    @gamma_out = gamma_out
    @volume = volume
    @flow_rate = flow_rate
    @coolant_temp = coolant_temp
    @diameter = (volume/(ALPHA*2*PI))**(1.0/3)*2
    @height = @diameter*ALPHA
    @surface = @diameter*PI*@height
    calc_power(0)
  end

  def calc_power(power)
    heat_of_reaction = HEAT_OF_POLY*(@flow_rate*DENSITY*(@gamma_in-
        @gamma_out)/3.6)/BUTADIENE_M
    heat = heat_of_reaction + power
    h = heat/@surface/(REACTION_TEMP-@coolant_temp)

    # viscosity [Pa s]
    viscosity = ((POLYMER_LENGTH)**1.7)*((1-(@gamma_out/@gamma_0))
        **2.5)*exp(21.0*@gamma_0)*1e-3

    # dimensionless numbers
    pr = viscosity*TOLUENE_SPECIFIC_HEAT/THERMAL_CONDUCTIVITY
    nu = h*@diameter/THERMAL_CONDUCTIVITY
    re = (2*nu/pr**(1/3.0))**1.5

    # power consumption
    revolution = re*viscosity/DENSITY/(@diameter/2)**2
    np = 14.6*re**(-0.28)
    power_new = np*DENSITY*(revolution**3)*(@diameter/2)**5

    if (power_new - power).abs < ACCURACY
      @reynolds = re
      @revolution = revolution
      @power = power_new
      @heat = heat
    else
      return calc_power(power_new)
    end
  rescue SystemStackError
    @reynolds = nil
    @revolution = nil
    @power = nil
    @heat = nil
  end
```

```
end
```

Listing 4: optimize.rb

```ruby
include Math
require "matrix"

# f = lambda{ |coordinate|
# x=coordinate[0]
# y=coordinate[1]
# (x-1)**2+50*(y-1)**2
# }
#
# g = lambda{ |coordinate|
# x=coordinate[0]
# y=coordinate[1]
# (x-1)**2+100*(x**3-y)**2
# }

class Simplex
  def initialize(function, initial_point, delta)
    p1 = initial_point
    p2 = initial_point + Vector[0.001, 0]
    p3 = initial_point + Vector[0, 0.1]
    @points = [p1, p2, p3]
    @function = function
    @prev_move_point = -1
    @delta = delta
  end

  def optimize
    while area > @delta
      step
    end
    return (1.0/3)*@points.inject(:+)
  end

  def step
    highest_point = @points.index(@points.max_by{ |coordinate|
      @function.call(coordinate)
    })
    other_points = [0, 1, 2] - [highest_point]

    if highest_point == @prev_move_point
      @points[highest_point] = (1.0/3)*@points.inject(:+)
      @prev_move_point = -1
    else
      @points[highest_point] = @points[other_points[0]] + @points[
          other_points[1]] - @points[highest_point]
      @prev_move_point = highest_point
    end
  end

  def area
    a = @points[1] - @points[0]
    b = @points[2] - @points[0]
    ip = a.inner_product(b)
```

```
    return 0.5*sqrt(a.norm**2*b.norm**2-ip**2)
  end
end
```