AAE 550 MULTIDISCIPLINARY DESIGN OPTIMIZATION

I. DESIGN PROBLEM

BeanCo produces 2 products: standard coffee machines and premium coffee + expresso machines. BeanCo currently produces all their products manually with 4 workers per assembly line that are paid \$15/hour, but BeanCo can install automated lines with only a single roboticist per line that is paid \$35/hour. Manual assembly lines cost \$10,000 maintain annually and can produce 5 standard or 2 premium machines per hour. Robotic lines cost \$200,000 to maintain annually but can produce 8 standard or 4 premium machines per hour. All workers will work at their normal rate of pay for up to 1800 hours per year and 1.5x overtime for up to 540 hours annually. Since BeanCo's coffee machines are indirect competitors on the market, the expected demand for each machine is a function of both machines' price:

$$Q_s(P_s, P_p) = 2{,}000{,}000 P_s^{-0.5} - 500{,}000 P_p^{-0.8} - 200{,}000$$

$$Q_p(P_s, P_p) = 2{,}000{,}000 P_p^{-0.4} - 100{,}000 P_s^{-0.8} - 200{,}000$$

Materials costs are \$6 for standard and \$20 for premium machines. Maximize profit by choosing the number of manual and robotic assembly lines, market price for standard and premium coffee machines, and the quantity of standard and premium machines to produce manually and automatically. For simplicity, neglect switchover time for assembly lines.

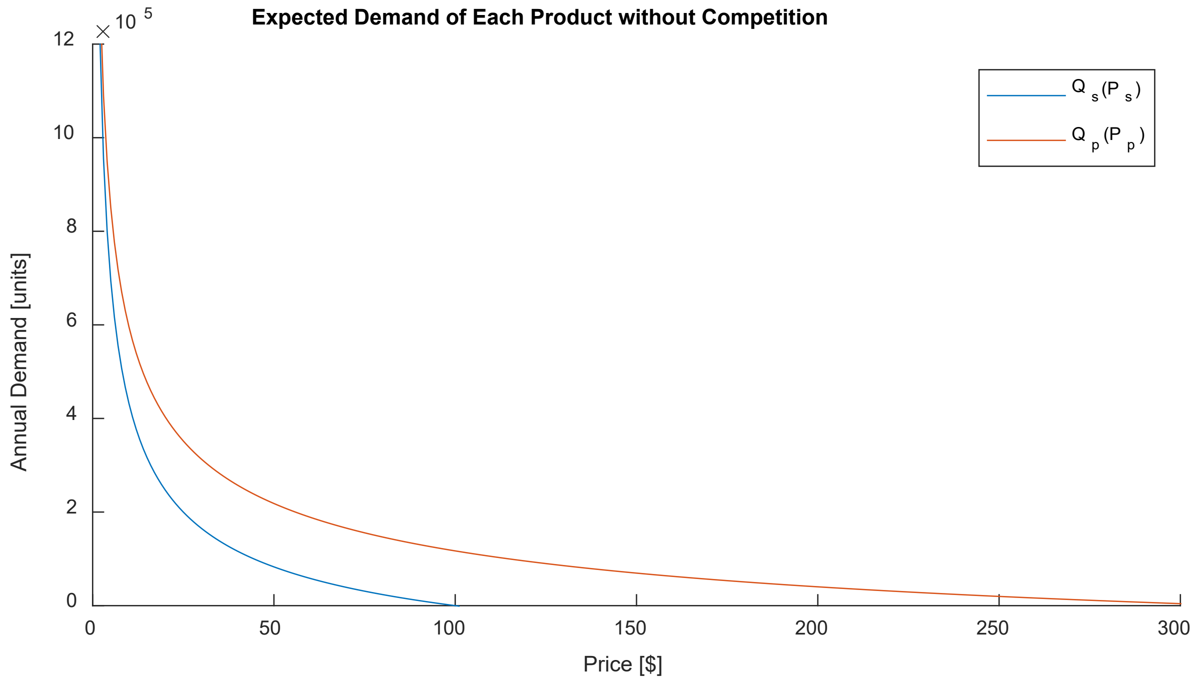

Figure 1: Reference of Demand Curves without Indirect Competition Factored in

II. FORMAL PROBLEM STATEMENT

Minimize	$[200000L_a$ $+10000L_m$ $+35\cdot L_a\cdot B_a(x)$ $+4\cdot 15\cdot L_m\cdot B_m(x)$ $-(Q_{s,a}+Q_{s,m})*(P_s-6)$ $-(Q_{p,a}+Q_{p,m})*(P_p-20)]$	Minimize [Automated line costs + Manual line costs + Cost of Labor on Automatic Lines + Cost of Labor on Manual Lines - Standard Machine Profits - Premium Machine Profits]
x =	$[L_a\ ,\ L_m\ ,\ Q_{s,a}\ ,\ Q_{s,m}\ ,$ $Q_{p,a}\ ,\ Q_{p,m}\ ,\ P_s\ ,\ P_p]$	Design Variable Vector

Subject to:	$(Q_{s,a}+Q_{s,m})/200{,}000$ $-10{P_s}^{-0.5}+2.5{P_p}^{-0.8}+1\le 0$	Standard production, limited by demand (order lowered)
	$(Q_{p,a}+Q_{p,m})/200{,}000$ $-10{P_p}^{-0.4}+0.5{P_s}^{-0.8}+1\le 0$	Premium production, limited by demand (order lowered)
	$\frac{Q_{s,m}}{11700}+\frac{Q_{p,m}}{4680}-L_m\le 0$	Manual production, limited by # of lines
	$\frac{Q_{s,a}}{18720}+\frac{Q_{p,a}}{9360}-L_a\le 0$	Automated production, limited by # of lines

Where:	$L_a\ge 0$	Automated Lines (integer)
	$L_m\ge 0$	Manual Lines (integer)
	$Q_{s,a}\ge 0$	Standard, Automated Production (integer)
	$Q_{s,m}\ge 0$	Standard, Manual Production (integer)
	$Q_{p,a}\ge 0$	Premium, Automated Production (integer)
	$Q_{p,m}\ge 0$	Premium, Manual Production (integer)
	$P_s\ge 0$	Price of Standard Coffee Machine
	$P_p\ge 0$	Price of Premium Coffee Machine

Other Useful Values:	$B_a(x)$ and $B_m(x)$	Billable hours of automated and manual lines (Used in objective function) (Piecewise function) Billable Hours = (Normal hours + 1.5 x Overtime hours)
	$T_m=\frac{\frac{Q_{s,m}}{5}+\frac{Q_{p,m}}{2}}{L_m}$	Total annual hours for each line worker
	$T_a=\frac{\frac{Q_{s,a}}{8}+\frac{Q_{p,a}}{4}}{L_a}$	Total annual hours for each roboticist

III. METHODOLOGY.

Some of the variable in this problem are integers (for example, a fractional number for assembly lines doesn't make sense) and since the functions involved in the problem become discontinuous with integers, I decided to compare a 2 solution approaches.

1. Ignore integer constraints and use Sequential Quadratic Programming (SQP) with MATLAB's fmincon function to get an approximate 'best case' answer to use as a sanity check and inform bounds for a Genetic Algorithm setup.
2. Use a Genetic Algorithm (GA) with the GA550 implementation (see appendix A4 for coding) because it can directly encode integer variables and handle the discontinuous problem as it is written.

SQP: I did not use user defined gradients because I knew my final solution would likely come from the GA and I judged the effort to figure out gradients to be higher than the computational cost of using numerical gradients. I made use of MATLAB's fmincon SQP ability to accept linearized constraints for g_3 and g_4, slightly speeding up the evaluation.

GA: I used the SQP results in **[IV. RESULTS TABLES]** to inform my setup:

- For x_1 and x_2 I chose to use 5-bit encoding starting from 0 and representing integers up to 31 as I was seeing no results approaching this high of a value
- For x_3 to x_6 I selected the lowest value of integer encoding starting from 0 that would represent up to 150000 products as this was roughly the total production found with SQP. This yielded 18 bits with an upper bound of 262143 for each variable.
- For price I selected a max price of \$100 and \$300 for x_7 and x_8 respectively and then selected 15 bits so that the resolution was ~\$0.01

I varied my N_{pop} and P_{mut} values but I did start at the class recommendation to begin with.

IV. RESULTS TABLES

Direct Constrained Minimization (fmincon with SQP)			
	Run 1	Run 2	Run 3
$\boldsymbol{x^0}$	$\begin{Bmatrix} 10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \end{Bmatrix}$	$\begin{Bmatrix} 4.4 \\ 16.72 \\ 0.01 \\ 80232 \\ 40521 \\ 28089 \\ 47 \\ 145 \end{Bmatrix}$ ***Close to x^* from run 1***	$\begin{Bmatrix} 50 \\ 50 \\ 500000 \\ 500000 \\ 500000 \\ 500000 \\ 1 \\ 1 \end{Bmatrix}$
$\boldsymbol{x}^*$	$\begin{Bmatrix} 4.33 \\ 16.72 \\ 0.06 \\ 80232.87 \\ 40521.47 \\ 28089.15 \\ 47.71 \\ 145.07 \end{Bmatrix}$	$\begin{Bmatrix} 7.39 \\ 8.86 \\ 0 \\ 79770.19 \\ 69205.35 \\ 0 \\ 47.85 \\ 144.29 \end{Bmatrix}$	$\begin{Bmatrix} 7.39 \\ 8.86 \\ 0 \\ 79768.04 \\ 69205.19 \\ 0.00 \\ 47.85 \\ 144.29 \end{Bmatrix}$
$f(\boldsymbol{x}^*)$	-8693200.14	-8739876.36	-8739876.29
$g(\boldsymbol{x}^*)$	$\begin{Bmatrix} -1.3050e-06 \\ -7.6233e-08 \\ -3.8578 \\ -8.8818e-16 \end{Bmatrix}$	$\begin{Bmatrix} -6.2741e-12 \\ -5.6428e-12 \\ -2.0454 \\ 0 \end{Bmatrix}$	$\begin{Bmatrix} 1.1102e-16 \\ -8.8818e-16 \\ -2.0453 \\ 0 \end{Bmatrix}$
# of iterations	62	57	141
Function Count	749	679	1458

Genetic Algorithm			
	Best of Run 1-5	Best of Run 6-10	Best of Run 11-15
Population Size	448 (Class recommended)	448	1792
Mutation Rate	0.001126 (Class recommended)	0.018017	0.004504
Number of Generations	151	1000	364
$\boldsymbol{x}^*$	$\begin{Bmatrix} 11 \\ 14 \\ 83668 \\ 45986 \\ 57070 \\ 30489 \\ 34.52 \\ 121.27 \end{Bmatrix}$	$\begin{Bmatrix} 9 \\ 19 \\ 29709 \\ 63675 \\ 59942 \\ 12427 \\ 43.33 \\ 139.59 \end{Bmatrix}$	$\begin{Bmatrix} 10 \\ 16 \\ 61342 \\ 44641 \\ 59514 \\ 22161 \\ 39.65 \\ 128.03 \end{Bmatrix}$
$f(\boldsymbol{x}^*)$	-7806642.59	-8315877.32	-8160866.33
$g(\boldsymbol{x}^*)$	$\begin{Bmatrix} -2.0079e-05 \\ -1.2994e-06 \\ -3.5548 \\ -0.4333 \end{Bmatrix}$	$\begin{Bmatrix} -0.0042 \\ -5.5722e-04 \\ -10.9024 \\ -1.0089 \end{Bmatrix}$	$\begin{Bmatrix} -0.0066 \\ -0.0010 \\ -7.4493 \\ -0.3649 \end{Bmatrix}$
Function Count	67648	448000	652288

Note that for each column of results for the genetic algorithm above, I took the best of 5 total runs on the same settings to help account for RNG woes

V. DISCUSSION

To find the final design to present I first returned to run 2 of SQP.

Best SQP result:

$$x^* = \begin{Bmatrix} 7.39 \\ 8.86 \\ 0 \\ 79770.19 \\ 69205.35 \\ 0 \\ 47.85 \\ 144.29 \end{Bmatrix}, f(x^*) = -8739876.36\,, g(x^*) = \begin{Bmatrix} -6.2741e-12 \\ -5.6428e-12 \\ -2.0454 \\ 0 \end{Bmatrix}$$

$$(Profit = \$8{,}739{,}876.36)$$

With a small margin of error, this appears to be the global optimum of the SQP problem as several iterations of SQP with different x^0 values converged to this design. Since the best profit recorded with the GA was $8,315,877.32, the SQP result is obviously 'better' but it is not a valid design without the integer consideration applied to x_{1-6}. I wanted to see if this solution could be made valid with some rounding and minor manual tweaks to variables, so I used MATLAB's floor() function on the design variable vector and arrived at:

$$x^* = \begin{Bmatrix} 7 \\ 8 \\ 0 \\ 79770 \\ 69205 \\ 0 \\ 47 \\ 144 \end{Bmatrix}, f(x^*) = -8334289.38\,, g(x^*) = \begin{Bmatrix} -0.0115 \\ -0.0203 \\ -1.1821 \\ 0.3937 \end{Bmatrix}$$

$$(Profit = \$8{,}334{,}289.38)$$

Notice that this modified solution is only a $18,412.06 gain on the best result from the GA and the solution is now infeasible on constraint g_4 indicating (with some math) that Q_{pa} (x_5) needs to be reduced by 3686 units. At the chosen P_p (x_8) this would result in a profit of only $7,944,034.13 – much lower than the best GA result. From this I concluded that my initial intuition was correct; the problem needs to be tackled directly with integer encoding.

My next step in analysis was to examine the result of the genetic algorithm more closely.

Best GA result:

$$x^* = \begin{Bmatrix} 9 \\ 19 \\ 29709 \\ 63675 \\ 59942 \\ 12427 \\ 43.33 \\ 139.59 \end{Bmatrix}, f(x^*) = -8315877.32\,, g(x^*) = \begin{Bmatrix} -0.0042 \\ -5.5722e-04 \\ -10.9024 \\ -1.0089 \end{Bmatrix}$$

$$(Profit = \$8{,}315{,}877.32)$$

This solution is obviously better than any other valid solution I had found but I wondered if I could once again try to slightly modify my results manually. The intuition of the problem that I gained from analysis of the SQP results told me that the large negative g_3 and g_4 were likely indicating that I could drop a few assembly lines and still meet the production required. To play with this idea, I decided to drop L_m (x_2) by increments of 1 until the total profit stopped increasing or g_3 became positive, then repeated this process for L_a (x_1) referencing against the g_4 constraint now. With these changes I arrived at this final design to present:

Final Design:

$$\boldsymbol{x^*} = \begin{Bmatrix} \mathbf{8} \\ \mathbf{11} \\ \mathbf{29709} \\ \mathbf{63675} \\ \mathbf{59942} \\ \mathbf{12427} \\ \mathbf{43.33} \\ \mathbf{139.59} \end{Bmatrix}, \boldsymbol{f(x^*)} = \mathbf{-8564019.37}, \boldsymbol{g(x^*)} = \begin{Bmatrix} \mathbf{-0.0042} \\ \mathbf{-5.5722e-04} \\ \mathbf{-2.9024} \\ \mathbf{-0.0089} \end{Bmatrix}$$

$$\boldsymbol{(Profit = \$8{,}564{,}019.37)}$$

An improvement of $248,142.05 is a great result from very easy changes to the design vector and this design is valid and feasible. Since this is $175,856.99 in profit below the SQP non-integer result, I could imagine that there would be some room for improvement, but another method should be implemented from this point to try for further improvements.

VI. CONCLUSION

This problem is informative as to why integer compatible algorithms are necessary to find valid solutions to problems that *seem* solvable by SQP or other methods. It should be noted how the implementation here throws caution to the wind for reducing computational cost though. My best GA run required 448000 function evaluations compared to 679 for my best SQP run and that doesn't even consider that I ran the GA 15 times total which was very time consuming compared to the SQP approximations I ran. I also made some manual adjustments at the end of optimization in order to arrive at my final solution which requires some intuition and effort on the optimizing engineer's part. If I needed to solve a very large version of this problem or many similar problems simultaneously, I would need to refine this method some. One method I would investigate if asked to solve this again is using modifications to simulated annealing to refine my final answer from the GA by "searching around" the solution a little for improvements. I also know that there are more optimization methods that cover integers, but I am not familiar enough to adapt techniques into the AAE550 toolkit.

VII. APPENDICES.

A1.1 – Objective function for SQP Minimization [objective.m]

```
function [f] = objective(x)
%   This function calculates the objective value for this project for use
%   in the SQP minimization technique
%Created by Matt Pugsley - 2022

L_a = x(1);       %Number of Automatic Lines
L_m = x(2);       %Number of Manual Lines
Q_sa = x(3);      %Production of Standard Machines on Automated Lines
Q_sm = x(4);      %Production of Standard Machines on Manual Lines
Q_pa = x(5);      %Production of Premium Machines on Automated Lines
Q_pm = x(6);      %Production of Premium Machines on Manual Lines
P_s = x(7);       %Price of a Standard Coffee Machine
P_p = x(8);       %Price of a Premium Coffee Machine

%Annual hours of each type of worker
T_m = (Q_sm/5 + Q_pm/2)/L_m;
T_a = (Q_sa/8 + Q_pa/4)/L_a;

%Figure out the billable hours for Roboticists
if T_a>1800
    B_a = T_a + 0.5*(T_a-1800);
else
    B_a = T_a;
end

%Figure out the billable hours for line workers
if T_m>1800
    B_m = T_m + 0.5*(T_m-1800);
else
    B_m = T_m;
end

%calculate the objective function
f = 200000*L_a ...
    + 10000*L_m ...
    + 35*L_a*B_a ...
    + 4*15*L_m*B_m ...
    - (Q_sa + Q_sm)*(P_s-6) ...
    - (Q_pa + Q_pm)*(P_p-20);
end
```

A1.2 – Objective function for GA Minimization [objectiveGA.m]

```
function [phi] = objectiveGA(x)
%   This function calculates the modified objective function to be used
%   with the genetic algorithm approach to my final project.
%   Note that this includes the calculation of constraint values directly
%   without another function call
%Created by Matt Pugsley - 2022

L_a = x(1);       %Number of Automatic Lines
L_m = x(2);       %Number of Manual Lines
Q_sa = x(3);      %Production of Standard Machines on Automated Lines
Q_sm = x(4);      %Production of Standard Machines on Manual Lines
Q_pa = x(5);      %Production of Premium Machines on Automated Lines
Q_pm = x(6);      %Production of Premium Machines on Manual Lines
P_s = x(7);       %Price of a Standard Coffee Machine
P_p = x(8);       %Price of a Premium Coffee Machine

c = [1 1 1 1];    %Constraint Multipliers

g(1) = (Q_sa+Q_sm)/200000-10*P_s^(-0.5)+2.5*P_p^(-0.8)+1;
g(2) = (Q_pa+Q_pm)/200000-10*P_p^(-0.4)+0.5*P_s^(-0.8)+1;
g(3) = Q_sm/11700+Q_pm/4680-L_m;
g(4) = Q_sa/18720+Q_pa/9360-L_a;

%Penalty Function
P = sum(c .* (~(g<=0)) .* (1+g));

%Annual hours of each type of worker
T_m = (Q_sm/5 + Q_pm/2)/L_m;
T_a = (Q_sa/8 + Q_pa/4)/L_a;

%Figure out the billable hours for Roboticists
if T_a>1800
    B_a = T_a + 0.5*(T_a-1800);
else
    B_a = T_a;
end

%Figure out the billable hours for line workers
if T_m>1800
    B_m = T_m + 0.5*(T_m-1800);
else
    B_m = T_m;
end

%calculate the objective function
f = 200000*L_a ...
    + 10000*L_m ...
    + 35*L_a*B_a ...
    + 4*15*L_m*B_m ...
    - (Q_sa + Q_sm)*(P_s-6) ...
    - (Q_pa + Q_pm)*(P_p-20);

%penalty multiplier - being pretty aggressive bc of magnitude of obj
rp = 100000000;

%Modified objective
phi = f + rp*P;

end
```

A2 – Constraint Function for SQP Minimization [nonlinearConstraints.m]

```
function [C,Ceq] = nonlinearConstraints(x)
%   This function calculates the 2 nonlinear constraints in this project
%   but it is useful to note that there are 2 linear constraints handled
%   directly by the SQP implementation. The constraints are formatted to
%   directly be received by fmincon as a C vector
%Created by Matt Pugsley - 2022

    L_a = x(1);       %Number of Automatic Lines
    L_m = x(2);       %Number of Manual Lines
    Q_sa = x(3);      %Production of Standard Machines on Automated Lines
    Q_sm = x(4);      %Production of Standard Machines on Manual Lines
    Q_pa = x(5);      %Production of Premium Machines on Automated Lines
    Q_pm = x(6);      %Production of Premium Machines on Manual Lines
    P_s = x(7);       %Price of a Standard Coffee Machine
    P_p = x(8);       %Price of a Premium Coffee Machine

    C = [(Q_sa + Q_sm)/200000-10*P_s.^(-0.5)+2.5*P_p.^(-0.8)+1;
        (Q_pa + Q_pm)/200000-10*P_p.^(-0.4)+0.5*P_s.^(-0.8)+1];
    Ceq = [];
end
```

***Note that linear constraints are handled in the main script for SQP**

***Note that there is no constraint function for the GA because they are incorporated in the objectiveGA.m function directly.**

Direct MATLAB code representing all 4 constraint equations in one place is below:

```
g1 = (Q_sa+Q_sm)/200000-10*P_s^(-0.5)+2.5*P_p^(-0.8)+1

g2 = (Q_pa+Q_pm)/200000-10*P_p^(-0.4)+0.5*P_s^(-0.8)+1

g3 = Q_sm/11700+Q_pm/4680-L_m

g4 = Q_sa/18720+Q_pa/9360-L_a
```

A3.1 – Script for calling SQP [FinalProject_SQP.m]

```
clc
clear all
close all
%Created by Matt Pugsley - 2022

format bank

X0 = [10 10 10 10 10 10 10 10]';

%L_a = x(1)     Number of Automatic Lines
%L_m = x(2)     Number of Manual Lines
%Q_s,a = x(3)   Production of Standard Machines on Automated Lines
%Q_s,m = x(4)   Production of Standard Machines on Manual Lines
%Q_p,a = x(5)   Production of Premium Machines on Automated Lines
%Q_p,m = x(6)   Production of Premium Machines on Manual Lines
%P_s = x(7)     Price of a Standard Coffee Machine
%P_p = x(8)    Price of a Premium Coffee Machine

demand_s = @(Ps,Pp) 2000000*Ps.^(-0.5)-500000*Pp.^(-0.8)-200000;
demand_p = @(Ps,Pp) 2000000*Pp.^(-0.4)-100000*Ps.^(-0.8)-200000;

A = [0 -1 0 1/11700 0 1/4680 0 0;
     -1 0 1/18720 0 1/9360 0 0 0];
B = [0;0];

A_eq = [];
B_eq = []';

LB = [0 0 0 0 0 0 0 0]';
UB = [inf inf inf inf inf inf inf inf]';

options = optimoptions('fmincon', 'Algorithm', 'sqp','MaxFunctionEvaluations',5000);

[xstar,fval,exitflag,output] =
fmincon(@objective,X0,A,B,A_eq,B_eq,LB,UB,@nonlinearConstraints,options);

% Let's look at some results ------------------------------------------
L_a = xstar(1)      %Number of Automatic Lines
L_m = xstar(2)      %Number of Manual Lines
Q_sa = xstar(3)     %Production of Standard Machines on Automated Lines
Q_sm = xstar(4)     %Production of Standard Machines on Manual Lines
Q_pa = xstar(5)     %Production of Premium Machines on Automated Lines
Q_pm = xstar(6)     %Production of Premium Machines on Manual Lines
P_s = xstar(7)      %Price of a Standard Coffee Machine
P_p = xstar(8)      %Price of a Premium Coffee Machine
T_m = (Q_sm/5 + Q_pm/2)/L_m     %Total annual hours for each line worker
T_a = (Q_sa/8 + Q_pa/4)/L_a     %Total annual hours for each roboticist
profit = -fval      %Total Profit
```

```
Manual_production = Q_sm+Q_pm       %Total products made by humans
Automated_production = Q_sa+Q_pa    %Total products made by robots
Standard_production = Q_sa+Q_sm     %Standard coffee machines produced
Premuim_production = Q_pa+Q_pm      %Premuim coffee machines produced
standard_demand = demand_s(P_s,P_p) %Demand for Standard Machines
premium_demand = demand_p(P_s,P_p)  %Demand for Premium Machines

xstar = xstar
fval = fval
format short
g1 = (Q_sa+Q_sm)/200000-10*P_s^(-0.5)+2.5*P_p^(-0.8)+1
g2 = (Q_pa+Q_pm)/200000-10*P_p^(-0.4)+0.5*P_s^(-0.8)+1
g3 = Q_sm/11700+Q_pm/4680-L_m
g4 = Q_sa/18720+Q_pa/9360-L_a
```

A3.2 – Script for calling GA [FinalProject_GA.m]

```
clc

clear all
close all
%Created by Matt Pugsley - 2022

format bank

%L_a = x(1)     Number of Automatic Lines
%L_m = x(2)     Number of Manual Lines
%Q_s,a = x(3)   Production of Standard Machines on Automated Lines
%Q_s,m = x(4)   Production of Standard Machines on Manual Lines
%Q_p,a = x(5)   Production of Premium Machines on Automated Lines
%Q_p,m = x(6)   Production of Premium Machines on Manual Lines
%P_s = x(7)     Price of a Standard Coffee Machine
%P_p = x(8)    Price of a Premium Coffee Machine

demand_s = @(Ps,Pp) 2000000*Ps.^(-0.5)-500000*Pp.^(-0.8)-200000;
demand_p = @(Ps,Pp) 2000000*Pp.^(-0.4)-100000*Ps.^(-0.8)-200000;

options = goptions([]);

vlb = [0 0 0 0 0 0 0 0];  %Lower bound of each gene
vub = [31 31 262143 262143 262143 262143 100 300]; %Upper bound of each gene
bits = [5 5 18 18 18 18 15 15];   %number of bits describing each gene

%bit string affinity
% options(2) = 0.90

%class recommended population size
npop = 4*sum(bits);
%Setting actual population size for this problem
options(11) = npop;

%class recommended mutation chance
pmut = (sum(bits)+1)/(2*npop*sum(bits));
%Setting Actual Pmut for this problem
options(13) = pmut;

%Maximum iterations
options(14) = 1000;

[xstar,fval,stats,nfit,fgen,lgen,lfit] = ...
    GA550('objectiveGA',[ ],options,vlb,vub,bits);

% Let's look at some results ------------------------------------------
L_a = xstar(1)      %Number of Automatic Lines
L_m = xstar(2)      %Number of Manual Lines
```

```
Q_sa = xstar(3)     %Production of Standard Machines on Automated Lines
Q_sm = xstar(4)     %Production of Standard Machines on Manual Lines
Q_pa = xstar(5)     %Production of Premium Machines on Automated Lines
Q_pm = xstar(6)     %Production of Premium Machines on Manual Lines
P_s = xstar(7)      %Price of a Standard Coffee Machine
P_p = xstar(8)      %Price of a Premium Coffee Machine
T_m = (Q_sm/5 + Q_pm/2)/L_m     %Total annual hours for each line worker
T_a = (Q_sa/8 + Q_pa/4)/L_a     %Total annual hours for each roboticist
profit = -fval      %Total Profit
Manual_production = Q_sm+Q_pm       %Total products made by humans
Automated_production = Q_sa+Q_pa    %Total products made by robots
Standard_production = Q_sa+Q_sm     %Standard coffee machines produced
Premuim_production = Q_pa+Q_pm      %Premuim coffee machines produced
standard_demand = demand_s(P_s,P_p) %Demand for Standard Machines
premium_demand = demand_p(P_s,P_p)  %Demand for Premium Machines

xstar = xstar
fval = fval
format short
g1 = (Q_sa+Q_sm)/200000-10*P_s^(-0.5)+2.5*P_p^(-0.8)+1
g2 = (Q_pa+Q_pm)/200000-10*P_p^(-0.4)+0.5*P_s^(-0.8)+1
g3 = Q_sm/11700+Q_pm/4680-L_m
g4 = Q_sa/18720+Q_pa/9360-L_a
```

A4 – Genetic Algorithm Coding [GA550.m]

```
function [xopt,fopt,stats,nfit,fgen,lgen,lfit] = GA550(fun, ...
    x0,options,vlb,vub,bits,P1,P2,P3,P4,P5,P6,P7P,P8,P9,P10)
%GA550 minimizes a fitness function using a simple genetic algorithm.
%
%        X=GA550('FUN',X0,OPTIONS,VLB,VUB) uses a simple
%       genetic algorithm to find a minimum of the fitness function
%       FUN.  FUN can be a user-defined M-file: FUN.M, or it can be a
%        string containing the function itself.  The user may define all
%       or part of an initial population X0. Any undefined individuals
%        will be randomly generated between the lower and upper bounds
%        (VLB and VUB).  If X0 is an empty matrix, the entire initial
%        population will be randomly generated.  Use OPTIONS to specify
%        flags, tolerances, and input parameters.  Type HELP GOPTIONS
%       for more information and default values.
%
%        X=GA550('FUN',X0,OPTIONS,VLB,VUB,BITS) allows the user to
%        define the number of BITS used to code non-binary parameters
%        as binary strings.  Note: length(BITS) must equal length(VLB)
%        and length(VUB).  If BITS is not specified, as in the previous
%        call, the algorithm assumes that the fitness function is
%        operating on a binary population.
%
%        X=GA550('FUN',X0,OPTIONS,VLB,VUB,BITS,P1,P2,...) allows up
%        to ten arguments, P1,P2,... to be passed directly to FUN.
%        F=FUN(X,P1,P2,...). If P1,P2,... are not defined, F=FUN(X).
%
%        [X,FOPT,STATS,NFIT,FGEN,LGEN,LFIT]=GA550(<ARGS>)
%          X       - design variables of best ever individual
%          FOPT    - fitness value of best ever individual
%          STATS   - [min mean max stopping_criterion] fitness values
%                    for each generation
%          NFIT   - number of fitness function evalations
%          FGEN    - first generation population
%          LGEN    - last generation population
%          LFIT    - last generation fitness
%
%       The algorithm implemented here is based on the book: Genetic
%       Algorithms in Search, Optimization, and Machine Learning,
%       David E. Goldberg, Addison-Wiley Publishing Company, Inc.,
%       1989.
%
%        Originally created on 1/10/93 by Andrew Potvin, Mathworks, Inc.
%        Modified on 2/3/96 by Joel Grasmeyer.
%        Modified on 11/12/02 by Bill Crossley.
%        Modified on 7/20/04 by Bill Crossley.

% Make best_feas global for stopping criteria (4/13/96)
global best_feas
global gen
global fit_hist
% Load input arguments and check for errors
if nargin<4,
    error('No population bounds given.')
elseif (size(vlb,1)~=1) | (size(vub,1)~=1),
    % Remark: this will change if algorithm accomodates matrix variables
    error('VLB and VUB must be row vectors')
elseif (size(vlb,2)~=size(vub,2)),
    error('VLB and VUB must have the same number of columns.')
elseif (size(vub,2)~=size(x0,2)) & (size(x0,1)>0),
    error('X0 must all have the same number of columns as VLB and VUB.')
elseif any(vlb>vub),
    error('Some lower bounds greater than upper bounds')
else
    x0_row = size(x0,1);
    for i=1:x0_row,
        if any(x0(x0_row,:)<vlb) | any(x0(x0_row,:)>vub),
            error('Some initial population not within bounds.')
        end % if initial pop not within bounds
    end % for initial pop
end % if nargin<4

if nargin<6,
    bits = [];
elseif (size(bits,1)~=1) | (size(bits,2)~=size(vlb,2)),
    % Remark: this will change if algorithm accomodates matrix variables
    error('BITS must have one row and length(VLB) columns')
elseif any(bits~=round(bits)) | any(bits<1),
    error('BITS must be a vector of integers >0')
end % if nargin<6

% Form string to call for function evaluation
if ~( any(fun<48) | any(fun>122) | any((fun>90) & (fun<97)) | ...
        any((fun>57) & (fun<65)) ),
    % Only alphanumeric characters implies that 'fun' is a separate m-file
    evalstr = [fun '(x'];
    for i=1:nargin-6,
        evalstr = [evalstr,',P',int2str(i)];
    end
else
    % Non-alphanumeric characters implies that the function is contained
    % within the single quotes
    evalstr = ['(',fun];
end

% Determine all options
% Remark: add another options index for type of termination criterion
if size(options,1)>1,
    error('OPTIONS must be a row vector')
else
```

```
    % Use default options for those that were not passed in
    options = goptions(options);
end
PRINTING = options(1);
BSA = options(2);
fit_tol = options(3);
nsame = options(4)-1;
elite = options(5);

% Since operators are tournament selection and uniform crossover and
% default coding is Gray / binary, set crossover rate to 0.50 and use
% population size and mutation rate based on Williams, E. A., and Crossley,
% W. A., "Empirically-derived population size and mutation rate guidelines
% for a genetic algorithm with uniform crossover," Soft Computing in
% Engineering Design and Manufacturing, 1998.  If user has entered values
% for these options, then user input values are used.
if options(11) == 0,
    pop_size = sum(bits) * 4;
else
    pop_size = options(11);
end
if options(12) == 0,
    Pc = 0.5;
else
    Pc = options(12);
end
if options(13) == 0,
    Pm = (sum(bits) + 1) / (2 * pop_size * sum(bits));
else
    Pm = options(13);
end
max_gen = options(14);
% Ensure valid options: e.q. Pc,Pm,pop_size,max_gen>0, Pc,Pm<1
if any([Pc Pm pop_size max_gen]<0) | any([Pc Pm]>1),
    error('Some Pc,Pm,pop_size,max_gen<0 or Pc,Pm>1')
end

% Encode fitness (cost) function if necessary
ENCODED = any(any(([vlb; vub; x0]~=0) & ([vlb; vub; x0]~=1))) |  ....
    ~isempty(bits);
if ENCODED,
    [fgen,lchrom] = encode(x0,vlb,vub,bits);
else
    fgen = x0;
    lchrom = size(vlb,2);
end

% Display warning if initial population size is odd
if rem(pop_size,2)==1,
    disp('Warning: Population size should be even.  Adding 1 to population.')
    pop_size = pop_size +1;
```

```
end

% Form random initial population if not enough supplied by user
if size(fgen,1)<pop_size,
    fgen = [fgen; (rand(pop_size-size(fgen,1),lchrom)<0.5)];
end
xopt = vlb;
nfit = 0;
new_gen = fgen;
isame = 0;
bitlocavg = mean(fgen,1);  % initial bit string affinity
BSA_pop = 2 * mean(abs(bitlocavg - 0.5));
fopt = Inf;
stats = [];

% Header display
if PRINTING>=1,
    if ENCODED,
        disp('Variable coding as binary chromosomes successful.')
        disp('')
        fgen = decode(fgen,vlb,vub,bits);
    end
    disp('                   Fitness statistics')
    if nsame > 0
        disp('Generation Minimum      Mean         Maximum       isame')
    elseif BSA > 0
        disp('Generation Minimum      Mean         Maximum       BSA')
    else
        disp('Generation Minimum      Mean         Maximum       not used')
    end
end

% Set up main loop
STOP_FLAG = 0;
for generation = 1:max_gen+1,
    old_gen = new_gen;

    % Decode binary strings if necessary
    if ENCODED,
        x_pop = decode(old_gen,vlb,vub,bits);
    else
        x_pop = old_gen;
    end

    % Get fitness of each string in population
    for i = 1:pop_size,
        x = x_pop(i,:);
        fitness(i) = eval([evalstr,')']);
        nfit = nfit + 1;
    end
    % Store minimum fitness value from previous generation (except for
    % initial generation)
    if generation > 1,
        min_fit_prev = min_fit;
        min_gen_prev = min_gen;
        min_x_prev = min_x;
    end

    % identify worst (maximum) fitness individual in current generation
    [max_fit,max_index] = max(fitness);

    % impose elitism - currently only one individual; this replaces worst
    % individual of current generation with best of previous generation
    if (generation > 1 & elite > 0),
        old_gen(max_index,:) = min_gen_prev;
        x_pop(max_index,:) = min_x_prev;
        fitness(max_index) = min_fit_prev;
    end

    % identify best (minimum) fitness individual in current generation and
    % store bit string and x values
    [min_fit,min_index] = min(fitness);
    min_gen = old_gen(min_index,:);
    min_x = x_pop(min_index,:);

    % Store best fitness and x values
    if min_fit < fopt,
        fopt = min_fit;
        xopt = min_x;
    end

    % Compute values for isame or BSA_pop stopping criteria
    if nsame > 0
        if generation > 1
            if min_fit_prev == min_fit
                isame = isame + 1;
            else
                isame = 0;
            end
        end
    elseif BSA > 0
        bitlocavg = mean(old_gen,1);
        BSA_pop = 2 * mean(abs(bitlocavg - 0.5));
    end


    % Calculate generation statistics
    if nsame > 0
        stats = [stats; generation-1,min(fitness),mean(fitness), ...
            max(fitness), isame];
    elseif BSA > 0
```

```
        stats = [stats; generation-1,min(fitness),mean(fitness), ...
            max(fitness), BSA_pop];
    else
        stats = [stats; generation-1,min(fitness),mean(fitness), ...
            max(fitness), 0];
    end

    % Display if necessary
    if PRINTING>=1,
        disp([sprintf('%5.0f %12.6g %12.6g %12.6g %12.6g', stats(generation,1), ...
                stats(generation,2),stats(generation,3), stats(generation,4),...
                stats(generation,5))]);
    end

    % Check for termination
    % The default termination criterion is bit string affinity.  Also
    % available are fitness tolerance across five generations and number of
    % consecutive generations with same best fitness.  These can be used
    % concurrently.
    if fit_tol>0,    % if fit_tol > 0, then fitness tolerance criterion used
        if generation>5,
            % Check for normalized difference in fitness minimums
            if stats(generation,1) ~= 0,
                if abs(stats(generation-5,1)-stats(generation,1))/ ...
                        stats(generation,1) < fit_tol
                    if PRINTING >= 1
                        fprintf('\n')
                        disp('GA converged based on difference in fitness minimums.')
                    end
                    lfit = fitness;
                    if ENCODED,
                        lgen = x_pop;
                    else
                        lgen = old_gen;
                    end
                    return
                end
            else
                if abs(stats(generation-5,1)-stats(generation,1)) < fit_tol
                    if PRINTING >= 1
                        fprintf('\n')
                        disp('GA converged based on difference in fitness minimums.')
                    end
                    lfit = fitness;
                    if ENCODED,
                        lgen = x_pop;
                    else
                        lgen = old_gen;
                    end
                    return
                end
            end
        end
    elseif nsame > 0,     % consecutive minimum fitness value criterion
            if isame == nsame
                if PRINTING >= 1
                    fprintf('\n')
                    disp('GA stopped based on consecutive minimum fitness values.')
                end
                lfit = fitness;
                if ENCODED,
                    lgen = x_pop;
                else
                    lgen = old_gen;
                end
                return
            end
    elseif BSA > 0,  % bit string affinity criterion
        if BSA_pop >= BSA,
            if PRINTING >=1
                fprintf('\n')
                disp('GA stopped based on bit string affinity value.')
            end
            lfit = fitness;
            if ENCODED,
                lgen = x_pop;
            else
                lgen = old_gen;
            end
            return
        end
    end

    % Tournament selection
    new_gen = tourney(old_gen,fitness);

    % Crossover
    new_gen = uniformx(new_gen,Pc);

    % Mutation
    new_gen = mutate(new_gen,Pm);

    % Always save last generation.  This allows user to cancel and
    % restart with x0 = lgen
    if ENCODED,
        lgen = x_pop;
    else
        lgen = old_gen;
    end


end % for max_gen
```

```
% Maximum number of generations reached without termination
lfit = fitness;
if PRINTING>=1,
    fprintf('\n')
    disp('Maximum number of generations reached without termination')
    disp('criterion met.  Either increase maximum generations')
    disp('or ease termination criterion.')
end


% end genetic

function [gen,lchrom,coarse,nround] = encode(x,vlb,vub,bits)
%ENCODE Converts from variable to binary representation.
%        [GEN,LCHROM,COARSE,nround] = ENCODE(X,VLB,VUB,BITS)
%       encodes non-binary variables of X to binary.  The variables
%       in the i'th column of X will be encoded by BITS(i) bits.  VLB
%       and VUB are the lower and upper bounds on X.  GEN is the binary
%       representation of these X.  LCHROM=SUM(BITS) is the length of
%       the binary chromosome.  COARSE(i) is the coarseness of the
%       i'th variable as determined by the variable ranges and
%       BITS(i).  ROUND contains the absolute indices of the
%       X which where rounded due to finite BIT length.
%
%        Copyright (c) 1993 by the MathWorks, Inc.
%        Andrew Potvin 1-10-93.

% Remark: what about handling case where length(bits)~=length(vlb)?


lchrom = sum(bits);
coarse = (vub-vlb)./((2.^bits)-1);
[x_row,x_col] = size(x);

gen = [];
if ~isempty(x),
   temp = (x-ones(x_row,1)*vlb)./ ...
          (ones(x_row,1)*coarse);
   b10 = round(temp);
   % Since temp and b10 should contain integers 1e-4 is close enough
   nround = find(b10-temp>1e-4);
   gen = b10to2(b10,bits);
end

% end encode


function [x,coarse] = decode(gen,vlb,vub,bits)
%DECODE Converts from binary Gray code to variable representation.
%        [X,COARSE] = DECODE(GEN,VLB,VUB,BITS) converts the binary
%       population GEN to variable representation.  Each individual
%       of GEN should have SUM(BITS).  Each individual binary string
%       encodes LENGTH(VLB)=LENGTH(VUB)=LENGTH(BITS) variables.
%       COARSE is the coarseness of the binary mapping and is also
%       of length LENGTH(VUB).
%
%  this *.m file created by combining "decode.m" from the MathWorks, Inc.
%  originally created by Andrew Potvin in 1993, with "GDECODE.FOR" written
%  by William A. Crossley in 1996.
%
%       William A. Crossley, Assoc. Prof. School of Aero. & Astro.
%  Purdue University, 2001
%
%  gen is an array [population size , string length], each row is one individual's
chromosome
%  vlb is a row vector [number of parameters], each entry is the lower bound for a
variable
%  vub is a row vector [number of parameters], each entry is the upper bound for a
variable
%  bits is a row vector [number of parameters], each entry is number of bits used for
a variable
%

no_para = length(bits); % extract number of parameters using number of rows in bits
vector
npop = size(gen,1);               % extract population size using number of rows in
gen array
x = zeros(npop, no_para);  % sets up x as an array [population size, number of
parameters]
coarse = zeros(1,no_para); % sets up coarse as a row vector [number of parameters]

for J = 1:no_para,  % extract the resolution of the parameters
        coarse(J) = (vub(J)-vlb(J))/(2^bits(J)-1); % resolution of parameter J
end

for K = 1:npop,  % outer loop through each individual (there may be a more efficient
way to operate on the
                  % gen array) BC 10/10/01
        sbit = 1;                 % initialize starting bit location for a parameter
        ebit = 0;                 % initialize ending bit location

   for J = 1:no_para,     % loop through each parameter in the problem
        ebit = bits(J) + ebit;    % pick the end bit for parameter J
                accum = 0.0;                              % initialize the running
sum for parameter J
      ADD = 1;                                            % add / subtract flag for
Gray code; add if(ADD), subtract otherwise
      for I = sbit:ebit,                   % loop through each bit in parameter J
         pbit = I + 1 - sbit;              % pbit determines value to be added or
subtracted for Gray code
```

```
         if (gen(K,I))                                   % if "1" is at current
location
            if (ADD)                                             % add if
appropriate
               accum = accum + (2.0^(bits(J)-pbit+1) - 1.0);
               ADD = 0;                                  % next time subtract
            else
               accum = accum - (2.0^(bits(J)-pbit+1) - 1.0);
               ADD = 1;                                  % next time add
            end
         end
      end                                                                 % end of I
loop through each bit
      x(K,J) = accum * coarse(J) + vlb(J);               % decoded parameter J for
individual K
      sbit = ebit + 1;
                 % next parameter starting bit location
   end                                          % end of J loop through each
parameter
end                                     % end of K loop through each individual

%end gdecode


function [new_gen,mutated] = mutate(old_gen,Pm)
%MUTATE Changes a gene of the OLD_GEN with probability Pm.
%        [NEW_GEN,MUTATED] = MUTATE(OLD_GEN,Pm) performs random
%       mutation on the population OLD_POP.  Each gene of each
%       individual of the population can mutate independently
%       with probability Pm.  Genes are assumed possess boolean
%       alleles.  MUTATED contains the indices of the mutated genes.
%
%        Copyright (c) 1993 by the MathWorks, Inc.
%        Andrew Potvin 1-10-93.

mutated = find(rand(size(old_gen))<Pm);
new_gen = old_gen;
new_gen(mutated) = 1-old_gen(mutated);

% end mutate


function [new_gen,nselected] = tourney(old_gen,fitness)
%TOURNEY Creates NEW_GEN from OLD_GEN, based on tournament selection.
%         [NEW_GEN,NSELECTED] = TOURNEY(OLD_GEN,FITNESS) selects
%        individuals from OLD_GEN by competing consecutive individuals
%         after random shuffling.  NEW_GEN will have the same number of
%         individuals as OLD_GEN.
%        NSELECTED contains the number of copies of each individual
%         that survived.  This vector corresponds to the original order
%         of OLD_GEN.
```

```
%
%         Created on 1/21/96 by Joel Grasmeyer

% Initialize nselected vector and indices of old_gen
new_gen = [];
nselected = zeros(size(old_gen,1),1);
i_old_gen = 1:size(old_gen,1);

% Perform two "tournaments" to generate size(old_gen,1) new individuals
for j = 1:2,

  % Shuffle the old generation and the corresponding fitness values
  [old_gen,i_shuffled] = shuffle(old_gen);
  fitness = fitness(i_shuffled);
  i_old_gen = i_old_gen(i_shuffled);

  % Keep the best of each pair of individuals
  index = 1:2:(size(old_gen,1)-1);
  [min_fit,i_min] = min([fitness(index);fitness(index+1)]);
  selected = i_min + [0:2:size(old_gen,1)-2];
  new_gen = [new_gen; old_gen(selected,:)];

  % Increment counters in nselected for each individual that survived
  temp = zeros(size(old_gen,1),1);
  temp(i_old_gen(selected)) = ones(length(selected),1);
  nselected = nselected + temp;

end

% end tourney


function [new_gen,index] = shuffle(old_gen)
%SHUFFLE Randomly reorders OLD_GEN into NEW_GEN.
%         [NEW_GEN,INDEX] = MATE(OLD_GEN) performs random reordering
%        on the indices of OLD_GEN to create NEW_GEN.
%         INDEX is a vector containing the shuffled row indices of OLD_GEN.
%
%         Created on 1/21/96 by Joel Grasmeyer

[junk,index] = sort(rand(size(old_gen,1),1));
new_gen = old_gen(index,:);

% end shuffle


function [new_gen,sites] = uniformx(old_gen,Pc)
%UNIFORMX Creates a NEW_GEN from OLD_GEN using uniform crossover.
%          [NEW_GEN,SITES] = UNIFORMX(OLD_GEN,Pc) performs uniform crossover
%         on consecutive pairs of OLD_GEN with probability Pc.
%          SITES shows which bits experienced crossover.  1 indicates
%          allele exchange, 0 indicates no allele exchange.  SITES has
%          size(old_gen,1)/2 rows.
%
%          Created 1/20/96 by Joel Grasmeyer

new_gen = old_gen;
sites = rand(size(old_gen,1)/2,size(old_gen,2)) < Pc;
for i = 1:size(sites,1),
  new_gen([2*i-1 2*i],find(sites(i,:))) = old_gen([2*i
2*i-1],find(sites(i,:)));
end

% end uniformx
```