Change 1 – Custom temperature profile using the sigmoid and hyperbolic tangent function

**A custom temperature profile was tested using the tanh and sigmoid functions. After testing the tanh function provided the best option for temperature ramping. This is largely due to the large derivative of the tanh function at values close to 0. This provides quick initial temperature ramping that levels off at values approaching 2; while the sigmoid function levels off much slower (Figure 1). Of course you can increase the x range of the sigmoid so that is it comparable to tanh in terms of slope, but the tanh function still provided the best results overall.**

%0:default, 1:linear, 2:adaptiveV1, 3:adaptive(tanh), 4:adaptive(sig)

tempFunction = [3,3];

idx = linspace(0,3,numCoolingLoops); % Set the boundaries for tanh/sig Temp function

case 3 %'tanh'

fracFunction = @(idx) tanh(-idx);

tCurrentFunction = @(frac,tCurrent,tEnd) tCurrent\*frac + tEnd;

case 4 %'sig'

fracFunction = @(idx) 1/(1+exp(idx));

tCurrentFunction = @(frac,tCurrent,tEnd) tCurrent\*frac + tEnd;



Figure 1 – tanh and sigmoid functions. The right plot indicates the function derivatives.

Change 2 – Custom temperature profile using an adaptive approach

**An adaptive approach was explored but did not perform as well as the tanh function.**

case 2 %'adaptiveV1'

fracFunction = @(D\_b,D\_j,tCurrent) 1.0+((D\_b-D\_j)/D\_b);

tCurrentFunction = @(frac,tCurrent) frac \* tCurrent;

Change 3 – Custom route perturbation method

**This innovative perturbation approach was used to perturb the route following each equilibrium step. This custom approach takes three inputs: the first initializes the option, the second sets the perturbation range, and the final one sets the likelihood of occurrence:**

customPermRoute = [false,5,0.1]; %Custom perm route input

**Once the method is initialized, the likelihood of occurrence is tested randomly against a** rand(1) **function, if successful, it will progress. Else, it will use the default** perturbRoute **function provided.**

if (customPermRoute(1) == true) && (rand(1) < (customPermRoute(3)-(0.001\*numAcceptedSolutions)))

cityRoute\_j = perturbRouteV2(numCities,cityRoute\_b,customPermRoute(2));

else

cityRoute\_j = perturbRoute(numCities,cityRoute\_b);

end

**The function operates by randomly selecting an index within the range of cities (**randIndex1**). The perturbation range then selects an upper and lower index to randomize (**randIndex2**). If both indices exist, then one is chosen at random, else, the one that exists in the range is chosen. For example, if a random index of 108 is chosen with a perturbation range of 5, then the upper and lower index would be 113 and 103, respectively. Since 113 doesn’t exist, it defaults to 103. If 100 is chosen, there is a random chance 95 or 105 is chosen. Once the range is determined, the range of indices –** randIndex1 **to** randIndex2 **– is randomized, then placed back in the same order in the original city route. The perturbation range greatly increases the disordering of the sequence, as mixing up 5 cities has a lower impact than mixing up 30 cities. As well, built into the likelihood of occurrence is an additional metric which decreases the likelihood of occurrence if many improved solutions have been found (**numAcceptedSolutions**). If the loops are lagging and are not finding optimal solutions, then the chances of initializing the function are greater (This was not modularized as I couldn’t find the best values to decrease the likelihood with different numbers of equilibrium/cooling loops).**

function [theCityRoute] = perturbRouteV2(numCities,theCityRoute,permValue)

randIndex1 = randi(numCities);

randRange = [randIndex1-permValue, randIndex1+permValue];

idx = find(randRange >= 1 & randRange <= 110);

if length(idx) ~= 1

if round(rand == 1)

randIndex2 = randRange(1);

else

randIndex2 = randRange(2);

end

else

randIndex2 = randRange(idx);

end

if randIndex1 < randIndex2

permRange = theCityRoute(randIndex1:randIndex2);

theCityRoute(randIndex1:randIndex2) = permRange(randperm(length(permRange)));

else

permRange = theCityRoute(randIndex2:randIndex1);

theCityRoute(randIndex2:randIndex1) = permRange(randperm(length(permRange)));

end

end

Change 4 – Custom Loop Schedule

**With 110 cities the solution space is too large too imagine. Therefore, intuitively it makes sense that quick searches of the solution space is preferable over fewer, longer searches of smaller regions of the space. A custom loop schedule was employed which starts at a lower number of cooling and equilibrium cycles, in an attempt to quickly find good starting routes. After 90% of the iterations are made,** overRideBest()**is used to keep the best routes from the first 90% of iterations. With the better route resulting from a large number of search spaces, this route can be used to dig deeper in the search space for the following 10% if iterations. Since the Loop Schedule is ramped as well (See Figure 2), it does a better job at searching the immediate search space for better routes when a good route is found.**

numCoolingLoops = [1000,10000];

numEquilbriumLoops = [500,1000];

testLength = 10;

numEquilbriumLoopsRange = linspace(numEquilbriumLoops(1),numEquilbriumLoops(2),testLength);

numCoolingLoopsRange = linspace(numCoolingLoops(1),numCoolingLoops(2),testLength);

if i < 0.9\*testLength

[fracFunction,tCurrentFunction] = tempFunctionSelection(tempFunction(2));

[D\_b,D,tCurrentRange,cR,fracAccepted] = annealEUCFunctionV1(pStart,pEnd,numCoolingLoopsRange(i),numEquilbriumLoopsRange(i),...

fracFunction,tCurrentFunction,numCities,cC,tempFunction(2),breakCondition,cR\_Best,overRideBest(1),customPermRoute);

else

[fracFunction,tCurrentFunction] = tempFunctionSelection(tempFunction(2));

[D\_b,D,tCurrentRange,cR,fracAccepted] = annealEUCFunctionV1(pStart,pEnd,numCoolingLoopsRange(i),numEquilbriumLoopsRange(i),...

fracFunction,tCurrentFunction,numCities,cC,tempFunction(2),breakCondition,cR\_Best,overRideBest(2),customPermRoute);

end

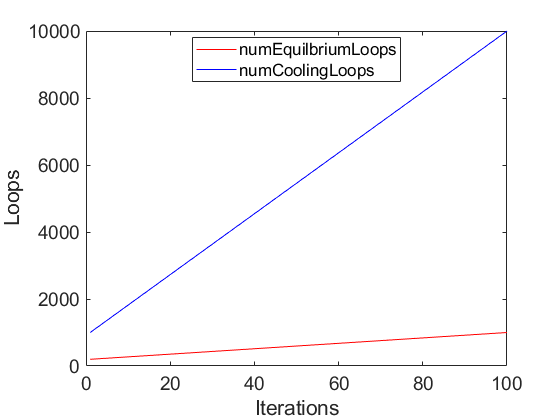
****

Figure 2 – loop ramping with increase in number of iterations

Change 5 – Break Condition

**In some cases during testing a break condition was used. It was found that the break condition lowers run times, but did not increase annealing performance. Can be useful when searching the search space for the first 90% of iterations.**

breakCondition = true;

if breakCondition == true

if i~=1

if (abs(D(i)-D(i-1))/D(i-1) < 1E-7); break; end

end

end

Change 6 – Override Best Condition

**As mentioned above, rather than starting a new route the best previous route can be used. This should be used sparingly as once you are set on a route, it is almost impossible to perturb the route out of a bad local minimum.**

overRideBest = [false,true];

if isempty(cR\_Best) || overRideBest == false

cityRoute\_i = randperm(numCities); % Get initial route

else

cityRoute\_i = cR\_Best;

end

Change 7 – Excel exporter to store best values for testing purposes

**For testing purposes, you can iterate through any hyperparameter you wish to use and it will store the best value, and export to an excel sheet. This is useful for testing purposes, as it will log all the values and settings set initially. See testdata.xlsx for previous runs of the model. In sheet 2, the column H demonstrates the best distances observed. Every other column refers to other parameters selected (**[pStart,pEnd,numEquilbriumLoops(2), numCoolingLoops(2),tempFunction(2),D\_b,length(cC),D\_bBest,cR\_Best]]**) in that order.**

%% Store Optimal Values in Range

if D\_bBest == 0

D\_bBest = D\_b;

cR\_Best = cR;

elseif D\_bBest > D\_b

D\_bBest = D\_b;

cR\_Best = cR;

D\_Best = D;

tCurrentBestRange = tCurrentRange;

if exist('m') == 1; BestIndex(1,:) = m; end %pStart

if exist('l') == 1; BestIndex(2,:) = l; end %numEquilib

if exist('k') == 1; BestIndex(3,:) = k; end %numCooling

if exist('n') == 1; BestIndex(4,:) = n; end %pEnd

end

%% Determine Optimal parameters to Export

if exist('BestIndex') == 1

if BestIndex(1) ~= 0

pStart = pStartRange(BestIndex(1));

elseif BestIndex(2) ~= 0

numEquilbriumLoops = numEquilbriumLoopsRange(BestIndex(2));

elseif BestIndex(3) ~= 0

numCoolingLoops = numCoolingLoopsRange(BestIndex(3));

elseif BestIndex(4) ~= 0

pEnd = pEndRange(BestIndex(4));

end

end

%% Export Into Excel Sheet

filename = 'testdata.xlsx';

[existData] = xlsread(filename,'Sheet1');

newData = [existData;[pStart,pEnd,numEquilbriumLoops(2),numCoolingLoops(2),tempFunction(2),D\_b,length(cC),D\_bBest,cR\_Best]];

xlswrite(filename,newData,'Sheet1'); % write new data into excel sheet.

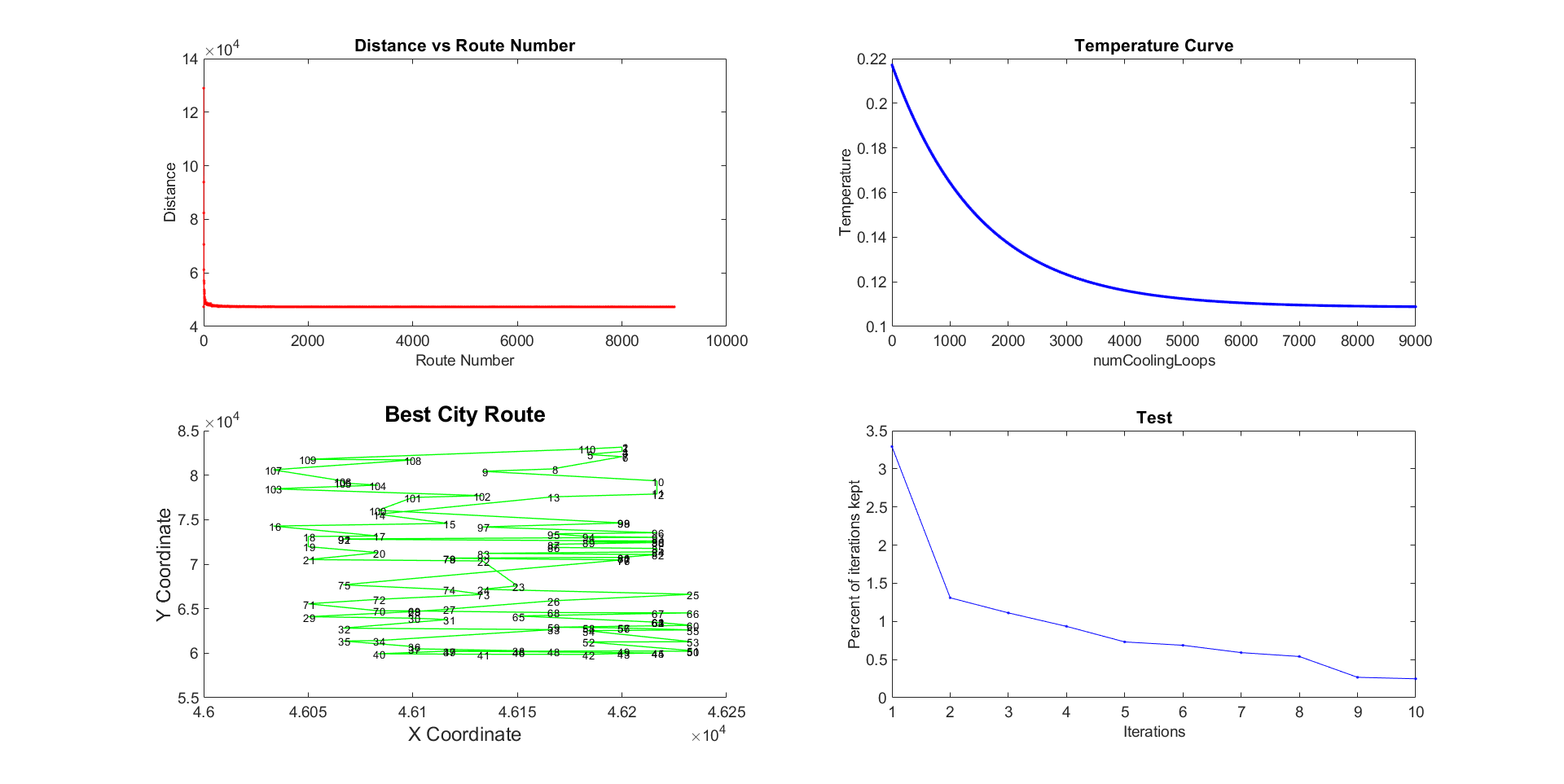


Figure 3 – Output of the best distance achieved D\_b = 47,234; route can be found in BestCR.txt. Past routes can be found in testdata.xlsx. The plot titled “Test” shows the number of solutions that were better in each iteration. Ideally, with increased number of iterations, you should find fewer better solutions. The percent of iterations kept was found to be in the range of 0.5 - 4%.