Simulated Annealing Hw5 ME 575

Ryan Day

March 7, 2019

1 Parameters, solution, and performance

Starting Pt	Cycles	Iter/cycle	Pstart	Pfinish	Max Perturb	Fun Calls
5,5	10	6	1e-6	1e-30	4.5	60

1.1 Methodology

I implemented the simulated annealing algorithm in a function. I then created a run routine that runs the simulated annealing algorithm with factorial combinations of parameters I choose. This run routine would run each combination 1000 times to reduce noise and count the number of times that combination found the optimum. My routine was fast enough that this did not put too big of a burden on the CPU. This helped me explore which parameters helped the simulated annealing algorithm converge the most.

1.2 Starting Points

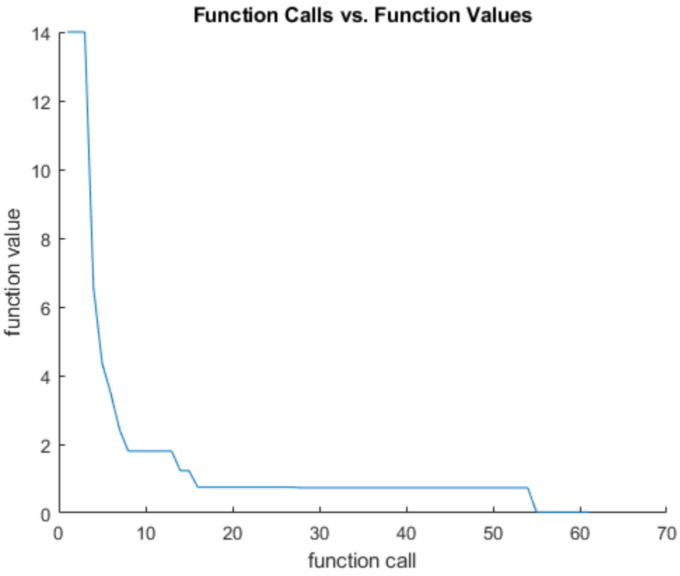
I based my algorithm's performance on how it did from the edge points (5,5;-5,-5;-5,5;5,-5). The edge points did the worst because they were most likely to perturb in a direction that was out of bounds, and so getting the algorithm to get out of the corner was a little slower than if I started more near the center. The points near the middle did better than the edge points in helping the algorithm to converge no matter what parameters I used.

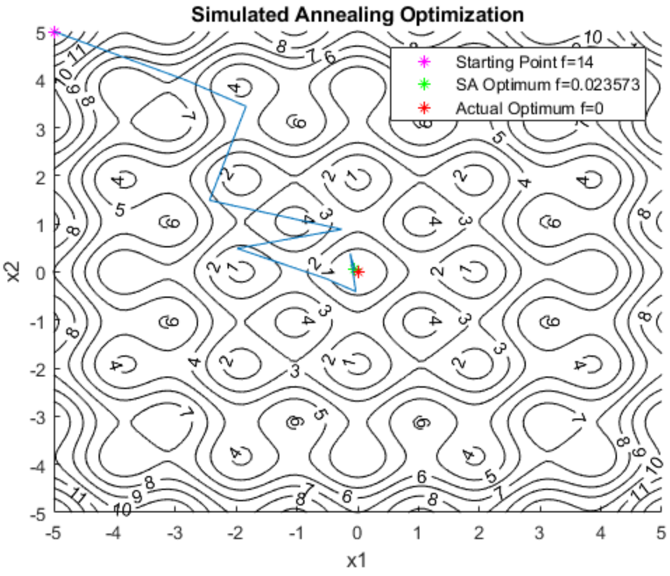
1.3 Parameters

I chose parameters in the table above to minimize function calls and meet the around 7 out of 10 runs reaching the optimum that you provided us with. The algorithm performs better with greater number of cycles and iterations per cycle because it has more chances to perturb the variables to the optimum and stay there. The next parameters were the probability starting and ending values. These optimal parameters also depended on the step size. I chose a higher step size, 4.5. Since we are searching a bounded box of 5 by 5 for an optimum, a step size of 4.5 with enough cycles and iterations practically guarantees that the algorithm will find the optimum eventually. This is because from any point, the algorithm can go to almost any other point. Since the optimum is in the middle, there was a very high chance the algorithm would find this optimum as the parameters were randomly perturbed. Because I used a high step size, it was actually better to have a very low Pstart and low Pfinish. This prevents the algorithm from going uphill, but since the design space was so small, you know that it could always find the global optimum with enough iterations since with a step size of 4.5 the algorithm could reach practically anywhere in the design space from any other point. In this situation, the algorithm would go downhill by directly jumping to another minimum instead of following the curves of the contour plot.

If the design space was much bigger with more variables with many more local minima, it would not be feasible to use the parameters that I did because it would be too computationally expensive. It would be better to have a higher Pstart, somewhere between 0.1 and 0.9 so that the algorithm wasn't relying on randomly finding the optimum with perturbations and instead was doing more valley descending and ascending. With higher dimensional space, it is not as simple to find optima. This is where the true power of simulated annealing lies. It saves computational power, and is advantageous to a random search because it automatically gravitates towards large wells without getting stuck in a local minima.

1.4 Figures for a run with parameters in above table





As you can see, because I gave it a very low Pstart, the algorithm never wandered around too much, staying at about 4 or 5 locations for all 60 perturbations. It would go directly to the next best optimum it found. Like I said previously, while this works in a simple two dimensional problem, with a larger design space and a more complex non-linear function it would be better to have a higher P-start so that the algorithm doesn't get stuck in local optima.

2 Code

2.1 Simulated Annealing algorithm with plotting and contour plots

```
function [x_final, f_final] = simulatedAnnealing(xInit,input,plotIt,
    upperLimit,lowerLimit)
% Initialize parameterized variables
N = input(1);
iterationsPerCycle = input(2);
```

```
perturbValue = input(3);
Pstart = input(4);
Pfinish = input(5);

Tstart = -1/(log(Pstart));
Tfinish = -1/(log(Pfinish));

F = (Tfinish/Tstart)^(1/(N-1)); % Reduction factor per cycle
numberVariables = size(xInit,1);

T = Tstart;
x = xInit;
f = fun(x(1),x(2));

% Initialize histories for plotting
allx = zeros(numberVariables,iterationsPerCycle*N+1);
allf = zeros(iterationsPerCycle*N+1,1);
allx(:,1) =  x;
allf(1) = f;

cycles = 1;
changeFavg = 0;
firstFlag = true;
for index1 = 1:N
    for iteration_index = 1:iterationsPerCycle
        totalIndex = index1*iterationsPerCycle-(iterationsPerCycle-iteration_index)+1;
        [x, f, changeFavg] = iterate(x,f,changeFavg,T,numberVariables,upperLimit,lowerLimit,perturbValue,firstFlag);
        if totalIndex == 2
             firstFlag = false;
        end
        allx(:,totalIndex) = x;
        allf(totalIndex,1) = f;
        cycles = cycles + 1;
    end
        T = F*T;
end
x_final = x;
f_final = f;
if plotIt
    plotCycles(allf,cycles)
    contourPlot(allx,allf)
end


function [x,f,avgChangeF] = iterate(x,f,avgChangeFprev,T,numberVariables,upperLimit,lowerLimit,perturbValue,firstFlag)
    xNew = x;
    % Randomly perturb the variables
    for index =  1:numberVariables
        perturbation = perturb(perturbValue);
        xnewAtIndex = xNew(index)+perturbation;
        % If value passes the limits, set it at the limit
        if xnewAtIndex < lowerLimit(index)
            xnewAtIndex = lowerLimit(index);
        elseif xnewAtIndex > upperLimit(index)
            xnewAtIndex = upperLimit(index);
        end
        xNew(index) = xnewAtIndex;
    end
    fnew = fun(xNew(1),xNew(2));
    changeF = fnew-f;
    % Calculate avg change F
    if firstFlag
        avgChangeF = abs(changeF);
    else
        avgChangeF = (abs(changeF)+avgChangeFprev)/2;
    end
    % If the function is minimized, keep it.
    if changeF < 0
        x = xNew;
        f = fnew;
    % If function is not minimized, check to keep it or not.
    else
        P = boltzmannProb(T,changeF,avgChangeF);
        if rand() < P
            x = xNew;
            f = fnew;
        end
    end
end

function answer = perturb(value)
    answer = rand()*value-(value/2);
end

function prob = boltzmannProb(T,changeE,changeEavg)
    prob = exp(-changeE/(changeEavg*T));
end

function output = fun(x1,x2)
    output = 2.0+0.2.*x1.^2+0.2.*x2.^2 - cos(pi.*x1) - cos(pi.*x2);
end

function plotCycles(allf,cycles)
    clf
    figure(1)
    hold on
    xlabel('function call');
    ylabel('function value');
    title('Function Calls vs. Function Values');
    plot(1:cycles,allf)
end

function contourPlot(allx,allf)
    meshResolution = 0.1;
    [x1,x2] = meshgrid(-5:meshResolution:5,-5:meshResolution:5);
    output = fun(x1,x2);
    figure(2)
    hold on;
    % Plot X0
    x0 = allx(1,1);
    y0 = allx(2,1);
    plot(x0,y0,'m*')
    % Plot SA Optimum
    xSA_Opt = allx(1,end);
    ySA_Opt = allx(2,end);
    plot(xSA_Opt,ySA_Opt,'g*')
    % Plot Optimum
    xOpt = 0;
    yOpt = 0;
    plot(xOpt,yOpt,'r*')

    % Plot Contour
    [C,h] = contour(x1,x2,output,[1:13],'k-'); % Plot Contour
    clabel(C,h,'Labelspacing',500);
    title('Simulated Annealing Optimization');
    xlabel('x1');
    ylabel('x2');
    hold on;

    % Path Lines
    x_pt = allx(1,:);
    y_pt = allx(2,:);
    line(x_pt', y_pt');
    xlim([-5,5])
    ylim([-5,5])
    legend(['Starting Point f=',num2str(allf(1))],['SA Optimum f=',num2str(allf(end))],'Actual Optimum f=0')
end

end
```

2.2 Simulated Annealing runner

```
clear

xInit = [5;-5];

plotIt = false;

Nall = [10];
iterationsPerCycleAll = [6];
perturbValueAll = [1,2,3,4.5];
PstartAll = [1e-6,0.5];
PfinishAll = [1e-6,1e-30];
upperLimit = [5;5];
lowerLimit = [-5;-5];

dFF = fullfact([size(Nall,2),size(iterationsPerCycleAll,2),size(perturbValueAll,2),size(PstartAll,2),size(PfinishAll,2)]);

numberOfExperiments = size(dFF,1);
inputwin = 0;
xwin = 0;

for index = 1:numberOfExperiments
```

```
    xhist = 0;
    N = Nall(dFF(index,1));
    iterationsPerCycle = iterationsPerCycleAll(dFF(index,2));
    perturbValue = perturbValueAll(dFF(index,3));
    Pstart = PstartAll(dFF(index,4));
    Pfinish = PfinishAll(dFF(index,5));
    input = [N,iterationsPerCycle,perturbValue,Pstart,Pfinish];
    for i = 1:1000
        [x,f] = simulatedAnnealing(xInit,input,plotIt,upperLimit,lowerLimit);
        if (abs(x(1)) < 1.0) && (abs(x(2)) < 1.0)
            xhist = xhist+1;
        end
    end
    if xhist>xwin
        inputwin = input;
        xwin = xhist;
    end
    input
    xhist
    index
end
inputwin
xwin
```