Mechanical Engineering 575
Homework #5

Gregory Vernon

March 25, 2020

1 Problem Summary

1.1 About

Printed circuit boards (PCBs) are the principle component of nearly every modern electronic assembly. PCBs not only provide a platform on which to mount electronic the various electronic components of an assembly but also act as a substrate through which to electronically connect the components. PCBs are a planar geometry, having a thickness that is usually much smaller than its other two dimensions[1]. In aerospace applications PCB assemblies are subject to high dynamic load environments that can induce fatigue-based and/or strength-based failure modes in PCBs. The predominant strategy to decrease the effects of the dynamic loads on a PCB is to increase the first natural frequency (i.e. the *fundamental frequency*) as much as possible. This is primarily done through the use of various *support mechanisms* such as screw mounts or wedge supports.

Our project will use Sandia's *DAKOTA* optimization framework to wrap a finite element code (*CF Crunch*) being developed by Chris and his employer, Coreform LLC. As of the time of this report, while most of the framework is complete, due to the recent COVID-19 pandemic our group did not get a chance to merge our separate work for this assignment. All the components are believed to be finished, but we need to schedule a series of video conferences to *stitch* the components together. Thus I was unable to use the desired framework for this homework problem. Instead, I assembled a *similar* workflow using MATLAB and a student edition of the *ABAQUS* finite element code – this is a similar workflow to that which we have been developing.

1.2 Description of Optimization Effort

Using MATLAB, I wrote a function that performs a differential evolution algorithm to optimize pin locations to maximize the disk's fundamental frequency. While I had intended to construct additional optimization routines (e.g. pattern-search, Latin-hypercube, finite-difference gradient methods, etc), due to the COVID-19 pandemic I had to leave campus where my research-group's Abaqus workstation is located. Therefore I was only able to run this single algorithm. This shouldn't be a problem for the final project, as the workstation we will be using for the final project is a non-Windows machine and is accessible through CAEDEM VPN.

The MATLAB code, included in section 2 of this report, includes routines to export a video of the optimization process. Due to time constraints - again due to COVID-19, I was only able to complete simulations with 1 and 2 pins, however during my initial development I tested up to 10 pin arrangements. I've uploaded

[1] For instance a Micro-Star 845E Max mainboard has dimensions 300mm x 200mm x 1.5mm

the videos of these optimizations to YouTube, the links are provided below. The differential evolution algorithm I implemented assigns each function evaluation a unique *child index* and conducts a tournament wherein for each child index it compares the best-ever entry against the active entry. After the tournament, the next generation is created using the now-updated best-ever children – which we call *parents*. The videos I've uploaded show the evolution of the *parents* as the number of generations increases.
As each function evaluation is independent of all other function evaluations, and as each function evaluation is expensive (15 seconds) I used MATLAB's *Parallel Computing Toolbox*'s parallel for-loop functionality, `parfor`, to evaluate the objective functions in parallel. This did lead to several practical issues that required special attention.

1. Upon job submission, ABAQUS writes an environment file that is specific to the job it has just submitted. This environment file is only needed for a split-second to transfer data from the GUI environment to the finite element executable. However, this environment file doesn't have a unique name. Therefore I had to hardcode an arbitrary pause that allowed this functionality and avoided a read+write segfault. See lines 136-137 in section 2.

2. Cubit's Python interpreter is not thread-safe and thus required me to use MATLAB if I wished to have parallel computing. This also meant that I had to treat Cubit as a standalone executable that would terminate after each mesh was built. This meant that I had to pass in textfiles with the relevant instructions (i.e. pin locations) and each file needed to be appended with its simulation index within the current generation (i.e.inputData_02.csv).

A further limitation that required its own optimization routine is that the student edition of ABAQUS only allows a maximum of 1000 nodes to be used in a simulation. Therefore I wrote a simple bisection routine in the Cubit meshing script that selects an mesh seed-size that tries to place as close to 1000 nodes as possible, without exceeding this limitation.

1. 1-Pin: https://www.youtube.com/watch?v=P1R95vS2fzg

2. 2-Pin: https://www.youtube.com/watch?v=YkONICllWTE

I noted that the differential evolution almost always was able to find the general region of the correct result, however it struggled to converge with any reasonable rate once it "identified" the region of the solution. We will try to implement a *hybrid global-local* optimization routine in our final project so that we can get the benefit of rapid convergence of gradient-based local solvers, combined with the robustness of finding the global minimum provided by gradient-free "global" solution methods such as differential evolution.

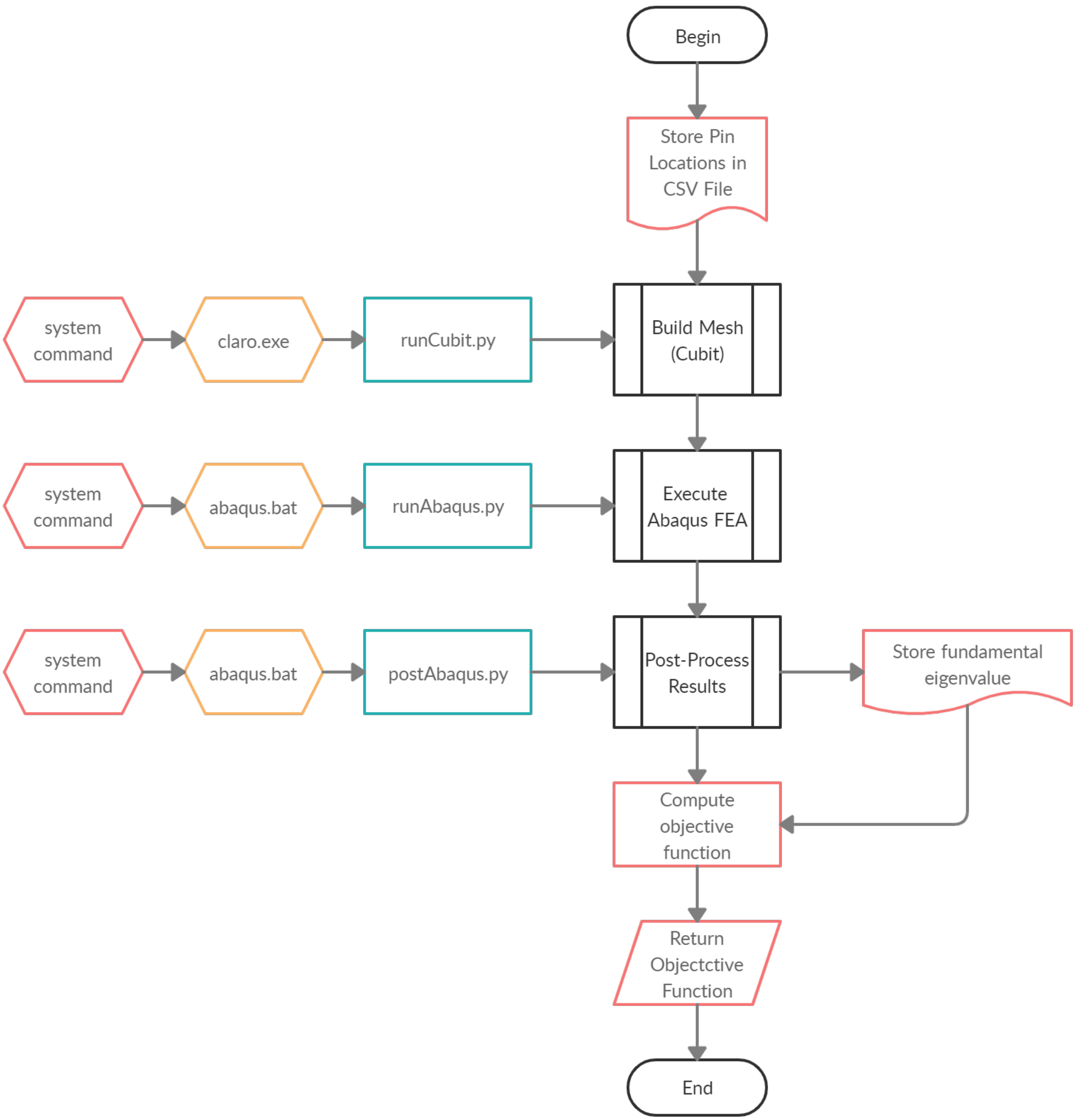

Figure 1: Overview of the FEA workflow contained within the function evaulation routine. General workflow concepts are shown in black boxes, MATLAB functionality is shown in red boxes, external executables are shown in orange boxes, and Python routines are shown in green boxes.

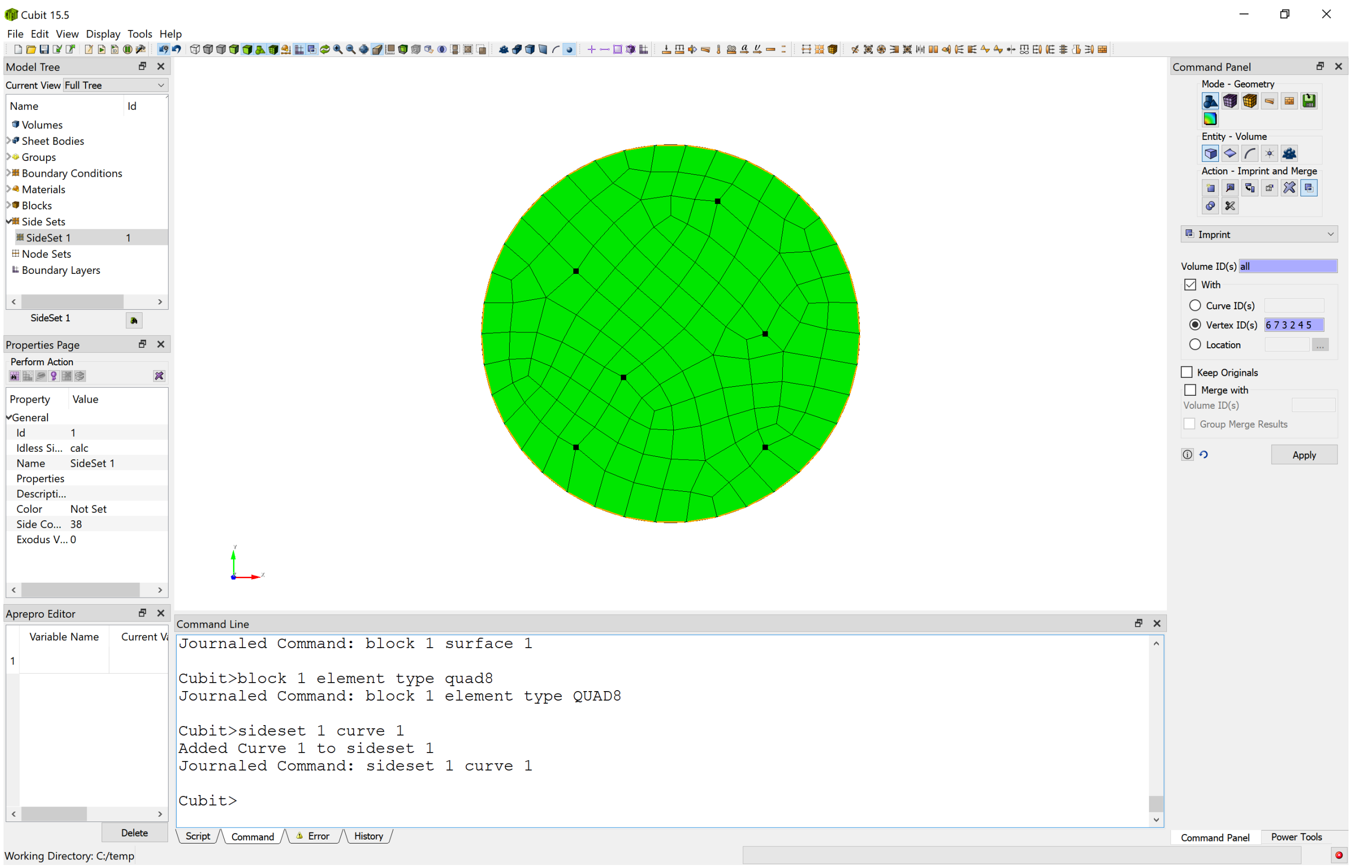

Figure 2: Example of a mesh within interactive Cubit GUI, showing the mesh conforming to the pin locations in a 6-pin optimization

2 Assignment

2.1 Problem #2

```
%% Specify optimization options
Opt.nDOF = 3;
Opt.nChildren = 20;
% Opt.diffWeight = 0.8;
Opt.crossProb = 0.95;
Opt.InitFile = "";

%% Initialize Video Output
VID = VideoWriter("drumEigen.mp4",'MPEG-4');
VID.FrameRate = 4;
VID.Quality = 100;
open(VID);

%% Execute Optimization
Parent = main(Opt,VID);

%% Main Function
% This function is the outer function that executes a differential
% evolution algorithm.
function Parent = main(Opt,VID)
for g = 1:1000
    disp("Generation: " + num2str(g))
    Opt.generation = g;
    if g == 1
        if Opt.InitFile == ""
            Parent = initialize(Opt);
            Child = Parent;
        else
            m = matfile(Opt.InitFile);
            Parent = m.Parent;
            Child = Parent;
        end
    else
        Child = createNextGeneration(Opt,Parent);
        parfor c = 1:Opt.nChildren
            disp("Evaluating function: " + num2str(c))
            f = functionEvaluation(Child(c));
            Child(c).objVal = f;
        end

        Parent = tournament(Parent,Child);
    end
    writeOutput(Opt,Parent,Child);
    plotParents(Opt,Parent,VID);
end
close(VID)
end

%% Create the initial generation
function Parent = initialize(Opt)
nChildren = Opt.nChildren;
nDOF = Opt.nDOF;

```

```
    maxRadius = 1;
    rTol = 0.02;

    x = cell(nChildren,1);
    y = cell(nChildren,1);
    for ii = 1:nChildren
        x{ii} = zeros(nDOF,2);
        for n = 1:nDOF
            theta = rand * 2*pi;
            radius = rand * maxRadius;
            x{ii}(n,:) = [radius*cos(theta) radius*sin(theta)];
        end
        Parent(ii).x = x{ii};
        Parent(ii).objVal = NaN;
        Parent(ii).generation = Opt.generation;
        Parent(ii).individual = ii;
    end

    parfor ii = 1:nChildren
        disp("Evaluating function: " + num2str(ii))
        f = functionEvaluation(Parent(ii));
        Parent(ii).objVal = f
    end
    end

    %% Create the next generation using differential evolution
    function Child = createNextGeneration(Opt,Parent)
    nChildren = Opt.nChildren;
    nDOF = Opt.nDOF;

    parentIDS = 1:nChildren;
    for ii = 1:nChildren
        Child(ii).generation = Opt.generation;
        Child(ii).individual = ii;
        isValid = false;
        while isValid == false
            F = rand + 0.5;
            % Pick three distinct parents
            availParents = true(nChildren,1);
            availParents(ii) = false;
            availParents = parentIDS(availParents);
            pID = availParents(randperm(nChildren-1,3));

            % Evaluate differential evolution
            A = Parent(pID(1)).x;
            B = Parent(pID(2)).x;
            C = Parent(pID(3)).x;
            for n = 1:nDOF
                Child(ii).x(n,:) = A(n,:) + F * (B(n,:) - C(n,:));
            end

            % Evaluate crossover
            for n = 1:nDOF
                r = rand();
                if r >= Opt.crossProb
                    Child(ii).x(n,:) = Parent(ii).x(n,:);
                end
            end

```

```
        radius = sqrt(Child(ii).x(:,1).^2 + Child(ii).x(:,2).^2);
        if all(radius <= 0.975)
            isValid = true;
        end
    end
end
end

%% Function Evaluation -- Execute FEA Workflow
function f = functionEvaluation(Child)
x = Child.x;
simID = Child.individual;
abqPath = 'C:\Program Files\SIMULIA\Commands\';
cubPath = 'C:\Program Files\Cubit 15.4\bin\';
%%% Write input data file
fName = "inputData_" + num2str(simID) + ".csv";
csvwrite(fName,x);
%%% Create Mesh in Cubit
command = string(['"' cubPath 'claro.exe" -nobanner -nographics -nojournal -noecho ...
    -information off -batch simID=' num2str(simID) ' runCubit.py']);
command = strjoin(command);
[status,~] = system(command);

%%% Run Abaqus Simulation
workerID = labindex;
pause(workerID/10); % Helps avoid .rec file name clashing
command = string(['"' abqPath 'abaqus.bat" cae noGUI=runAbaqus.py -- ', ...
    num2str(simID)]);
command = strjoin(command);
[status,~] = system(command);
success = isfile("drumEigen_"+num2str(simID)+".odb");
if success == true
    %%% Post-process Abaqus Simulation
    command = string(['"C:\Program Files\SIMULIA\Commands\abaqus.bat" cae ...
        noGUI=postAbaqus.py -- ', num2str(simID)]);
    command = strjoin(command);
    [status,~] = system(command);
    %%% Read in objective function
    success = isfile("objectiveFunction_" + num2str(simID) + ".csv");
    if success == true
        f = fileread("objectiveFunction_" + num2str(simID) + ".csv");
        f = str2double(f);
        f = -f;
        delete("objectiveFunction_" + num2str(simID) + ".csv");
    else
        f = inf;
    end
else
    f = inf;
end
end


%% Tournament
function Parent = tournament(Parent,Child)
for ii = 1:length(Parent)
    if Child(ii).objVal < Parent(ii).objVal
        Parent(ii) = Child(ii);
    end
```

```
    end
end

%% Save current parents to disk
function writeOutput(Opt,Parent, Child)
if Opt.generation == 1
    save("drumEigen_results.mat",'-v7.3',"Child","Opt","Parent");
else
    m = matfile("drumEigen_results.mat",'Writable',true);
    m.Opt = Opt;
    m.Parent = Parent;
    m.Child = Child;
end

for ii = 1:length(Parent)
    if Parent(ii).generation == Opt.generation
        try
            copyfile("drumEigen_" + num2str(ii) + ".cae","drumEigen_Parent_" + ...
                num2str(ii) + ".cae");
            copyfile("drumEigen_" + num2str(ii) + ".odb","drumEigen_Parent_" + ...
                num2str(ii) + ".odb");
        end
    end
end
end

%% Plot the current parents and write frame to video
function plotParents(Opt,Parent,VID)
X = {Parent.x};
close all
figure
hold on;

t = linspace(0,2*pi,10000);
x = cos(t);
y = sin(t);
plot(x,y);

for ii = 1:length(X)
    scatter(X{ii}(:,1),X{ii}(:,2),60,'.')
end
axis equal
title("Parents @ Generation: " + num2str(Opt.generation))
drawnow

f = getframe(gcf);
writeVideo(VID,f);
end
```

2.2 Problem #3

```
clear
close all

n = 2:20;
```

```
time = nan(7,length(n));
err = nan(7,length(n));
fCount = nan(7,length(n));

for ii = 1:length(n)
    N = n(ii)
    x0 = 20*(rand(N,1)-0.5);
    LB = -10 * ones(size(x0));
    UB =  10 * ones(size(x0));

    objFun = @(x) rosenbrock(x);
    [f,df] = objFun(rand(N)); % Warmup
    exactSolution = ones(N,1);
```

Gradient-Based

Exact Gradient

```
    options = optimoptions('fminunc');
    options.MaxFunctionEvaluations = 1e6;
    options.MaxIterations = 1e6;
    options.SpecifyObjectiveGradient = true;
    options.Display = "off";
    tic;
    [sol,~,~,output] = fminunc(objFun,x0,options);
    t_elapsed = toc;
    err(1,ii) = norm(exactSolution - sol);
    time(1,ii) = t_elapsed;
    fCount(1,ii) = output.funcCount;
```

Forward Difference

```
    options = optimoptions('fminunc');
    options.MaxFunctionEvaluations = 1e6;
    options.MaxIterations = 1e6;
    options.FiniteDifferenceType = "forward";
    options.OptimalityTolerance = 1e-12;
    options.Display = "off";
    tic;
    [sol,~,~,output] = fminunc(objFun,x0,options);
    t_elapsed = toc;
    err(2,ii) = norm(exactSolution - sol);
    time(2,ii) = t_elapsed;
    fCount(2,ii) = output.funcCount;
```

Centered Difference

```
    options = optimoptions('fminunc');
```

```
options.MaxFunctionEvaluations = 1e6;
options.MaxIterations = 1e6;
options.FiniteDifferenceType = "central";
options.OptimalityTolerance = 1e-12;
options.Display = "off";
tic;
[sol,~,~,output] = fminunc(objFun,x0,options);
t_elapsed = toc;
err(3,ii) = norm(exactSolution - sol);
time(3,ii) = t_elapsed;
fCount(3,ii) = output.funcCount;
```

Gradient Free

Simulated Annealing

```
options = optimoptions("simulannealbnd");
options.MaxFunctionEvaluations = 1e30;
options.MaxIterations = 1e30;
options.Display = "off";
tic;
[sol,~,~,output] = simulannealbnd(objFun, x0, LB, UB, options);
t_elapsed = toc;
err(4,ii) = norm(exactSolution - sol);
time(4,ii) = t_elapsed;
fCount(4,ii) = output.funccount;
```

Particle Swarm

```
options = optimoptions("particleswarm");
options.MaxIterations = 1e30;
options.Display = "off";
tic;
[sol,~,~,output] = particleswarm(objFun,N,LB,UB,options);
t_elapsed = toc;
err(5,ii) = norm(exactSolution - sol);
time(5,ii) = t_elapsed;
fCount(5,ii) = output.funccount;
```

Genetic Algorithm

```
options = optimoptions("ga");
options.MaxGenerations = 1e30;
options.Display = "off";
tic;
[sol,~,~,output] = ga(objFun,N,[],[],[],[],LB,UB,[],[],options);
t_elapsed = toc;
err(6,ii) = norm(exactSolution - sol);
```

```
    time(6,ii) = t_elapsed;
    fCount(6,ii) = output.funccount;
```

Pattern Search

```
    options = optimoptions("patternsearch");
    options.MaxFunctionEvaluations = 1e30;
    options.MaxIterations = 1e30;
    options.Display = "off";
    tic;
    [sol,~,~,output] = patternsearch(objFun,x0,[],[],[],[],[],[],[],options);
    t_elapsed = toc;
    err(7,ii) = norm(exactSolution - sol);
    time(7,ii) = t_elapsed;
    fCount(7,ii) = output.funccount;
```

```
end

legendNames = ["Exact Gradient", "Forward Difference", "Centered Difference", ...
    "Simulated Annealing", "Particle Swarm", "Genetic Algorithm", "Pattern Search"];

figure();  % Plot Time per method vs nVar
plot(n, time)
legend(legendNames)
xlabel("# Variables")
ylabel("Elapsed Time (s)")
ax = gca;
ax.YScale = "log";

figure();  % Plot Err per method vs nVar
plot(n, err)
legend(legendNames)
xlabel("# Variables")
ylabel("Error Norm")
ax = gca;
ax.YScale = "log";

figure();  % Plot nFuncEval per method vs nVar
plot(n,fCount)
legend(legendNames)
xlabel("# Variables")
ylabel("# Function Evaluations")
ax = gca;
ax.YScale = "log";
```

Functions

```
function [f, df] = rosenbrock(x)
% Compute function value
```

```
f = 0;
for ii = 1:length(x)-1
    f = f + 100*(x(ii+1) - x(ii)^2)^2 + (1-x(ii))^2;
end

if nargout > 1 % gradient required
    N = length(x);
    df = zeros(N,1);
    for ii = 1:N
        if ii == 1
            df(ii) = 2*x(ii) - 400*x(ii)*(x(ii+1) - x(ii)^2) - 2;
        elseif ii == N
            df(ii) = 200*x(ii) - 200*x(ii-1)^2;
        else
            df(ii) = -200*x(ii-1)^2 + 202*x(ii) - 400*x(ii)*(x(ii+1)-x(ii)^2) - 2;
        end
    end
end
end
```

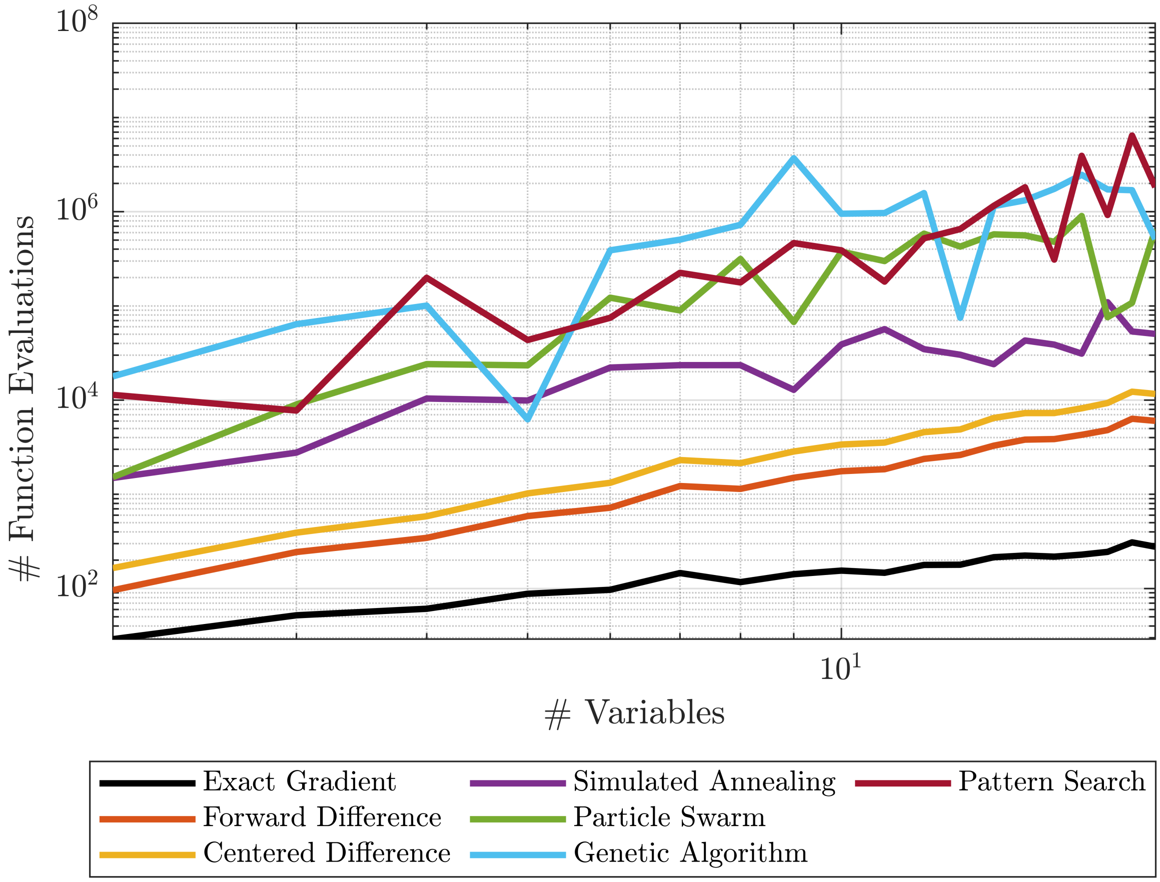

Figure 3: Comparing the total number of function calls for gradient-based solvers and gradient-free solvers on the N-D Rosenbrock problem from N = 2:20

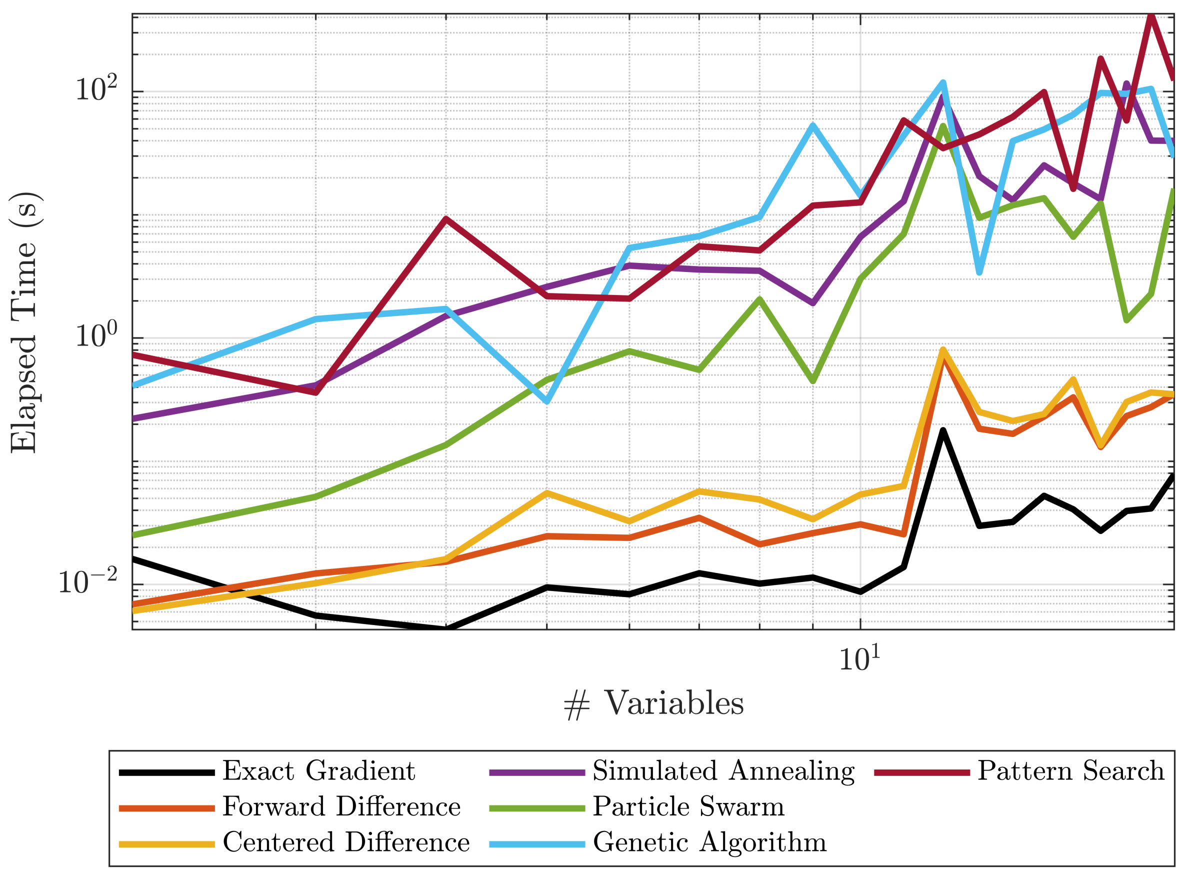

Figure 4: Comparing the wall-clock time for gradient-based solvers and gradient-free solvers on the N-D Rosenbrock problem from N = 2:20

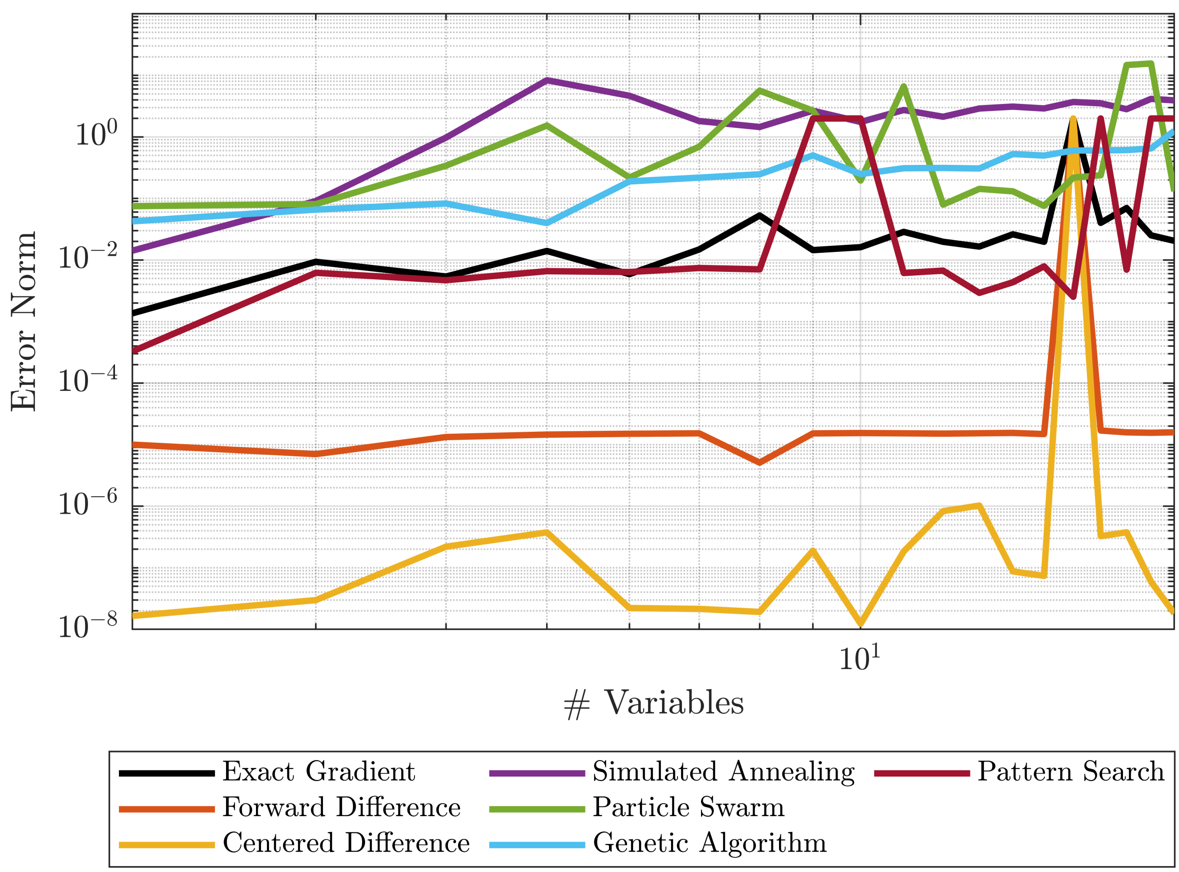

Figure 5: Comparing the norm of the error for gradient-based solvers vs gradient-free solvers on the N-D Rosenbrock problem from N = 2:20. Note the odd spike at N=16 for the gradient based methods.

3 Discussion

I discussed future plans for the project in section 1, so here I will discuss the results from section 3. I executed this code within MATLAB Live Editor, which negatively impacted the performance of all the routines. Due to the long runtimes for the gradient-free methods, I was only able to evaluate problems up to $N \approx 20$ within a reasonable amount of time (≈ 30 min). It's difficult to verify trends, however it is clear that the gradient-free methods require a considerably higher amount of function evaluations (figure 3) than their gradient-based counterparts. This directly corresponds to a higher wall-clock time (figure 4) for these gradient-free methods. Interestingly, as shown in figure 5, while using the `CheckGradients` method in `fminunc` showed no issues with the provided gradient, the exact gradient solutions were hardly distinguishable from the gradient-free methods. Additionally, all of the gradient-based methods struggled to find the solution at $N = 16$. It is unknown to me what the cause of this might be.