Thomas Satterly
AAE 550, Final Project Report

Geometry Optimization of a Scramjet Combustor

Objective

This project aims to optimize the geometry of a rectangular profile scramjet combustor for maximum thrust by varying the contour of one of the combustor walls. For simplification purposes, it is assumed that the nozzle perfectly and isentropically expands the resultant flow, producing the maximum thrust possible for a given combustor design.

For this project, it is assumed the scramjet is operating at a 20 km altitude, cruising at Mach 5, and ingesting 100 kg/s of air with a 0.3 pressure recovery factor and a 1/3 Mach recovery factor through the inlet and isolator system. The combustor has a rectangular cross section with a 5:1 aspect ratio and is 3 meters long. A single wall is split into 20 equal segments whose angles can be individually altered to create a contour. Hydrogen fuel is used at an equivalency ratio of 0.9. Further details on the fuel injection regime can be found in the appendix.

The maximum jet thrust produced by the combustor is a function of the exit conditions of the combustor, which themselves are a function of the combustor properties. For easier representation, the thrust equation is reduced to the following form:

$$T(x) = F(x, a_i)$$
$$x = [\alpha_1, \alpha_2\ \dots \alpha_n\,]^{\mathrm{T}}$$

In this representation, α is the angle of the n^{th} wall section that together compose the design variable matrix x, and a_i is the collection of parameters that defines the initial starting conditions. Transforming the problem from a maximization to minimization problem, the objective becomes:

$$Minimize: f(x) = 1/T(x)$$

Constraints

Constraints were placed on the model to keep the solution feasible and practical. First, and consecutive wall segments along the contour must be angles within 5 degrees of each other to prevent large changes in combustor geometry that could cause flow separation or strong shocks in the flow. Additionally, the wall angles were bounded to greater than or equal to 0, but less than or equal to 25 degrees. Second, at any point in the flow, the Mach number must be greater than or equal to 1.05. Perturbations in the upstream flow could cause the Mach number to drop, and if Mach 1 is reached, the flow chokes and risks unstarting the inlet system, causing a dramatic loss of thrust and, most likely, a failed mission. The exit Mach from the combustor must be at least 1.15 so that afterburning remains feasible without running a great risk of thermally choking the flow. Finally, the maximum temperature in the combustor is limited to 3000 K in order to keep heat loads manageable.

The nonlinear constraint functions, $g_i(x)$, derived from the physical constraints above are as follows:

$$\frac{\alpha_{i+1} - \alpha_i}{5} - 1 \leq 0$$

$$\frac{\alpha_i - \alpha_{i+1}}{5} - 1 \leq 0$$

$$1 - \frac{M_m}{1.05} \leq 0$$

$$1 - \frac{M_{last}}{1.15} \leq 0$$

$$\frac{T_m}{3000} - 1 \leq 0$$

As before, α_i is the angle of a wall section and is the design variable of interest. M_m is the Mach number at the m^{th} integration step, and likewise, T_m is the static temperature at the m^{th} integration step. The total number of nonlinear constraint equations depends upon the total number of wall sections (whose angle can be varied) and the total number of integration steps along the combustor. In a model with 20 wall sections and an integration step of 1 cm produces 679 constraint equations. The lower and upper bounds for wall angles are linear constraints, and as such, are simple and not represented in the above equations.

Scramjet Combustor Model

The scramjet combustor model is given by Shapiro as a 1-D analysis, of which the main relations are given below.

$$\frac{dM}{dx} = M\left(\frac{1 + \frac{\gamma - 1}{2}M^2}{1 - M^2}\right)\left(-\frac{1}{A}\frac{dA}{dx} + \frac{1 + \gamma M^2}{2}\frac{1}{T_t}\frac{dT_t}{dx}\right)$$

$$\frac{1}{T_t}\frac{dT_t}{dx} = \frac{1}{\dot{m}}\frac{d\dot{m}}{dx}\left(\frac{h}{c_p T_t} - 1\right)$$

Standard isentropic relations for compressible flows as well as the continuity equations combine with those provided by Shapiro to create a unified model. The changing values of the specific heat (c_p) and the ratio of specific heats (γ) are calculated using NASA's Chemical Equilibrium with Applications (CEA) program, which is a chemical property solver developed for determining the thermodynamic properties and physical state of complex mixtures of reactants. The effects of cooling at the wall and wall friction were ignored for simplification purposes. To determine combustor properties, the above equations are numerically integrated while using CEA to update thermodynamic properties at each integration step.

The major functional portions of code are provided in the appendix. The full code base, which is too large to reasonably attach in PDF form, is available in full upon request.

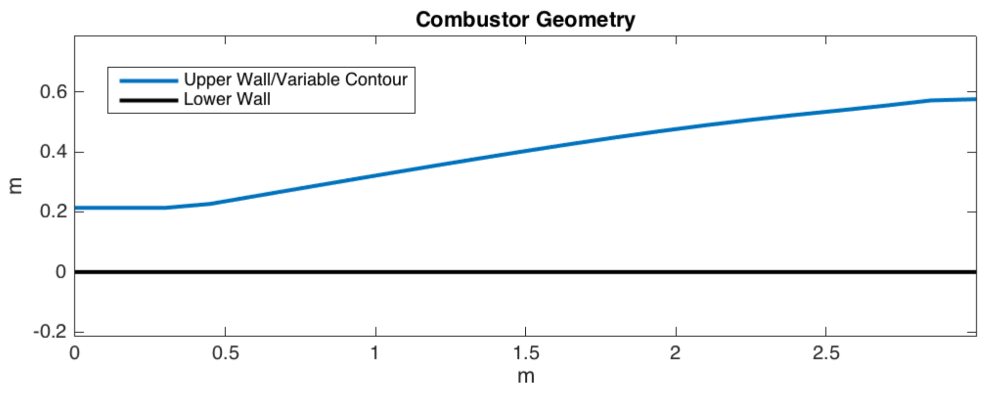
Methodology

The Sequential Quadratic Programing (SQP) optimization algorithm was used for a multitude of reasons. With the combustor taking significant computation time to solve, the fast and relatively efficient nature of SQP will make the problem more feasible in the allotted time. There is a significant chance the optimization procedure will venture into unfeasible territory easily, which SQP can handle. Additionally, the non-linear nature of the flow field constraints will be better represented by SQP than other strictly linear methods. There was a risk of stepping outside of the continuous domain of the objective function, which occurs when the Mach number in the combustor drops below 1, but the constraints put in place due to the physical ramifications of this scenario prevent such discontinuities from occurring. Matlab's "fmincon" function with the SQP algorithm was used in practice.

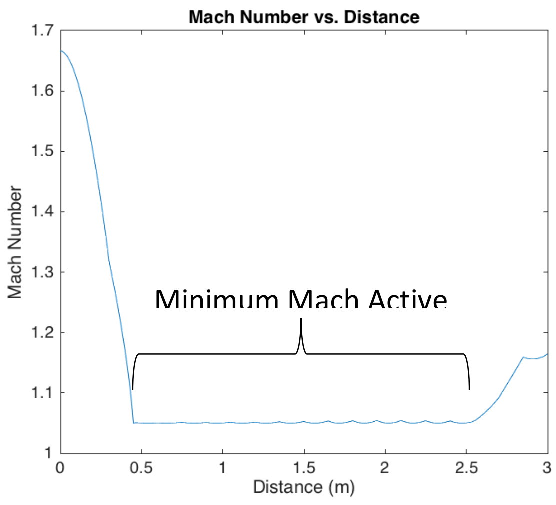
The scramjet combustor model was also developed in Matlab. Calculating the gradients of the objective and constraint functions was not possible to do analytically, and extremely time consuming when done numerically. To make gradient calculations more feasible, a significant amount of effort was put into making the model more computationally efficient as well as parallelizing numeric gradient calculations. All gradients were calculated using the forward difference method in order to minimize the number of objective function calls. The model was also developed to have flexible integration fidelity so that, should optimization become intractable within the allotted project time, a low-fidelity solution could be found.

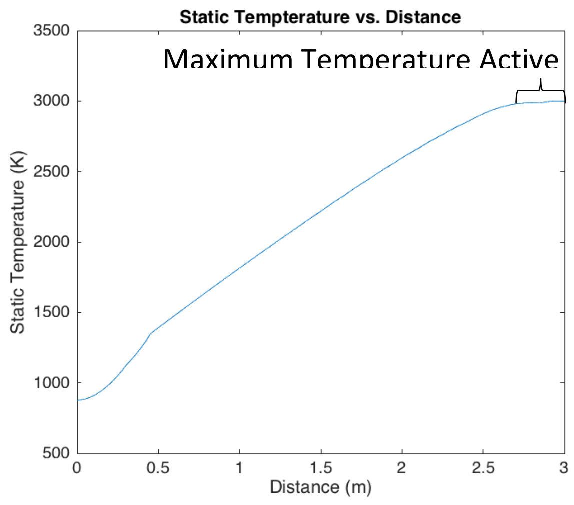
Results

A combustor was optimized using a 1 cm integration step resolution and 20 variable angle wall segments, which produces a thrust of 332.982 kN. The SQP algorithm produced an optimal solution (local minimum that satisfies constraints) after 120 iterations and 2856 function calls. The starting point (x_0) for optimization was chosen as a linear progression of wall angles from 15 degrees at the start to 20 degrees at the end, which was chosen to satisfy all constraints. A side view of the optimized geometry for the combustor is shown below.



For most of the length of the combustor, the minimum Mach of 1.05 was the active constraint. At the start of the combustor, the minimum wall angle bound of 0 degrees was active, and towards the end of the combustor, the maximum static temperature constraint became active. Neither the minimum exit Mach, upper bound on wall angles, or maximum sequential angle difference were active at any point, and no constraints were violated. The figures below highlight the active constraint sections along the combustor.

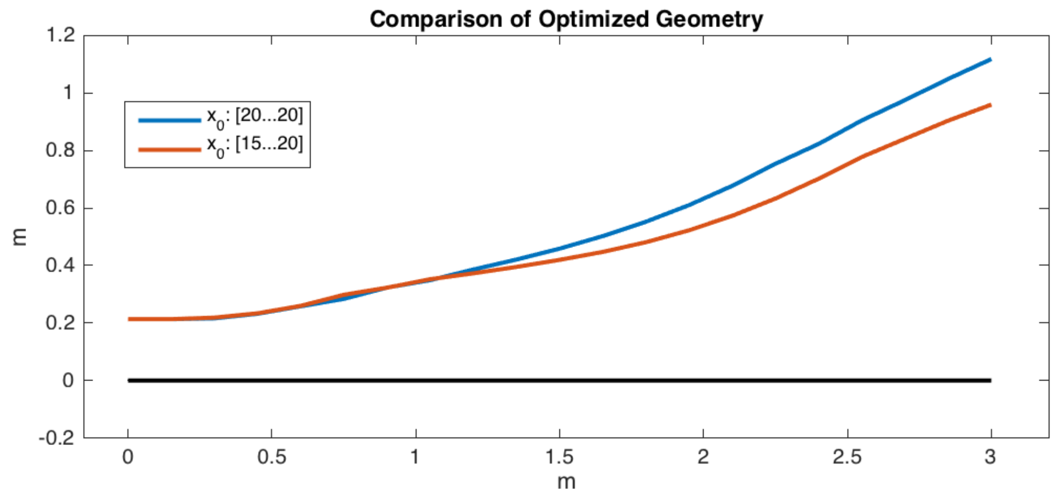




Two more lower fidelity optimizations with a 10 cm integration step size were performed with a reduced maximum temperature of 2700 K and with difference starting points to test if the solution space was multimodal. The first optimization used the same starting point as before, while the second used the starting point of 20 degree angles for all wall sections. It was found that each optimization provided significantly different results, suggesting that the objective function may be multimodal. A table summarizing the differences is shown below.

Thrust (kN)	**Iterations**	**Function Calls**	**x0**
310.505	55	7035	[15 … 20]
303.997	105	11088	[20 … 20]

Although the thrust for each design is within 2.6% of each other, the geometry is much more evidently different. An overlaid comparison of the geometry is shown below.



This result is not conclusive that the same behavior would occur on the higher fidelity model. As the integration step size increases, flow property calculations become inaccurate, and a 10 cm step size was chosen for this comparison solely for the purpose of time constraints. A similar analysis using a 1 cm integration step size as before would have taken two days.

Final Remarks

The approach outlined and executed in the project has shown that the optimization of a scramjet combustor is possible using 1-D analysis and non-constant flow properties, but the most significant result is realizing the amount of processing power needed to perform such optimizations. A single optimization using a 1 cm integration step took 23 hours to complete on a high-performance consumer machine whilst using four logical compute nodes to run parallel code. A bulk of this time was devoted to computing the numerical gradients, as analytic gradients were not possible for the complex flow model used. If a fully realized optimization were desired, including multiple starting points, and higher resolution integration, the time required would rise to the point where dedicated and large compute clusters are required.

The issue of compute time also highlights the importance of optimization algorithm selection. For example, a lower fidelity model was shown to be multimodal, suggesting that a global search method might be more effective than a multi-start SQP optimization. However, global search methods tend to take more objective function calls than a direct approach, which could once again push the compute time required into intractable territory.

Lastly, the effect of model resolution on the number function calls required to find a solution should be noted. On the initial high-fidelity model, optimization took 2856 function calls. By comparison, each of the lower fidelity models took significantly more function calls, at least 7035. Although this was not investigated further in the project, it seems worth mentioning that a minimum optimum fidelity might need to be considered in similar types of problems so that the gradients of the model represent the true gradients accurately enough for efficient optimization.

Works Cited

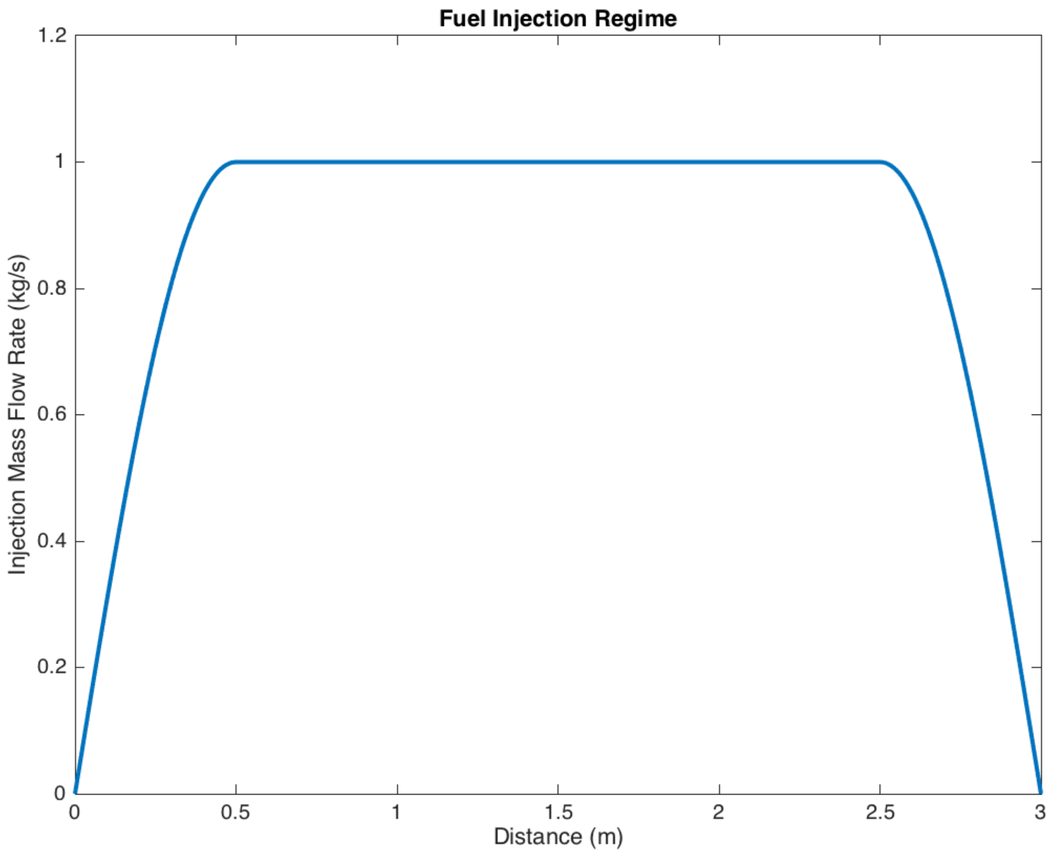
Shapiro, A. H. (1953). *The Dynamics and Thermodynamics of Compressible Fluid Flow*. New York.

Chemical Equilibrium with Applications. (n.d.). Retrieved November 27, 2017, from https://www.grc.nasa.gov/www/CEAWeb/ceaHome.htm

Appendix

The fuel injection regime is defined as follows, in kg/s:

$$\frac{d\dot{m}}{dx} = \sin(\pi x)\,, 0 \le x \le 0.5$$
$$\frac{d\dot{m}}{dx} = 1, 0.5 < x \le 2.5$$
$$\frac{d\dot{m}}{dx} = \sin\big(\pi(x-2)\big)\,, 2.5 < x \le 3$$



The Shapiro 1-D compressible fluid flow equations used in this project are listed below:

$$\frac{dM}{dx} = M\left(\frac{1+\frac{\gamma-1}{2}M^2}{1-M^2}\right)\left(-\frac{1}{A}\frac{dA}{dx}+\frac{1+\gamma M^2}{2}\frac{1}{T_t}\frac{dT_t}{dx}\right)$$

$$\frac{1}{T_t}\frac{dT_t}{dx} = \frac{1}{\dot{m}}\frac{d\dot{m}}{dx}\left(\frac{h}{c_p T_t}-1\right)$$

M: Local Mach number
γ: Local Ratio of specific heats
x: Distance along the combustor
A: Local combustor cross sectional area
T_t: Local total temperature
$\dot{m}$: Local mass flow rate
c_p: Local specific heat at constant pressure

```
classdef Burner < handle
    %BURNER Burner assembly

    properties (SetAccess = private)
        segments = {}; % Cell array of burner segments
        maxStep = 1e-3; % [m] Maximum solution step size
        states; % State array from solutios
        startFlow; % Starting flow of the burner
        injectionFunc; % Function handle of injection function

        width;
        startHeight;
        lengths;
        angles = []; % [deg] Array of angles for each element

        h; % Fuel heating value

        cea;
    end

    methods

        function obj = Burner()
            obj.cea = nasa.CEARunner();
        end

        function setHeatingValue(obj, h)
            obj.h = h;
        end

        function setMaxStep(obj, step)
            obj.maxStep = step;
        end

        function setInjectionFunc(obj, fh)
            assert(isa(fh, 'function_handle'), 'Injection function
 must be a handle!');
            obj.injectionFunc = fh;
        end

        function setStartFlow(obj, flow)

 assert(isa(flow, 'aeroBox.flowFields.FlowElement'), 'Starting flow
 must be burner flow type!');
            obj.startFlow = flow;
        end

        function setGeometry(obj, w, h, l, a)
            assert(numel(l) == numel(a), 'Must have consistant
 dimensions!');
            obj.angles = a;
            obj.width = w;
            obj.startHeight = h;
            obj.lengths = l;
        end

        function setup(obj)
            % Sets up the burner for solving

            % Make assertions to verify all componenets are ready to
setup
            assert(~isempty(obj.injectionFunc), 'Missing injection
function!');
            assert(~isempty(obj.startFlow), 'Missing starting flow!');
            assert(~isempty(obj.angles), 'Missing segment angles!');
            assert(~isempty(obj.h), 'Missing fuel heating value!');

            for i = 1:numel(obj.angles)
                % Create the burner segment
                obj.segments{i} = aae550.final.BurnerSegment('cea',
obj.cea);

                % Set the geometry
                obj.segments{i}.geometry.setWidth(obj.width);
                if i == 1

obj.segments{i}.geometry.setHeight(obj.startHeight);
                else
                    obj.segments{i}.geometry.setHeight(obj.segments{i
- 1}.geometry.getHeight(obj.lengths(i - 1)));
                end
                obj.segments{i}.geometry.setLength(obj.lengths(i));
                obj.segments{i}.geometry.setAngle(obj.angles(i));

                % Set the injection function
                obj.segments{i}.setInjectionFunc(obj.injectionFunc);

                % Set the heating value of the fuel
                obj.segments{i}.setHeatingValue(obj.h);

                % Set to use global injection
                obj.segments{i}.setGlobalInjection();

            end
        end

        function solve(obj)
            % Solves throught the combustor
            obj.setup(); % Sets up the combustor
            x = 0;
            tempFlow = obj.startFlow;
            obj.states = [];

            totalLength = sum(obj.lengths);
            waitString = @(x) sprintf('%0.2f m of %0.2f m', x,
totalLength);
```

```
            wh = waitbar(0, waitString(x));
            updateFH = @(x) waitbar(x / totalLength, wh,
waitString(x));
            for i = 1:numel(obj.segments)
                % Solve the current burner segment
                obj.segments{i}.setWaitFH(updateFH, max(totalLength /
1000, 0.01));
                obj.segments{i}.setFlowElement(tempFlow);
                [tempFlow, newStates] =
obj.segments{i}.solve(ceil(obj.lengths(i) / obj.maxStep), x);
                % Get the length of the combustor for the next
iteration
                x = x + obj.segments{i}.geometry.length;
                obj.states = [obj.states newStates];
            end
            close(wh);
        end

        function plotGeometry(obj)
            % Plots the burner segments
            fh = figure();
            x = [];
            y = [];
            xx = 0;
            for i = 1:numel(obj.segments)
                [xnew, ynew] = obj.segments{i}.getPlotArrays();
                x = [x xnew + xx];
                y = [y ynew];
                xx = xx + obj.lengths(i);
            end
            plot(x, y);
        end

    end

end
```

Published with MATLAB® R2015b

```
classdef BurnerFlow < handle
    %BURNERFLOW Flow with fast kinetic burning, 1D Shapiro relations
    % Created by Thomas Satterly

    properties (SetAccess = private)
        flowElement;
        geometry;
        injectionFunc; % Function for dm_dot/dx
        h; % Heating value of the fuel
        %mdot; % Mass flow rate

        localInjection = 1; % Flag for using local x values for the
 injection function
        cea;
        waitFH;
        dx;
    end

    methods

        function obj = BurnerFlow(varargin)
            np = aeroBox.inputParser();
            np.addParameter('cea', @ishandle);
            np.parse(varargin{:});
            if ~isempty(np.results.cea)
                obj.cea = np.results.cea;
            else
                obj.cea = nasa.CEARunner();
            end
        end

        function setWaitFH(obj, fh, dx)
            obj.waitFH = fh;
            obj.dx =dx;
        end

        function setGlobalInjection(obj)
            obj.localInjection = 0;
        end

        function setLocalInjection(obj)
            obj.localInjection = 1;
        end

        function setHeatingValue(obj, h)
            obj.h = h;
        end

        function setGeometry(obj, geo)
            obj.geometry = geo;
        end
        function setInjectionFunc(obj, fh)
            assert(isa(fh, 'function_handle'), 'Must use function
handle for injection function!');
            obj.injectionFunc = fh;
        end

        function setMassFlowRate(obj, mdot)
            obj.flowElement.setMassFlow(mdot);
        end

        function setFlowElement(obj, flow)
            obj.flowElement = flow;
        end

        function [lastFlow, states] = solve(obj, numSteps, startX)
            if nargin < 2
                numSteps = 1;
            end

            maxX = obj.geometry.getLength();
            stepSize = maxX / numSteps;
            endFlow = obj.flowElement.getCopy();
            x = 0;
            genState = @(x, flow) struct('flow', flow.getCopy(), 'x',
x + startX);
            states = {};
            lastFlow = obj.flowElement.getCopy();

            if obj.localInjection
                injectionFH = obj.injectionFunc;
            else
                injectionFH = @(x) obj.injectionFunc(x + startX);
            end
            i = 0;
            lastWHUpdate = 0;
            while x < maxX
                i = i + 1;
                states{i} = genState(x, lastFlow);
                x = min(x + stepSize, maxX);
                if ~isempty(obj.waitFH)
                    if x > lastWHUpdate + obj.dx
                        obj.waitFH(x + startX);
                        lastWHUpdate = lastWHUpdate + obj.dx;
                    end
                end

                dTt_dx = lastFlow.Tt * (1 / lastFlow.mdot) *
injectionFH(x) * (obj.h / (lastFlow.cp * lastFlow.Tt) - 1);
                Tt = lastFlow.Tt + dTt_dx * stepSize;
                endFlow.setStagnationTemperature(Tt);


                dM_dx = lastFlow.M * ((1 + ((lastFlow.gamma - 1) / 2)
* lastFlow.M^2) / (1 - lastFlow.M^2)) * ...
```

```
                    ((-1 / obj.geometry.getArea(x - stepSize)) *
obj.geometry.getRateOfAreaChange + ...
                    ((1 + lastFlow.gamma * lastFlow.M^2) / 2) * (1 /
lastFlow.Tt) * dTt_dx);
                nextMach = max(1, lastFlow.M + stepSize * dM_dx);
                if nextMach == 1
                    %keyboard;
                end
                endFlow.setMach(lastFlow.M + stepSize * dM_dx);

                endFlow.setMassFlow(endFlow.mdot + injectionFH(x) *
stepSize);

                P = lastFlow.P * (obj.geometry.getArea(x -
stepSize) / obj.geometry.getArea(x)) * (lastFlow.M / endFlow.M) *
sqrt(endFlow.T / lastFlow.T);

                Pt = aeroBox.isoBox.calcStagPressure('mach',
endFlow.M, 'Ps', P, 'gamma', endFlow.gamma);
                endFlow.setStagnationPressure(Pt);
                try
                    params = obj.cea.run('prob', 'tp', 'p(bar)',
P/1e5, 't,k', endFlow.T(), 'reac', 'name', 'Air', 'wt%', 100, 'end');
                    endFlow.setGamma(params.output.gamma);
                    endFlow.setCp(params.output.cp * 1e3);
                catch
                    warning('CEA error....');
                end
                lastFlow = endFlow.getCopy();
            end
            states{end + 1} = genState(x, lastFlow);
            if ~isempty(obj.waitFH)
                obj.waitFH(x + startX);
            end
        end
    end

end
```

Published with MATLAB® R2015b

```
classdef BurnerSegment < aeroBox.flowFields.BurnerFlow
    %BURNERSEGMENT Burner with rectangular cross section, linear
 varying area
    % Created by Thomas Satterly
    properties
    end

    methods
        function obj = BurnerSegment(varargin)
            % Create with rectangular type
            obj@aeroBox.flowFields.BurnerFlow(varargin{:});
            obj.setGeometry(aeroBox.geometric.Rectangular());
        end

        function [x y] = getPlotArrays(obj)
            x = [0, obj.geometry.getLength()];
            y = [obj.geometry.getHeight(0),
 obj.geometry.getHeight(obj.geometry.getLength())];
        end

    end

end
```

Published with MATLAB® R2015b

```
classdef FlowElement < handle
    %FLOW Basic flow element
    % Created by Thomas Satterly
    properties (SetAccess = private)
        gamma; % Ratio of specific heats
        cp; % Specific heat at constant pressure
        R; % Gas constant
        Tt; % Stagnation temperature
        Pt; % Stagnation pressure
        rho_t; % Stagnation density
        M; % Mach number
        mdot; % Mass flow of stream
    end

    methods

        function fe = getCopy(obj)
            % Returns a deep copy of the flow element
            feh = getByteStreamFromArray(obj);
            fe = getArrayFromByteStream(feh);
        end

        function t = T(obj)
            % Returns the static temperature of the flow
            t = aeroBox.isoBox.calcStaticTemp('mach', obj.M, 'Tt',
 obj.Tt, 'gamma', obj.gamma);
        end

        function setCp(obj, cp)
            obj.cp = cp;
        end

        function setGamma(obj, gamma)
            obj.gamma = gamma;
        end

        function setR(obj, R)
            obj.R = R;
        end

        function setStagnationTemperature(obj, t)
            % Sets the stagnation temperature
            obj.Tt = t;
        end

        function setStaticTemperature(obj, t)
            % Sets the flow properties to match the desired static
 temperature
            obj.Tt = aeroBox.isoBox.calcStagTemp('mach', obj.M, 'Ts',
 t, 'gamma', obj.gamma);
        end
        function p = P(obj)
            % Returns the static pressure of the flow
            p = aeroBox.isoBox.calcStaticPressure('mach', obj.M, 'Pt',
obj.Pt, 'gamma', obj.gamma);
        end

        function setStagnationPressure(obj, p)
            % Sets the stagnation pressure
            obj.Pt = p;
        end

        function setStaticPressure(obj, p)
            % Sets the flow properties to match the desired static
pressure
            obj.Tt = aeroBox.isoBox.calcStagPressure('mach',
obj.M, 'Ps', p, 'gamma', obj.gamma);
        end

        function r = rho(obj)
            % Returns the static density
            r = aeroBox.isoBox.calcStaticDensity('mach',
obj.M, 'rho_t', obj.rho_t, 'gamma', obj.gamma);
        end

        function setStagnationDensity(obj, r)
            obj.rho_t = r;
        end

        function setStaticDensity(obj, r)
            obj.rho_t = aeroBox.isoBox.calcStagDensity('mach',
obj.M, 'rho', r, 'gamma', obj.gamma);
        end

        function m = u(obj)
            % Returns the mach number of the flow
            m = obj.M * obj.getSonicVelocity();
        end

        function setMach(obj, m)
            % Sets flow properties to match the desired mach number
            obj.M = m;
        end

        function a = getSonicVelocity(obj)
            % Returns the sonic velocity of the flow
            a = sqrt(obj.gamma * obj.R * obj.T());
        end

        function a = getArea(obj)
            % Returns the area of the flow
            a = obj.A;
        end

        function setMassFlow(obj, mdot)
```

```
            obj.mdot = mdot;
        end

%         function setMassFlow(obj, mdot, variable)
%             switch variable
%                 case 'density'
%                     rho = mdot / (obj.u * obj.A);
%                     obj.rho_t =
 aeroBox.isoBox.calcStagDensity('mach', obj.M, 'rho', rho, 'gamma',
 obj.gamma);
%                 case 'velocity'
%                     obj.u = mdot / (obj.rho() * obj.A);
%                 case 'area'
%                     obj.A = mdot / (obj.rho() * obj.u);
%                 otherwise
%                     error('Invalid input variable ''%s''',
 variable);
%             end
%         end
    end

end
```

Published with MATLAB® R2015b

```matlab
function [Thrust, M, T] = getBurnerThrust(angles)

% Designated fuel flow rate
dmdot_dt = @(x) 1 * ((x <= 0.5) * sin(pi * x) + ...
    (x > 0.5) * (x <= 2.5) * 1 + ...
    (x > 2.5) * (x <= 3) * sin(pi * (x - 2))) + ...
    (x > 3) * 0;

% Operating at 20 km
Pa = 5474.89; % [Pa] Ambient Pressure
Ta = 216.65; % [K] Ambient Temperature
burner = aae550.final.Burner();
burner.setMaxStep(1e-2);

% Set up the geometry
w = 1.067724; % need to calculate this
h = w / 5;

numSegments = numel(angles);
totalLength = 3;
width = w;
height = h;
lengths = ones(1, numSegments) * totalLength / numSegments;


% Setup the initial flow
M0 = 5; % Freestream mach
M3 = M0 / 3; % Mach at isolator exit
pr = 0.3; % Inlet/compression system total pressure recovery factor
mdot = 100; % [kg/s] Mass flow of air at isolator exit
h = 120908000;  % J/kg
startFlow = aeroBox.flowFields.FlowElement();
startFlow.setCp(1216); % J/kg*K
startFlow.setR(287.058); % J/kg*K

startFlow.setGamma(1.4);
startFlow.setMach(M3);
startFlow.setStagnationTemperature(aeroBox.isoBox.calcStagTemp('mach',
 M0, 'gamma', 1.4, 'Ts', Ta));
startFlow.setStagnationPressure(aeroBox.isoBox.calcStagPressure('mach',
 M0, 'gamma', 1.4, 'Ps', Pa) * pr);
startFlow.setMassFlow(mdot);

cea = nasa.CEARunner();
params = cea.run('prob', 'tp', 'p(bar)', startFlow.P()/1e5, 't,k',
 startFlow.T(), 'reac', 'name', 'Air', 'wt%', 100, 'end');

startFlow.setGamma(params.output.gamma);
startFlow.setCp(params.output.cp * 1e3);

burner.setGeometry(width, height, lengths, angles);
burner.setHeatingValue(h);
burner.setInjectionFunc(dmdot_dt);
burner.setStartFlow(startFlow);


% Setup solver


burner.solve();

states = burner.states;
M = zeros(1, numel(states));
for l = 1:numel(states)
     x(l) = states{l}.x;
    flow = states{l}.flow;
    M(l) = flow.M();
     mdot(l) = flow.mdot();
%    u(l) = flow.u();
%    Pt(l) = flow.Pt();
%    Tt(l) = flow.Tt();
%    R(l) =flow.R();
%    cp(l) = flow.cp();
    T(l) = flow.T();
%   P(l) = flow.P();
%    gamma(l) = flow.gamma();
end
if any(M < 1)
    Thrust = 1;
else
    endFlow = burner.states{end}.flow;
    Me = aeroBox.isoBox.machFromPressureRatio('Prat', Pa /
 endFlow.Pt, 'gamma', endFlow.gamma);
    ue = Me * endFlow.getSonicVelocity();
    Thrust = ue * endFlow.mdot();
end

shouldPlot = 1;
if shouldPlot
    figure;
    subplot(1, 2, 1);
    plot(x, M);
    xlabel('Distance (m)');
    ylabel('Mach Number');
    title('Mach Number vs. Distance');

    subplot(1, 2, 2);
    plot(x, T);
    xlabel('Distance (m)');
    ylabel('Static Temperature (K)');
    title('Static Tempterature vs. Distance');

    burner.plotGeometry();
    axis equal
end
end
```

Published with MATLAB® R2015b

```
classdef memBurner < handle
    % Memoized burner function
    properties
        fh; % Memoized function handle
    end
    methods

        function obj = memBurner()
            obj.fh = @(x) aae550.final.getBurnerThrust(x);
            try %#ok<TRYNC>
                obj.fh = memoize(obj.fh);
            end
        end

        function [Thrust, M, T] = getBurnerThrust(obj, angles)
            %MEMGETBURNERTHRUST Summary of this function goes here
            %   Detailed explanation goes here
            [Thrust, M, T] = obj.fh(angles);
        end
    end
end
```

Published with MATLAB® R2015b

```
%  Thomas Satterly
% SQP

clear all;
tic;

x0 = linspace(20, 20, 20);

maxAngleDiff = 5;
minMach = 1.05;
minEndMach = 1.15;
maxTemp = 2700;
memObj = aae550.final.memBurner();
f_x = @(angles) aae550.final.fx(angles, memObj);
g_x = @(angles) aae550.final.gx(angles, memObj, maxAngleDiff, minMach,
 minEndMach, maxTemp);

% no linear inequality constraints
A = [];
b = [];
% no linear equality constraints
Aeq = [];
beq = [];
% lower bounds (no explicit bounds in example)
lb = zeros(1, numel(x0));
% upper bounds (no explicit bounds in example)
ub = 40 * ones(1, numel(x0));
% set options for medium scale algorithm with active set (SQP as
 described
% in class; these options do not include user-defined gradients
options = optimoptions('fmincon','Algorithm','sqp', 'Display','iter-
detailed', ...
    'SpecifyObjectiveGradient', true, ...
    'SpecifyConstraintGradient', true, ...
    'UseParallel', false);
% initial guess  - note that this is infeasible

[x,fval,exitflag,output] =
 fmincon(f_x,x0,A,b,Aeq,beq,lb,ub,g_x,options);
toc;
```

Published with MATLAB® R2015b